Lecture #5: Heap Attacks

UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

Partly based on slides by Drew Davidson, University of Kansas and Robert Walls, WPI

Stack overflows aren’t the only memory attacks

- **Integer overflow**: numerical variables controlling memory allocation are overflown (see CWE 680)
  - Small memory allocated instead of large memory
  - Code accessing that memory will do bad stuff
- **Environment variable overflow**: programs use environment variables without performing bound checking
- **Heap overflow** (see CWE 122 and today’s discussion)

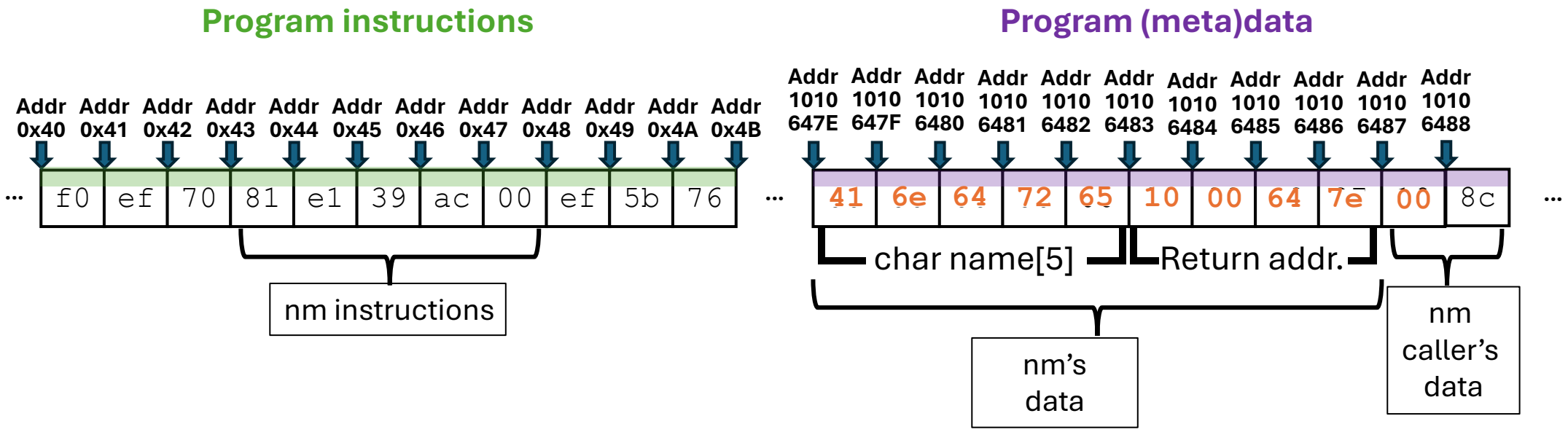
How is heap overflow different from stack smashing?

- Stack smashing gives us a **direct way** to control program execution
- Heap overflow gives us no such thing

Stack smashing: refresher

```
int nm()
{
    char name[5];
    printf("Enter your name: ");
    gets(name);
    printf("hi %s\n", name);
    return 0;
}
```

A	n	d	r	e	LF	LF	d	~	(end)
0x41	0x6e	0x64	0x72	0x65	0x10	0x10	0x64	0x7e	0x00



The heap does not store control flow data

- You can “smash the heap” all you want, all you are going to do is overwrite program data (and eventually write into unallocated memory)
- There is no data, stored in the heap, that affects the control-flow of a program (such as return addresses)
- So… how do we use the heap for exploits?

