CSE 127: Computer Security

Memory (un)safety

Deian Stefan

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

Today

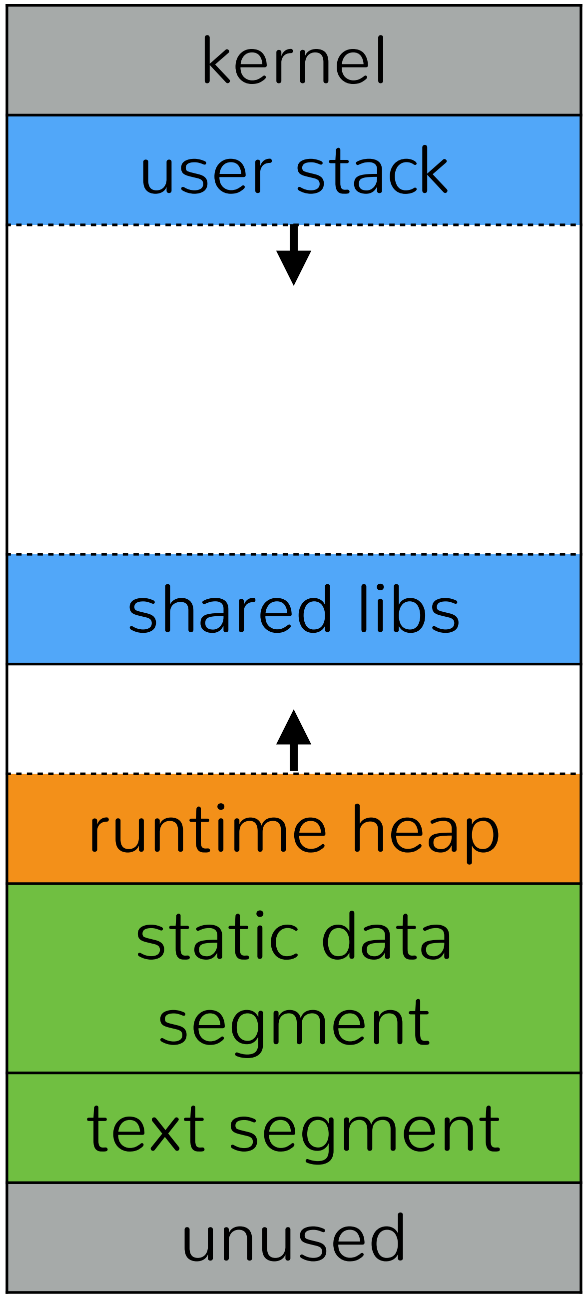
- Heap corruption
- Integer overflows

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()`
  - <u>API invariant:</u> 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()`
- 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: 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

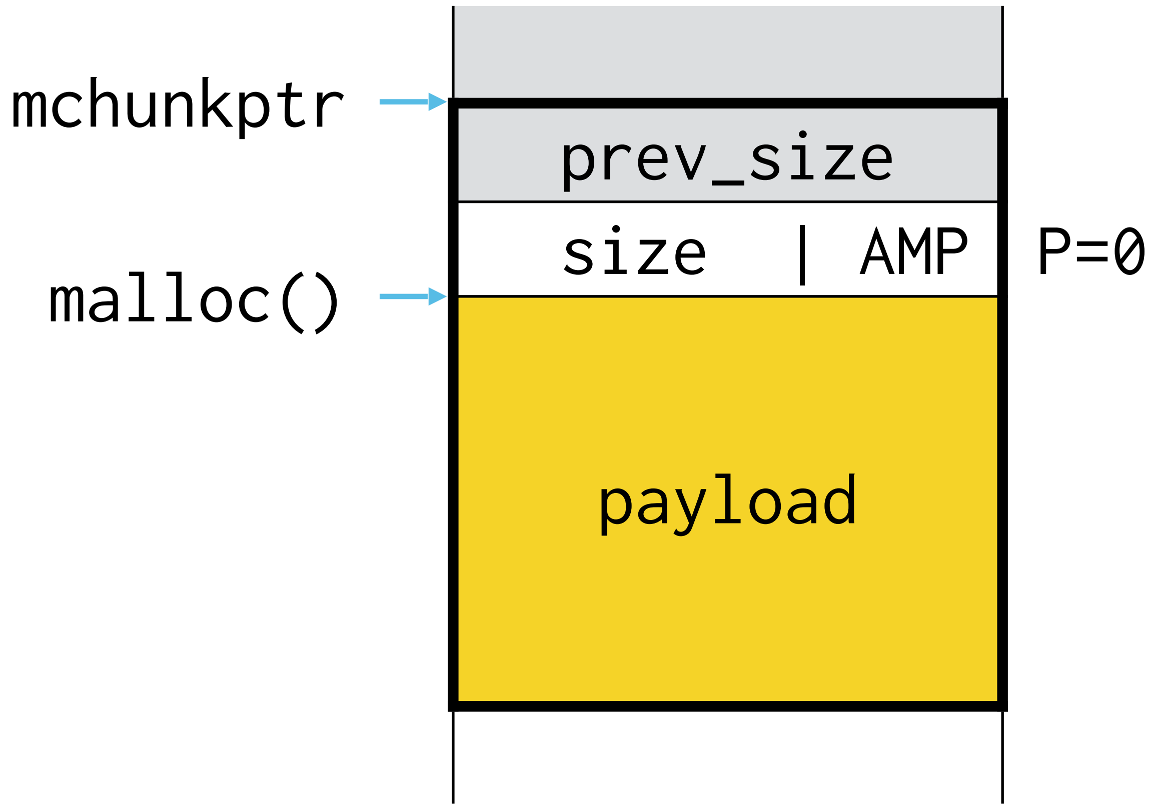
- Organized in contiguous **chunks** of memory
- Heap layout evolves with `malloc()`s and `frees()`s
  - Chunks may get allocated, freed, split, coalesced
- Free chunks are stored in doubly linked lists (bins)
  - Different kinds of bins: fast, unsorted, small, large, ...

