HITCON 2019 QUAL

Challenge dadadb

angelboy@chroot.org

@scwuaptx

Outline

- Description
- Vulnerability
- Exploit

Description

- Windows x64 on windows sever 2019
  - Similar to Windows 10 (1809)
- DEP
- ASLR
- CFG
- Disallow Child Process
  - You can not create new process.

Description

- It a windows heap challenge
  - If you are unfamiliar at windows heap, you can reference my slide
    - https://www.slideshare.net/AngelBoy1/windows-10-nt-heap-exploitation-english-version
    - https://slideshare.net/AngelBoy1/windows-10-nt-heap-exploitation-chinese-version

Description

- It a windows heap challenge
  - The following command in Windbg will be very helpful
    - !heap
    - !heap -a [heap address]
    - dt _HEAP [heap address]
    - dt _HEAP_LIST_LOOKUP
    - dt _LFH_HEAP

Description

- Login
  - Add
  - View
  - Remove
  - logout
- Private Heap
  - More stable than default heap

Description

- A simple database
- After login
  - You can add/read/remove data by key
  - Use array and linked list to store data

Description

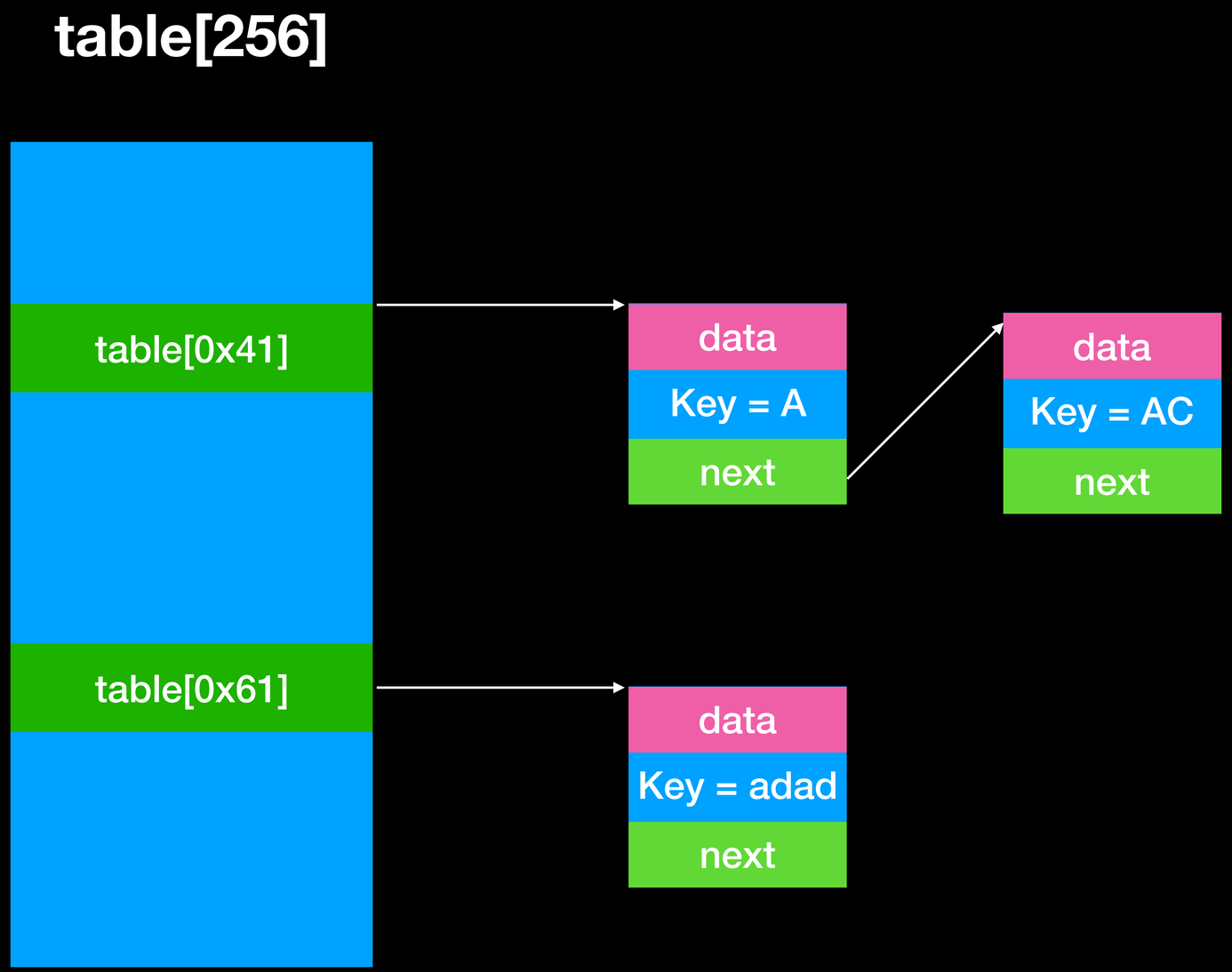
- Structure
  - KEY_SIZE 0x40

```
struct node{
    char* data;
    size_t size;
    char key[KEY_SIZE+1];
    struct node* next;

};

struct node* table[256];
```

Description



Description

- Login
  - Read user data from “user.txt”
  - If user & pass match
    - Set is_login flag

```
printf("User:");
read_input(user, 0x20);
printf("Password:");
read_input(pass, 0x20);
memset(buf, 0, 0x100);

if (!fp) {
    fopen_s(&fp,"user.txt", "r");
    if (!fp)
        _exit(0);
}
fread(buf, 0x100, 0x1, fp);
fseek(fp, 0, SEEK_SET);
snprintf(cmpbuf,0x20,"%s:", user);
ret = strstr(buf, cmpbuf);
```

Description

- ADD
  - Add data by key
- It will search node in table.
- If not found, it will create new node and insert node to the table
  - Insert into in front of linked list
- If found it will reuse the node and allocate a new data buffer

Description

- READ
  - Read data by key
- It will use key to search node in table
- If found, it will write data to stdout

Description

- REMOVE
  - Delete data by key
- It will use key to search node in table
- If found, it will delete the node in table and linked

Description

- Private Heap
  - Only use in my malloc
  - Independent memory pool

```
void* hcalloc(size_t cnt,size_t size) {
    return HeapAlloc(hHeap, HEAP_ZERO_MEMORY, size*cnt);
}

void hfree(void* ptr) {
    HeapFree(hHeap, 0, ptr);
}
```

Vulnerability

- Heap Overflow
- When it reuse the node
  - It will use the size of old data buffer when reading data to new buffer
- It will lead to heap overflow, when the old size of data buffer larger than new buffer

```
read_input(buf, KEY_SIZE);
target = search(buf);
if (target) {
    hfree(target->data);
    printf("Size:");
    size = read_long();
    if (size >= 0x1000)
        size = 0x1000;
    target->data = (char*)hcalloc(1,size) ;
    printf("Data:");
    read_input(target->data, target->size);
}
```

Exploit

- Leak
- Arbitrary memory writing
- ROP

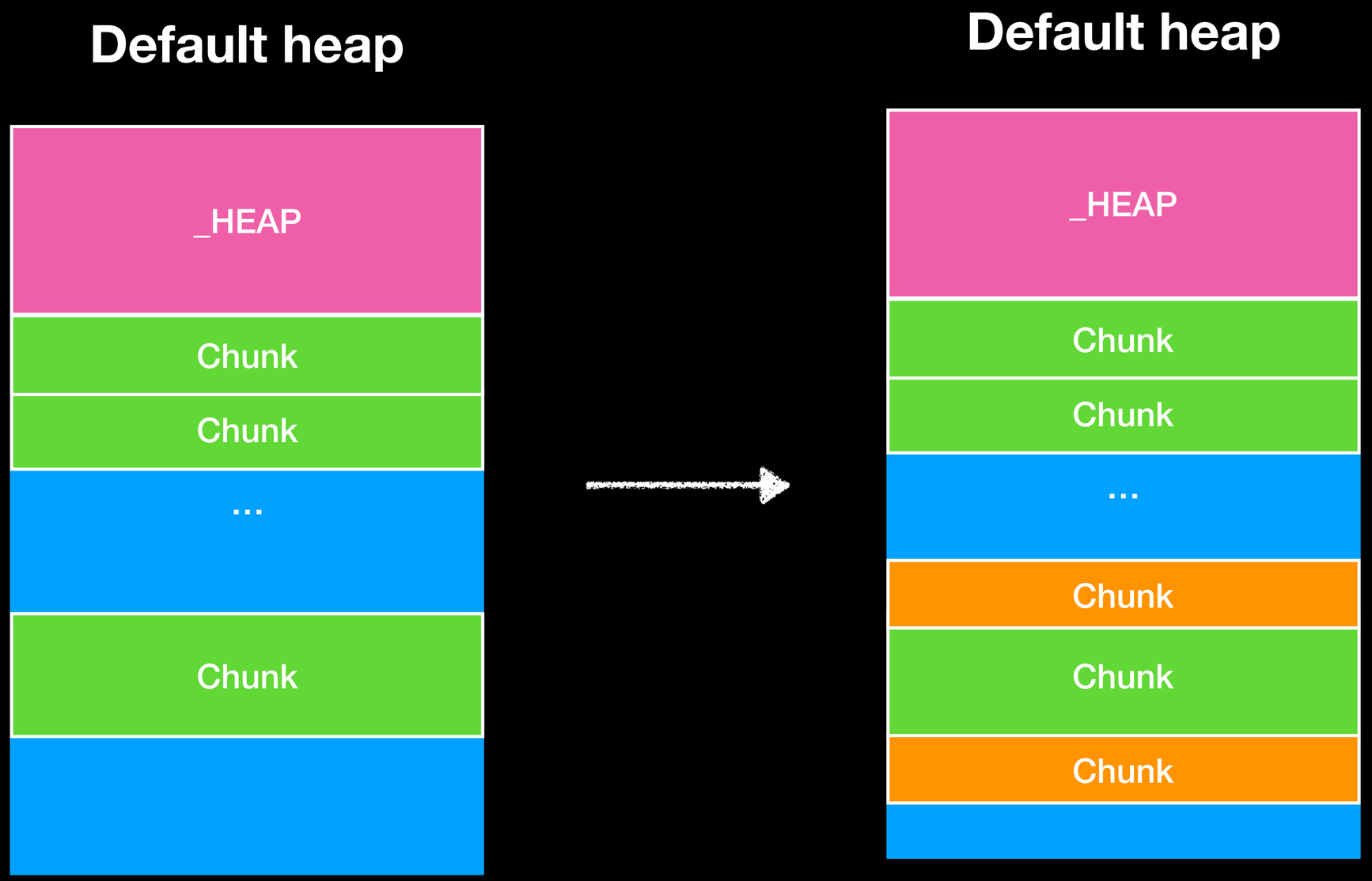
Exploit

- Leak
  - Because it use private heap, we can easy use heap overflow to overwrite data pointer to do arbitrary memory reading
  - But I will demonstrate in normal case (default heap)