Let’s look at how the heap works

- We are going to use the (at this point ancient) dlmalloc implementation from the article
- Designed by Doug Lea, SUNY Oswego, starting in the late ‘80s
- Details: https://gee.cs.oswego.edu/dl/html/malloc.html

Before we begin…

- Andrew Griffith's Exploit Education has a version of the phrack article's exploit framed as a CTF challenge: https://exploit.education/protostar/heap-three/
- LiveOverflow has an excellent rundown of the same: https://youtu.be/gL45bjQvZSU
  (I kept this explanation consistent with the video, so you need a refresher can just go back and watch that one)
- There are also a number of related tutorials which I used:
  - https://infosecwriteups.com/the-toddlers-introduction-to-heap-exploitation-unsafe-unlink-part-4-3-75e00e1b0c68
  - https://tc.gts3.org/cs6265/2019/tut/tut09-02-advheap.html

Code example (from Protostar Heap #3)

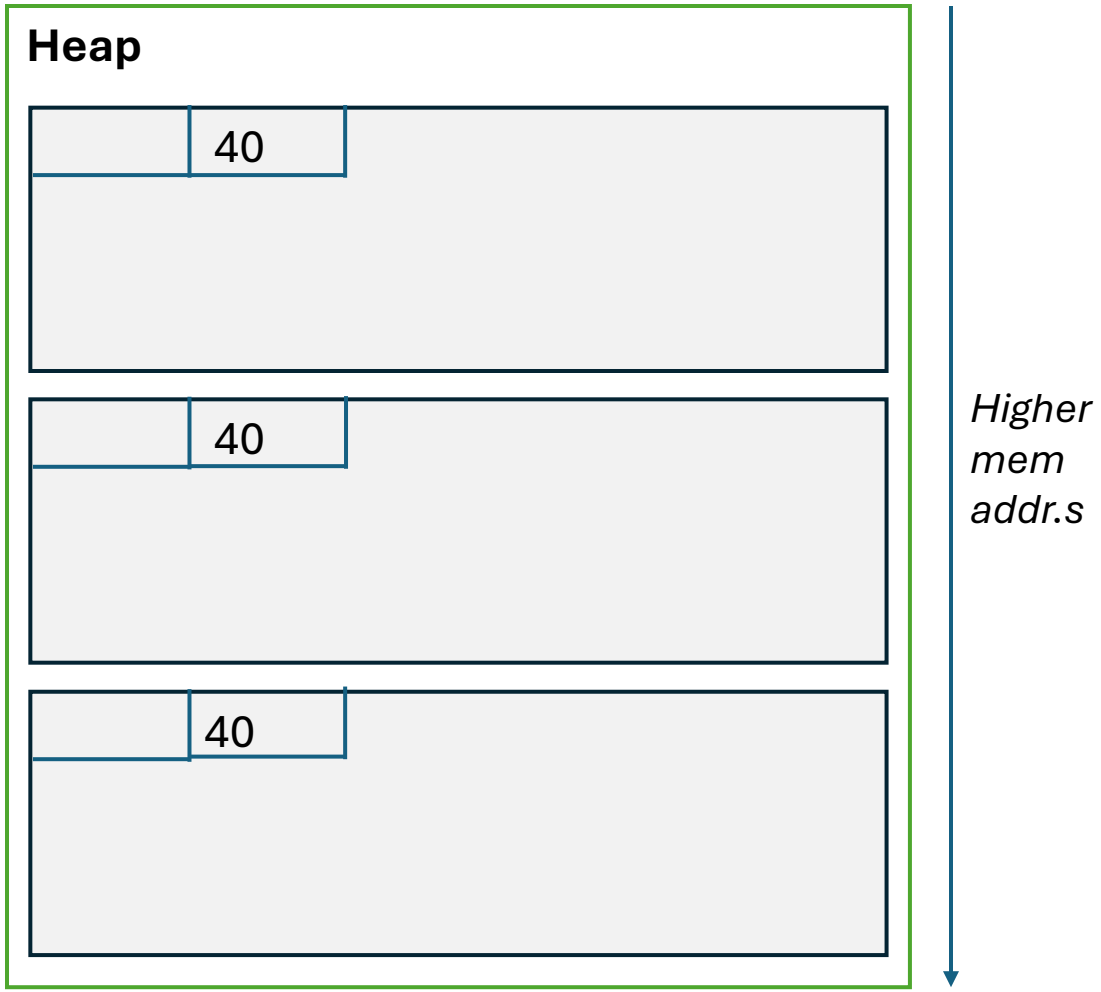
```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```

https://exploit.education/protostar/heap-three/