What’s a chunk?

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

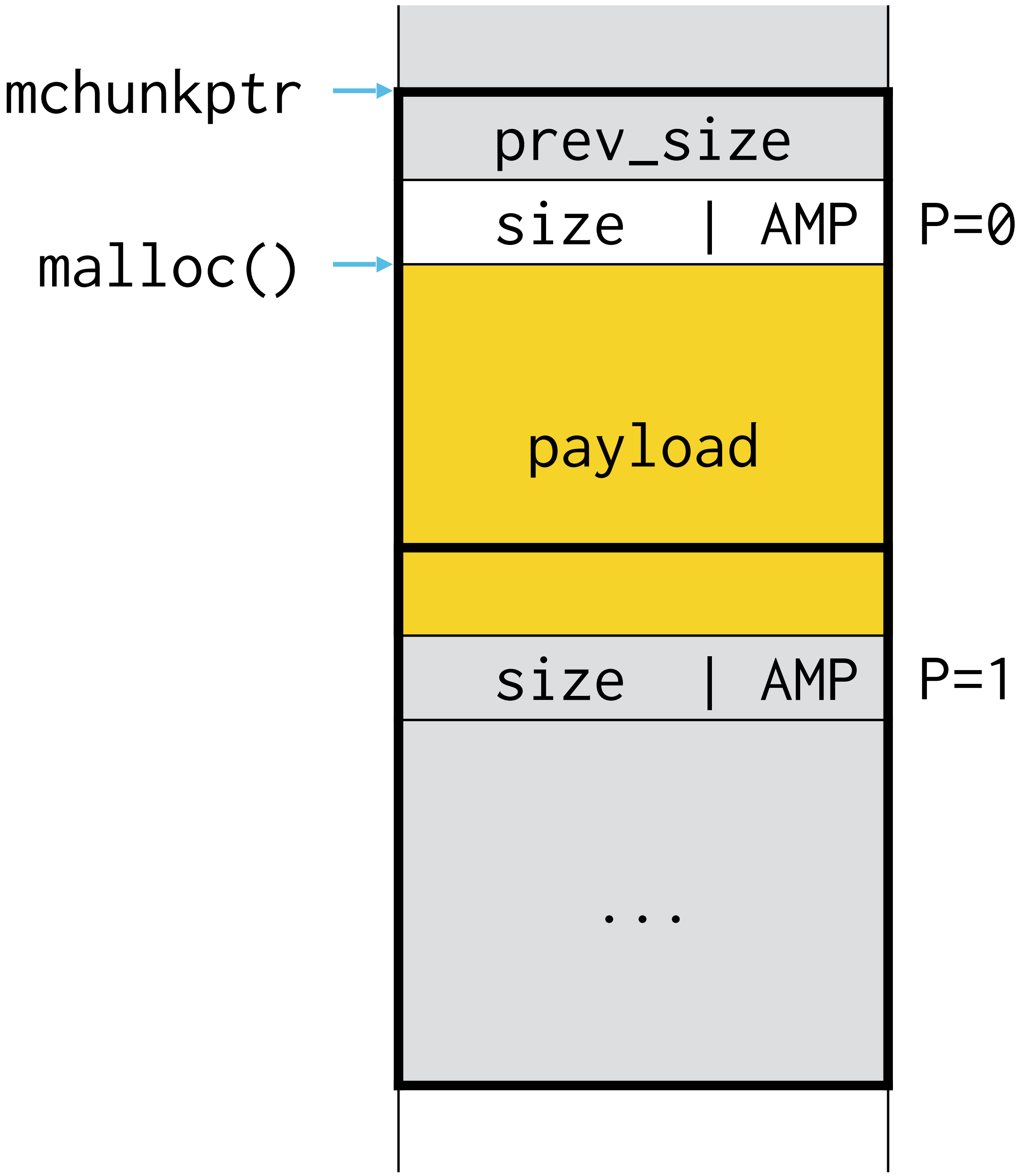
In use chunk

- 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



