CSE 127: Computer Security

Memory (un)safety

Deian Stefan

Some slides adopted from Stefan Savage, Raluca Popal, and David Wagner

Today

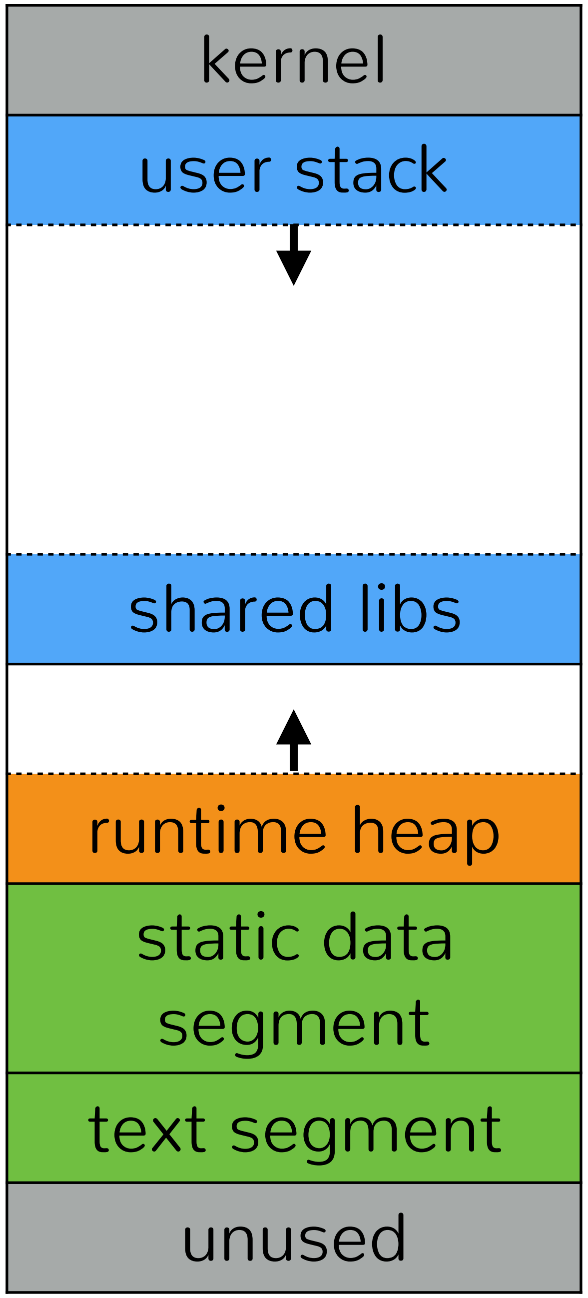
- Heap corruption
- Integer overflows

Memory management in C/C++

- C uses explicit memory management
  - Data is allocated and freed <u>dynamically</u>
  - Dynamic memory is accessed via pointers
- You are on your own
  - System does not track memory liveness
  - System doesn’t ensure that pointers are live or valid
- By default C++ has same issues

The heap

- Dynamically allocated data stored on the "heap"
- Heap manager exposes API for allocating and deallocating memory
  - `malloc()` and `free()`
  - API invariant: every memory allocated by `malloc()` **has to** be released by corresponding call to `free()`



How can things go wrong?

- Forget to free memory
- Write/read memory we shouldn't have access to
- Use pointers that point to freed object
- Free already freed objects

Most important: heap corruption

- Can bypass security checks (data-only attacks)
  - E.g., `isAuthenticated`, `buffer_size`, `isAdmin`, etc.
- Can overwrite function pointers
  - Direct transfer of control when function is called
  - C++ virtual tables are especially good targets
- Can overwrite heap management data
  - Program the heap weird machine

How does the heap work?

- Abstraction vs. reality of `malloc()` and `free()`

How does the heap work?

- Abstraction vs. reality of `malloc()` and `free()`
- Abstraction: magic!
  - Dynamically allocate and release memory as needed
  - Give me 20 bytes: `ptr = malloc(20);`
  - I don't need my 20 bytes: `free(ptr);`

How does the heap work?

- Abstraction vs. reality of `malloc()` and `free()`
- Abstraction: magic!
  - Dynamically allocate and release memory as needed
  - Give me 20 bytes: `ptr = malloc(20);`
  - I don't need my 20 bytes: `free(ptr);`
- Reality: not magic.
  - Where does the memory come from?
  - How does the system know how much memory to reclaim when `free(ptr)` is called?

How does the heap work?

- Heap is managed by the heap manager/memory allocator
- Many different heap managers, different tradeoffs:
  - Speed:
  - Space:
  - Security: avoid the pitfalls we'll talk about today
- Today: dlmalloc -> glibc dlmalloc -> ptmalloc2

How does the heap work?

- Heap is managed by the heap manager/memory allocator
- Many different heap managers, different tradeoffs:
  - Speed: allocation and deallocation should be free
  - Space: memory should used efficiently
  - Security: avoid the pitfalls we'll talk about today
- Today: dlmalloc -> glibc dlmalloc -> ptmalloc2

Heap management

- Organized in contiguous **chunks** of memory
- Heap layout evolves with `malloc()`s and `frees()`s
  - Chunks may get allocated, freed, split, coalesced
- Free chunks are stored in doubly linked lists (bins)
  - Different kinds of bins: fast, unsorted, small, large, ...

What’s a chunk?