What happens when malloc is called

Suppose I call malloc() three times:

1. a =malloc(32)
2. b = malloc(32)
3. c = malloc (32)



What happens when blocks are freed?

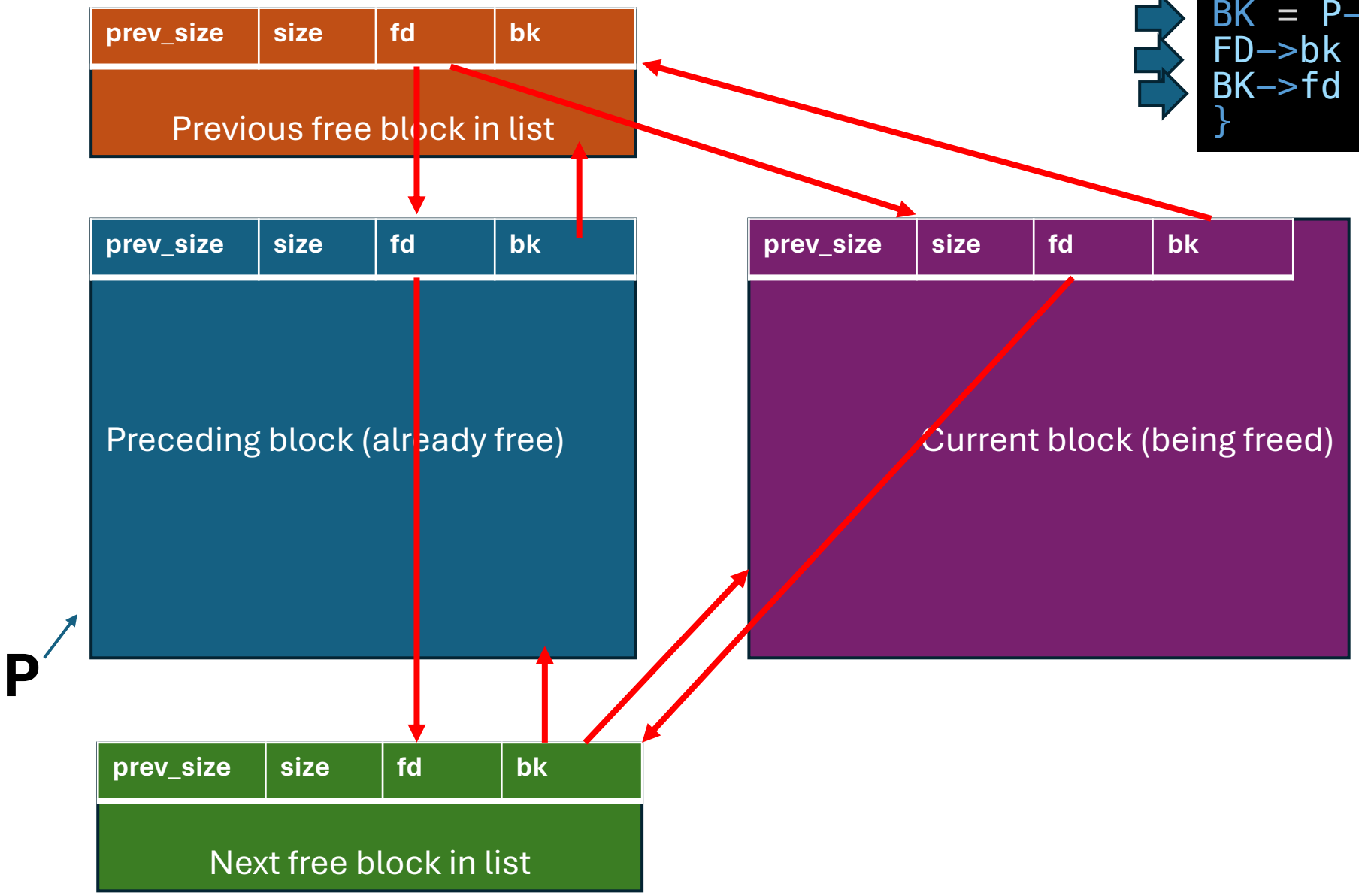
- Freed memory blocks are managed using a doubly-linked list
- Why? (this is actually pretty common any time you need to manage free space, e.g. file systems)
- Where are the pointers between blocks stored? Within the blocks themselves!

More about freeing

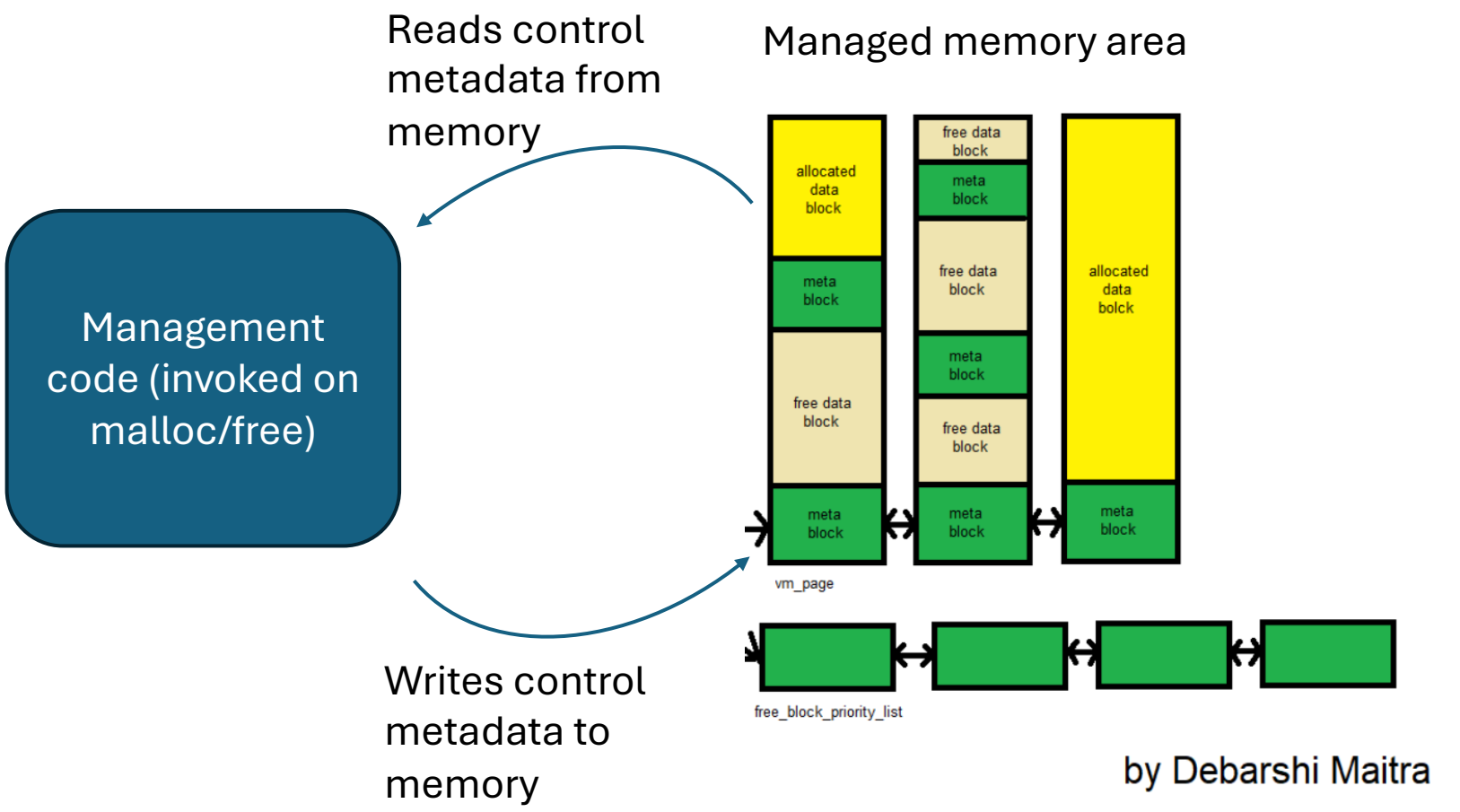
- When a block is freed, it is added to a linked list but…
- If it turns out that the chunks surrounding the current block are also unused, they are merged to the current block, and the resulting merged block is added to the list instead
- As a result, the preexisting blocks are removed from the list (using the infamous unlink function)

Let's see this in action

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



What can we conclude from this?



The million dollar question