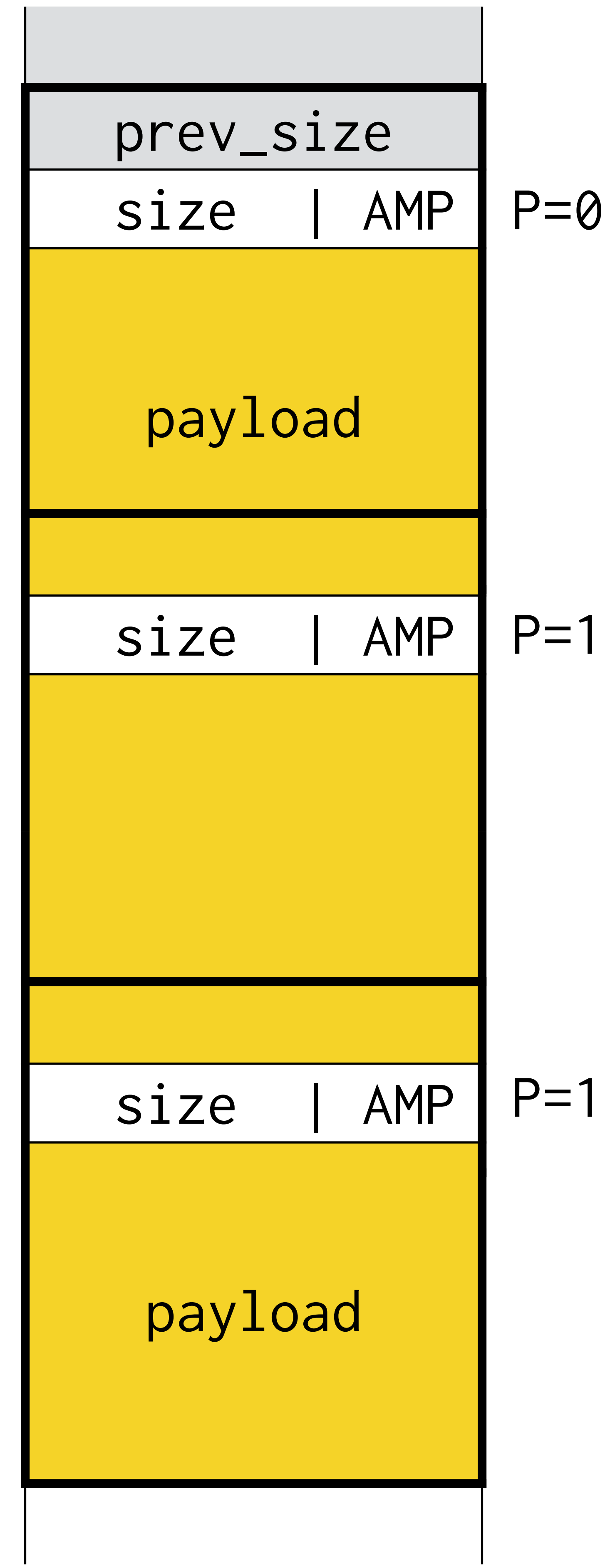
In use chunk

- 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



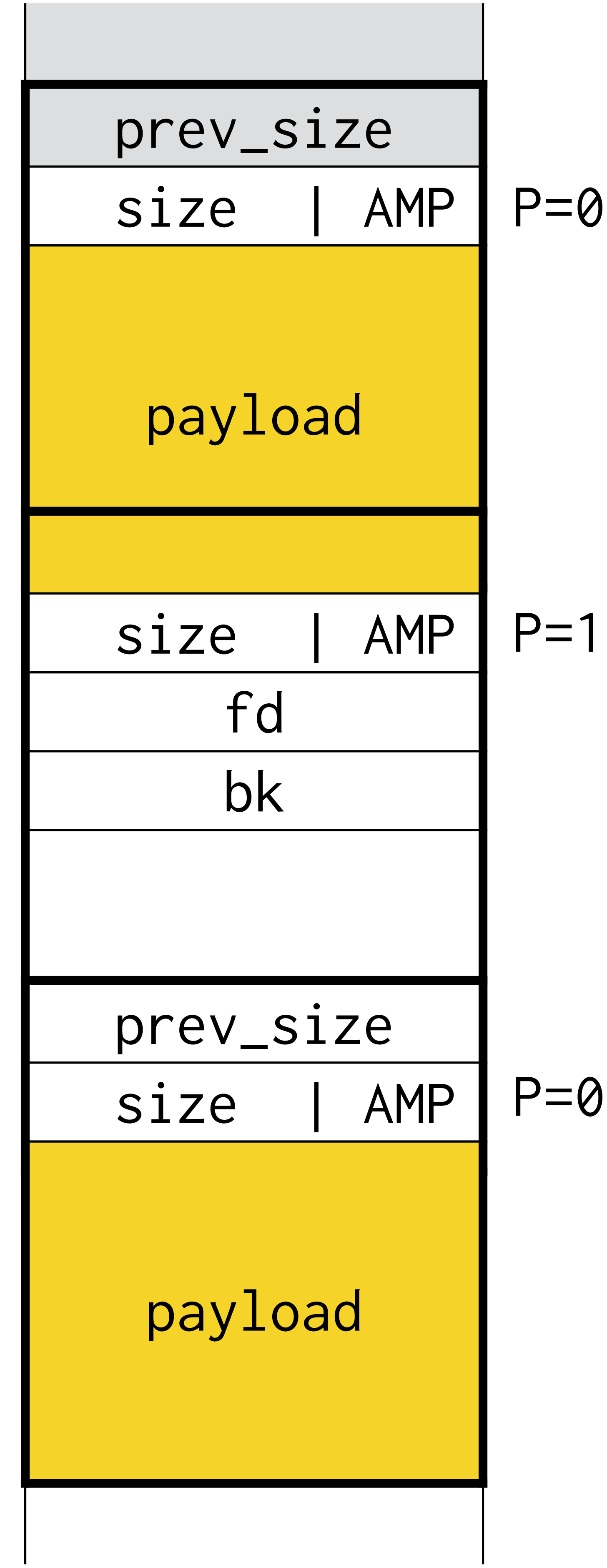
Free chunk

- 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