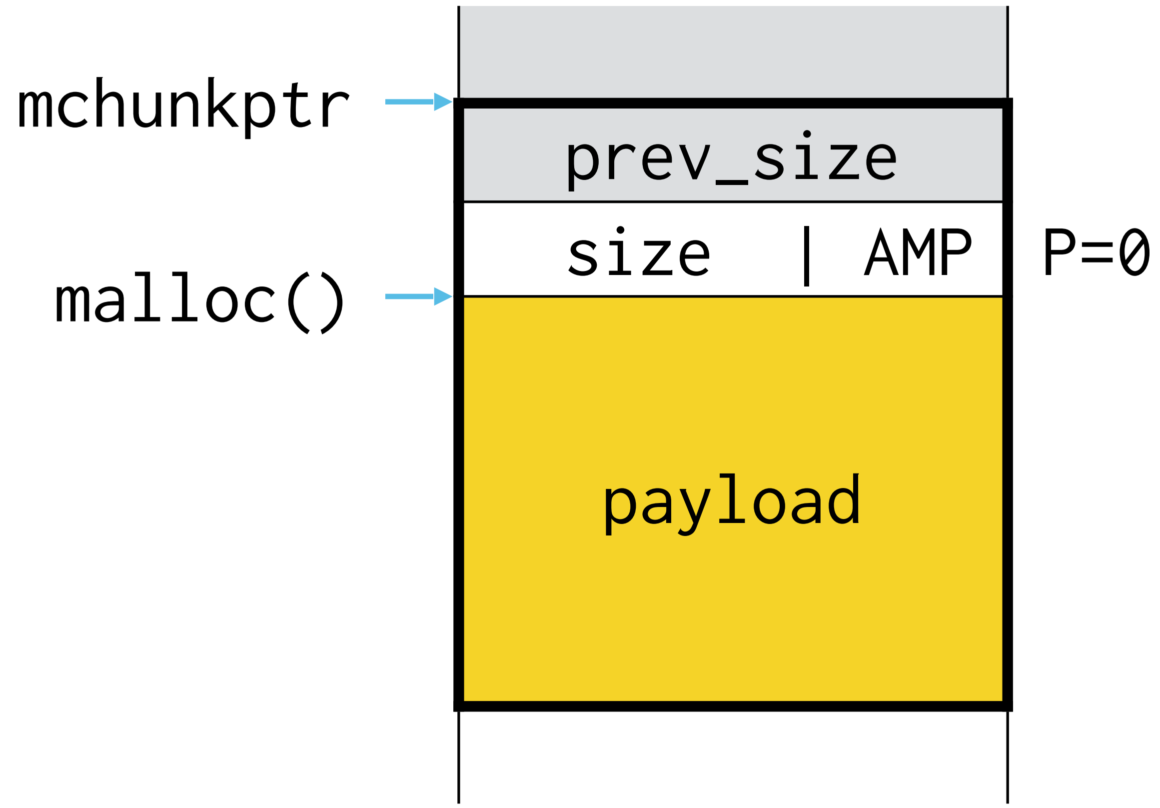
(let’s look at malloc.c)

What's a chunk?

- Basic unit of memory on the heap
  - Can be either <u>free</u> or <u>in-use</u>
- Metadata: size (8-bit aligned) + flags
  - chunk size | A | M | P
- What else?
  - Allocated chunk: payload
  - Free chunk:
    - links to next/previous chunks
    - size of previous chunk (same as size)

