House of Force

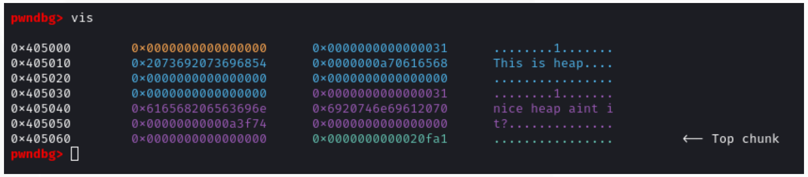
A Linux heap exploitation technique for GLIBC < 2.29

Prerequisites

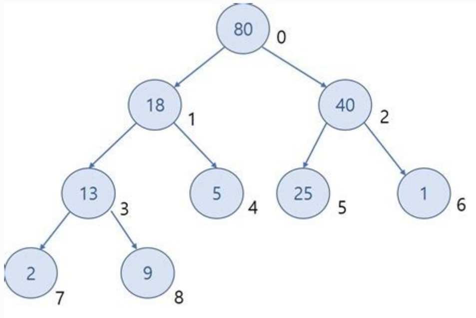
- A Linux distro of choice
  - Tested to work on Kali Linux 2021.1
- GDB – GNU Debugger
- Python3
- Pwntools – Python library for exploit development
- Pwndbg – GDB plug-in for exploit development
- Clone this repo:
  https://github.com/bad5ect0r/monsec-house-of-force

Data structure vs. memory allocation

- We are talking about this heap



- Not this one



What is the heap?

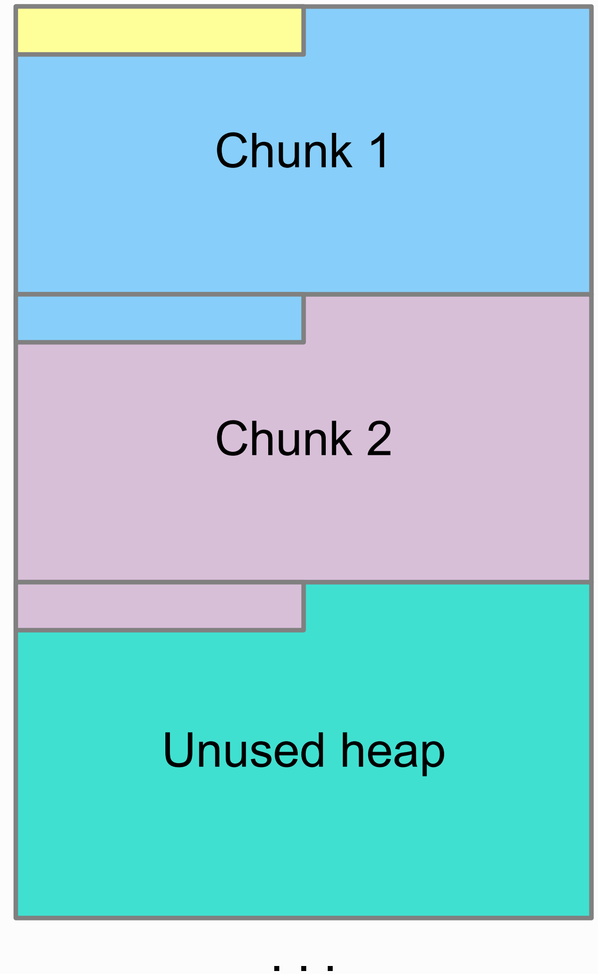
- A space in memory
- Pin board for objects
- Objects are dynamically allocated
- Accessible between functions

How does the heap work?

- **You ask for a chunk of the heap**
  - **It will create the heap if needed**
- **Once you're done with a chunk, you free it**
  - **The program makes that chunk available for future use**

```
pwndbg> vis

0x405000        0x0000000000000000      0x0000000000000031      ........1.......
0x405010        0x2073692073696854      0x0000000a70616568      This is heap....
0x405020        0x0000000000000000      0x0000000000000000      ................
0x405030        0x0000000000000000      0x0000000000000031      ........1.......
0x405040        0x616568206563696e      0x6920746e69612070      nice heap aint i
0x405050        0x00000000000a3f74      0x0000000000000000      t?..............
0x405060        0x0000000000000000      0x0000000000020fa1      ................        <-- Top chunk
pwndbg> 
```



How do I use the heap?