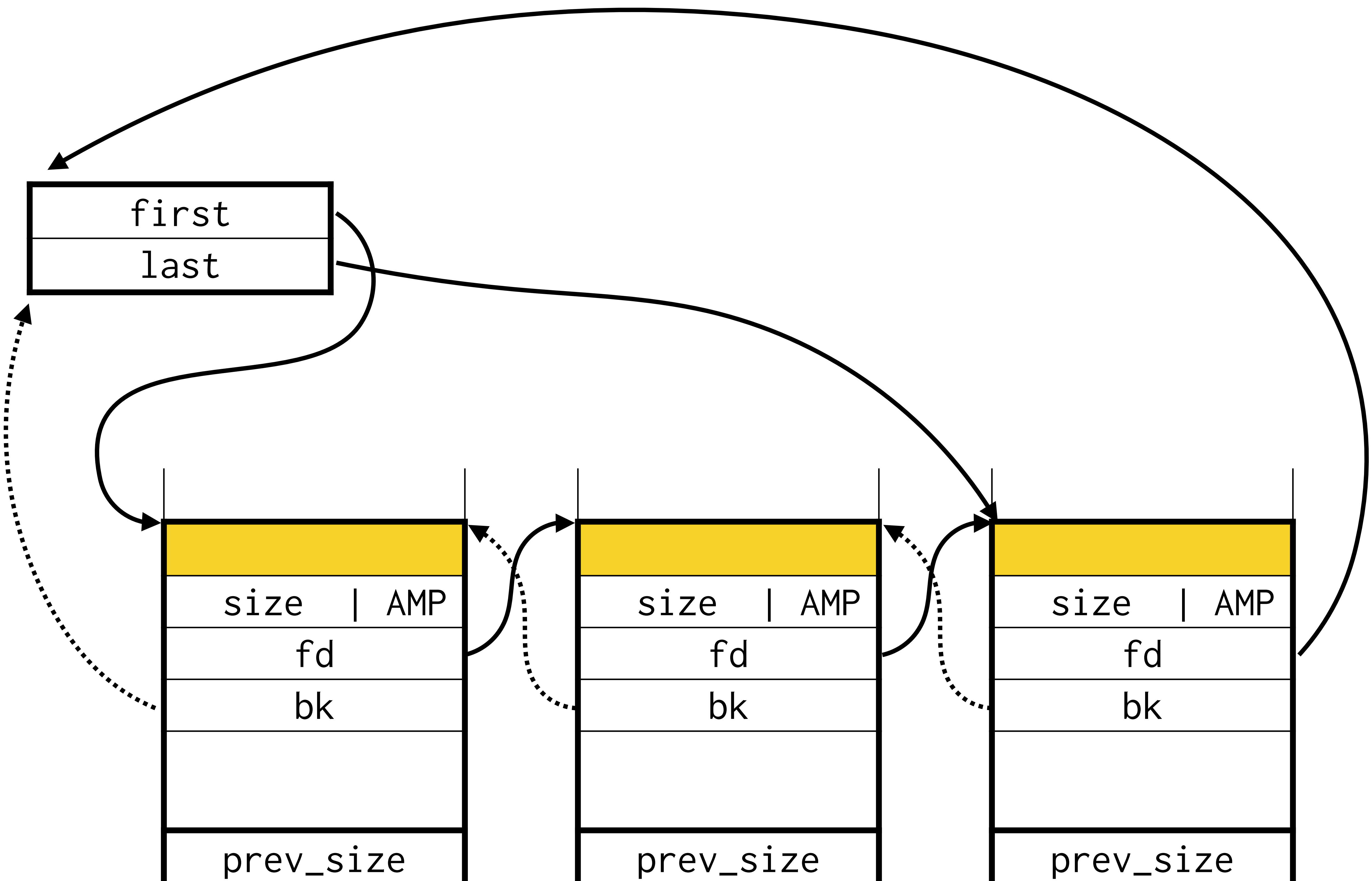


Free chunk

- 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

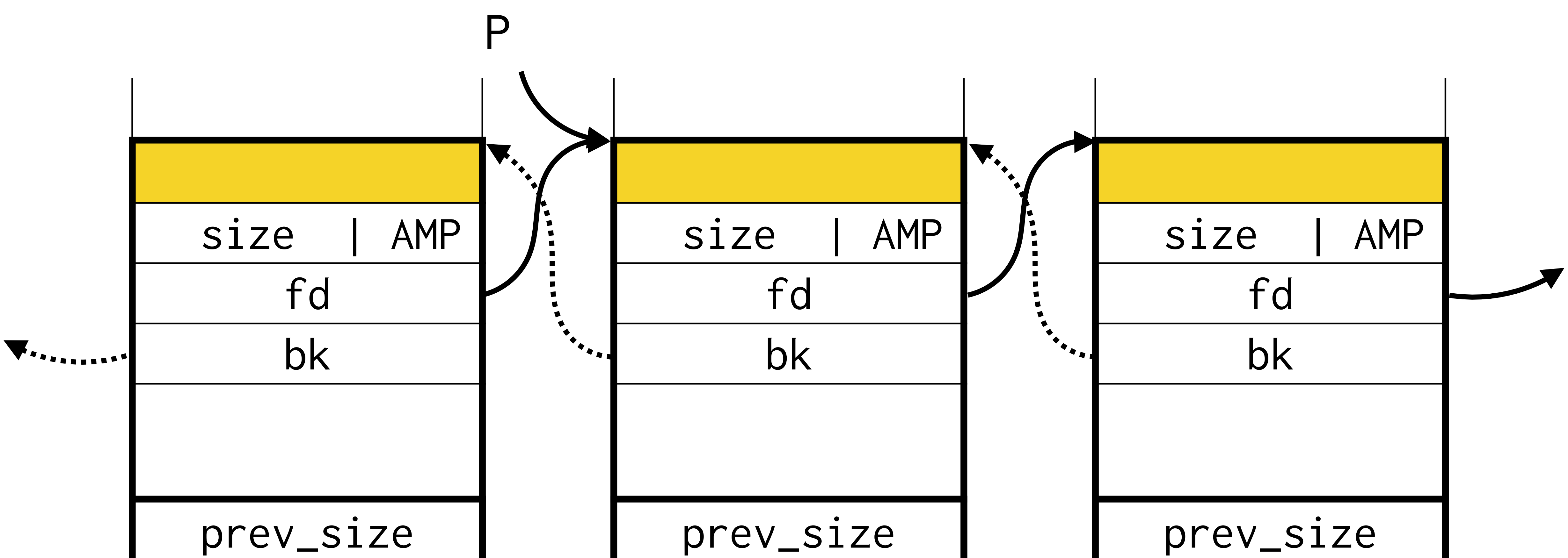


Free list (bin)