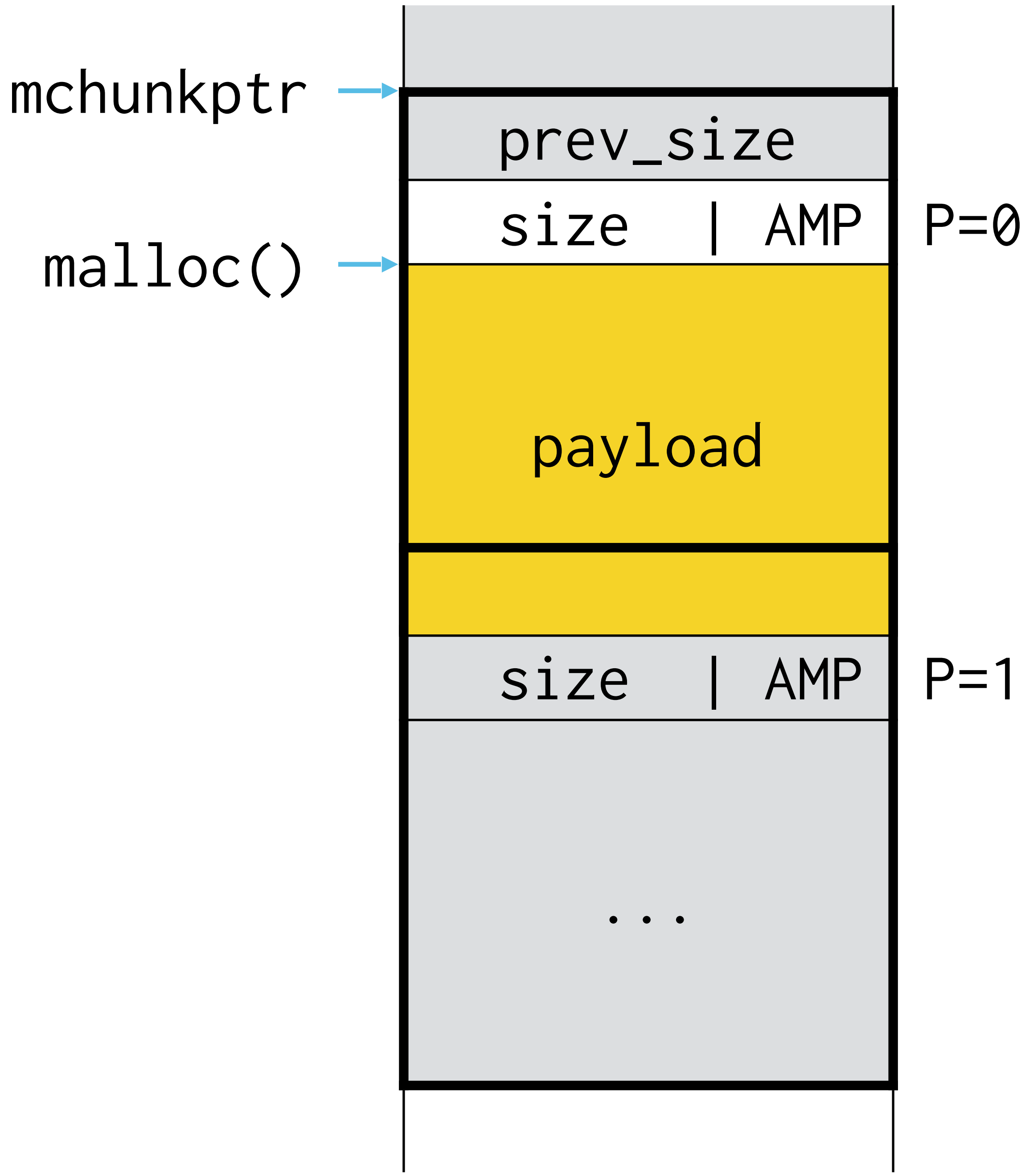
In use chunk

- malloc returns pointer to the payload
- How does free know how much to free?
  - Look at the metadata: chunk size
- Last word of payload is first word of next chunk



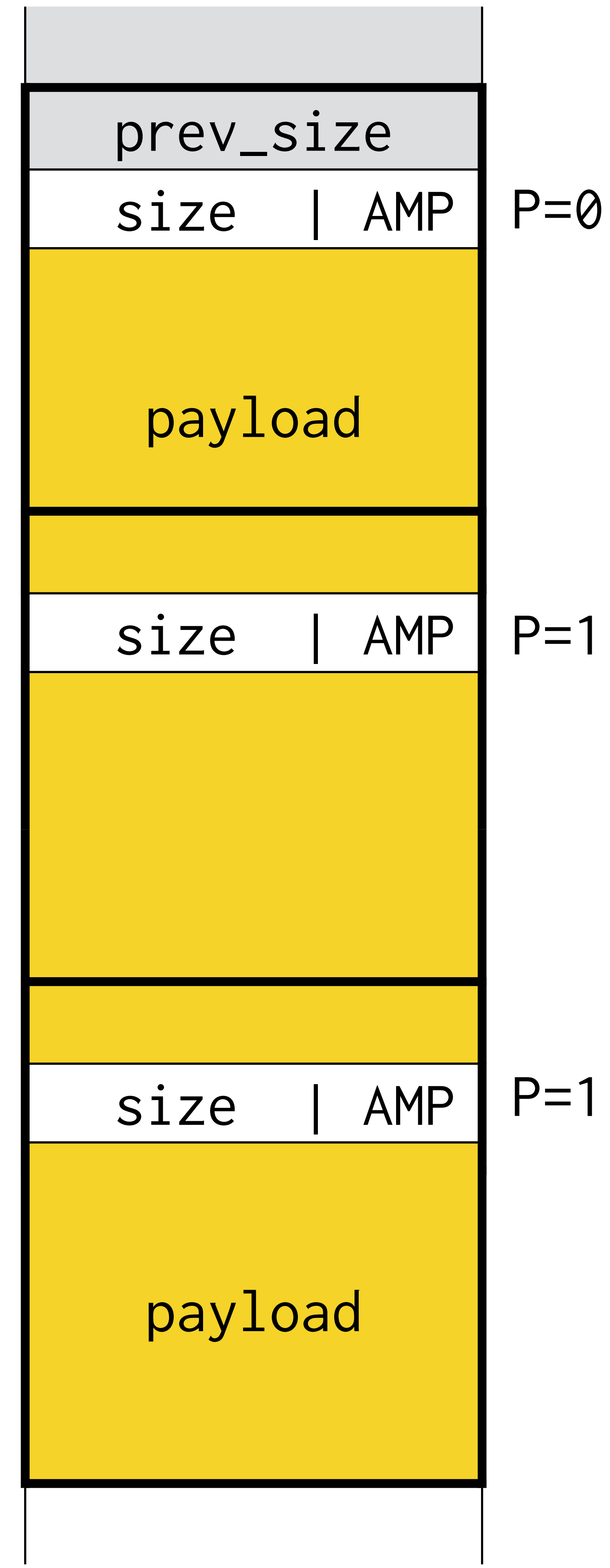
In use chunk

- malloc returns pointer to the payload
- How does free know how much to free?
  - Look at the metadata: chunk size
- Last word of payload is first word of next chunk