- Malloc gives you chunks of the heap to work with
- Free “gets rid” of chunks that you’re done with

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```

How do I use the heap? - Example

```c
#include <stdlib.h>

int main() {
        void *a = malloc(0x28);
        void *b = malloc(0x38);
        void *c = malloc(0x48);
        void *d = malloc(0x58);

        free(a);
        free(b);
        free(c);
        free(d);
}
```

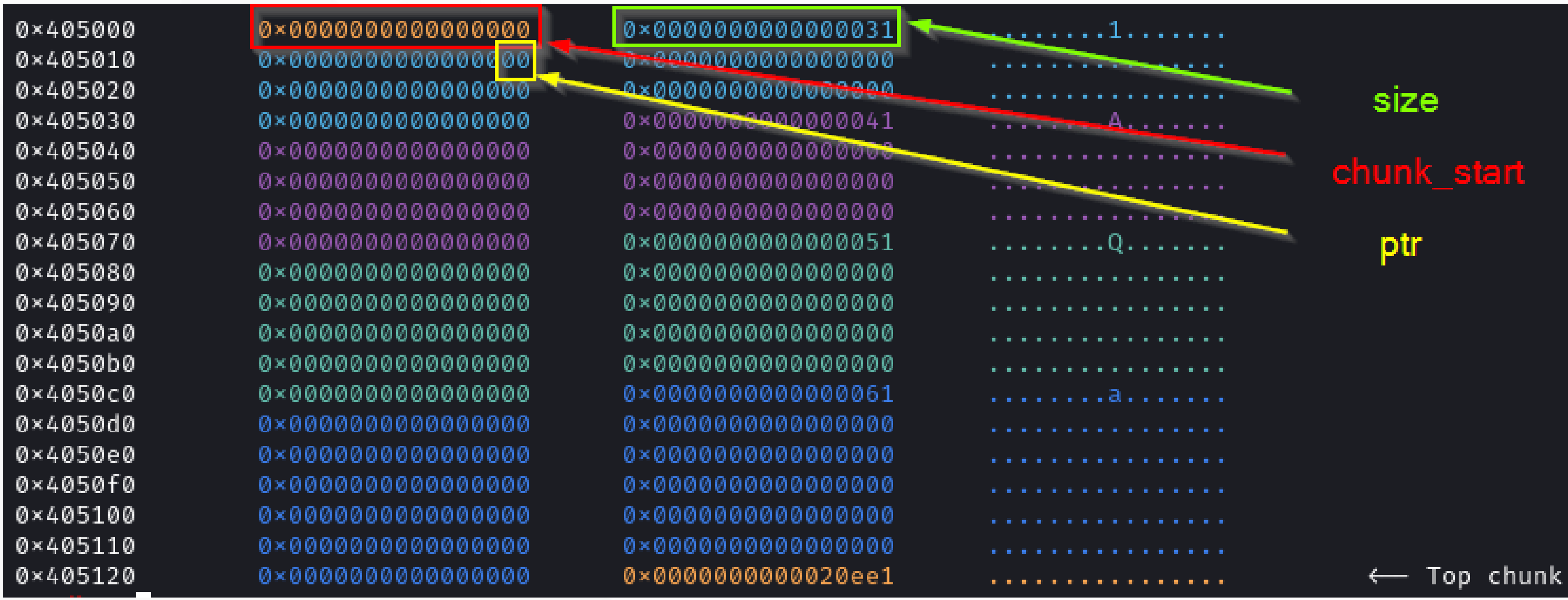
```
pwndbg> vis

0x405000	0x0000000000000000	0x0000000000000031	........1.......
0x405010	0x0000000000000000	0x0000000000000000	................
0x405020	0x0000000000000000	0x0000000000000000	................
0x405030	0x0000000000000000	0x0000000000000041	........A.......
0x405040	0x0000000000000000	0x0000000000000000	................
0x405050	0x0000000000000000	0x0000000000000000	................
0x405060	0x0000000000000000	0x0000000000000000	................
0x405070	0x0000000000000000	0x0000000000000051	........Q.......
0x405080	0x0000000000000000	0x0000000000000000	................
0x405090	0x0000000000000000	0x0000000000000000	................
0x4050a0	0x0000000000000000	0x0000000000000000	................
0x4050b0	0x0000000000000000	0x0000000000000000	................
0x4050c0	0x0000000000000000	0x0000000000000061	........a.......
0x4050d0	0x0000000000000000	0x0000000000000000	................
0x4050e0	0x0000000000000000	0x0000000000000000	................
0x4050f0	0x0000000000000000	0x0000000000000000	................
0x405100	0x0000000000000000	0x0000000000000000	................
0x405110	0x0000000000000000	0x0000000000000000	................
0x405120	0x0000000000000000	0x0000000000020ee1	................	 <-- Top chunk
pwndbg>
```

A closer look at a heap chunk (basics)

- chunk_start = ptr – 0x10;
- chunk_start* is reserved
  - prev_size
  - Not relevant today
- (chunk_start+0x8)* has the size field.
  - 3 least significant bits are flags
- Size is always in multiples of 16
  - I.E, ignore the first nibble (true_size = size & 0xfffffffffffffff0)
  - Because of the flag bits

A closer look at a heap chunk (basics)



The top_chunk

- The heap is like a pizza
- Each chunk you allocate reserves a slice of pizza
- The top_chunk is the whole pizza
- The top_chunk is a chunk in and of itself
  - Kinda like how a whole pizza is a slice of pizza
- Each chunk you allocate is taken away from the top chunk
  - Except for large malloc requests which need to be mmapped
    - Irrelevant for today