Exploit

- Leak
  - In default heap case, it will use Heapallocate when program start.
    - There are many unstable hole in the heap.
  - It also used in many Windows API, so it hard to locate heap and heap layout

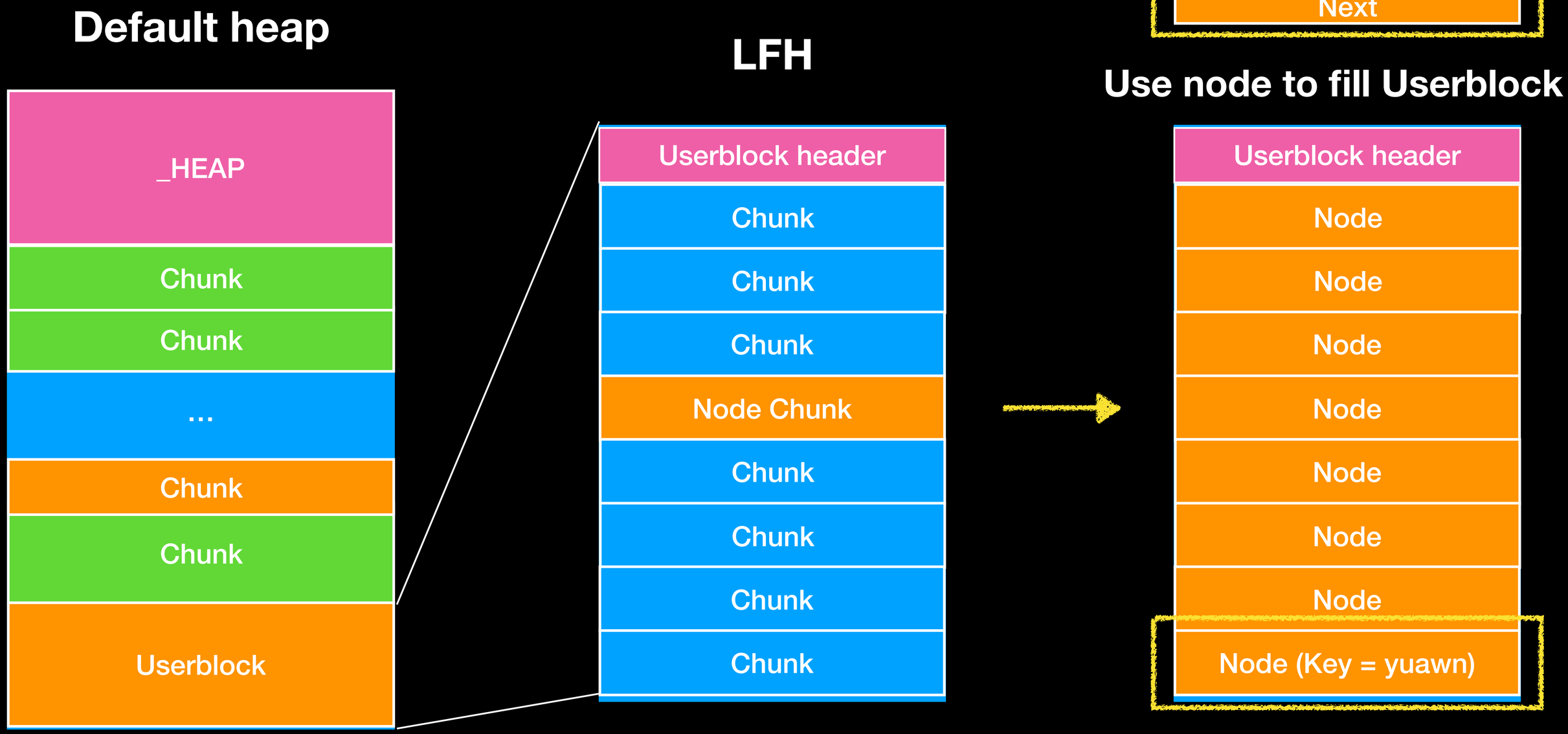
Exploit


Default heap
_HEAP
Chunk
Chunk
...
Chunk
Default heap
_HEAP
Chunk
Chunk
...
Chunk
Chunk
Chunk

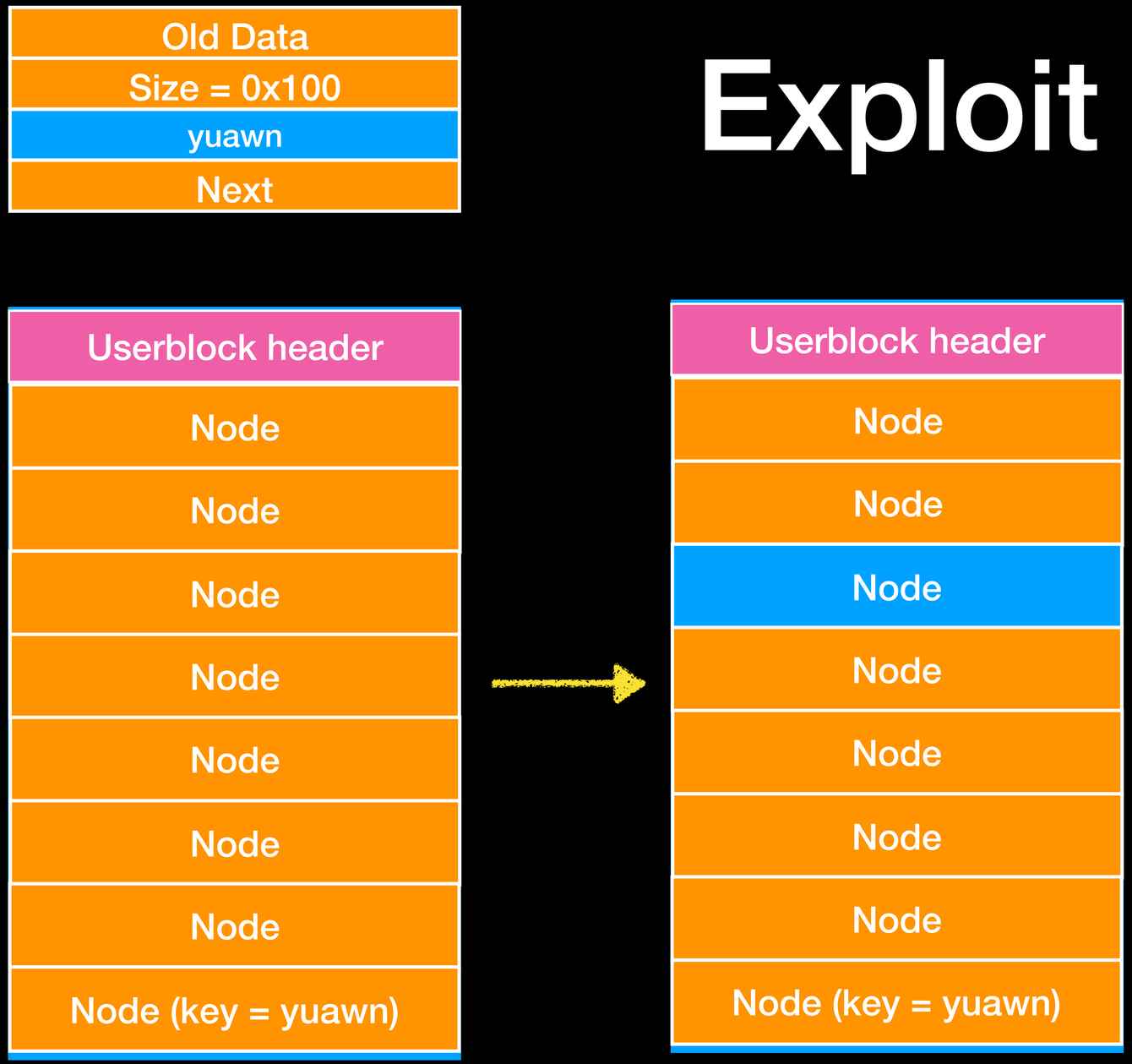
Exploit

- Leak
  - If we want to have a stable leakage, we can use LFH
    - There are less checks in LFH.
      - We can use it to prevent some heap detection.