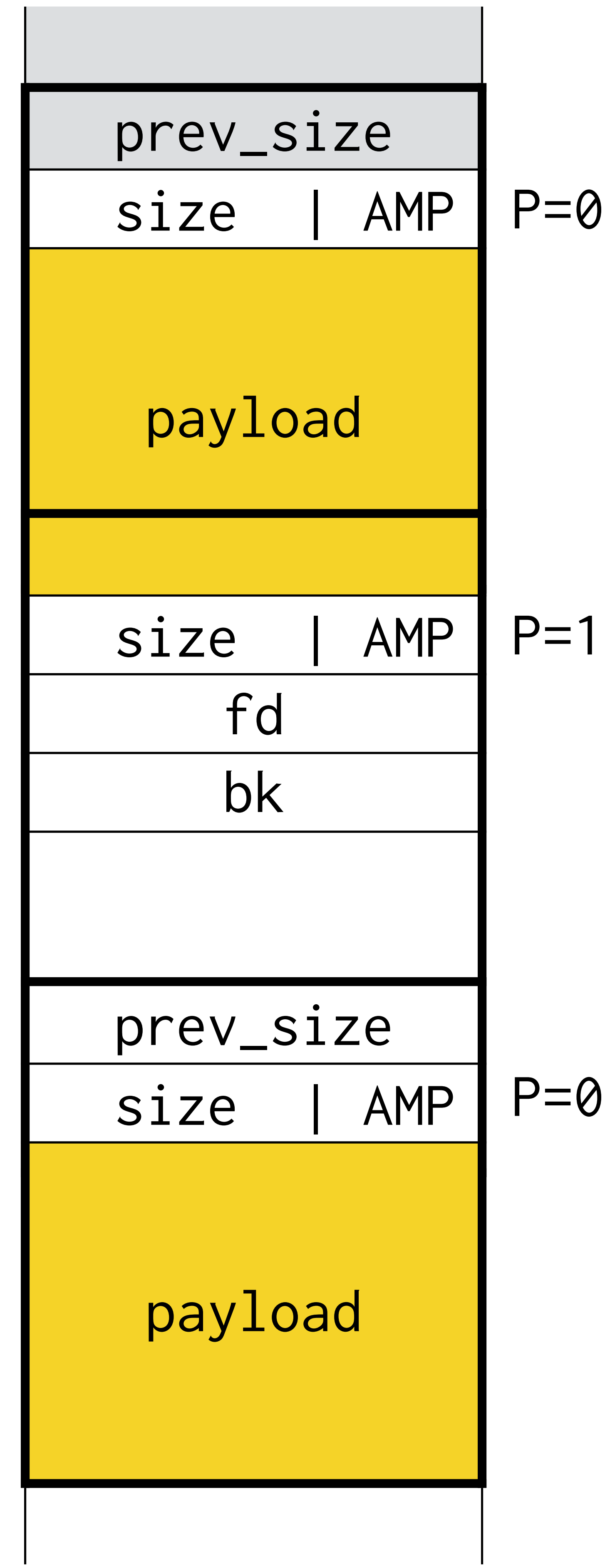
Free chunk

- Free chunks are kept in doubly linked list
  - ➤ Unused payload data is used to store link pointers
- Consecutive free chunks are coalesced
  - ➤ No two free chunks can be adjacent to eac hother
- Last word: size of the chunk

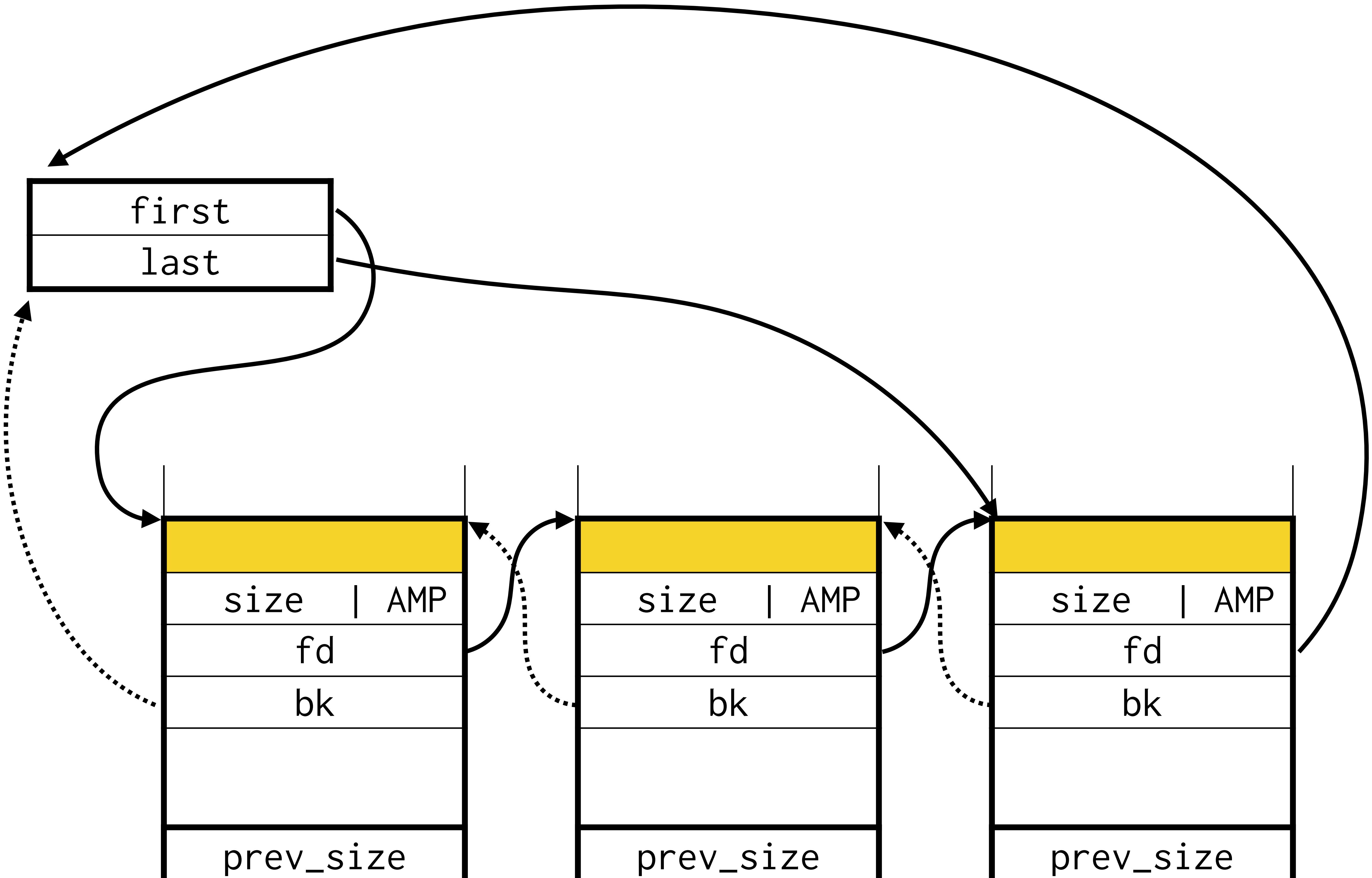


Free chunk

- Free chunks are kept in doubly linked list
  - ➤ Unused payload data is used to store link pointers
- Consecutive free chunks are coalesced
  - ➤ No two free chunks can be adjacent to each other
- Last word: size of the chunk

