Memory Corruption 2

Advanced Internet Security

Adrian Dabrowski
Christian Kudera
Georg Merzdovnik
Aljosha Judmayer

News from the Lab

- 21 people solved Challenge 6 so far!
- Challenge 7 will start tomorrow

News from the Field I

- CastHack
  - `https://casthack.thehackergiraffe.com/`
  - Quote: “Hacking Chromecasts/Google Homes/SmartTVs”
    - Played a video on the chromecast
  - Actually it seems to be a problem with routers allowing access to UPnP from the Internet
    - Which is bad, but not a chromecast fault
- NSA to release Ghidra at RSA conference
  - Reverse Engineering Tool (Disassembler)
  - https://github.com/nationalsecurityagency

News from the Field II

- unCAPTCHA2
  - 2017 researchers cracked the reCAPTCHA Audio challenge with about 85% accuracy[1]
  - Google responded and changed from numbers to words and also included bot detection
  - Welcome unCAPTCHA2: using the same engine, with additions against bot detection they cracked it again, now with 90% accuracy
    - Works since June 2018, notified Google
    - Code is on github[2]

[1] http://uncaptcha.cs.umd.edu/papers/uncaptcha_woot17.pdf
[2] https://github.com/ecthros/uncaptcha2

Format String Exploitation

A Short Introduction to printf

- `int printf(const char *format, ...)`
  - function with variable number of arguments
    - as usual, arguments are fetched from the stack
- `const char *format` is called format string
  - used to specify type of arguments
  - %d or %x for numbers
  - %s for strings

Format String Example

```c
#include <stdio.h>

void main(int argc, char** argv){
    char buf[100];
    fgets(buf, 100, stdin);
    printf(buf);
}
```

What is the Problem?

- User input passed as format string
  - Allows user to pass format string which will be interpreted

```
printf("Hello world\n"); // is ok
printf(user_input);      // vulnerable
```

- Allows to read values with format identifiers

Simple example

```
$ echo "AAAABBBB" | ./fmt_ex
    AAAABBBB
```

Simple example

```
$ echo "AAAABBBB" | ./fmt_ex
    AAAABBBB

$ echo "%p-%p-%p-%p" | ./fmt_ex
    0x7f6fed1fb730-0x7ffc2a9fb5d0-0xfbad2088-0x557eb97d226c
```

Simple example

```
$ echo "AAAABBBB" | ./fmt_ex
    AAAABBBB

$ echo "%p-%p-%p-%p" | ./fmt_ex
    0x7f6fed1fb730-0x7ffc2a9fb5d0-0xfbad2088-0x557eb97d226c

$ echo "AAAABBBB-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p" | ./fmt_ex
    AAAABBBB-0x7eff2e070730-0x7fff534d5520-0xfbad2088-
    0x557e9da9428a-0x77-0x7fff534d5678-0x100000000-
    0x4242424241414141-0x252d70252d70252d-
    0x2d70252d70252d70-0x70252d70252d7025
```

Simple example

```
$ echo "AAAABBBB" | ./fmt_ex
    AAAABBBB

$ echo "%p-%p-%p-%p" | ./fmt_ex
    0x7f6fed1fb730-0x7ffc2a9fb5d0-0xfbad2088-0x557eb97d226c

$ echo "AAAABBBB-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p" | ./fmt_ex
    AAAABBBB-0x7eff2e070730-0x7fff534d5520-0xfbad2088-
    0x557e9da9428a-0x77-0x7fff534d5678-0x100000000-
    0x4242424241414141-0x252d70252d70252d-
    0x2d70252d70252d70-0x70252d70252d7025
```

- If you look closely, you will notice that our format string is also on the stack

Leaking Data

- This means we can read data from the stack
- But the approach is somehow limited
  - Possibly limited input (format string) length

Leaking Data

- This means we can read data from the stack
- But the approach is somehow limited
  - Possibly limited input (format string) length
- So, can we improve this to read arbitrary data?

“$” Modifier

- actually we can give an **index** to format specifiers
  - Direct parameter access
- "%<n>$p"
  - Tells the interpreter to take the n^{th} argument from the stack

“$” Modifier

- actually we can give an **index** to format specifiers
  - Direct parameter access
- "%<n>$p"
  - Tells the interpreter to take the n^{th} argument from the stack

```
$ echo 'AAAABBBB-%5$p' | ./fmt_ex
    AAAABBBB-0x77
```

“$” Modifier

- actually we can give an **index** to format specifiers
  - Direct parameter access
- "%<n>$p"
  - Tells the interpreter to take the n^{th} argument from the stack

```
$ echo 'AAAABBBB-%5$p' | ./fmt_ex
    AAAABBBB-0x77

$ echo 'AAAABBBB-%8$p' | ./fmt_ex
    AAAABBBB-0x4242424241414141
```

Reading Arbitrary Data

- Currently we only leak data from the stack
- We can also extend this to leak arbitrary data

“%s” Modifier

- Allows to print NULL-terminated strings
  - Address of the string is passed as parameter

“%s” Modifier

- Allows to print NULL-terminated strings
  - Address of the string is passed as parameter
- Our own format string is also accessible
  - We can place the address we want to read inside the format string
  - Use this address to indirectly access data

Writing Arbitrary Data?

- ok, we can read anything
- but, can we also write something?

Writing Arbitrary Data!

- %n
  - from *man 3 printf*
    - The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. (This specifier is not supported by the bionic C library[3].) The behavior is undefined if the conversion specification includes any flags, a field width, or a precision.

[3] Google's standard C library for Android

Wrtiting Arbitrary Data!

```c
int i;
printf("01234%n", &i);
```

- Writes 5 into *i*
- This means we can write values
  - Basically the same way we used to read strings with %s

Writing Arbitrary Data!

- We can use the width modifier to write arbitrary values
  - for example, %.500d
  - even in case of truncation, the characters that would have been written are used for %n

Writing Arbitrary Data!

- Might still crash the program or take long for addresses
  - e.g. for address *0x0804a004* we would need 134520836 characters
- *h* and *hh* modifiers
  - A following integer conversion corresponds to a *signed short* or a *signed char* respectively
  - This means we can also only write 2 bytes (*%hn*) or a single byte (*%hhn*)
  - Also means we need the address (with according offest) more often on the stack

Taking Control of the Program

- Business as usual:
  - Overwrite function pointer
  - e.g. GOT entries

The Heap

Heap Management

- Implementations

Algorithm	Operating System
dlmalloc	Doug Lea' malloc (general purpose)
ptmalloc2	GNU LibC (based on dlmalloc)
jemalloc	FreeBSD and Firefox
tcmalloc	Google (thread-caching malloc)
….	….