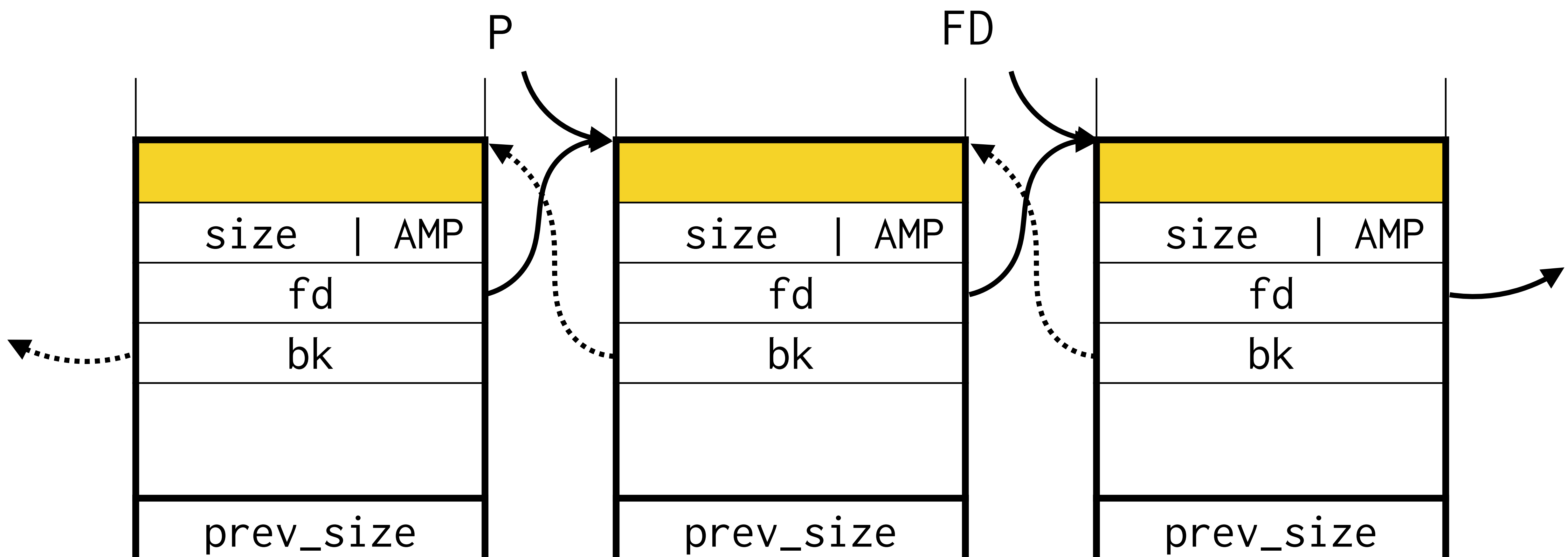
What happens when we allocate?

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



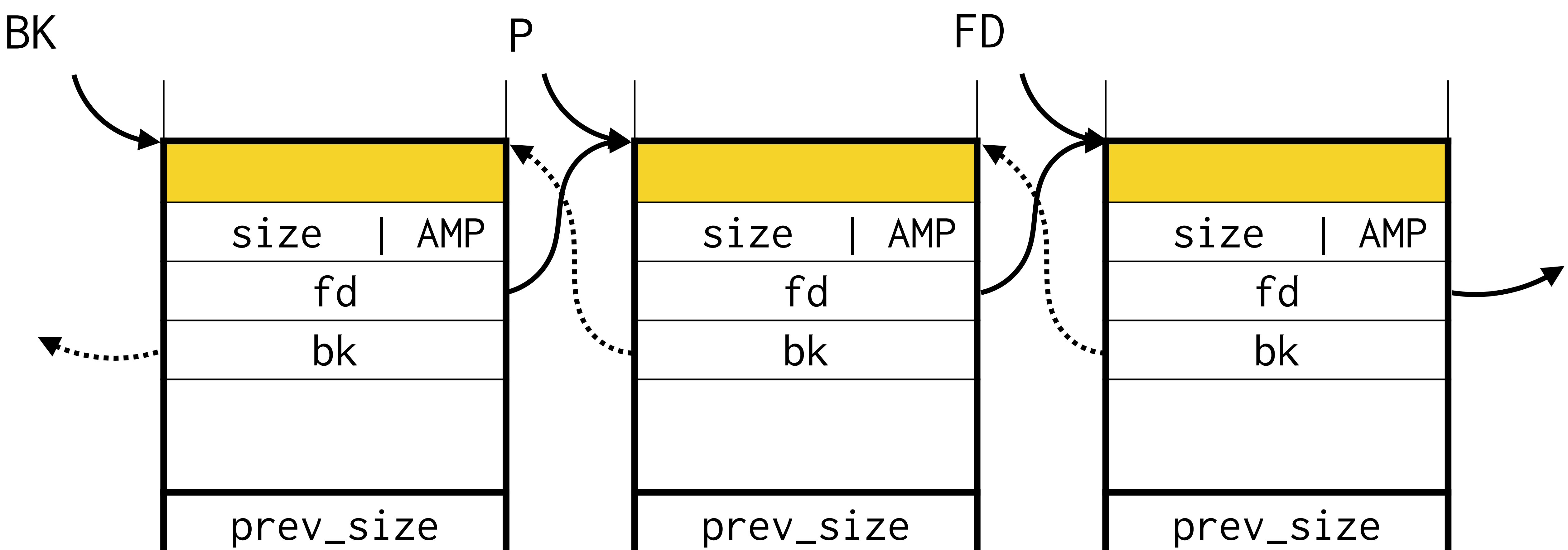
What happens when we allocate?