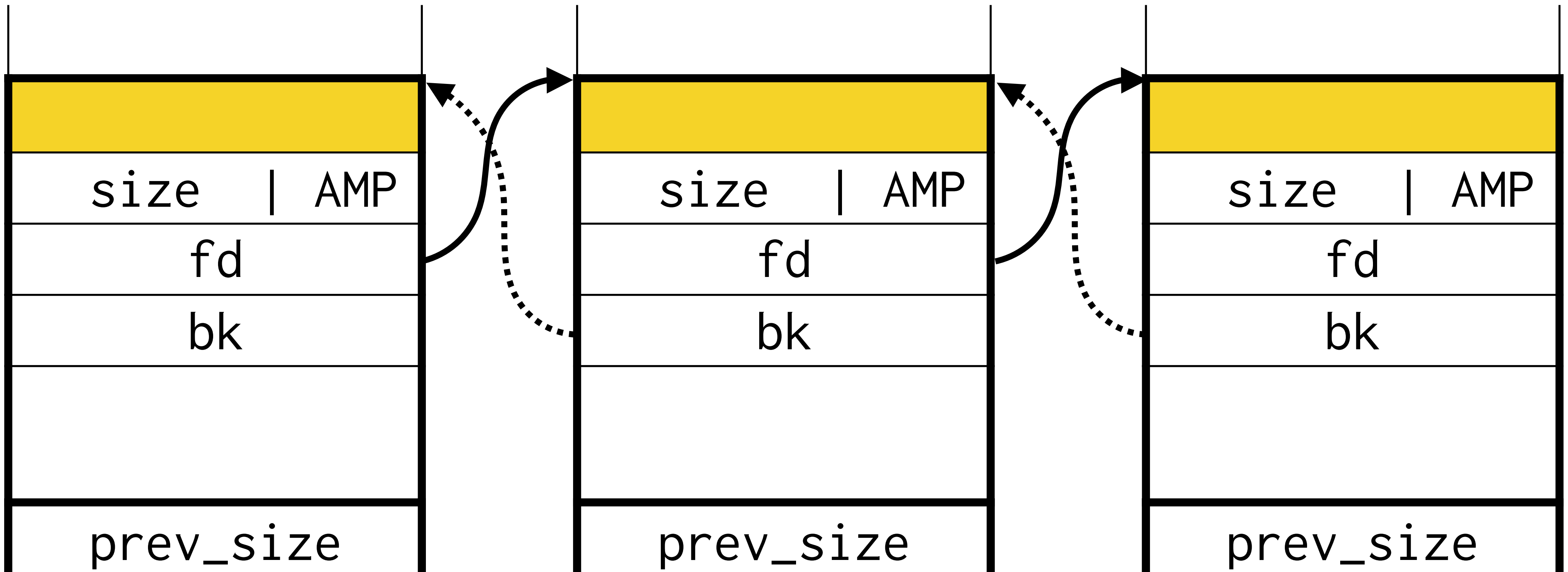


Free list (bin)



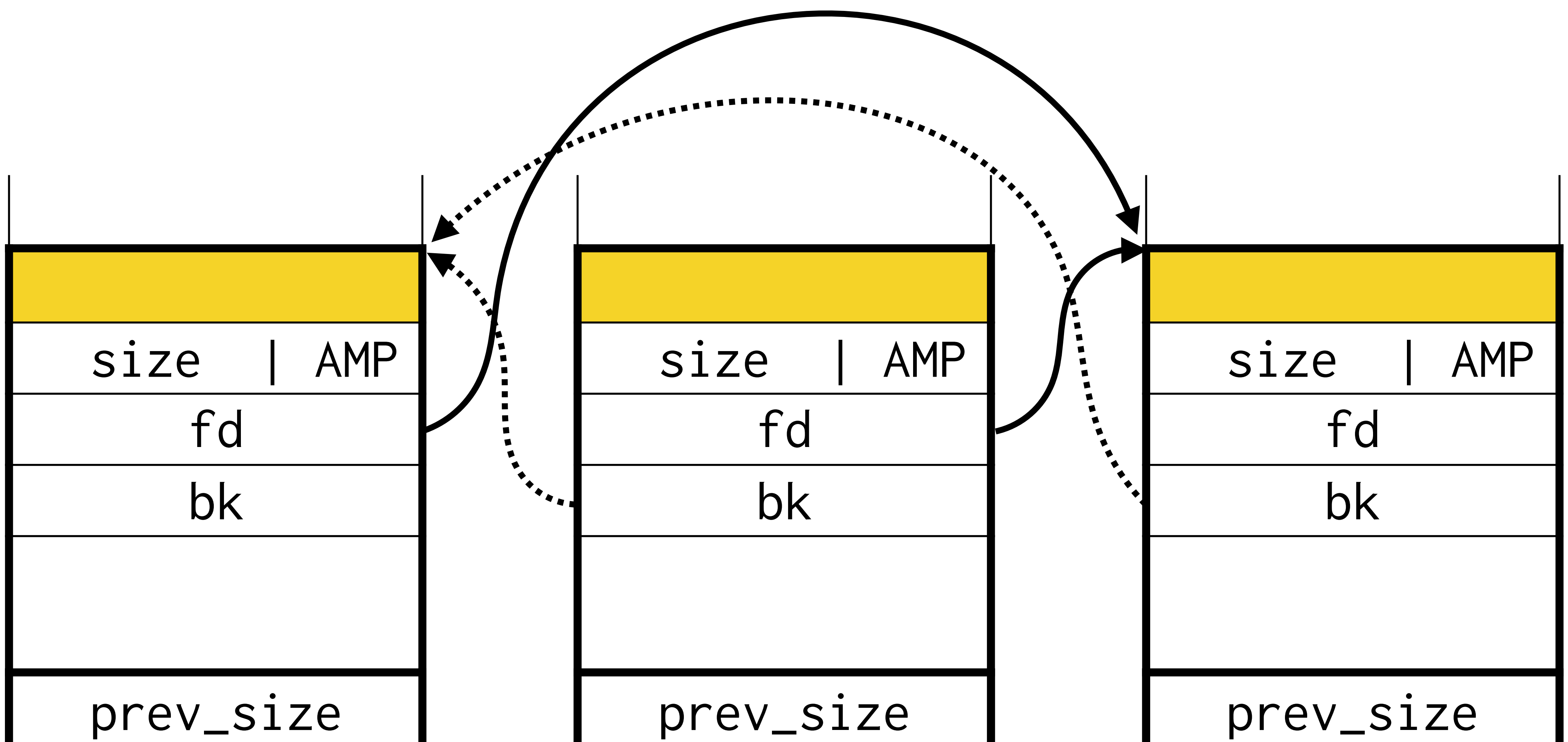
What happens when we allocate?