Exploit

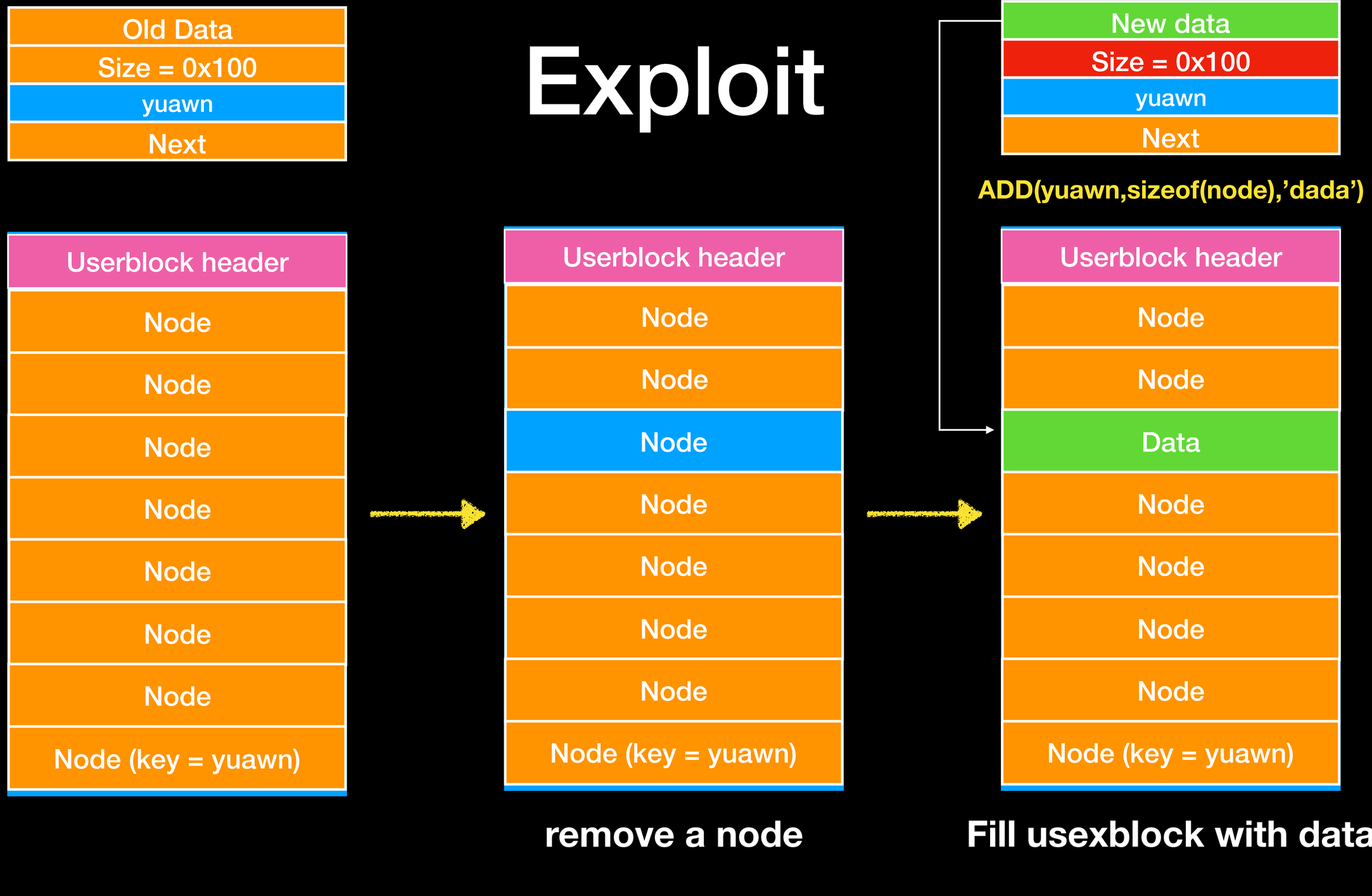

Default heap
_HEAP
Chunk
Chunk
...
Chunk
Chunk
Userblock
LFH
Userblock header
Chunk
Chunk
Chunk
Node Chunk
Chunk
Chunk
Chunk
Chunk
Use node to fill Userblock
Userblock header
Node
Node
Node
Node
Node
Node
Node
Old Data
Size = 0x100
yuawn
Next
Node (Key = yuawn)

Exploit

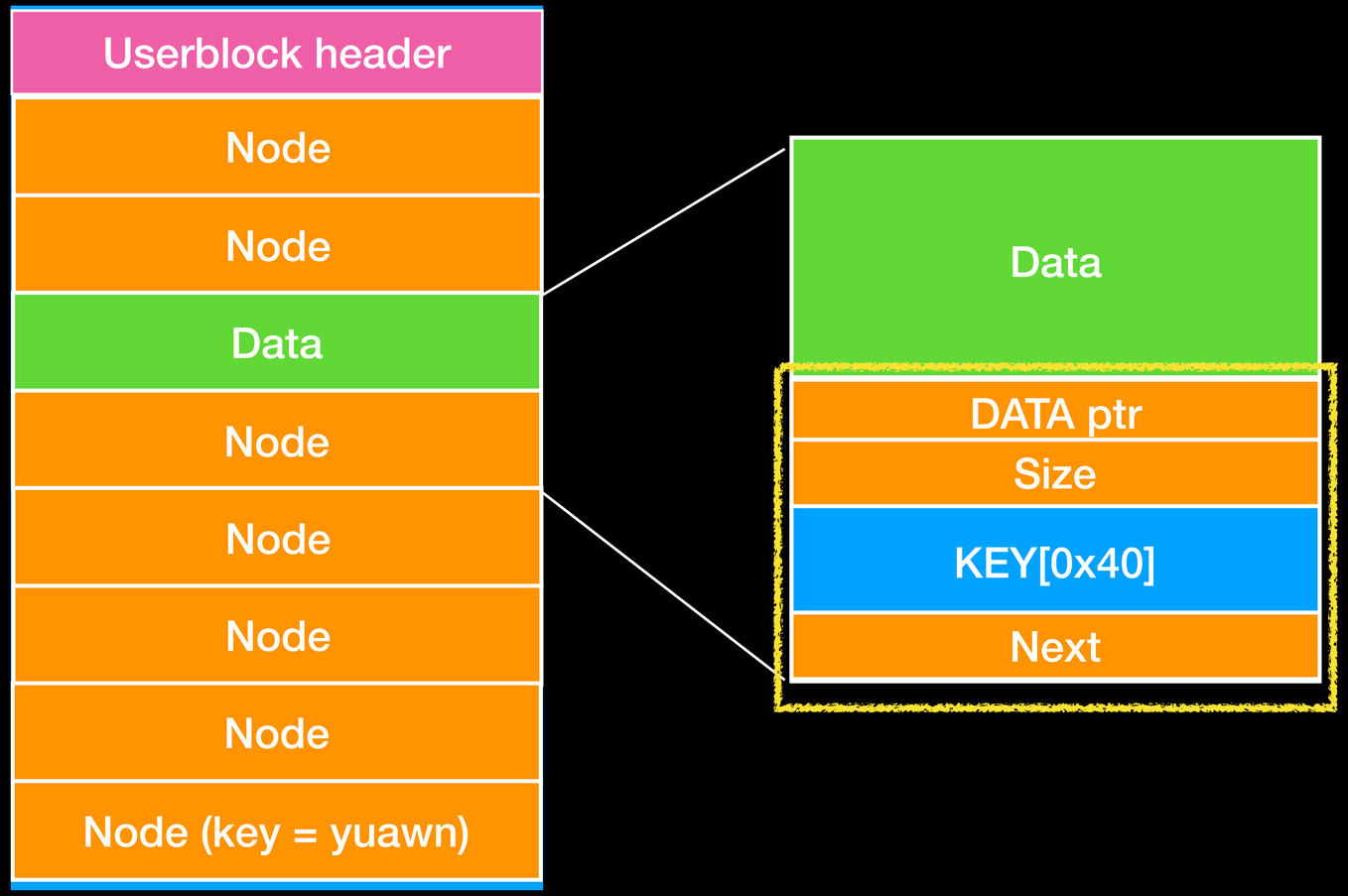


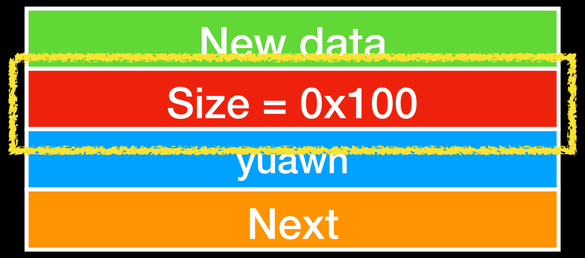
remove a node

Exploit


Old Data
Size = 0x100
yuawn
Next
New data
Size = 0x100
yuawn
Next
ADD(yuawn,sizeof(node),'dada')
Userblock header
Node
Node
Node
Node
Node
Node
Node
Node (key = yuawn)
Userblock header
Node
Node
Node
Node
Node
Node
Node
Node (key = yuawn)
Userblock header
Node
Node
Data
Node
Node
Node
Node
Node (key = yuawn)
remove a node
Fill usexblock with data

Exploit

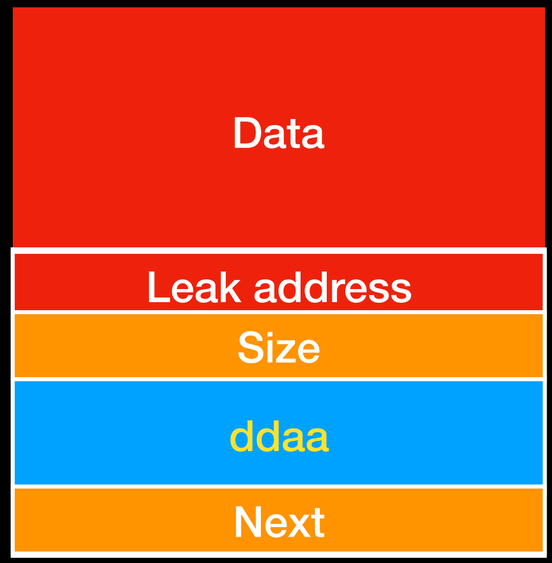




- Now we can use vulnerability to leak something in next chunk.
  - Heap address (from data ptr)
  - Key of next chunk (node)
- Assume key of next chunk is ddaa

Exploit

- Leak
  - After leaking heap address, we can use heap overflow to overwrite next chunk with address we want to leak
    - By the way, Heap overflow will corrupt header of next chunk.
    - When we use add to update data, it will free original data. In back-end allocator it will encounter heap corruption detect if you do not forge header.
    - In LFH, it does not check !
  - That is, we can do arbitrary memory reading by key(ddaa)



Exploit

- Leak
  - After we can do arbitrary memory reading
    - We can get address of
      - ntdll (from _HEAP->lock)
      - PEB (from ntdll)
      - binary (from ntdll!PebLdr)
      - kernel32 (from IAT of binary)

Exploit

- Leak
  - After we can do arbitrary memory reading
    - We can get address of
      - TEB (from PEB)
      - Stack address (from TEB)
        - The location of return address (scan return address at stack)

Exploit

- Arbitrary memory writing
  - Next step, we need to do arbitrary memory writing to overwrite return address.
  - Intended solution
    - Forge a fake chunk at bss to overwrite fp.
  - Unintended solution
    - Forge a fake chunk at stack to overwrite return address directly.