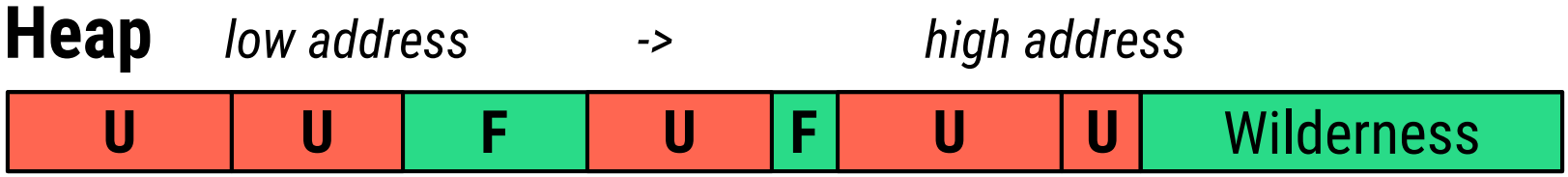
- Each application can use/implement it's own allocator

GlibC Memory Allocation

- Glibc integrated ptmalloc2 (there may be differences now between these two)
- Previously dlmalloc, but ptmalloc allows for better handling of threads
  - No need for locking/synchronisation
  - **Per-thread arena**

Glibc Malloc

- Memory Layout
  - heap is divided into continuous chunks of memory




```
U           ... used chunk
F           ... free chunk
Wilderness  ... topmost free chunk
```

- Wilderness chunk
  - only chunk that may be increased (with system call sbrk)
  - treated as bigger than all other chunks
    - If nothing else fits it will just be increased

Glibc Malloc

- Memory Chunk
  - continuous region of heap memory
  - can be allocated, freed, split, joined (two free chunks)
- Public and Internal routines

```
// allocate size bytes, memory not initialized
malloc(size_t n)
// allocate mem for array of elements, memory set to zero
calloc(size_t unit, size_t quantity)
// change size of memory block
realloc(void* ptr, size_t n)
// free memory space
free(void *ptr)
```

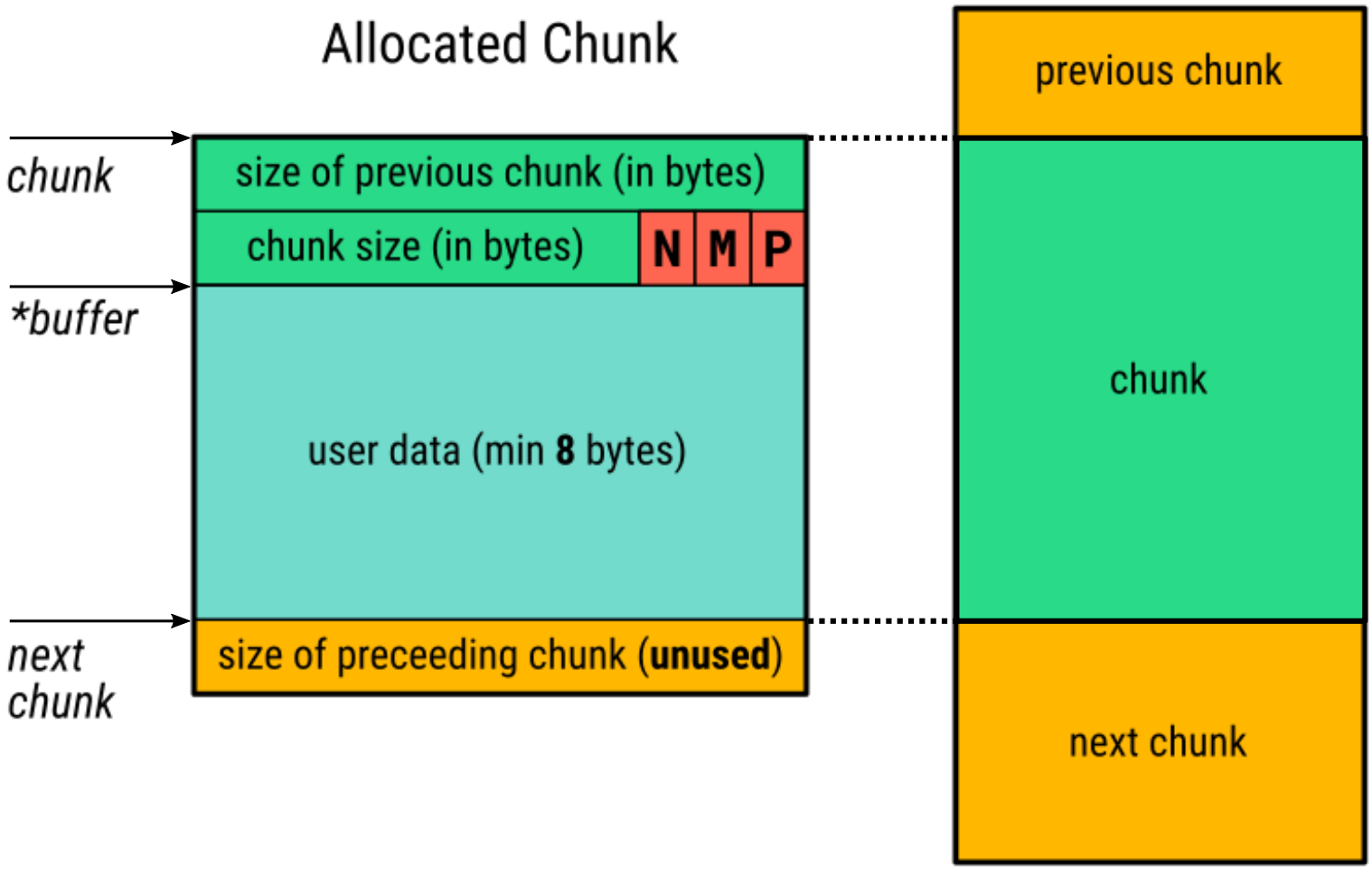
Glibc Malloc

- Boundary tag
  - holds chunk management information
  - stored in front of each chunk
  - 16 bytes large -> minimum allocated size