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



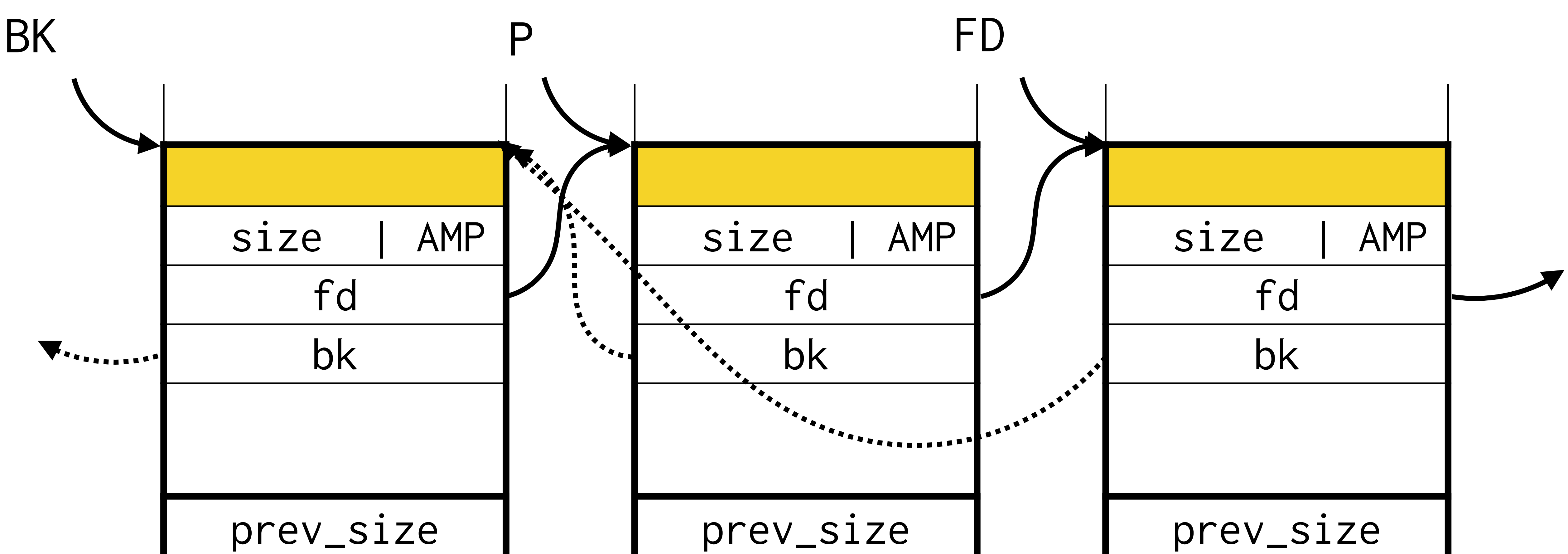
What happens when we allocate?

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



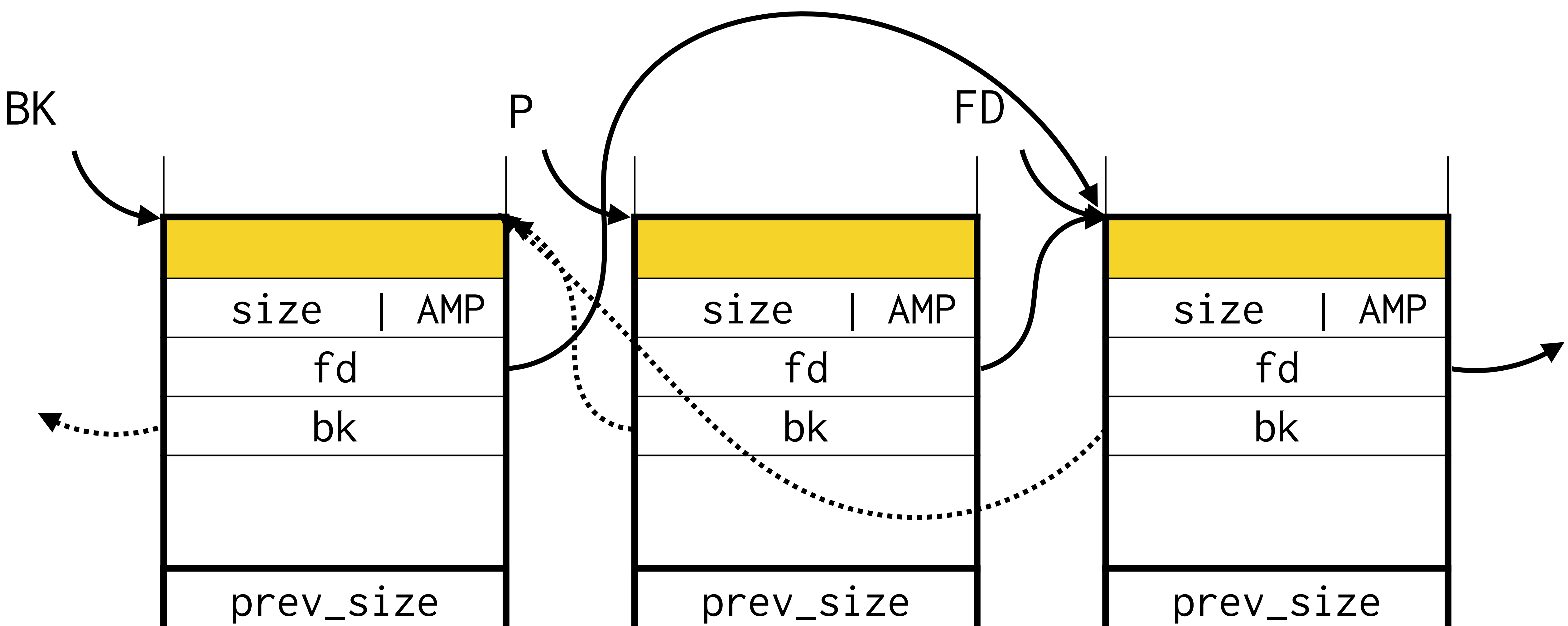
What happens when we allocate?