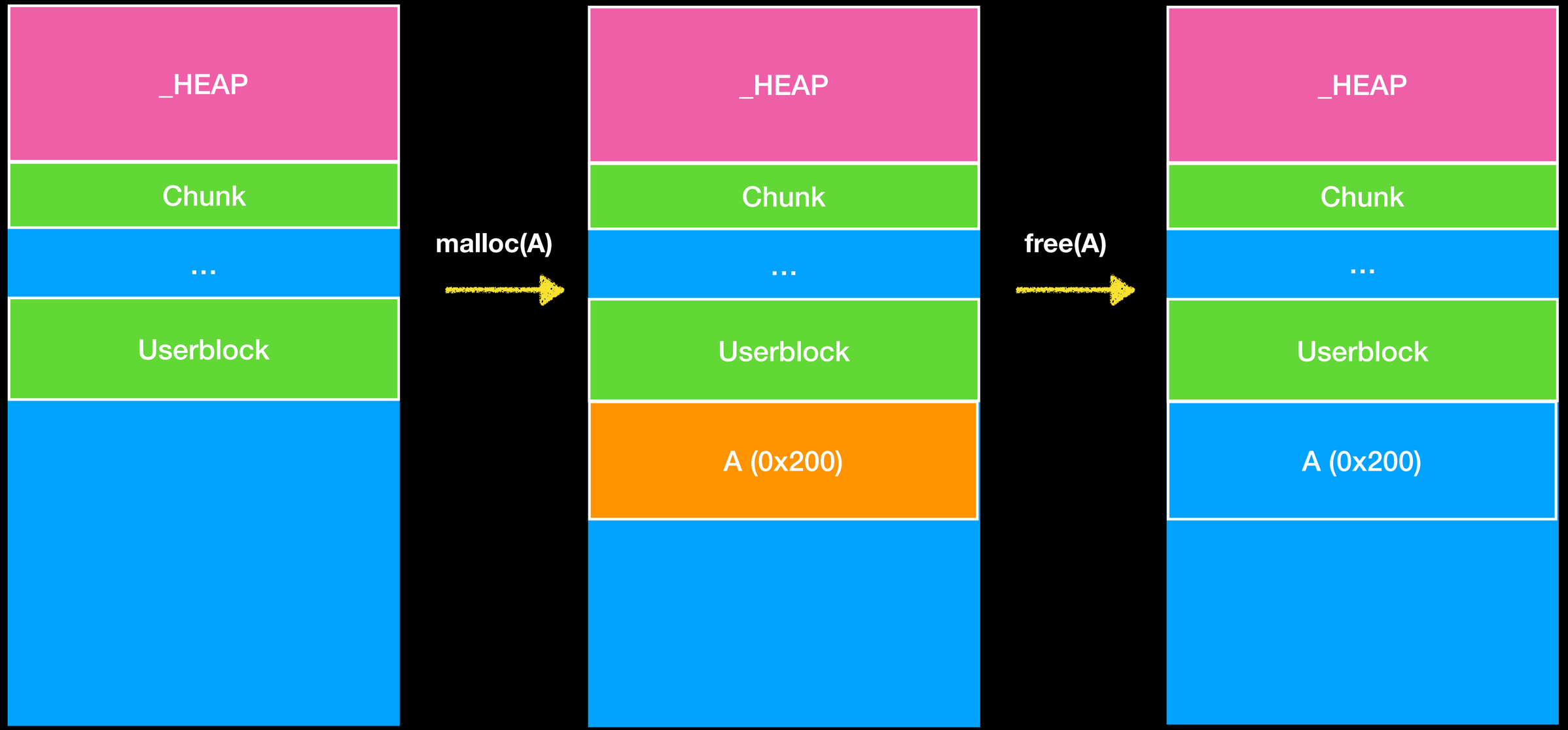
Exploit

- Arbitrary memory writing
  - In order to forge a fake chunk at bss to overwrite fp, we need to prepare some chunk in back-end allocator first.
  - I divide into three part
    - Prepare a chunk used to overwrite other chunk (chunk A)
    - Forge a legal chunk at password buffer
    - Prepare a free chunk to be overwrite with fake Flink & Blink.

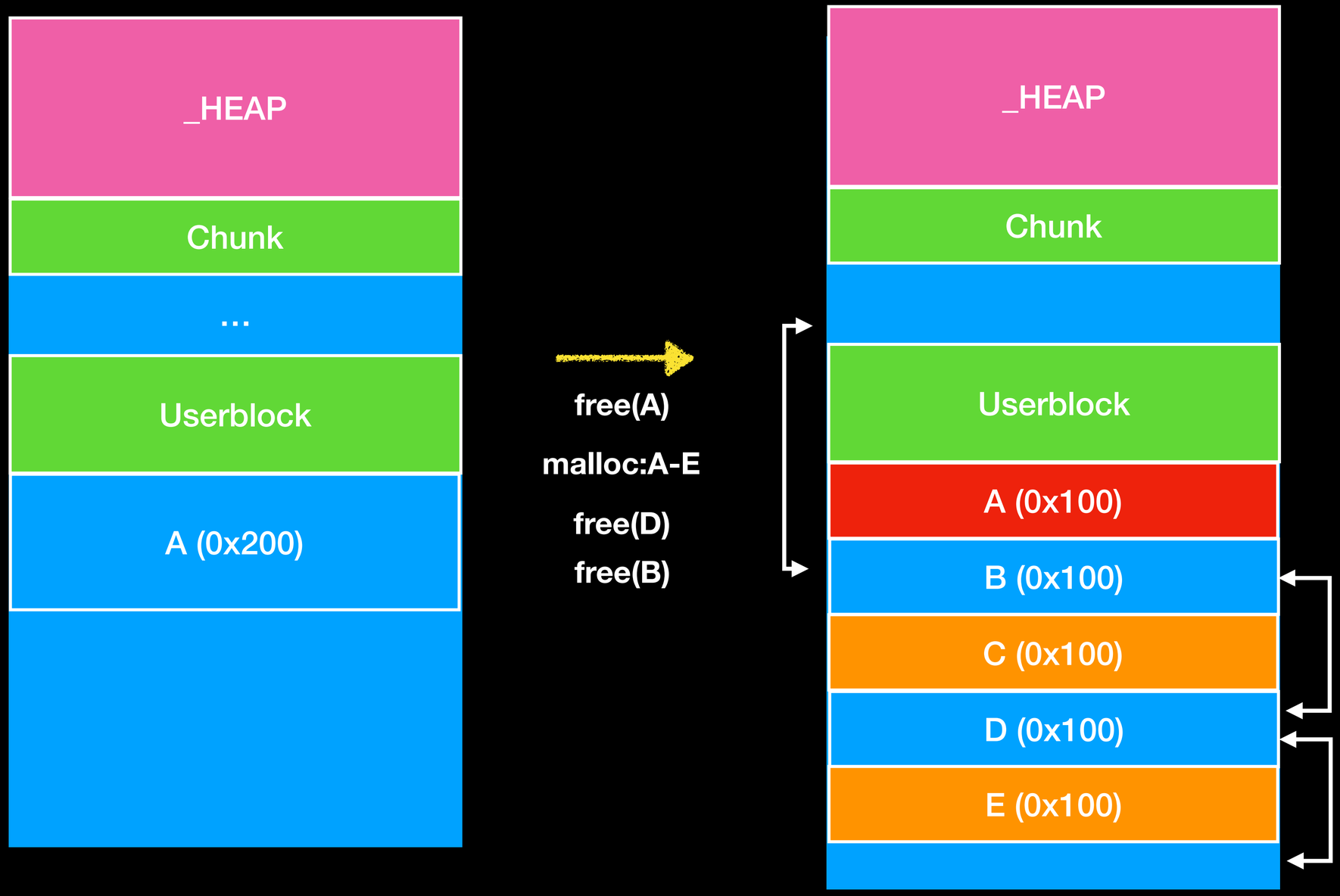
Exploit

- Arbitrary memory writing
  - Prepare a chunk used to overwrite other chunk
  - We can choose a suitable chunk such that allocate in the largest free chunk

Exploit

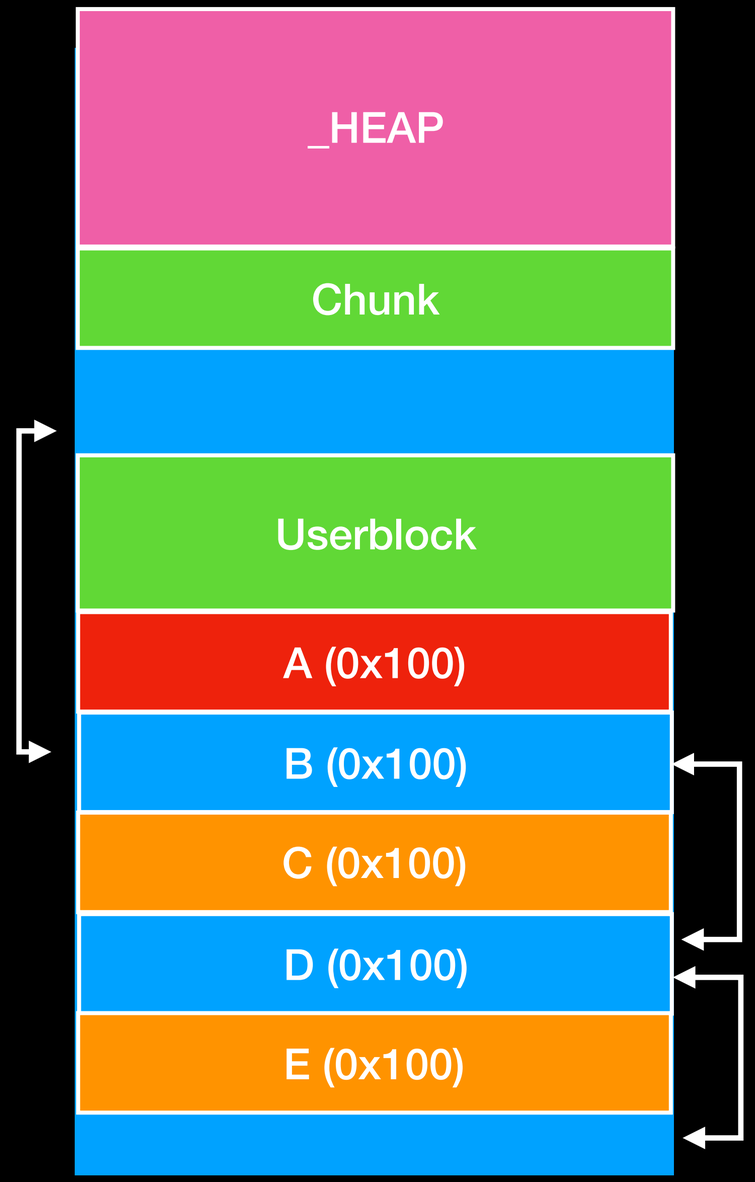

_HEAP
Chunk
...
Userblock
malloc(A)
_HEAP
Chunk
...
Userblock
A (0x200)
free(A)
_HEAP
Chunk
...
Userblock
A (0x200)