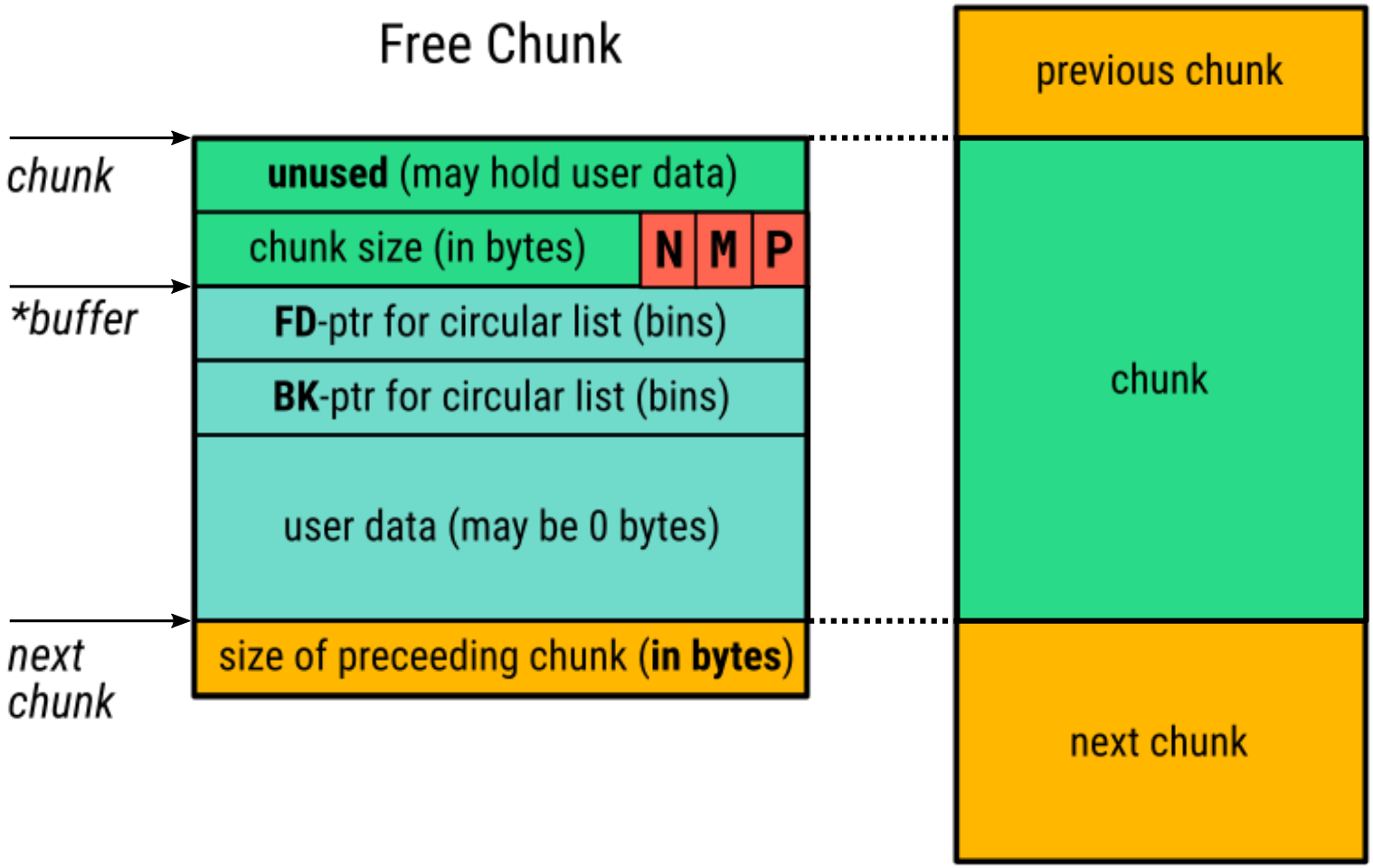
```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead.

  struct malloc_chunk* fd;         /* double links -- used only if free.
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free.
  struct malloc_chunk* bk_nextsize;
};
```

- pointer returned by malloc (for user) starts at fd
  - usually 8 bytes overhead for allocated chunks

Glibc Malloc



Glibc Malloc



Glibc Malloc

- Status Bits
  - Lower 3 bits of chunk size
  - Chunk size is always 8-byte aligned, so these would be unused otherwise

```
0x01 PREV_INUSE // set when previous chunk is in use
0x02 IS_MMAPPED // set if chunk was obtained with mmap()
0x04 NON_MAIN_ARENA // set if chunk belongs to a thread arena
```

Glibc Malloc - Bin Management

- available chunks are stored in **bins** on a circular doubly-linked list
- each bin holds chunks of a certain size range
- the bin itself consists of two pointers (forward/back) and acts as the corresponding list head
- each bin is initially empty
- chunks are maintained in decreasing sorted order by size
  - best fit algorithm

Heap Overflow

- Heap overflow requires modification of boundary tags
  - in-band management information
  - task is to fake these tags to trick *malloc* into overwriting addresses of attackers choice
- However, this strongly depends on the corresponding memory manager
  - They all have their implementation differences
- Often interesting information is stored on the heap
  - C++ vtable pointers, function pointers
  - Often easier to overwrite these objects

How2Heap

- Easiest way to learn is to try yourself and look at examples.
- https://github.com/shellphish/how2heap
  - A repository for learning various heap exploitation techniques.

Heap Details

- There is much more about heaps we did not cover in depth:
  - Arenas and Binning
    - Different heaps for threads and different bins for different chunk sizes
  - Chunk coalescing
    - How free chunks are merged
- These details depended on the underlying implementation

Resources and Further Reading

- `https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/`
  - Understanding glibc malloc, *sploitfun*
- `http://tukan.farm/2017/07/08/tcache/`
  - Thread local caching in glibc
  - relatively new feature
  - increased performance for programs
  - However: also impacts security

Heap Spraying

- Requirement:
  - we need control over memory allocations
  - must create many objects containing shellcode
- Solution: embedded scripts
  - today, many applications allow execution of user-provided scripts in the context of the application/document to enrich usability
  - JavaScript (browsers, pdf readers)
  - ActionScript (flash applications)
- Before exploiting a memory corruption bug, allocate many objects (e.g., strings) filled with shellcode
  - It's actually not a vulnerability, we just use the Heap