The top_chunk

- `void *a = malloc(0x28);`

```
pwndbg> vis

0x405000        0x0000000000000000      0x0000000000000031      ........1.......
0x405010        0x0000000000000000      0x0000000000000000      ................
0x405020        0x0000000000000000      0x0000000000000000      ................
0x405030        0x0000000000000000      0x0000000000020fd1      ................         <-- Top chunk
pwndbg>
```

The top_chunk

- `void *b = malloc(0x38);`

```
pwndbg> vis

0x405000        0x0000000000000000      0x0000000000000031      ........1.......
0x405010        0x0000000000000000      0x0000000000000000      ................
0x405020        0x0000000000000000      0x0000000000000000      ................
0x405030        0x0000000000000000      0x0000000000000041      ........A.......
0x405040        0x0000000000000000      0x0000000000000000      ................
0x405050        0x0000000000000000      0x0000000000000000      ................
0x405060        0x0000000000000000      0x0000000000000000      ................
0x405070        0x0000000000000000      0x0000000000020f91      ................         <-- Top chunk
pwndbg>
```

The top_chunk

- `void *c = malloc(0x48);`

```
pwndbg> vis

0x405000	0x0000000000000000	0x0000000000000031	........1.......
0x405010	0x0000000000000000	0x0000000000000000	................
0x405020	0x0000000000000000	0x0000000000000000	................
0x405030	0x0000000000000000	0x0000000000000041	........A.......
0x405040	0x0000000000000000	0x0000000000000000	................
0x405050	0x0000000000000000	0x0000000000000000	................
0x405060	0x0000000000000000	0x0000000000000000	................
0x405070	0x0000000000000000	0x0000000000000051	........Q.......
0x405080	0x0000000000000000	0x0000000000000000	................
0x405090	0x0000000000000000	0x0000000000000000	................
0x4050a0	0x0000000000000000	0x0000000000000000	................
0x4050b0	0x0000000000000000	0x0000000000000000	................
0x4050c0	0x0000000000000000	0x0000000000020f41	........A.......	 <-- Top chunk
pwndbg> 
```

The top_chunk

- `void *d = malloc(0x58);`

```
pwndbg> vis

0x405000        0x0000000000000000      0x0000000000000031      ........1.......
0x405010        0x0000000000000000      0x0000000000000000      ................
0x405020        0x0000000000000000      0x0000000000000000      ................
0x405030        0x0000000000000000      0x0000000000000041      ........A.......
0x405040        0x0000000000000000      0x0000000000000000      ................
0x405050        0x0000000000000000      0x0000000000000000      ................
0x405060        0x0000000000000000      0x0000000000000000      ................
0x405070        0x0000000000000000      0x0000000000000051      ........Q.......
0x405080        0x0000000000000000      0x0000000000000000      ................
0x405090        0x0000000000000000      0x0000000000000000      ................
0x4050a0        0x0000000000000000      0x0000000000000000      ................
0x4050b0        0x0000000000000000      0x0000000000000000      ................
0x4050c0        0x0000000000000000      0x0000000000000061      ........a.......
0x4050d0        0x0000000000000000      0x0000000000000000      ................
0x4050e0        0x0000000000000000      0x0000000000000000      ................
0x4050f0        0x0000000000000000      0x0000000000000000      ................
0x405100        0x0000000000000000      0x0000000000000000      ................
0x405110        0x0000000000000000      0x0000000000000000      ................
0x405120        0x0000000000000000      0x0000000000020ee1      ................         <-- Top chunk
pwndbg>
```