Exploit


_HEAP
Chunk
...
Userblock
A (0x200)
free(A)
malloc:A-E
free(D)
free(B)
_HEAP
Chunk
Userblock
A (0x100)
B (0x100)
C (0x100)
D (0x100)
E (0x100)

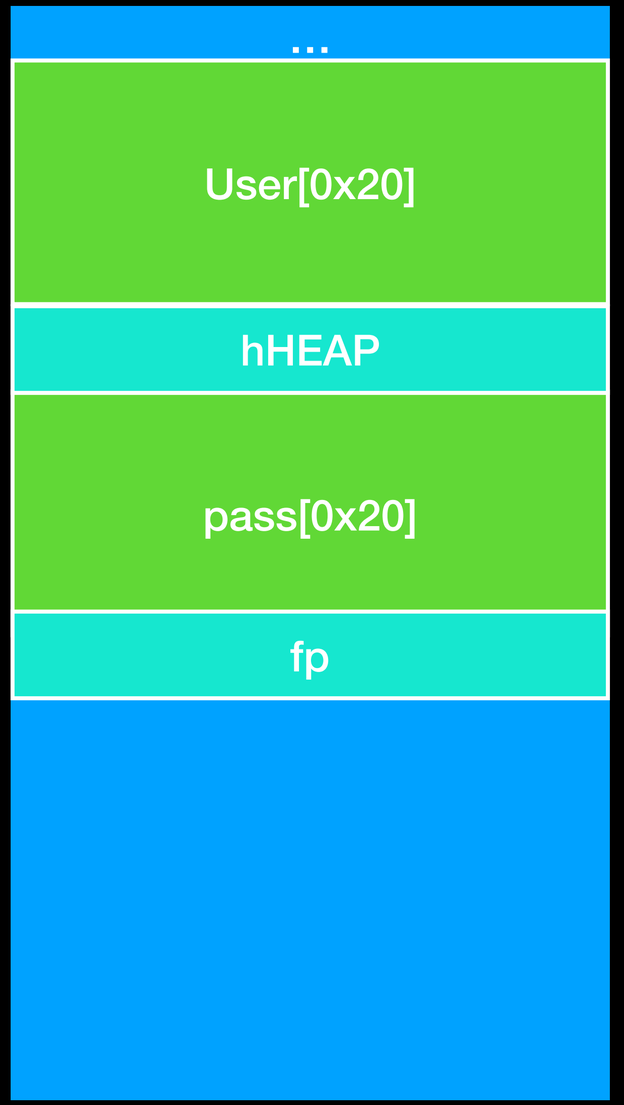
Exploit

- Arbitrary memory writing
  - After we prepare some chunks, the heap layout will like the diagram
    - A can be used to overflow B
    - B, D are free chunk with same size
  - We can use A to leak Flink and Blink of B
    - That is, we can get address of B and D



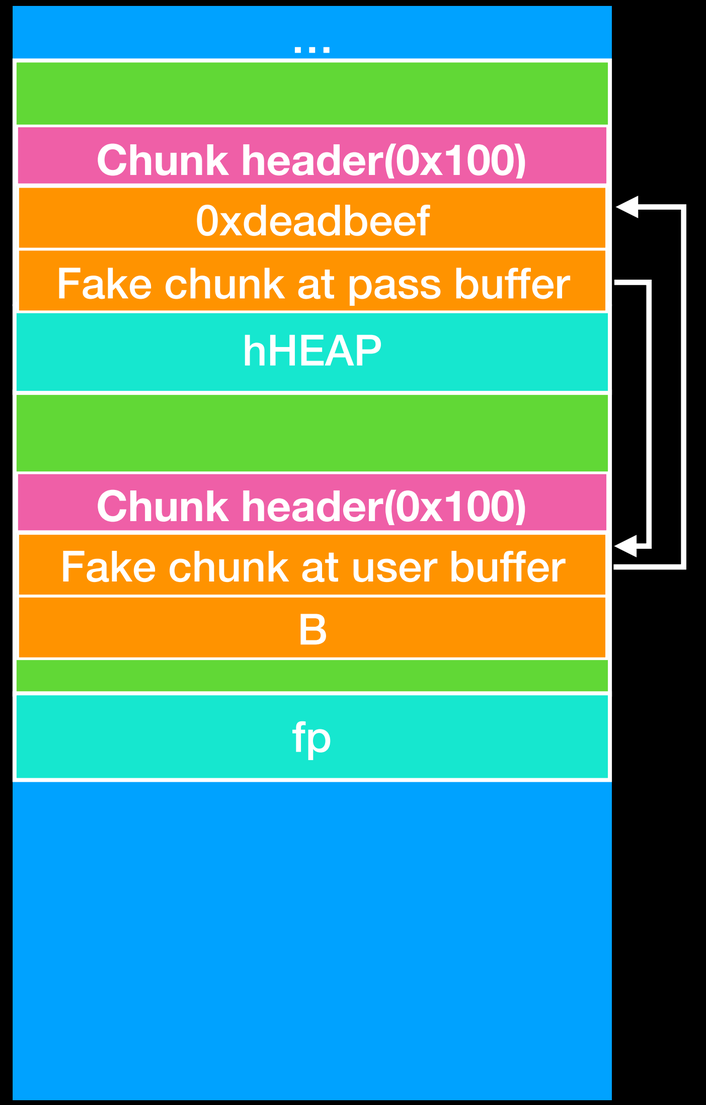
Exploit

- Arbitrary memory writing
  - Next step is forging a legal chunk at password buffer so that we can use malloc to get the chunk.
  - At first, the layout will like the diagram.



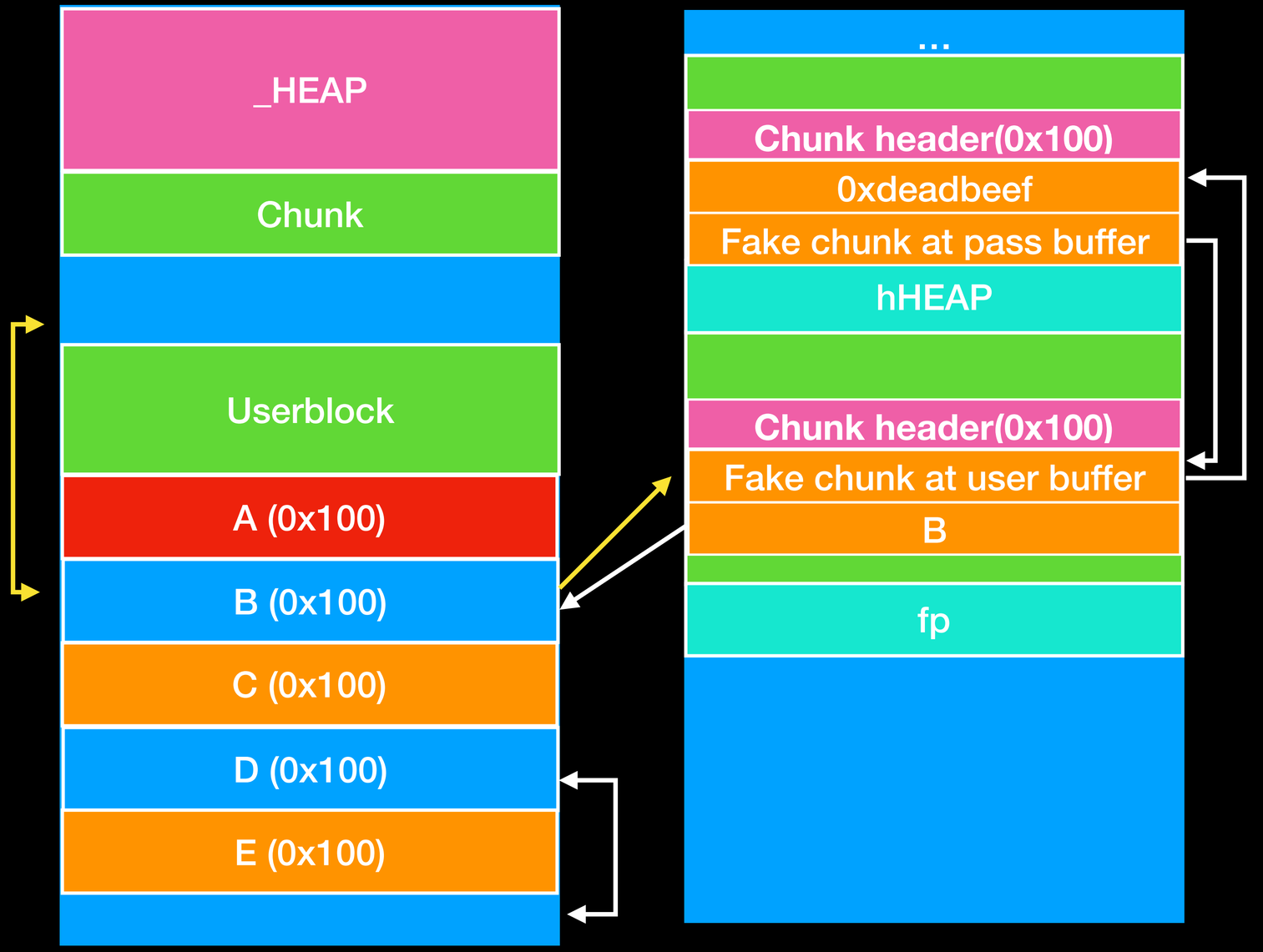
Exploit

- Arbitrary memory writing
  - We can forge a legal chunk with same header of chunk B