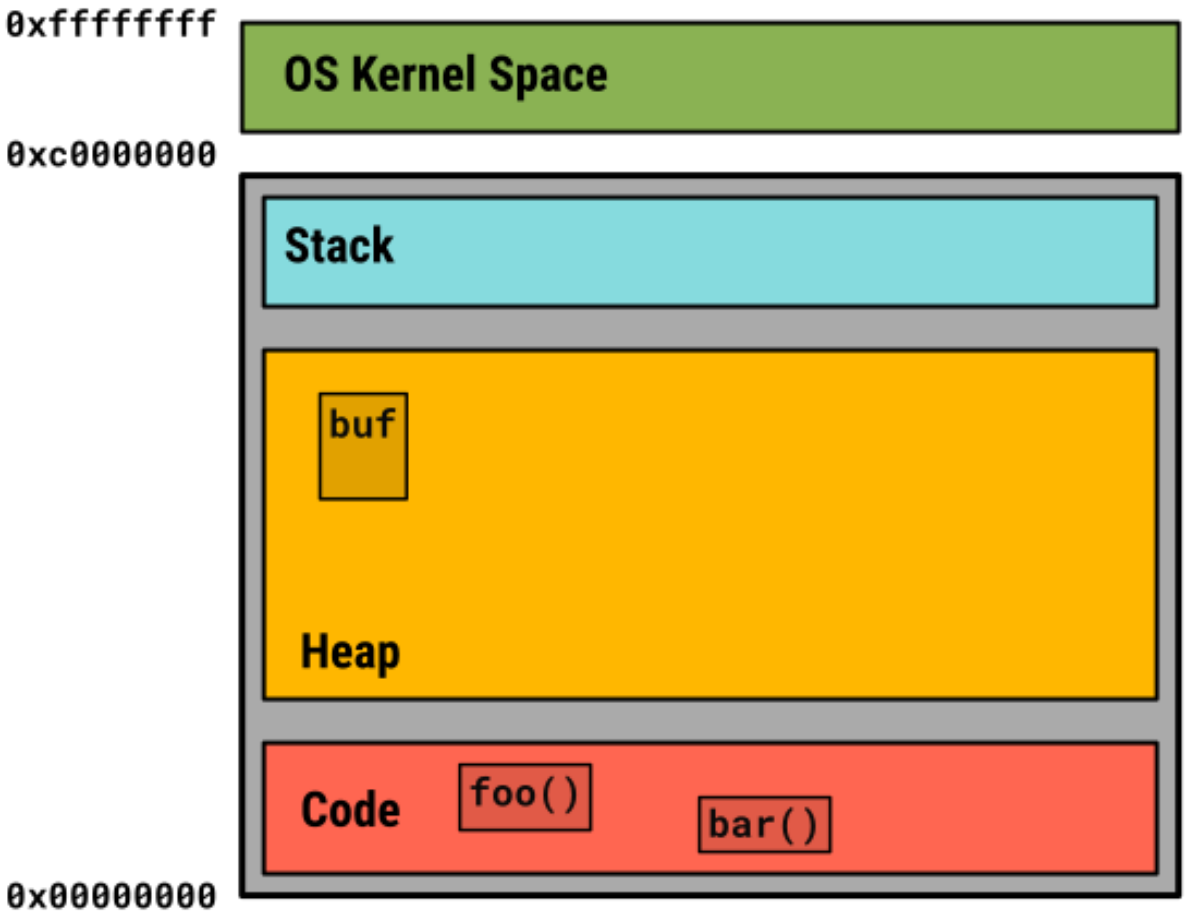
Heap Spraying - 32 vs 64 bit

- 32 bit systems have a maximum address space of 4GB
  - pretty easy to fill up
- 64 bit systems have and address space of 2^{64}
  - 18446744073709551616 bytes
  - over 18 exabytes
  - no chance to spray the full heap, but you could still do targeted spraying (e.g. if you are able to modify a heap pointer slightly)

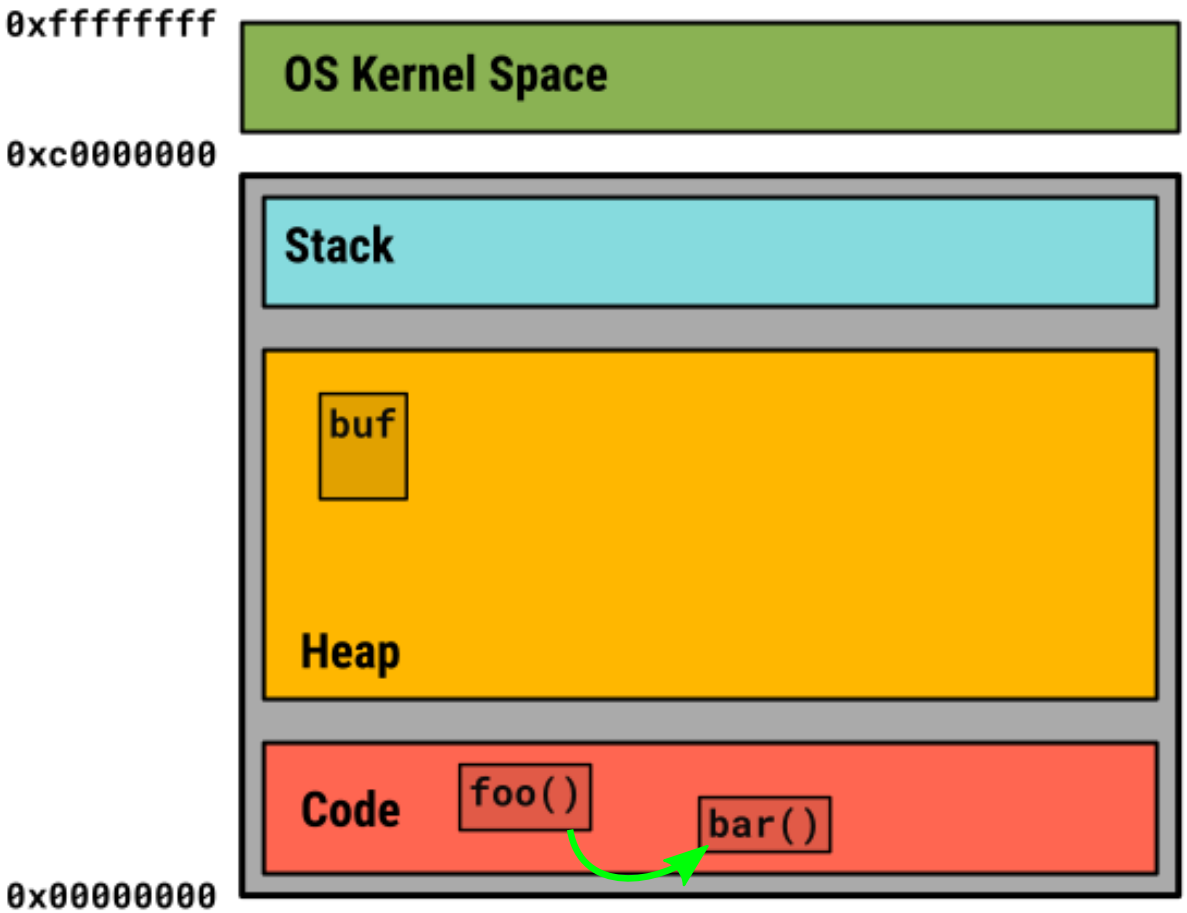
Payloads for Heap Spraying

- Previously it had been shellcodes (since heap was executable)
- Nowadays mostly fake objects or ROP chains