What can we do here that causes the free() routines to do our bidding?

... what is our bidding?

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  char *a, *b, *c;

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```

...let's talk about the paper

==Phrack Inc.==

Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12

|=----------=[Once upon a free()...]=----------=|
|=----------=----------=----------=----------=----------=|
|=----------=[anonymous <d45a312a@author.phrack.org>]=----------=|

On the Unix system, and later in the C standard library there are functions to handle variable amounts of memory in a dynamic way. This allows programs to dynamically request memory blocks from the system. The operating system only provides a very rough system call 'brk' to change the size of a big memory chunk, which is known as the heap.

On top of this system call the malloc interface is located, which provides a layer between the application and the system call. It can dynamically split the large single block into smaller chunks, free those chunks on request of the application and avoid fragmentation while doing so. You can compare the malloc interface to a linear file system on a large, but dynamically sized raw device.

There are a few design goals which have to be met by the malloc interface:

- stability
- performance
- avoidance of fragmentation
- low space overhead

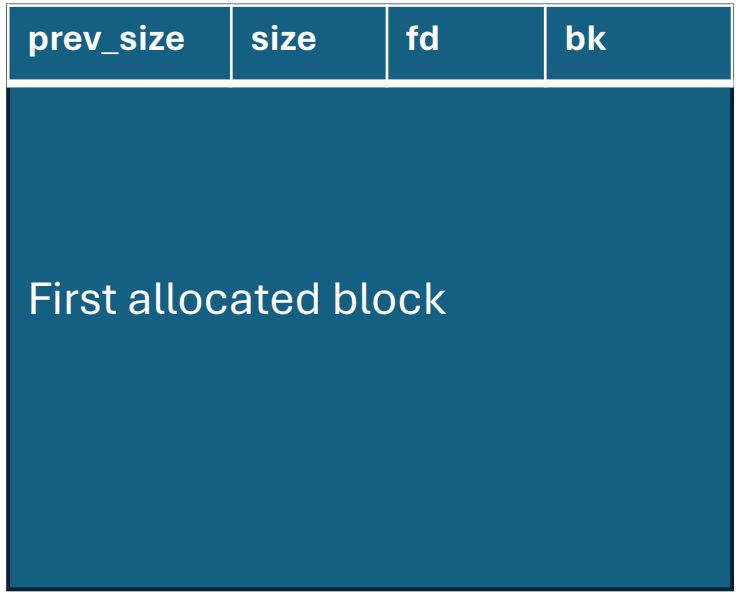
There are only a few common malloc implementations. The most common ones are the System V one, implemented by AT&T, the GNU C Library implementation and the malloc-similar interface of the Microsoft operating systems (RtlHeap*).

Here is a table of algorithms and which operating systems use them:

Algorithm	Operating System
BSD kingsley	4.4BSD, AIX (compatibility), Ultrix
BSD phk	BSDI, FreeBSD, OpenBSD
GNU Lib C (Doug Lea)	Hurd, Linux
System V AT&T	Solaris, IRIX
Yorktown	AIX (default)
RtlHeap*	Microsoft Windows *

Now… let’s attack the heap, shall we?

Let’s go back to free memory management



Recall the program is vulnerable