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



What happens when we allocate?

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



Heap corruption

- What can we do if we manage to get free to act on data we control?
  - What if we can cause the heap manager to act on fake chunks?
- What can we do if we can control what’s in `fd` and `bk` fields?
  - Arbitrary write gadget: write-what-where

Heap corruption

- How can attacker corrupt metadata in free chunk?
    - Simple overflow
    - Indirect overwrite
    - Use after free
    - Fake chunk

Don't need to abuse manager

- Don't need to bend the heap manager's control flow to hijack control flow
- What can we do instead?
  - Overflow code pointers on the heap
  - Use after free
  - Double free

Let’s look at some C code

Let’s look at some C++

C++ vtables

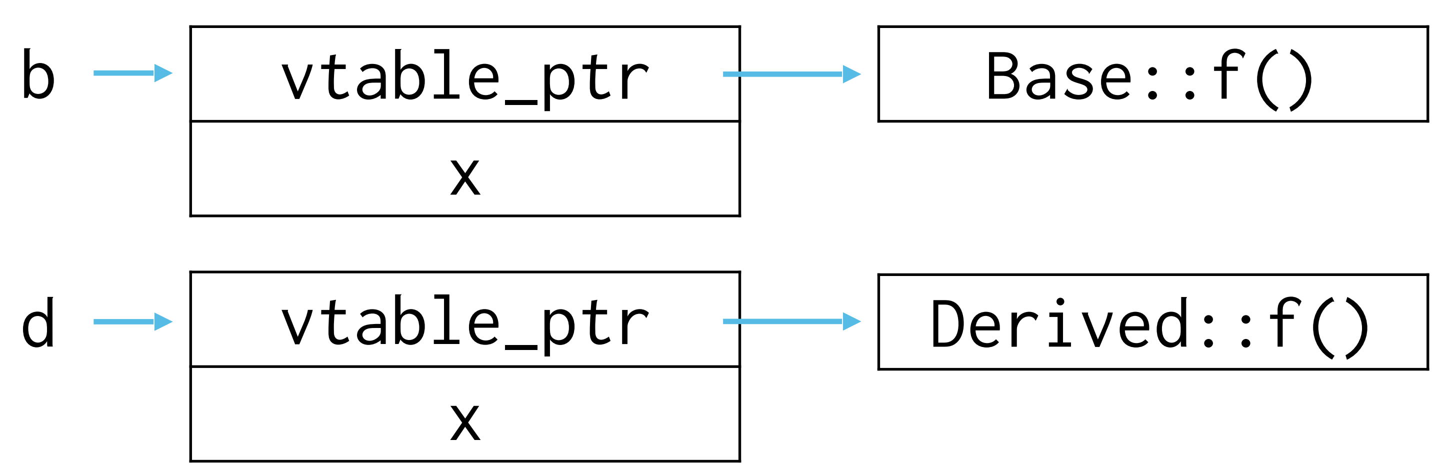
```
class Base {
  public:
   uint32_t x;
   Base(uint32_t x) : x(x) {};
   virtual void f() {
     cout << “base: ” << x;
   }
};

class Derived: public Base {
  public:
   Derived(uint32_t x) : Base(x) {};
   void f() {
     cout << “derived: ” << x;
   }
};

void bar(Base* obj) {
   obj->f();
}

int main(int argc, char* argv[])
{
  Base *b = new Base(42);
  Derived *d = new Derived(42);

  bar(b);
  bar(d);
 }
```

Q: What does this print out?

A: `base:42`
`derived: 42`

Q: What does `bar()` compile to?

A: `*(obj->vtable[0])(obj)`



UAF in C++

Victim: Free object: `free(obj);`

Attacker: Overwrite the vtable of the object so entry (e.g., `obj->vtable[0]`) points to attacker gadget

Victim: Use dangling pointer: `obj->foo()`

Why talk about these attacks?