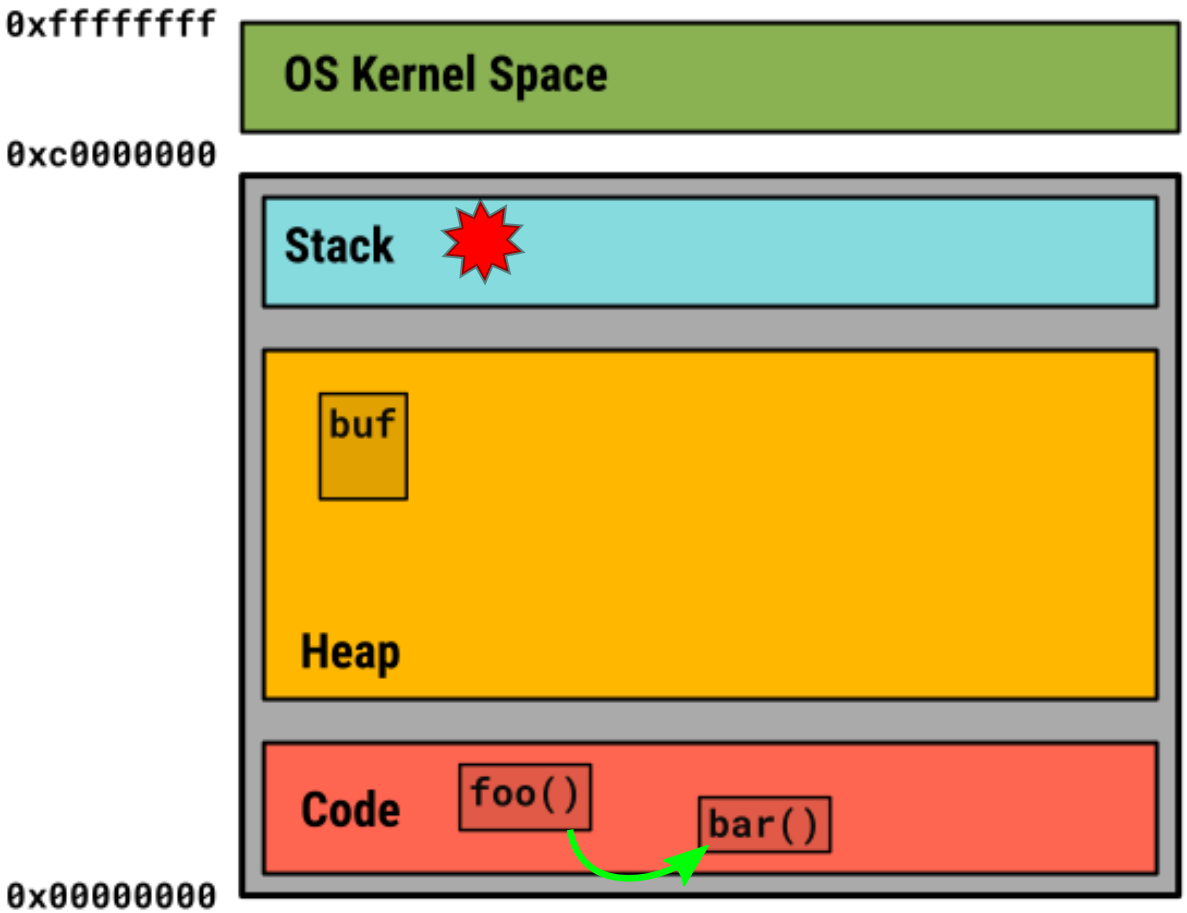
Heap Spraying



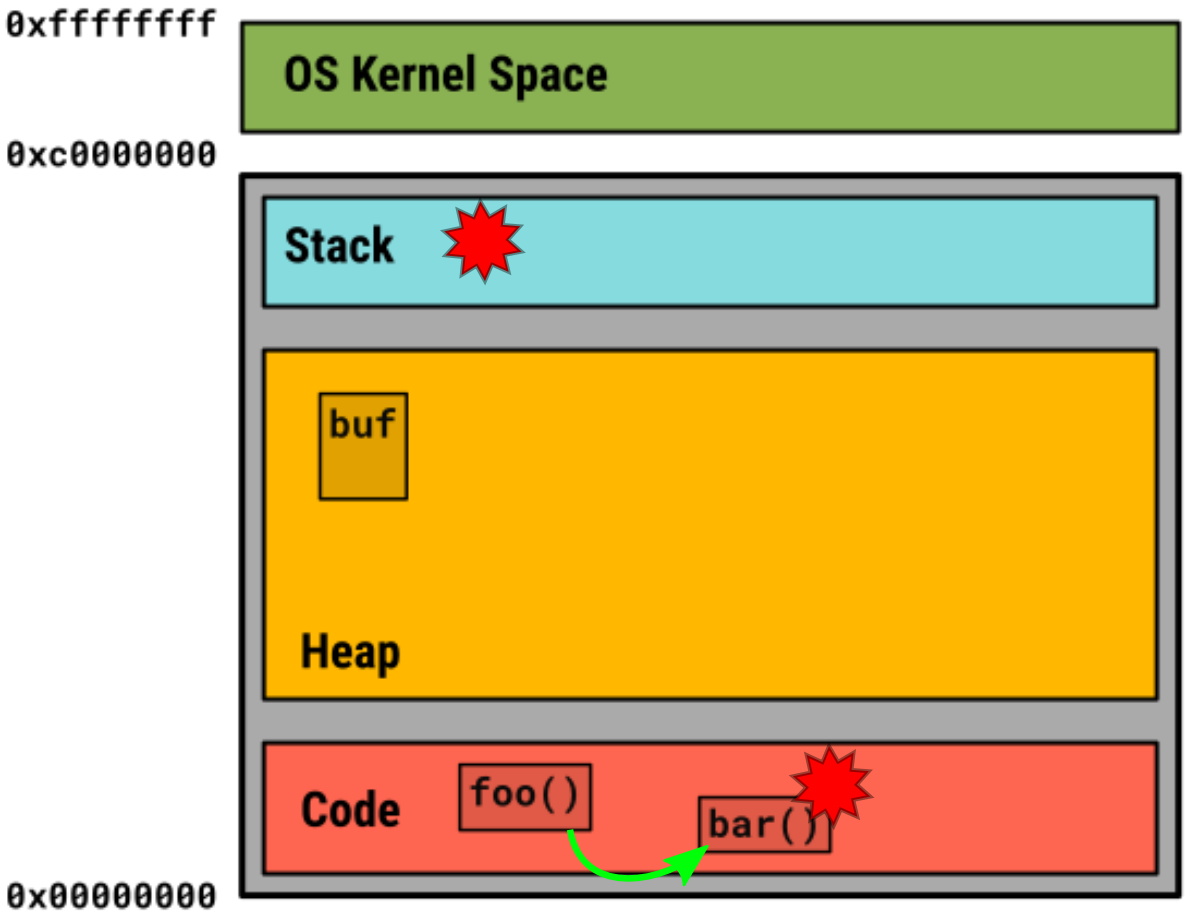
Heap Spraying



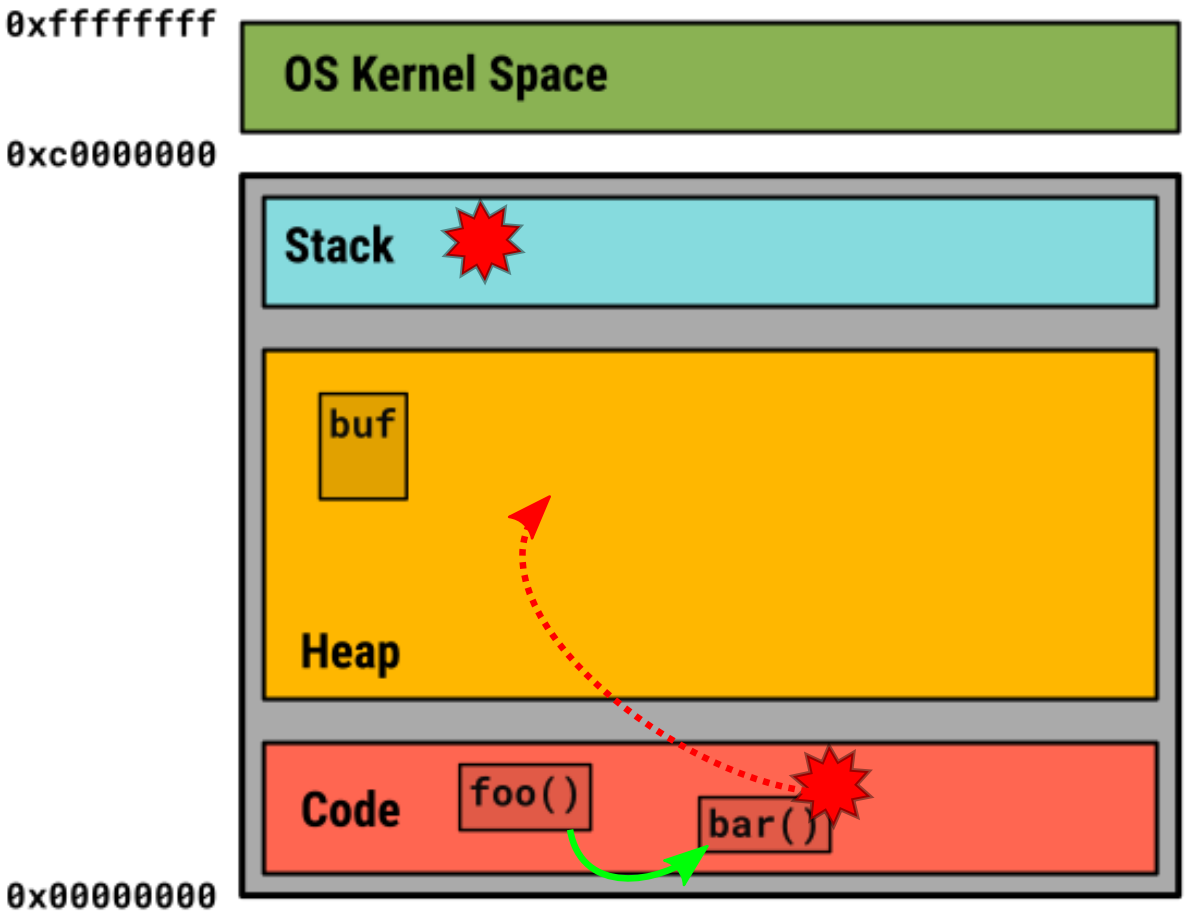
Heap Spraying



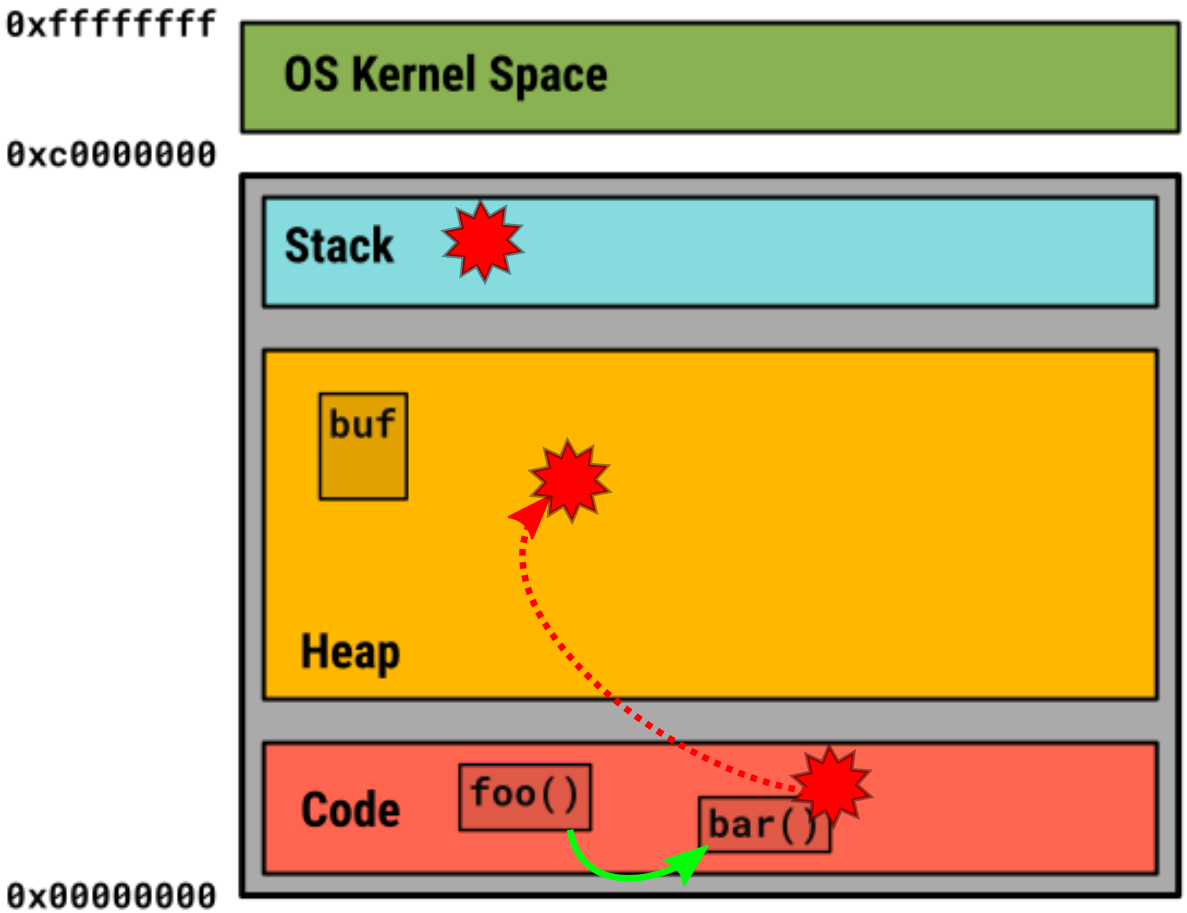
Heap Spraying



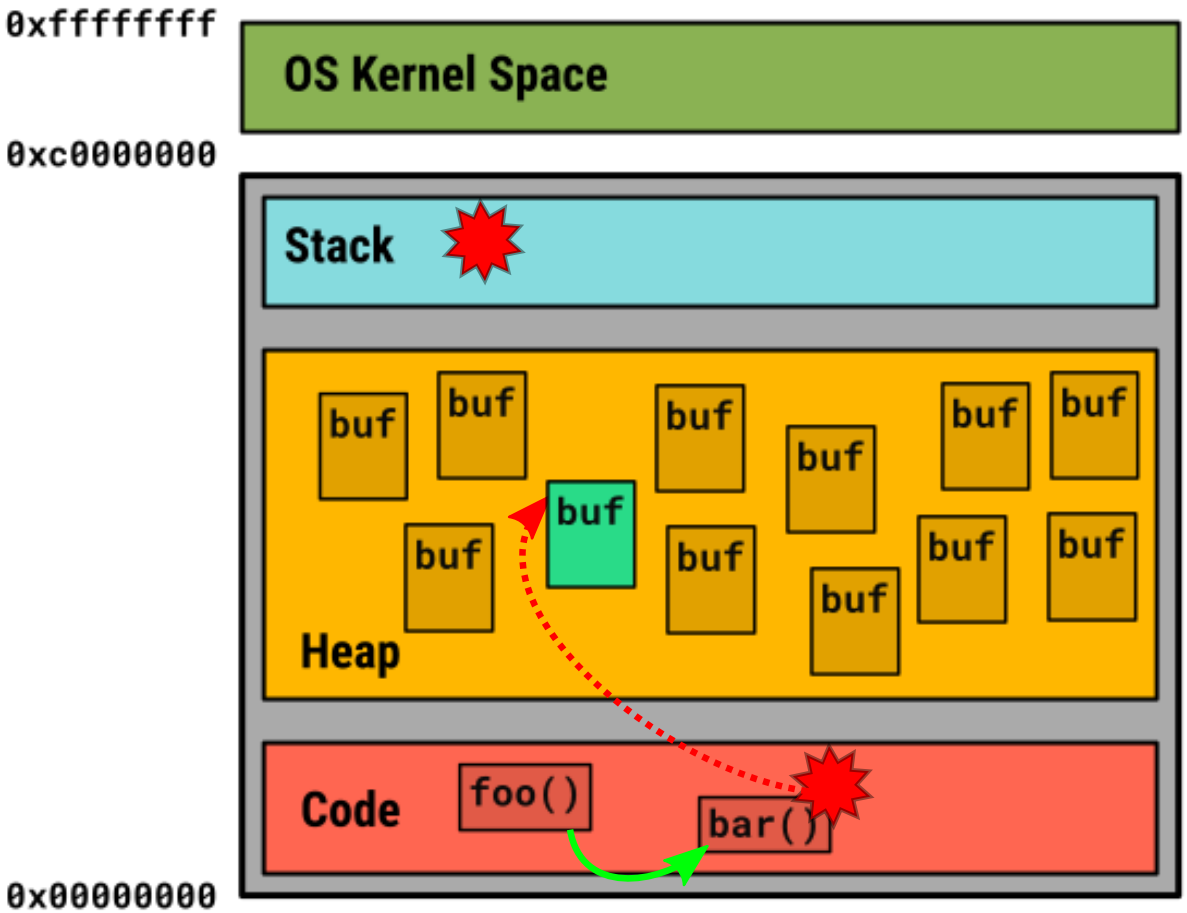
Heap Spraying



Heap Spraying



Heap Spraying



JIT Spraying

- Heap not executable
  - Can't *just* spray shellcode anymore
- JIT compilers need to create executable code on the fly
  - Spray the Heap with JIT code
    - JavaScript (Browser + PDF)
    - BPF (Kernel)
    - ...
- Code can include constants $\rightarrow$ Which could also be interpreted as code

Dangling Pointer

- A pointer that references data that is freed and which could be re-used by the program
- No guarantees can be made on the data anymore after it’s freed

Use after free (UAF)

- Happens when an object is free'd and then used again (dangling pointer)