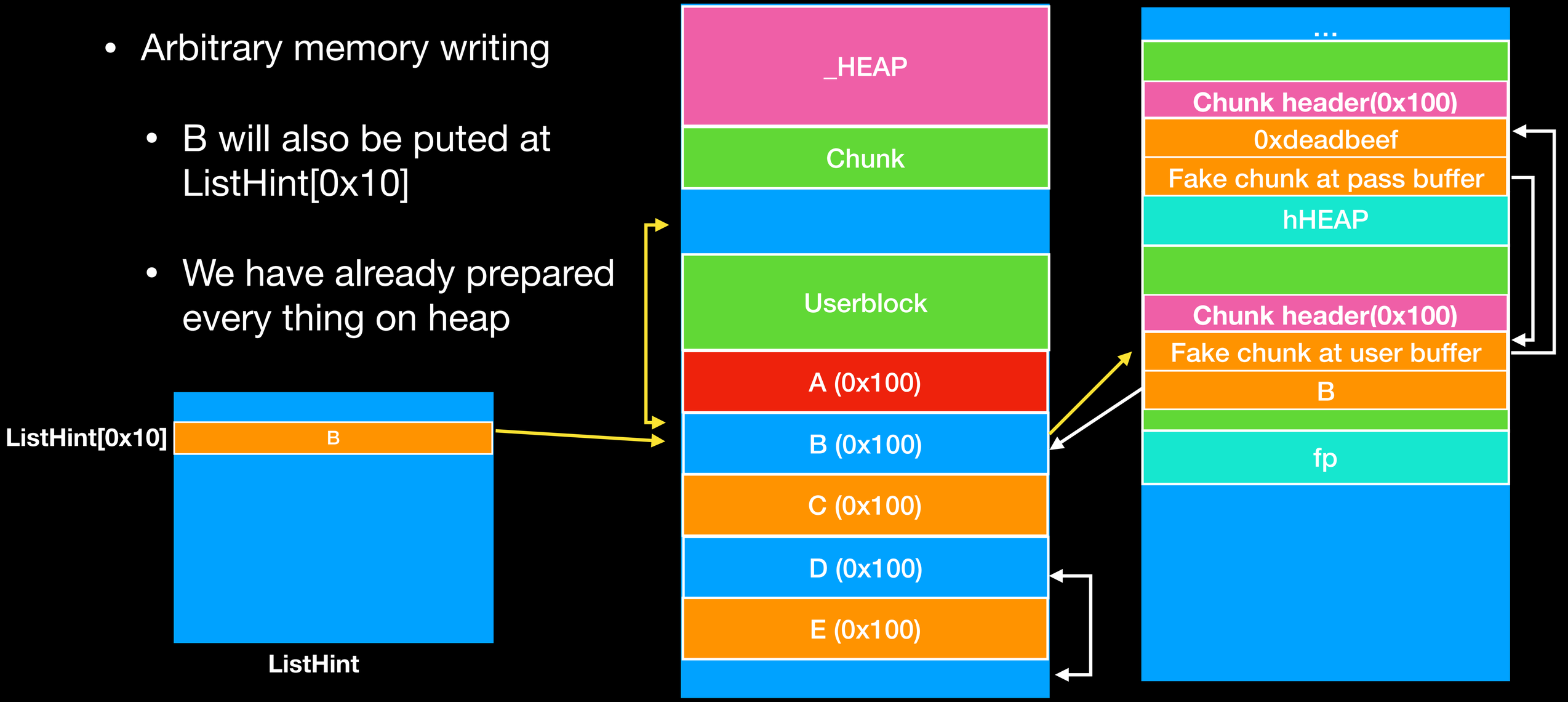
Exploit

- Arbitrary memory writing
  - Then use chunk A to overwrite Flink and Blink of chunk B
  - You should make sure that double linked does not corrupted.



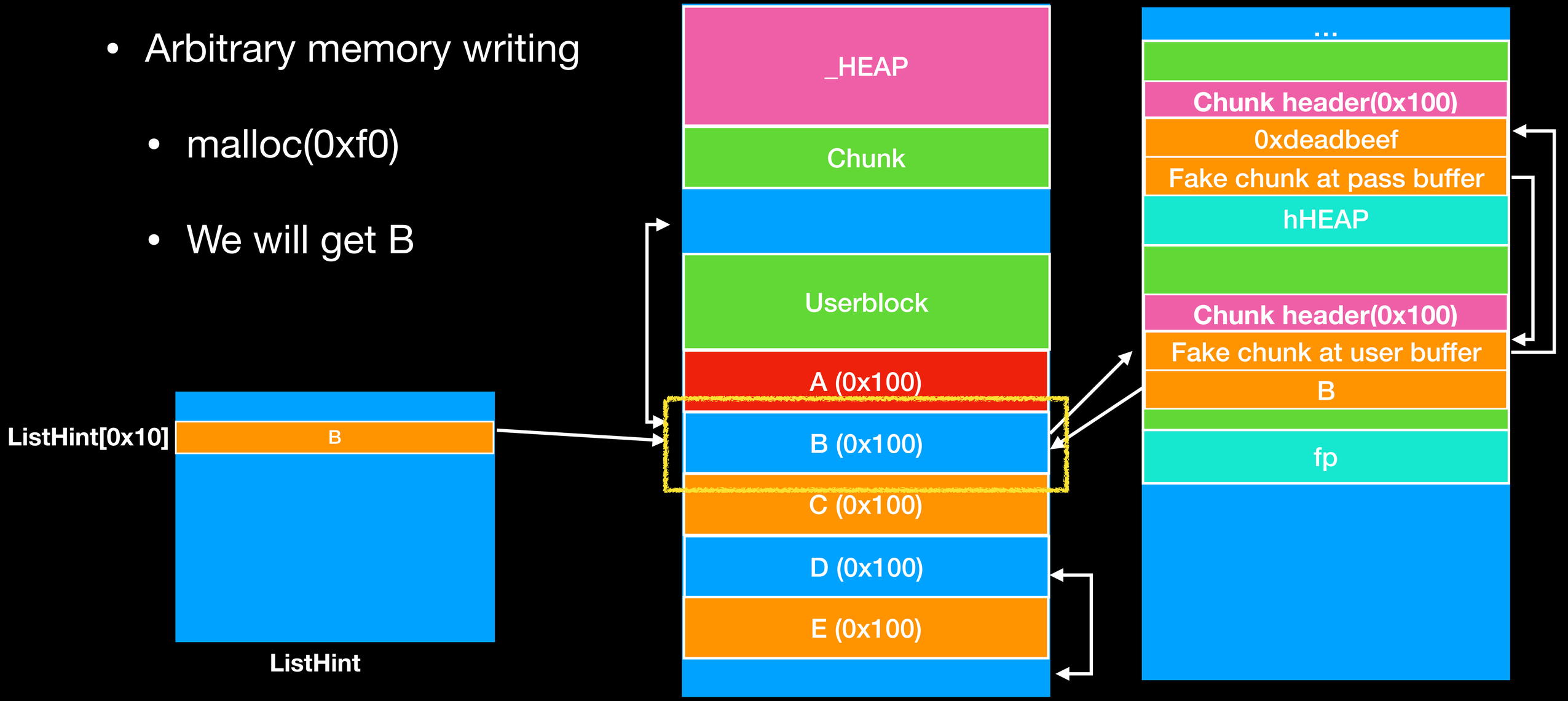
Exploit

- Arbitrary memory writing
  - B will also be puted at ListHint[0x10]
  - We have already prepared every thing on heap