Microsoft

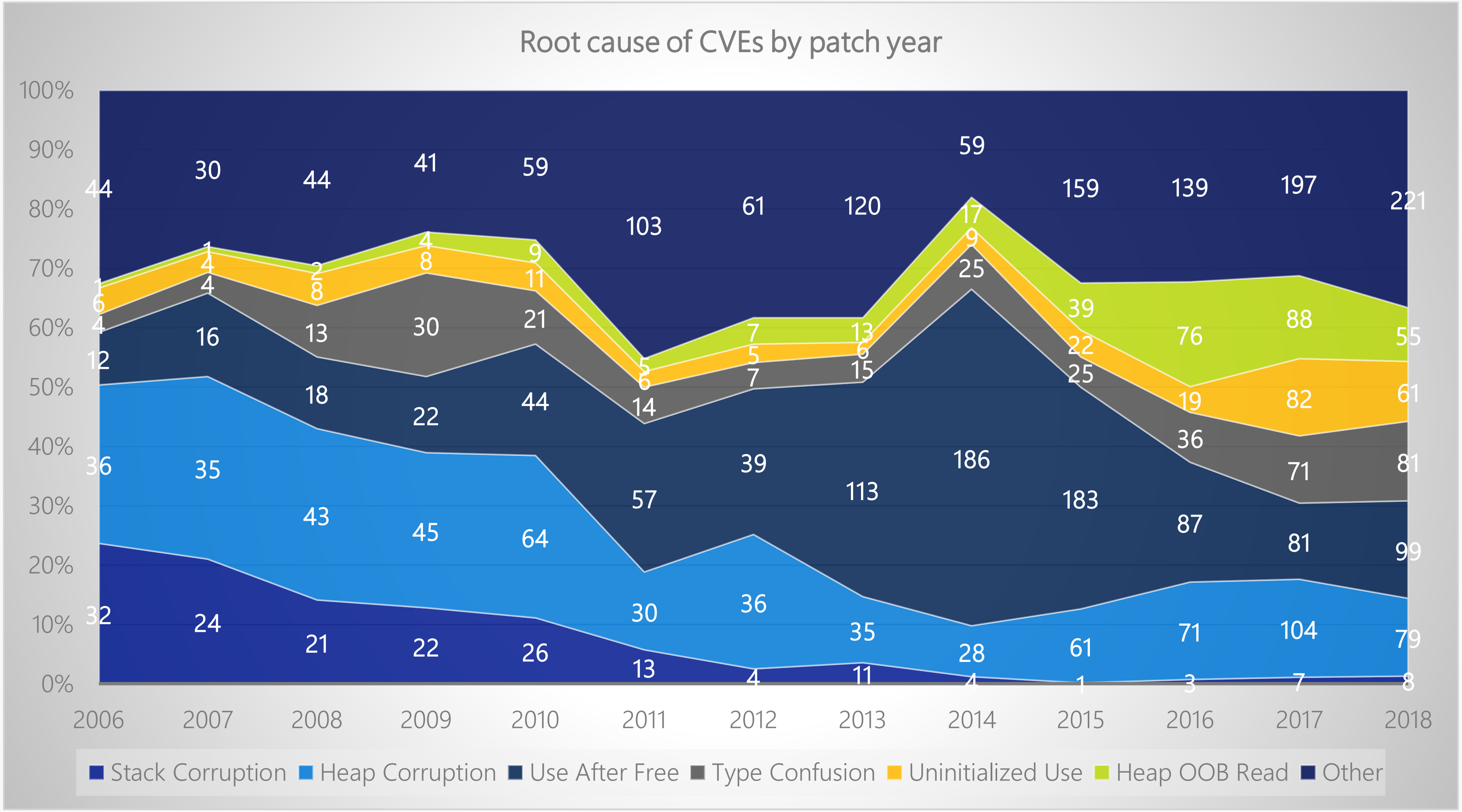
Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape

Matt Miller (@epakskape)
Microsoft Security Response Center (MSRC)

BlueHat IL
February 7th, 2019

Why talk about these attacks?

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

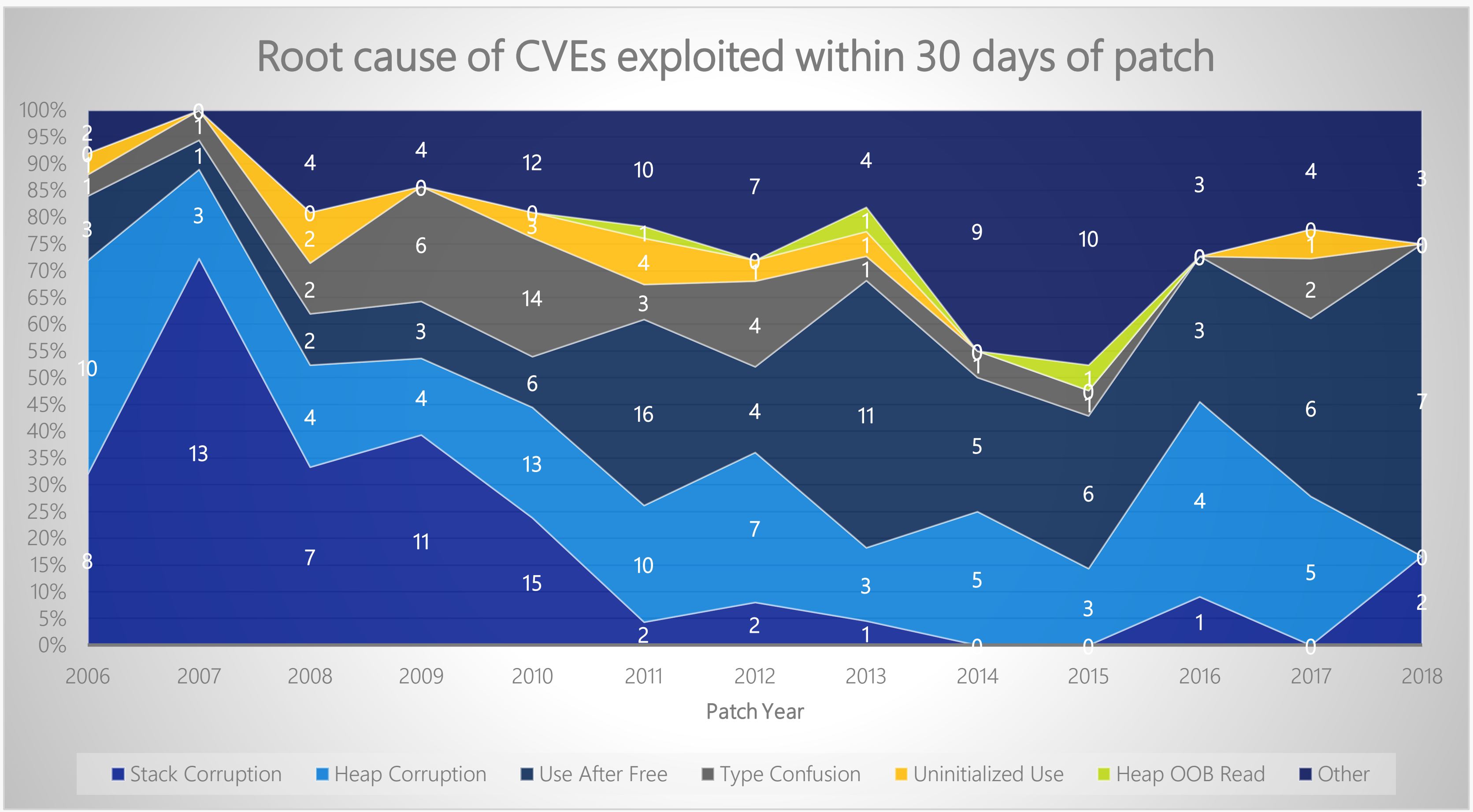
Top root causes since 2016: #1: heap out-of-bounds | #2: use after free | #3: type confusion | #4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Why talk about these attacks?