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



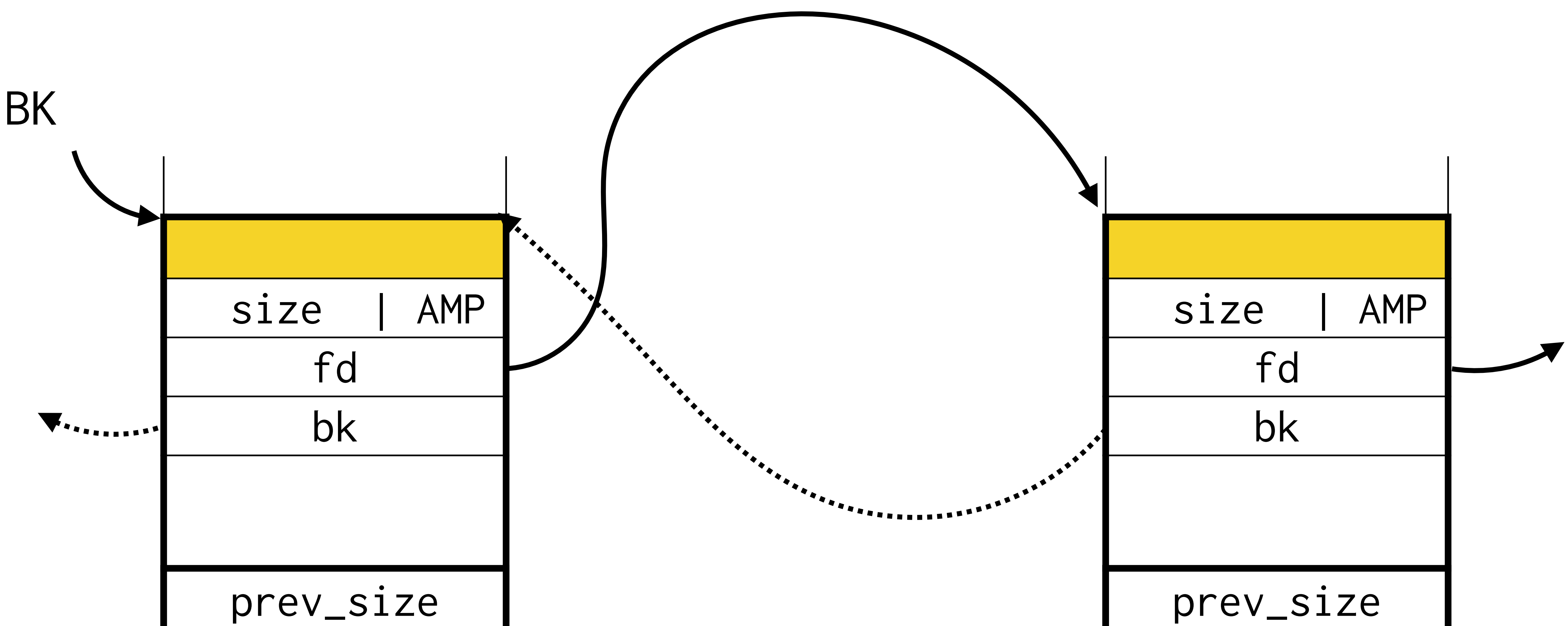
What happens when we allocate?

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



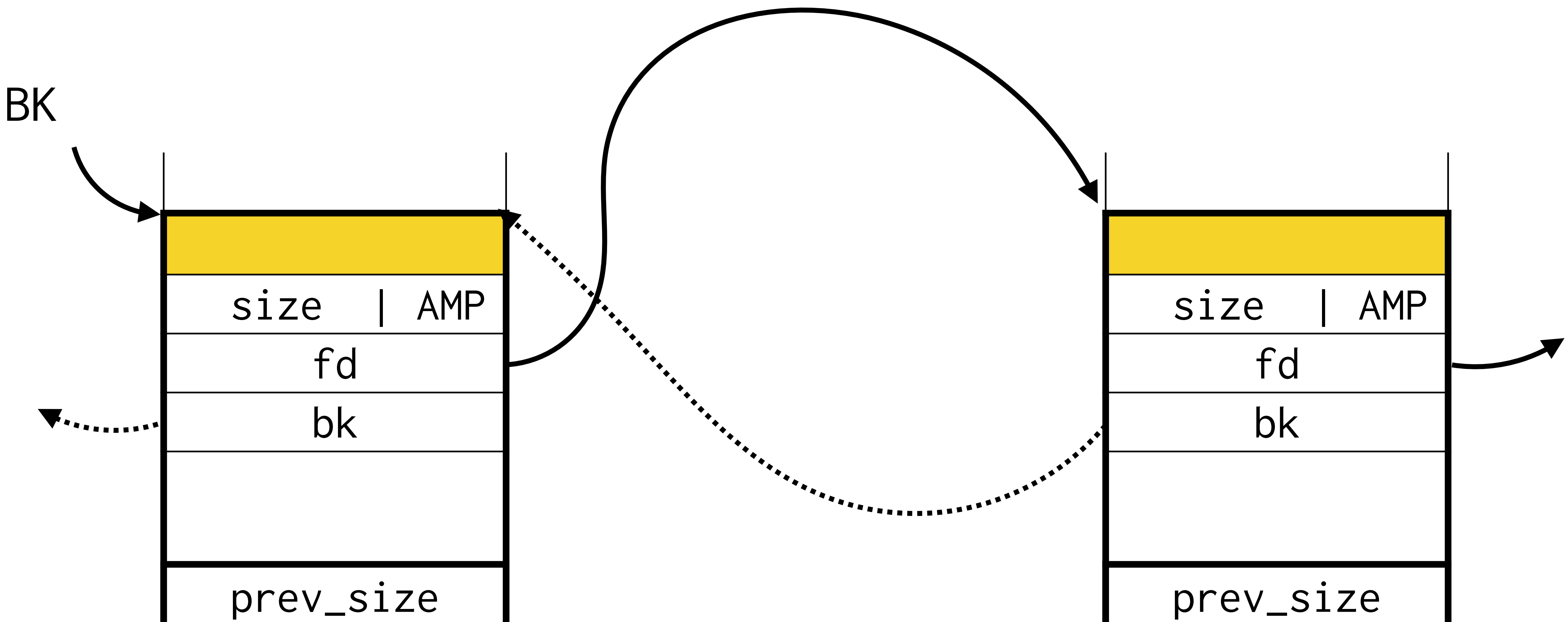
What happens when we allocate?