Demo 0/a.out

Let’s begin pwning…

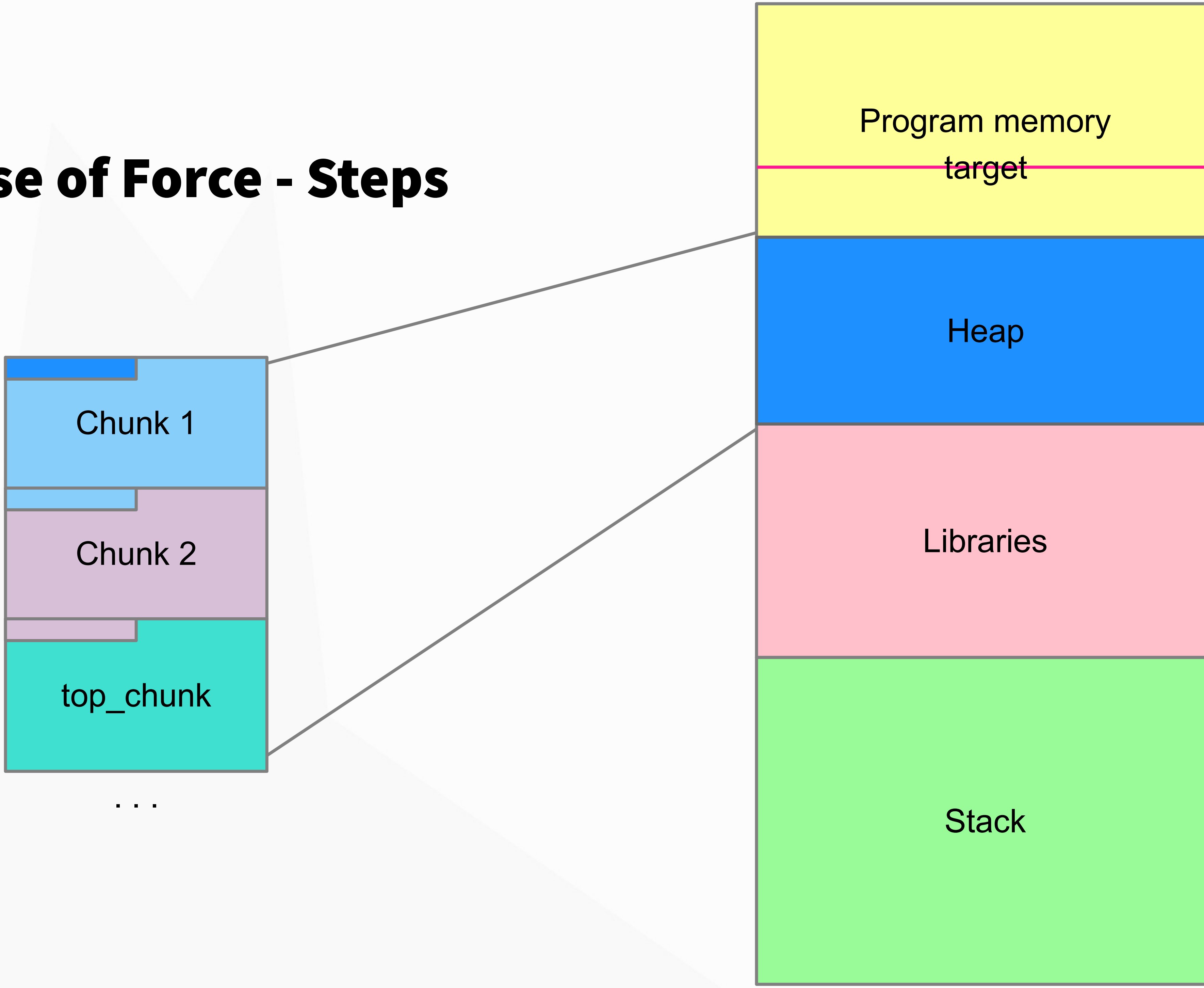
House of Force

- Arbitrary write primitive
- Caused by a lack of size checks
- Works on GLIBC < 2.29
  - 2.29 introduced a top_chunk size sanity check
  - 2.30 introduced a max allocation size
- Requires knowing top_chunk address
- Requires having the ability to overwrite top_chunk’s size field