Root causes of exploited vulnerabilities

The root cause of exploited vulnerabilities provide hints on attacker preference & ease of exploitability



Use after free and heap corruption continue to be preferably targeted

“Other” category consists of a few common types of issues:

- XSS & zone elevation issues
- DLL planting issues
- File canonicalization & symbolic link issues

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Even null pointer errors are tricky

- What does this code do?

```
char *p = NULL;

*p = 20;
```

- A null pointer is a pointer to address **0**

Even null pointer errors are tricky

- What does this code do?

```
char *p = NULL;

*p = 20;
```

- A null pointer is a pointer to address **0**
  - ➤ Dereferencing null pointer can lead to crash (DoS)
  - ➤ There is more to it though.. what's at address 0?

Return-to-user attack

- What if process mapped page 0 and…
- What if the process manages to trigger a null pointer dereference in the kernel
  - Instead of crashing the kernel will use attacker-controlled data on page 0

What to do?

- Disallow mapping 0
- Safe heap implementations
  - Safe unlinking
  - Cookies/canaries on the heap
  - Heap integrity check on malloc and free
- Use Rust or a safe GCed language

Today

- Heap corruption
- Integer bugs

What’s wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

What’s wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

MEMCPY(3) **Linux Programmer's Manual** **MEMCPY(3)**

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

What's wrong with this program?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

MEMCPY(3) **Linux Programmer's Manual** **MEMCPY(3)**

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

What's wrong with this program?

```
void vulnerable(int len = 0xffffffff, char *data) {
  char buf[64];
  if (len = -1 > 64)
    return;
  memcpy(buf, data, len = 0xffffffff);
}
```

MEMCPY(3) **Linux Programmer's Manual** **MEMCPY(3)**

NAME top

memcpy - copy memory area

SYNOPSIS top

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```

Let's fix it:

```
void safe(size_t len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

Is this program safe?

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL)
    return;
  memcpy(buf, data, len);
  buf[len] = ‘\n’;
  buf[len+1] = ‘\0’;
}
```

Is this program safe?

No!

```
void f(size_t len = 0xffffffff, char *data) {
  char *buf = malloc(len+2 = 0x000000001);
  if (buf == NULL)
    return;
  memcpy(buf, data, len = 0xffffffff);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

Three flavors of integer overflows

- Truncation bugs
  - E.g., assigning an int64_t into in32_t
- Arithmetic overflow bugs
  - E.g., adding huge unsigned number
- Signedness bugs
  - E.g., treating signed number as unsigned

Still relevant classes of bugs

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlfbr...@stanford.edu on Fri, Apr 12, 2019, 1:59 PM PDT

Code

VULNERABILITY DETAILS
Possible OOB with chrome_sqlite3_malloc

REPRODUCTION CASE
There's a pattern of using sqlite malloc functions that call chrome_sqlite3_malloc in combination with traditional memory operations (e.g., memcpy). There may be invariants that make this ok, or a principle here that I am not aware of. Thanks for your time.

chrome_sqlite3_malloc takes an int size argument, while memcpy takes a size_t size argument. On x86-64 this means that chrome_sqlite_3_malloc's size argument is width 32, while memcpy's is width 64. This can lead to potentially concerning wrapping behavior for extreme allocation sizes (depending on the compiler, optimizations, etc).

For example:

Function fts3UpdateDocTotals
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3399)

(1) a = sqlite3_malloc((sizeof(u32)+10)*nStat);
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3416)
...
(2) memset(a, 0, sizeof(u32)*(nStat));
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3434)

Depending on optimization level etc, this may turn into:

(1)
size = mul i32 nstat 14
chrome_sqlite3_malloc(size)

Today

- Heap corruption
- Integer overflows

What does this all tell us?

If you're trying to build secure systems, use a memory and type safe language.