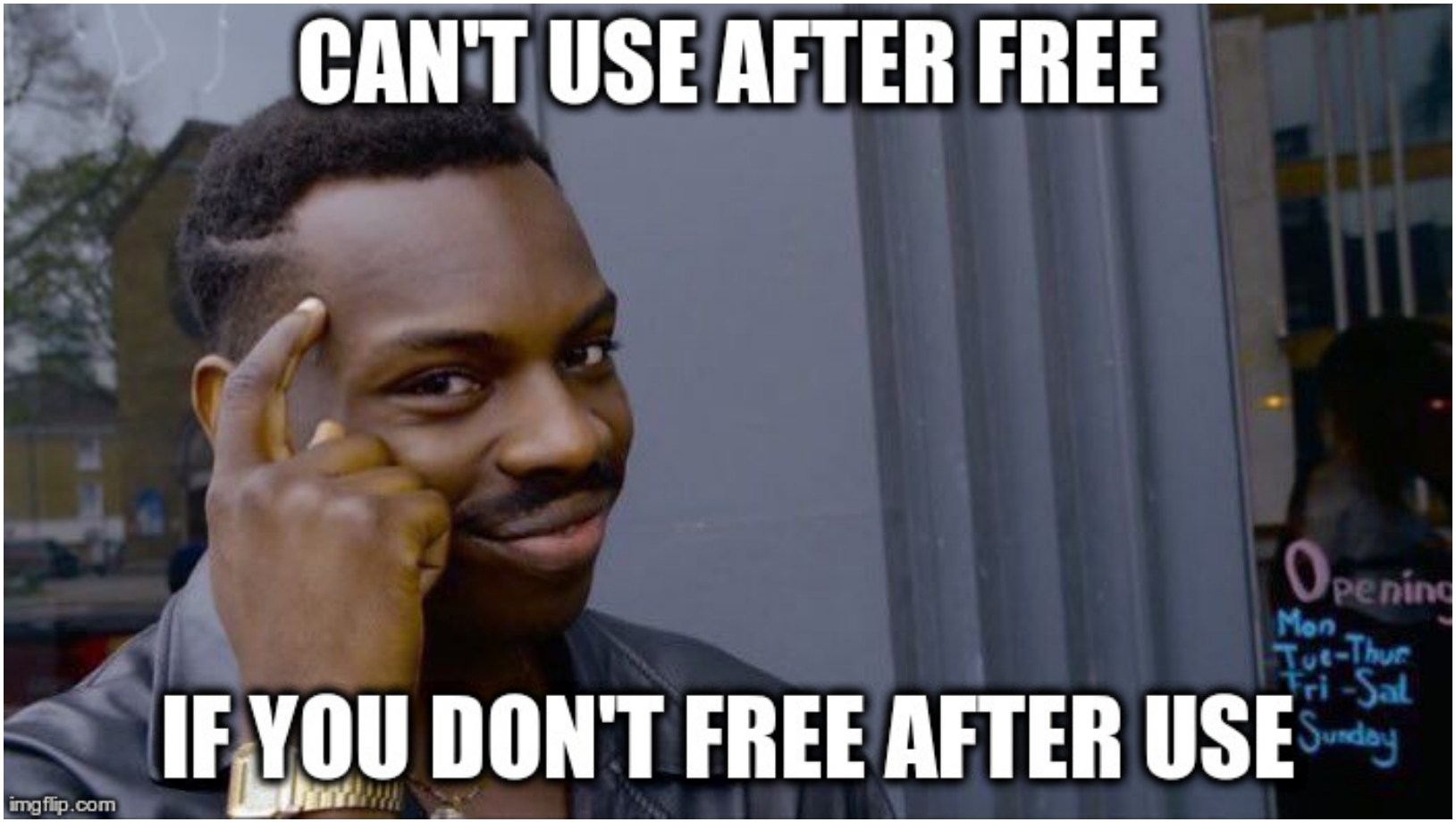
```
delete X;
...
X->func();
```

- Objects are located in memory
  - free/delete release the memory to be reused
  - If we can change the content of memory between the free and the use $\rightarrow$ Win
  - Especially interesting for function pointers (e.g C++ vtables)

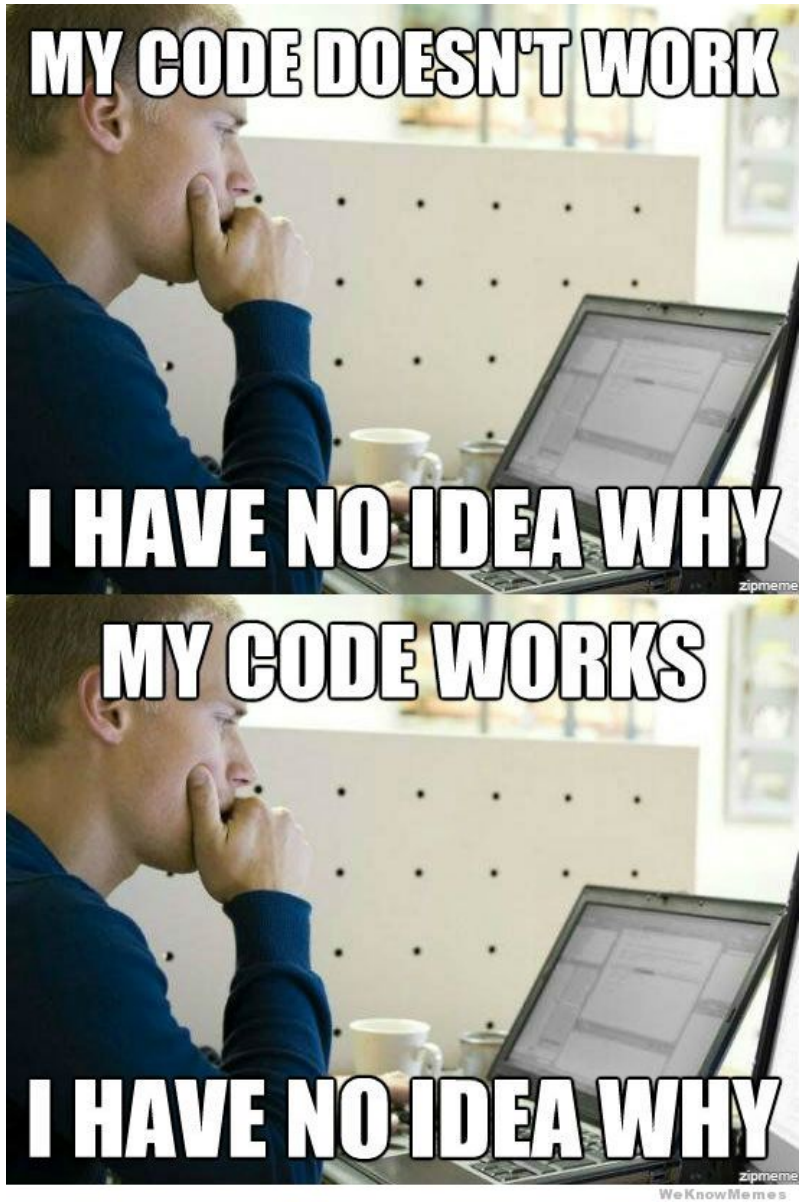
How to exploit UAF

- Free Object on Heap
- Create one or more smaller objects that fit into the free slot
  - Larger objects will not fit into the space
- If you can overwrite some function pointer that is reused:
  - Execute this function
  - WIN

How NOT to protect against it ;)



Other Attacks


MY CODE DOESN'T WORK
I HAVE NO IDEA WHY
zipmeme
MY CODE WORKS
I HAVE NO IDEA WHY
zipmeme
WeKnowMemes

Integer Overflows

- Simple **unsigned** 8 bit integer incremented

```
0x00
0x01
...
0xfe
0xff
???
```

- What happens here?

Integer Overflows

- Simple **unsigned** 8 bit integer incremented

```
0x00
0x01
...
0xfe
0xffF
0x00T
0x010
...
```

Integer Overflows

- What about **signed** integer overflow?

Integer Overflows

- What about **signed** integer overflow?
- This is actually **undefined behaviour** in C and C++
  - Might also be *optimized away*

Integer Overflows - Overflow Check Example

Demo Time

Kernel Exploitation

- Kernel?
- Usually the kernel was exploited by creating the shellcode in userspace and then jumping back from kernel space to execute
- Protection techniques against this
  - SMEP and SMAP

SMEP & SMAP

- SMEP (Supervisor Mode Execution Protection)
  - allows pages to be protected from supervisor-mode instruction fetches
  - Disable execution of userland pages
- SMAP (Supervisor Mode Access Protection)
  - allows pages to be protected from supervisor-mode data accesses
  - Disable access to userland pages
    - Protect against ROP/Stack Pivoting
- Both do not prevent exploitation, they just make it harder by removing possibility to access user space data from kernel space

Kernel ASLR Bypass

- Kernel got ASLR (kASLR)
- To write ROP chains we need to break this
  - Can be the same as for other programs, e.g. Memory leaks
- But do we need a kernel vulnerability to leak information?

Side channels to the rescue

- Two recent papers/techniques presented at CCS2016
- Get the kernel's code layout by leveraging processor/hardware **features**
- Patched/Protections in place now
- However, with all the recent news on CPU *features* most likely more to come

Summary

- Format String Vulnerability
- The Heap
- Other Problems

Summary

- Format String Vulnerability
- The Heap
- Other Problems

We touched a lot, but far from everything!