Safe unlink

Linux heap exploitation pt. 1

Intro

The binary for this challenge is pretty much the same as the former one, except that it is linked with a difference, more secured version of glibc, and that allocating and editing a chunk are different options right now:

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

===============
|   HeapLAB   |  Safe Unlink
===============

puts() @ 0x7f864dbd3af0

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

In the decompilation we can see the following:

```c
    }
    if (2 < choice) break;
    if (choice == 1) {
      if (index < 2) {
        printf("size: ");
        choice = read_num();
        if ((choice < 121) || (1000 < choice)) {
          puts("small chunks only - excluding fast sizes (120 < bytes <= 1000)");
        }
        else {
          returned_ptr = (char *)malloc(choice);
          m_array[index].user_data = returned_ptr;
          if (m_array[index].user_data == (char *)0x0) {
            puts("request failed");
          }
          else {
                /* m_array[index].request_size is being set to the user input */
            m_array[index].request_size = choice;
            index = index + 1;
          }
        }
      }
      else {
        puts("maximum requests reached");
      }
    }
  }
```

```c
    if (choice != 2) break;
    printf("index: ");
    choice = read_num();
    if (choice < index) {
      if (m_array[choice].user_data == (char *)0x0) {
        puts("cannot edit a free chunk");
      }
      else {
        printf("data: ");
              /* but when writin to that pointer, the heap is overflown with an extra 8-bytes
                 */
        read(0,m_array[choice].user_data,m_array[choice].request_size + 8);
      }
    }
    else {
      puts("invalid index");
    }
```

The allocated chunk pointers are being stored in an array of structs that looks like that:

```
{
        uint64_t user_data; /* the returned pointer from malloc */
        uint64_t request_size; /* the requested user data size */
}
```

There's an heap overflow bug, so that we can override 8-bytes of heap memory. We'll use this primitive to take advantage of the backward consolidation process.

Understanding the relevant internals of malloc

Let's look into the flow of backward consolidation in a macro perspective:

1. Free is being called to free chunk 'p'

2. Malloc will check if the former chunk is also freed - This is being by check the PREV_INUSE flag in p->size member (prev_in_use = p->size & PREV_INUSE).

3. Assuming that it is indeed freed. Malloc will want to consolidate the two chunks and in order to get the pointer to the former chunk it will do the following: p = p - prev_size(p) (where prev_size is the first quadword of metadata in a chunk IN CASE of the former chunk is freed, otherwise, it is the last quadword of user data accessible by the former chunk).

   - When a chunk is being freed, the the last quad word of user data is being set to that chunk size and it will be repurposed as the succeeding chunk's prev_size member

4. Chunk p will now be pointed by the former chunk (ie. p = p - prev_size(p)) so that the two chunks are now one.

5. Next, as a mitigation, malloc will check that prev_size is indeed equal to the size member of the former chunk.

6. If so, the next step taken is the unlink_chunk(arena, p) - malloc will:

   A. Check that: p->fd->bk == p && p->bk->fd == p

   B. Set p->fd->bk to p->bk

   C. Set p->bk->fd to p->fd

```c
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}
```

```c
/* Take a chunk off a bin list.  */
static void
unlink_chunk (mstate av, mchunkptr p)
{
  if (chunksize (p) != prev_size (next_chunk (p)))
    malloc_printerr ("corrupted size vs. prev_size");

  mchunkptr fd = p->fd;
  mchunkptr bk = p->bk;

  if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
    malloc_printerr ("corrupted double-linked list");

  fd->bk = bk;
  bk->fd = fd;
```

Exploiting

Basically, if we can control our own last quadword (ie. The succeeding chunk's prev_size member), and using the heap overflow, the size member of the succeeding chunk (and clear the PREV_INUSE bit), we can successfully trick malloc into thinking that chunk_A is freed. And then, by legally freeing chunk_B, cause the backward consolidation and unlinking process to fire, while still having a write primitive the one of the chunk being consolidated (chunk_A).

Even-though the meteorologic is pretty straight forward, there are some mitigations we have to take into an account:

1. When unlinking is perform, we have to ensure that p->fd->bk = p and that p->bk->fd = p.
    1. Solution: We can override the prev_size of chunk_B so that the consolidated chunk will not start at chunk_A, but at chunk_A's first quadword - that is being pointed by m_array
2. As a result of (1), the fake chunk's size will not be equal to the prev_size
    1. Solution: the size member of the fake chunk is actually the 2nd user data quadword of chunk_A, so we can control it to satisfy this problem

To put it together, We'll trick malloc to consolidate chunk_B with (chunk_A + 0x10 - being pointed by m_array), overriding the 2nd quadword with the value of prev_size, the 3rd quadword with the pointer to m_array-0x18 (so that &m_array->bk will be overriden) and the 4th quadword with &m_array-0x10 (so that m_array->fd will be overriden).

By doing this, we control m_array->user_data address member and get a write primitive to wherever.

Getting shell:

We exploit this powerful primitive by:

1. overriding m_array->user_data address with the address of __free_hook - 0x08
2. Writing "/bin/sh\0" to the 1st quadword of that location
3. Writing the address of system(…) to the 2nd quadword of that address

4. Calling free with chunk_A (that it is now pointed by __free_hook-0x08 - ie. A char pointer to “/bin/sh”) so that system will be called with “/bin/sh\0” char ptr as it’s first and only argument.

```
$ pwd
/home/ubuntu/Desktop/HeapLAB/safe_unlink
$ whoami
ubuntu
$ 
```