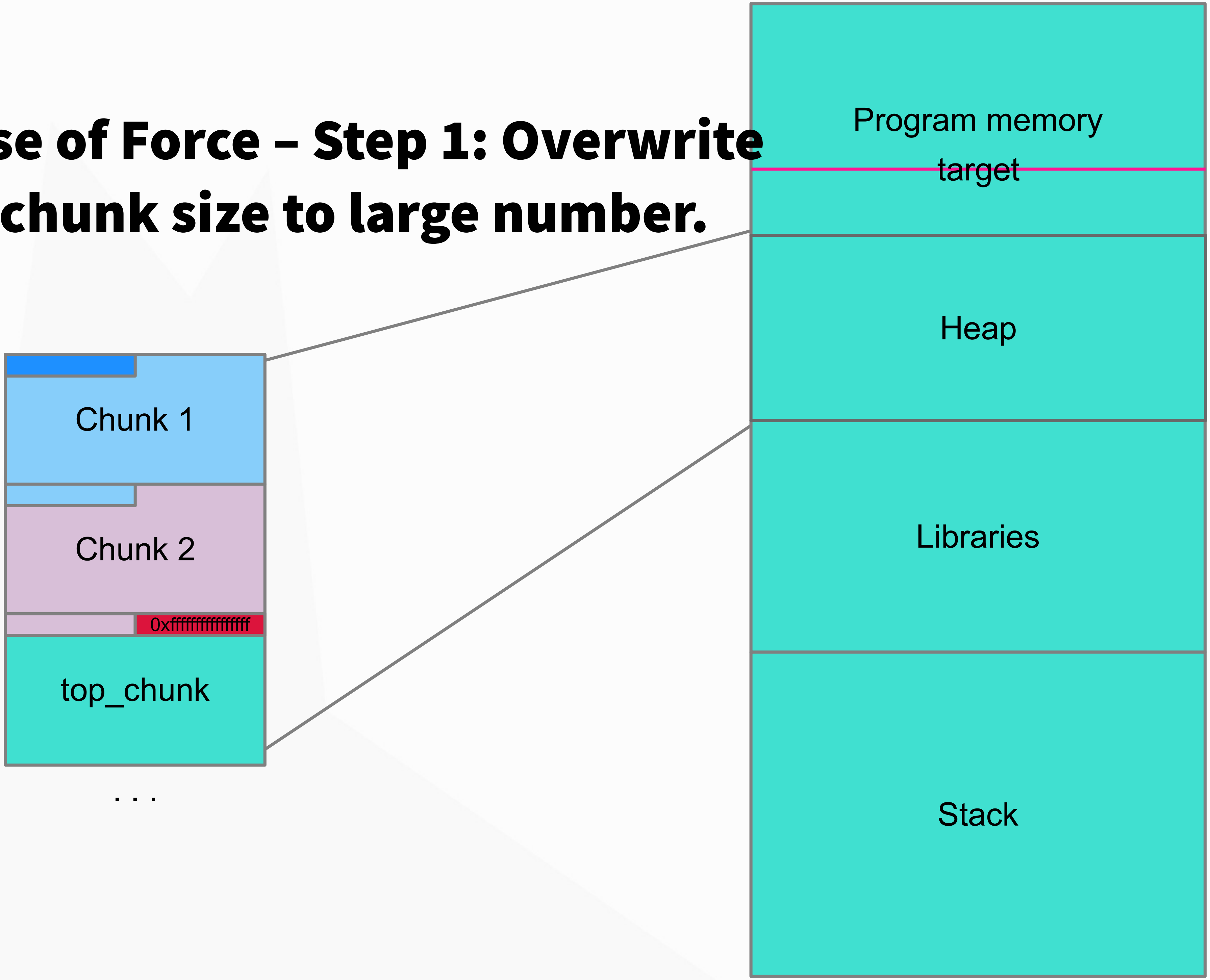
House of Force - Technique

1) Overwrite the top_chunk size field to be a massive number
    1) This will cause malloc to think the size of the heap covers the entire process memory map
2) Malloc a large enough chunk to move the top chunk close to the target memory address
3) Calling malloc again with sufficient size will return a pointer close to the target address