```
strcpy(a, argv[1]);
strcpy(b, argv[2]);
```

Suppose I have a way to overflow this block (like in the vulnerable code)

Then, I can write whatever I want into this one (not just in the allocated memory, but also in the metadata!)

Ok, I can overwrite the block, so what?

- Well, once you overwrite the block, you can write whatever you want in FD and BK
- Unlink is just a bunch of memory writes
- Once the block is freed, it will be executed

If I control BK and FD, I can cause free() to write the content of BK to FD+12

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

```
BK = *(P + 12);
FD = *(P + 8);
*(FD + 12) = BK;
*(BK + 8) = FD;
```

What I can do with this?

- Modern programs use a table called GOT (global address table) to store the address of library functions such as printf
- A call to printf() will result in a lookup in the GOT to find the function
- If I can overwrite data in the GOT for a given function…
- …whenever that function is called, execution will jump to an attacker-controlled address instead!

Idea

```
BK = *(P + 12);
FD = *(P + 8);
*(FD + 12) = BK;
*(BK + 8) = FD;
```

1. Write an address, stored on the head, to the GOT

2. Some content of the GOT is going to be copied to our heap. This is unavoidable, but we don't care

3. The address pointed to by BK is also on the heap (in the area we overflowed). At that address, we took care of writing our exploit code

Is it that simple?

- Not really! As usual, the devil is in the details
- For example, free() won't believe a block is unused just because the block says so. Metadata in the surrounding blocks must also be consistent
- When exploiting the heap overflow vulnerability, must make sure that both the new fake block and the following one are consistently initialized
- There are a few additional fields that need to be set up (basically must look at the code and ensure that all if conditions are satisfied to get to unlink)

Some take-away points

- This style of exploits challenge the notion (from stack smashing attacks) which must be able to manipulate control flow directly
- Instead, here we manipulate legitimate code so that it manipulates data for us
- This attack is considerably more complicated than stack smashing, as it requires to find the “right” sequence of memory manipulations

Some take away points/2

- Reviewing this and the previous paper should give us some sense of what exploit writers look for
- You don’t need to search the entire code for vulnerabilities
- What you are looking for is program statements that (i) modify memory, based on (ii) data indirectly or directly affected by userinput

Other heap-based attacks

- You may have heard of heap spraying
- This is not technically an attack, but a support technique for other attacks
- Suppose you have an attack that entails jumping into attacker-controlled memory
- Heap spraying helps by filling the heap with multiple copies of your code

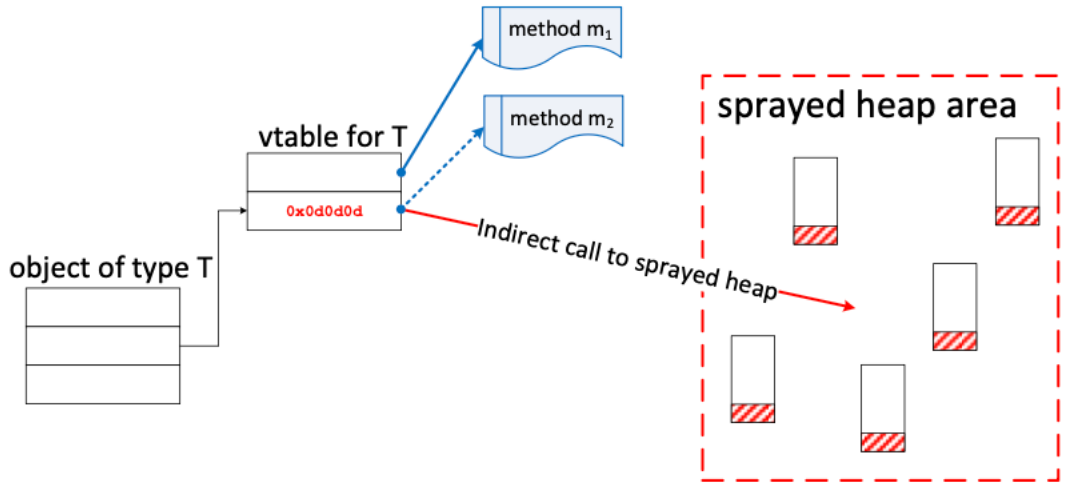


Figure 1: Schematic of a heap spraying attack.

(from Nozzle: A Defense Against Heap-spraying Code Injection Attacks, USENIX 2009)

That’s all for today!