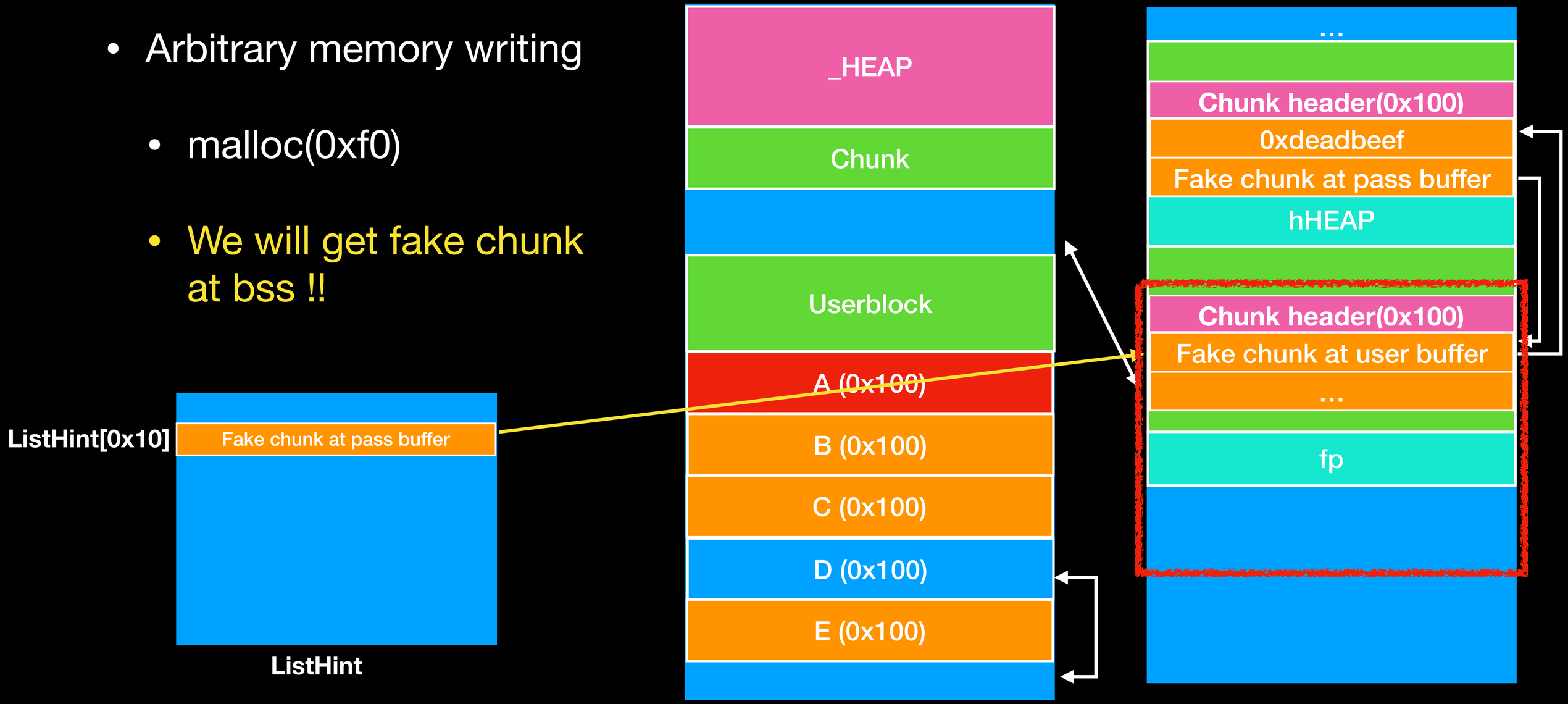
Exploit

- Arbitrary memory writing
  - malloc(0xf0)
  - We will get B



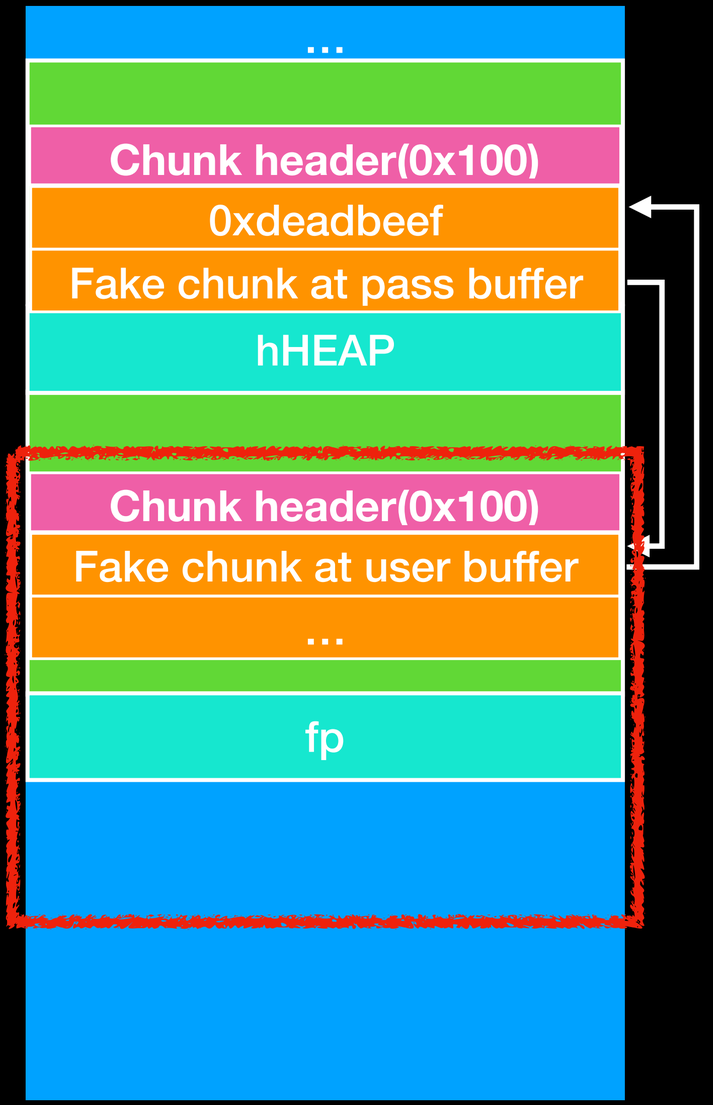
Exploit

- Arbitrary memory writing
  - malloc(0xf0)
  - We will get fake chunk at bss !!



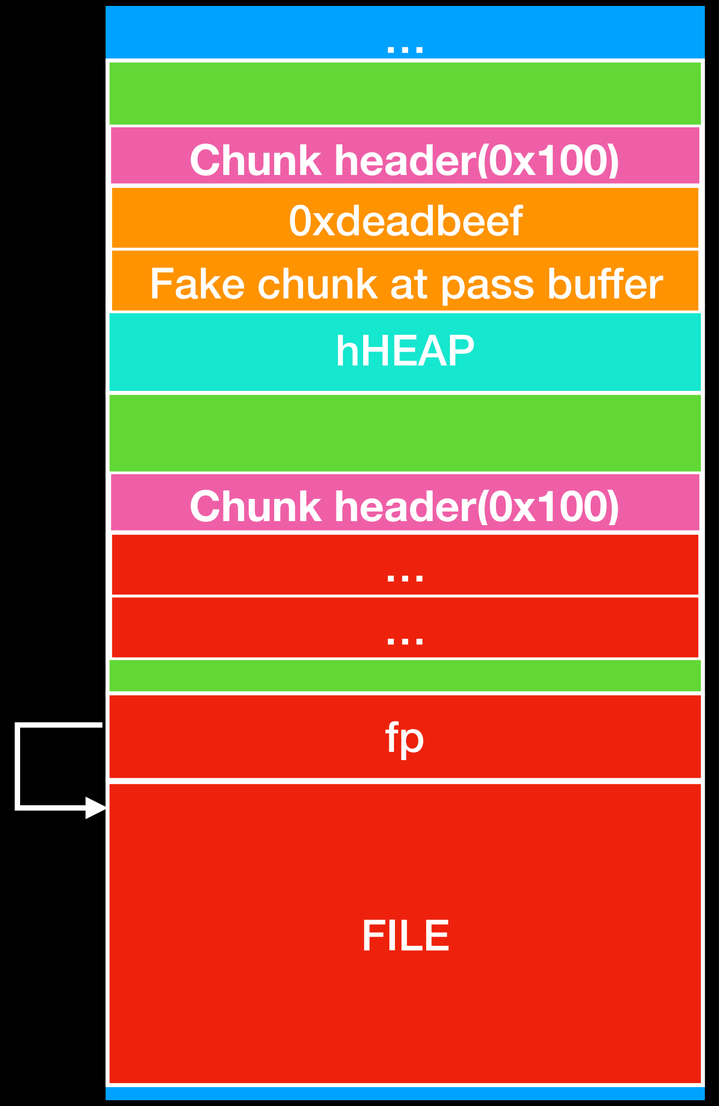
Exploit

- Arbitrary memory writing
  - By the way, it only check header and double linked list of chunk when it use malloc.
    - It does not check the header of next chunk
    - Sometimes it’s very useful



Exploit

- Arbitrary memory writing
  - Now, we can overwrite fp !!
    - We can forge the FILE structure follow by fp and overwrite fp with the fake FILE structure.
    - That is, we can use fread to do arbitrary memory writing !



Make FILE structure great again

- How about FILE on Windows ?
  - No vtable in FILE
  - It also has stream buffer pointer
    - You can corrupt it to achieve arbitrary memory reading and writing

Make FILE structure great again

- File Structure in ucrtbase.dll
  - _ptr
    - Current pointer
  - _base
    - The base of stream buffer

```
121 struct __crt_stdio_stream_data
122 {
123     union
124     {
125         FILE  _public_file;
126         char* _ptr;
127     };
128
129     char*            _base;
130     int              _cnt;
131     long             _flags;
132     long             _file;
133     int              _charbuf;
134     int              _bufsiz;
135     char*            _tmpfname;
136     CRITICAL_SECTION _lock;
137 };
138
```

Make FILE structure great again

- File Structure in ucrtbase.dll
  - _cnt
    - The rest of the buffer is not read out.

```
121 struct __crt_stdio_stream_data
122 {
123     union
124     {
125         FILE  _public_file;
126         char* _ptr;
127     };
128
129     char*             _base;
130     int               _cnt;
131     long              _flags;
132     long              _file;
133     int               _charbuf;
134     int               _bufsiz;
135     char*             _tmpfname;
136     CRITICAL_SECTION _lock;
137 };
138
```

Make FILE structure great again

- File Structure in ucrtbase.dll
  - _flags
    - Record the attribute of the File stream
      - Has buffer
      - Read/Write
      - …

```
121 struct __crt_stdio_stream_data
122 {
123     union
124     {
125         FILE  _public_file;
126         char* _ptr;
127     };
128
129     char*             _base;
130     int               _cnt;
131     long              _flags;
132     long              _file;
133     int               _charbuf;
134     int               _bufsiz;
135     char*             _tmpfname;
136     CRITICAL_SECTION _lock;
137 };
138
```

Make FILE structure great again

- File Structure in ucrtbase.dll
  - _file
    - File descriptor
  - _charbuf
    - Local buffer

```
121 struct __crt_stdio_stream_data
122 {
123     union
124     {
125         FILE  _public_file;
126         char* _ptr;
127     };
128
129     char*            _base;
130     int              _cnt;
131     long             _flags;
132     long             _file;
133     int              _charbuf;
134     int              _bufsiz;
135     char*            _tmpfname;
136     CRITICAL_SECTION _lock;
137 };
138
```

Make FILE structure great again

- File Structure in ucrtbase.dll
  - _bufsiz
    - Size of buffer
  - Lock
    - Just lock

```
121 struct __crt_stdio_stream_data
122 {
123     union
124     {
125         FILE  _public_file;
126         char* _ptr;
127     };
128
129     char*             _base;
130     int               _cnt;
131     long              _flags;
132     long              _file;
133     int               _charbuf;
134     int               _bufsiz;
135     char*             _tmpfname;
136     CRITICAL_SECTION _lock;
137 };
138
```

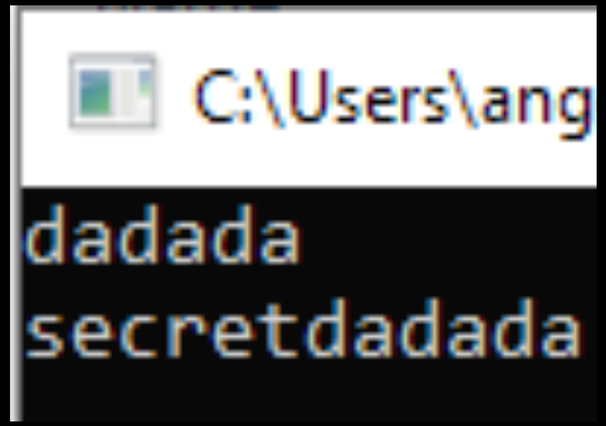
Make FILE structure great again

- Arbitrary memory reading
  - fwrite
    - Set the _file to the file descriptor of stdout
    - Set _flag to _IOWRITE | IOBUFFER_USER | _IOUPDATE
    - Set _cnt to 0
    - Set the _base & _ptr to memory address which you want to read
    - Elementsize*count > _bufsize