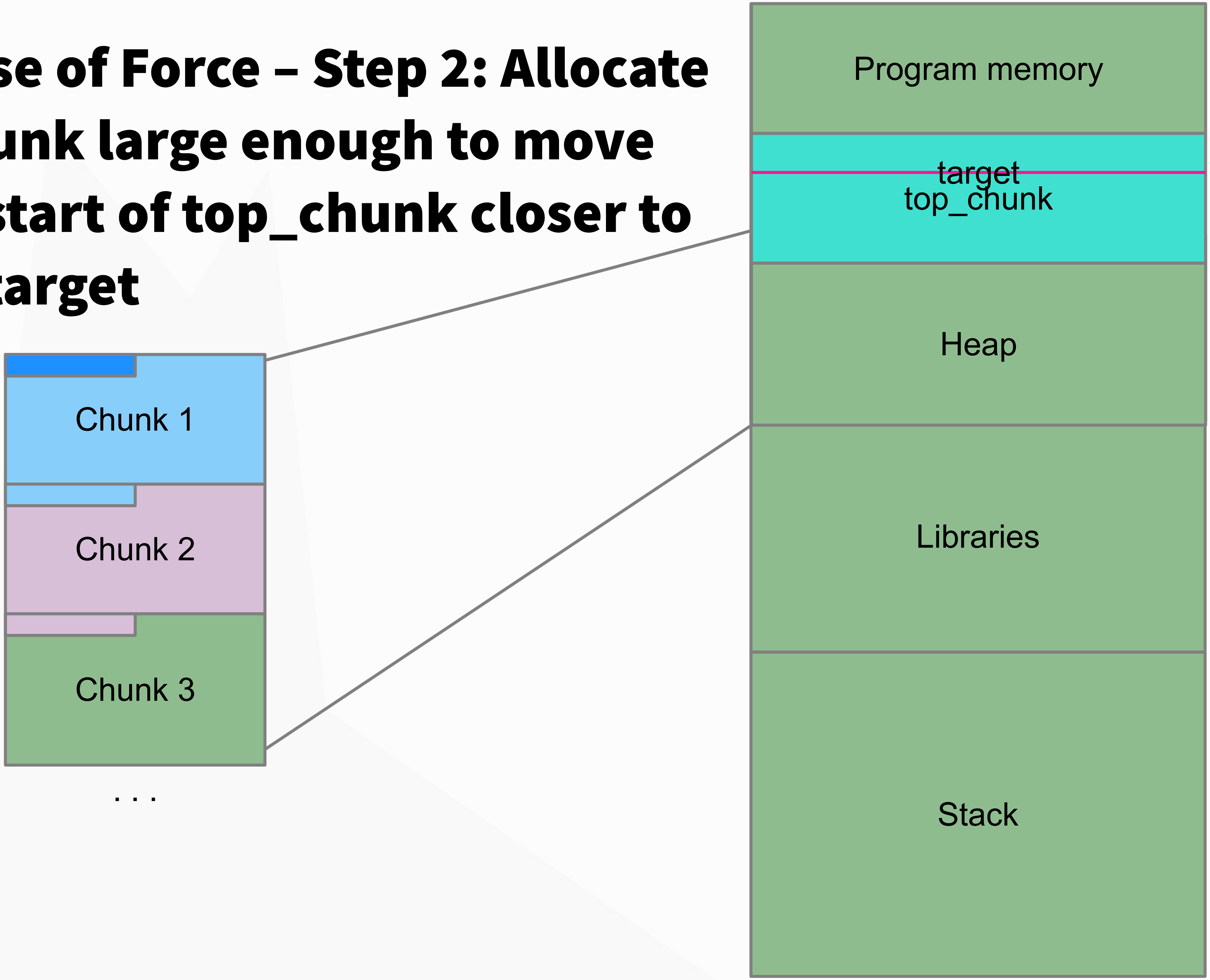
House of Force - Steps



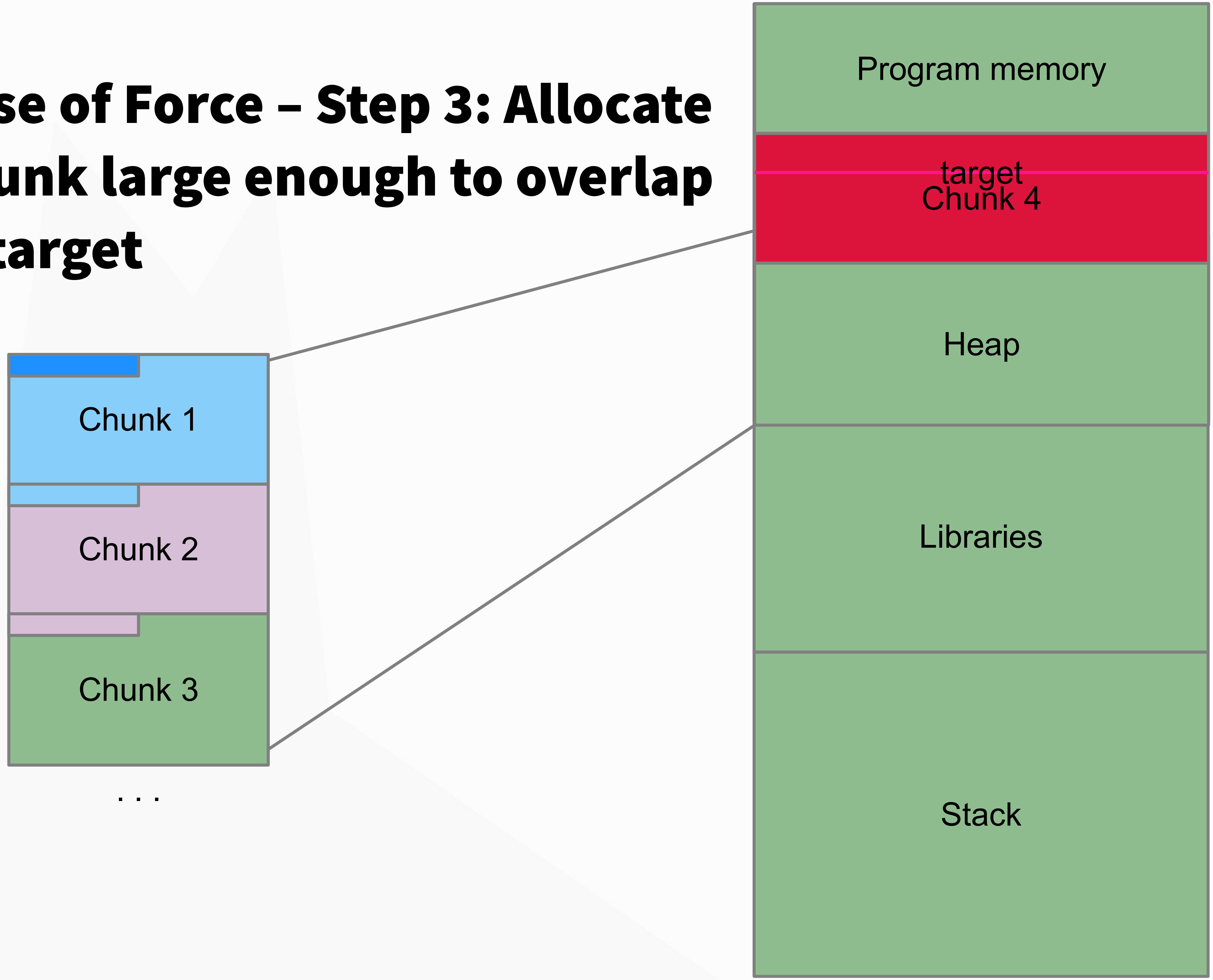
House of Force – Step 1: Overwrite top_chunk size to large number.



House of Force – Step 2: Allocate a chunk large enough to move the start of top_chunk closer to the target



House of Force – Step 3: Allocate a chunk large enough to overlap the target

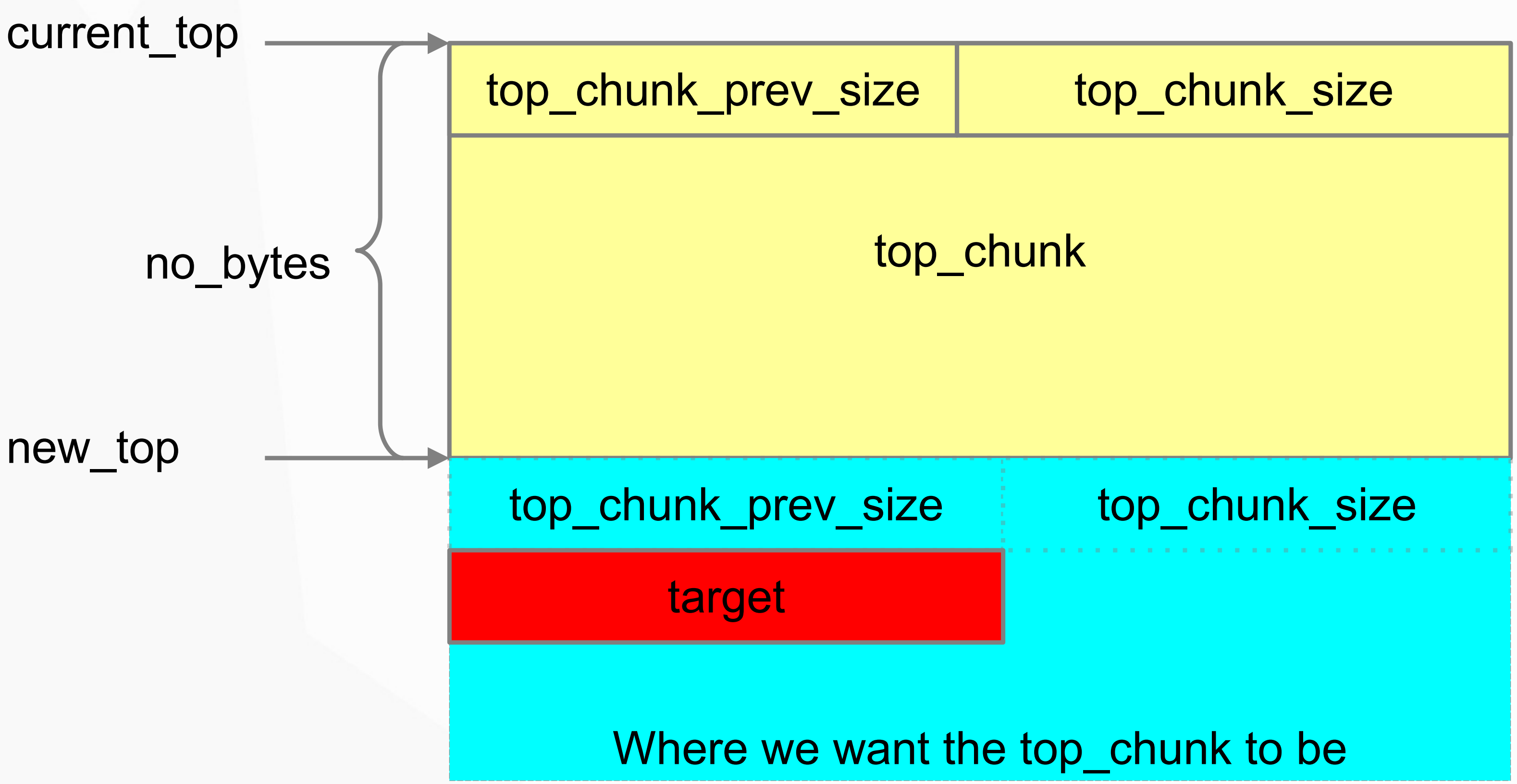


Demo 1/a.out

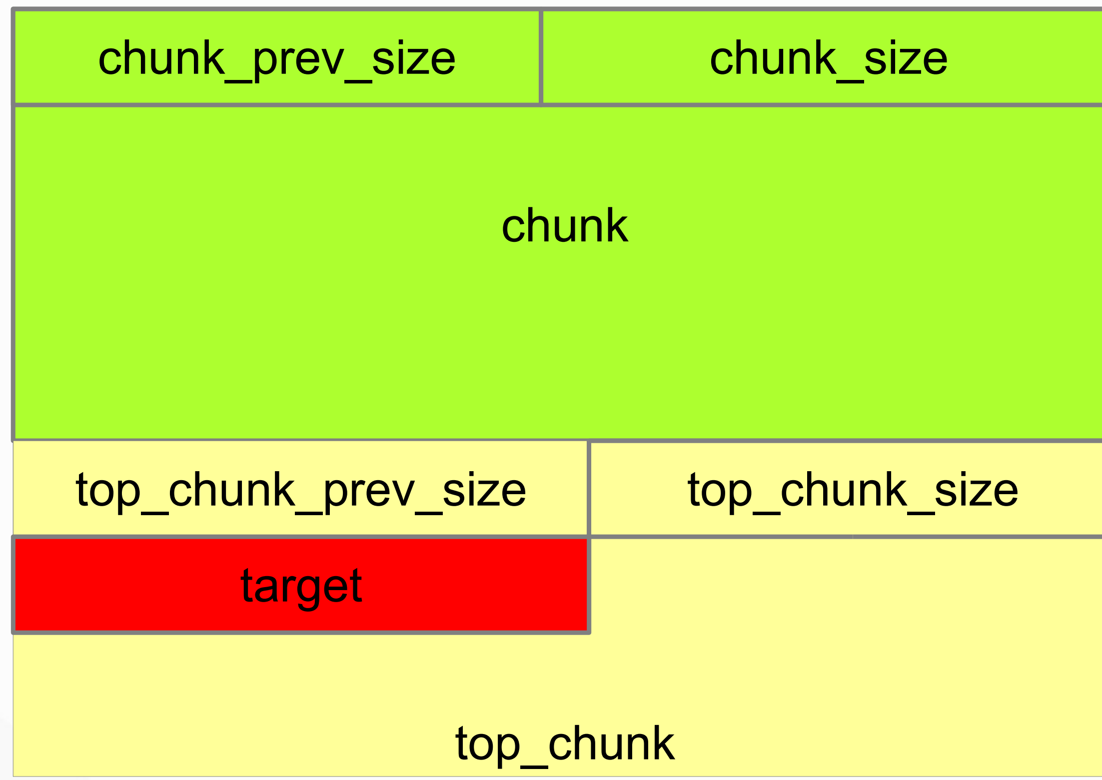
Step 2 malloc size calculation

- *new_top = current_top + no_bytes*
- *no_bytes = new_top – current_top*
- *no_bytes = malloced_size + sizeof(prev_size) + sizeof(size) = malloced_size + 8 + 8 = malloced_size + 16*
- *malloced_size + 16 = new_top – current_top*
- *malloced_size = new_top – current_top – 16*
- *new_top = target - sizeof(prev_size) - sizeof(size) = target - 8 - 8 = target - 16*