Make FILE structure great again

- Arbitrary memory reading

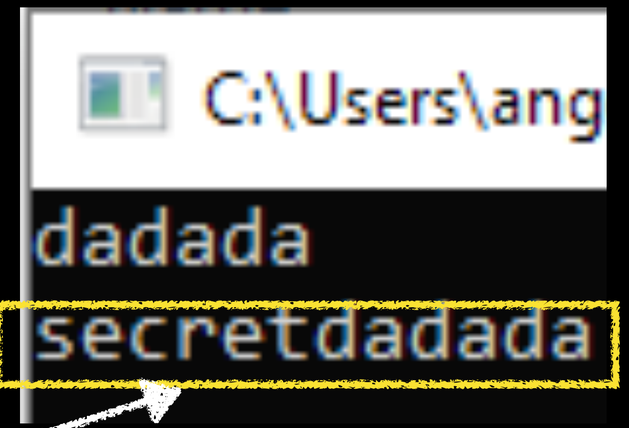
```
const char* msg = "secret";
FILE* fp = NULL;
char* buf = (char*)malloc(0x100);
_read(0, buf, 0x100);
fopen_s(&fp,"key.txt", "w");
((struct file_stream* )fp)->_flags = _IOBUFFER_USER | _IOWRITE | _I
((struct file_stream*)fp)->_cnt = 0x0;
((struct file_stream*)fp)->_bufsize = 0;
((struct file_stream*)fp)->_base = (char*)msg;
((struct file_stream*)fp)->_ptr = (char*)msg+6;
((struct file_stream*)fp)->_file = 0x1;
fwrite(buf, 1, 100, fp);
```



Make FILE structure great again

- Arbitrary memory reading

```
const char* msg = "secret";
FILE* fp = NULL;
char* buf = (char*)malloc(0x100);
_read(0, buf, 0x100);
fopen_s(&fp,"key.txt", "w");
((struct file_stream* )fp)->_flags = _IOBUFFER_USER | _IOWRITE | _I
((struct file_stream*)fp)->_cnt = 0x0;
((struct file_stream*)fp)->_bufsize = 0;
((struct file_stream*)fp)->_base = (char*)msg;
((struct file_stream*)fp)->_ptr = (char*)msg+6;
((struct file_stream*)fp)->_file = 0x1;
fwrite(buf, 1, 100, fp);
```



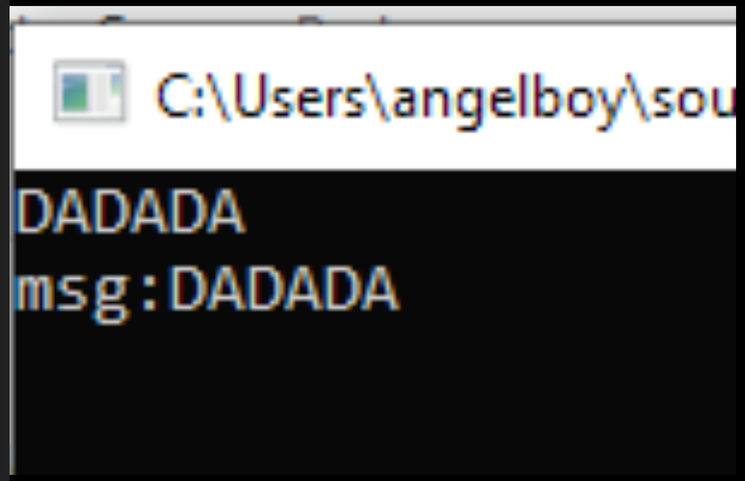
Make FILE structure great again

- Arbitrary memory writing
  - fread
    - Set the _file to file descriptor of stdin
    - Set _flag to _IOALLOCATED | _IOBUFFER_USER
    - Set _cnt to 0
    - Set the _base to memory address which you want to wirte
    - Elementsize*count < _bufsize

Make FILE structure great again

- Arbitrary memory writing

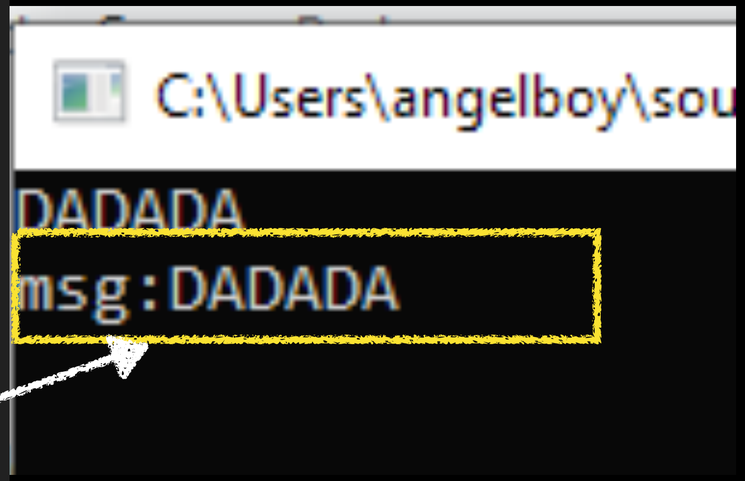
```
FILE* fp = NULL;
char* buf = (char*)malloc(0x100);
char msg[0x100];
memset(msg, 0, 0x100);
fopen_s(&fp, "key.txt", "r");
((struct file_stream*)fp)->_flags = _IOBUFFER_USER | _IOALLOCATED;
((struct file_stream*)fp)->_cnt = 0x0;
((struct file_stream*)fp)->_base = msg;
((struct file_stream*)fp)->_bufsize = 0x30;
((struct file_stream*)fp)->_file = 0x0;
fread(buf, 1, 6, fp);
printf("msg:%s\n",msg);
```



Make FILE structure great again

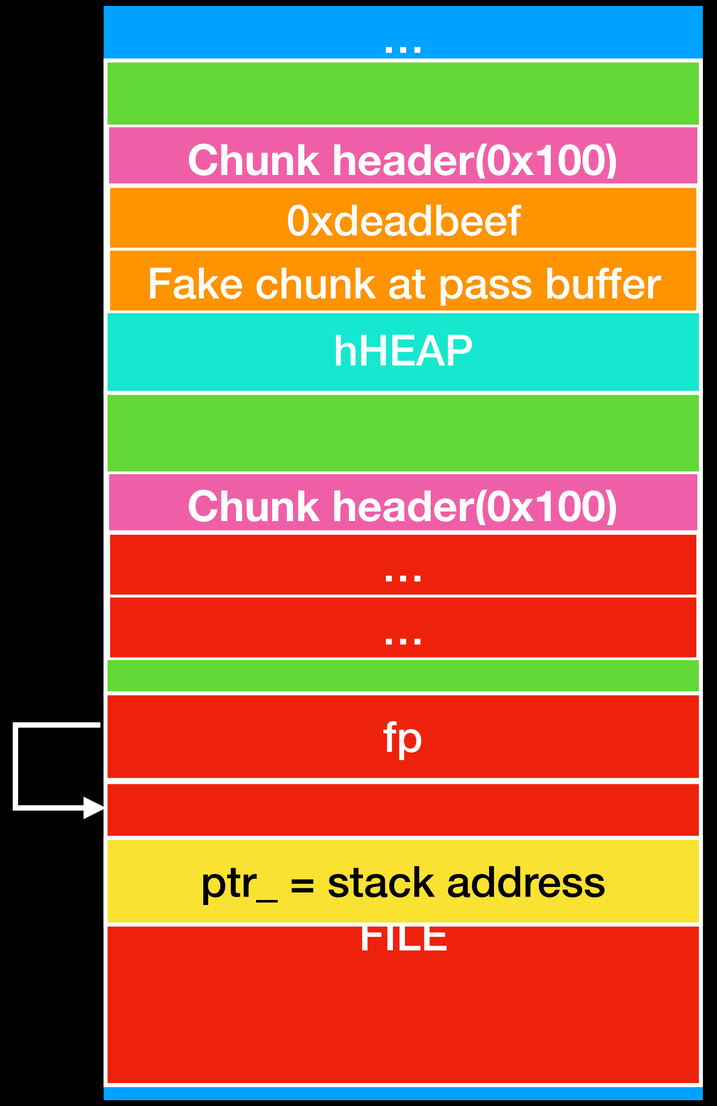
- Arbitrary memory writing

```
FILE* fp = NULL;
char* buf = (char*)malloc(0x100);
char msg[0x100];
memset(msg, 0, 0x100);
fopen_s(&fp, "key.txt", "r");
((struct file_stream*)fp)->_flags = _IOBUFFER_USER | _IOALLOCATED;
((struct file_stream*)fp)->_cnt = 0x0;
((struct file_stream*)fp)->_base = msg;
((struct file_stream*)fp)->_bufsize = 0x30;
((struct file_stream*)fp)->_file = 0x0;
fread(buf, 1, 6, fp);
printf("msg:%s\n",msg);
```



Exploit

- ROP
  - After we have arbitrary memory writing we can overwrite return address on stack with ROP
  - But it disallow child process , you can not create new process.
    - We need use ROP to read flag.txt, but it’s a little complicated.
    - So we use ROP to do VirtualProtect to change page permission so that we can jump to shellcode.



Exploit

- ROP
  - After we can run shellcode, we can read files more easily.
    - Kernel32
      - CreateFile/ReadFile/GetStdHandle/WriteFile
    - In default heap
      - You should create new heap for windows API, otherwise you will encounter heap detection
      - Overwrite _PEB->ProcessHeap/ucrtbase!crtheap/ntdll!ldrpheap with new heap.

Exploit

- Unintended Solution
  - Forge a fake chunk on stack to overwrite return address
    - You need to leak more data
      - Cookie
      - RSP