- *malloced_size = target - 16 – current_top – 16*
- **malloced_size = target – current_top - 32**

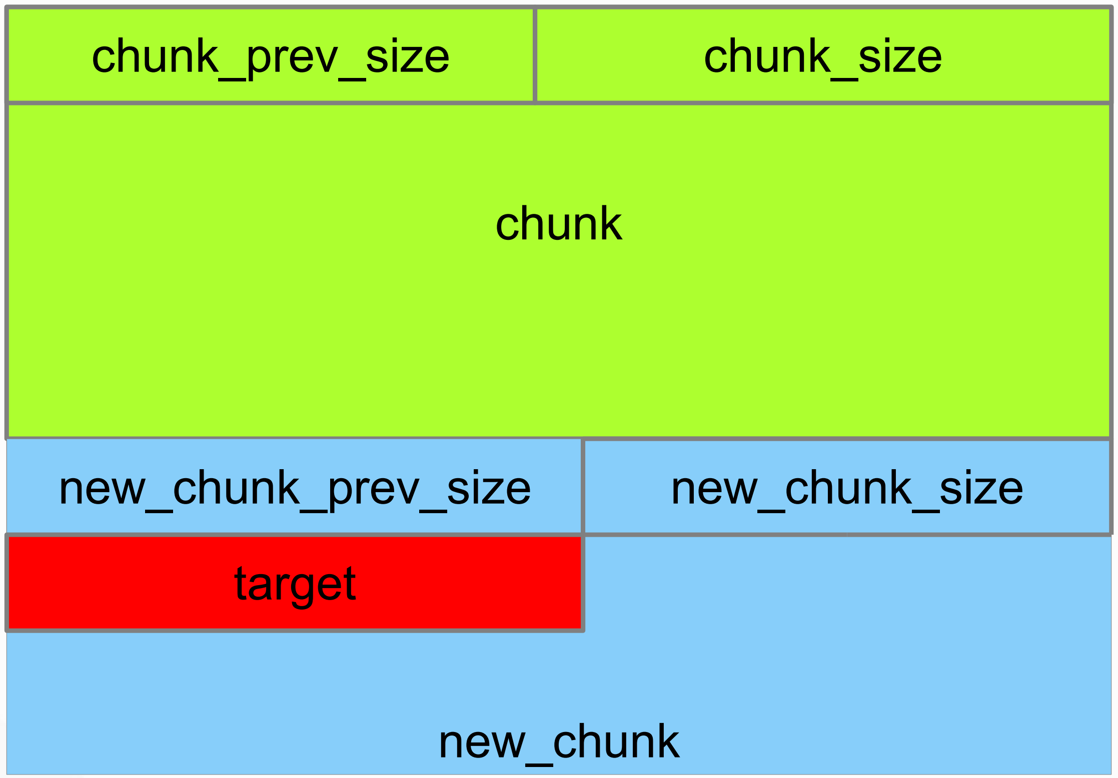
Step 2 malloc size calculation (Visualization)



Step 2 malloc size calculation (Visualization)



Step 3 (Visualization)



Challenges

- Should be accessible on https://ctf.monsec.io/challenges
- Some trivia stuff to make sure you understand the heap and chunks
- A remote challenge to try getting a shell.

Additional resources

- Dhaval Kapil’s GitBook on Heap Exploitation
- Max Kamper’s Linux Heap Exploitation courses on Udemy
- Shellphish’s how2heap repo
- Boston Key Party CTF: cookbook-6

end