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



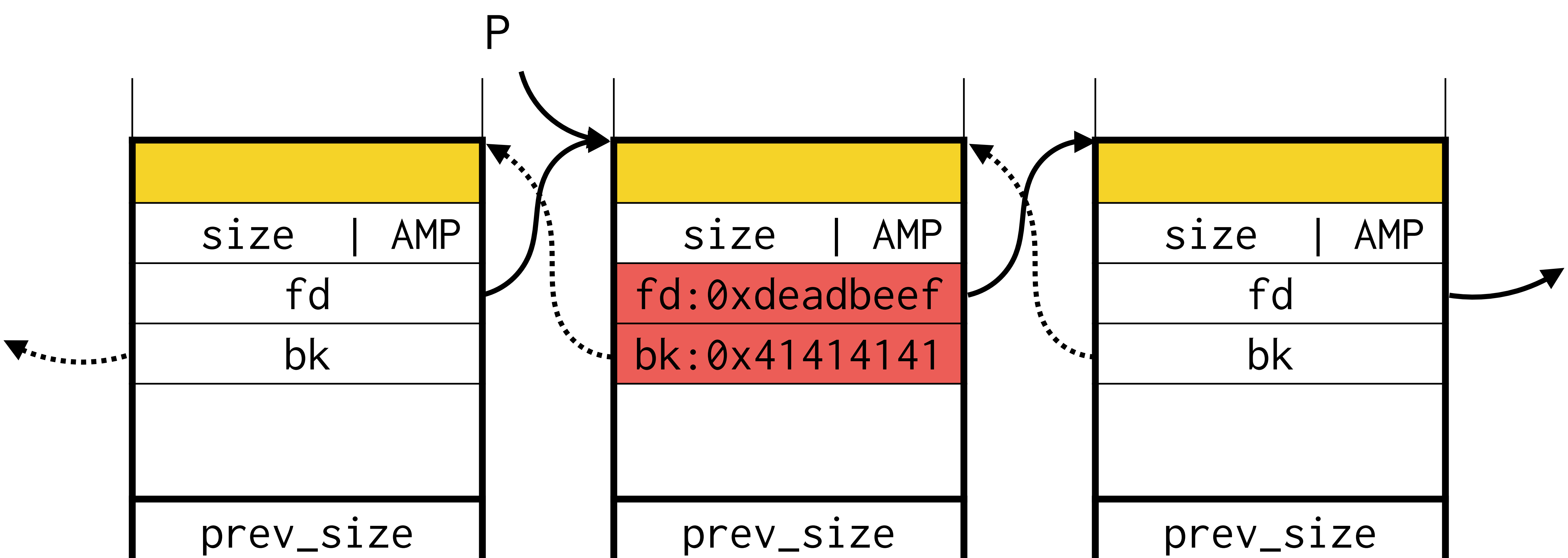
What happens when we allocate?

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



What if the attacker controls chunk?

```
#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

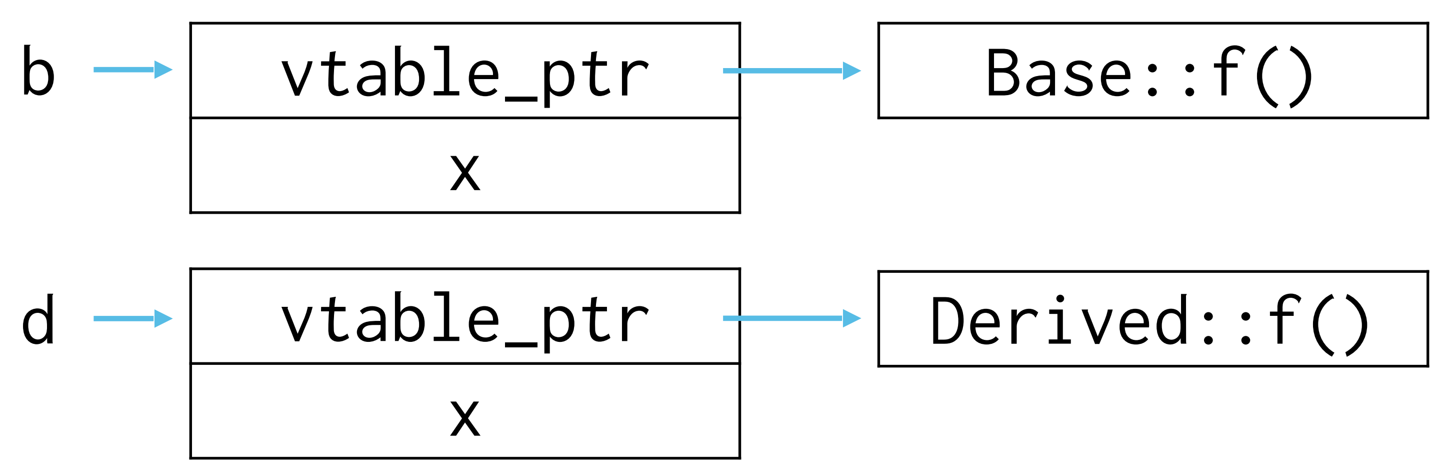
```
class Base {
  public:
   uint32_t x;
   Base(uint32_t x) : x(x) {};
   virtual void f() {
     cout << “base: ” << x;
   }
};

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

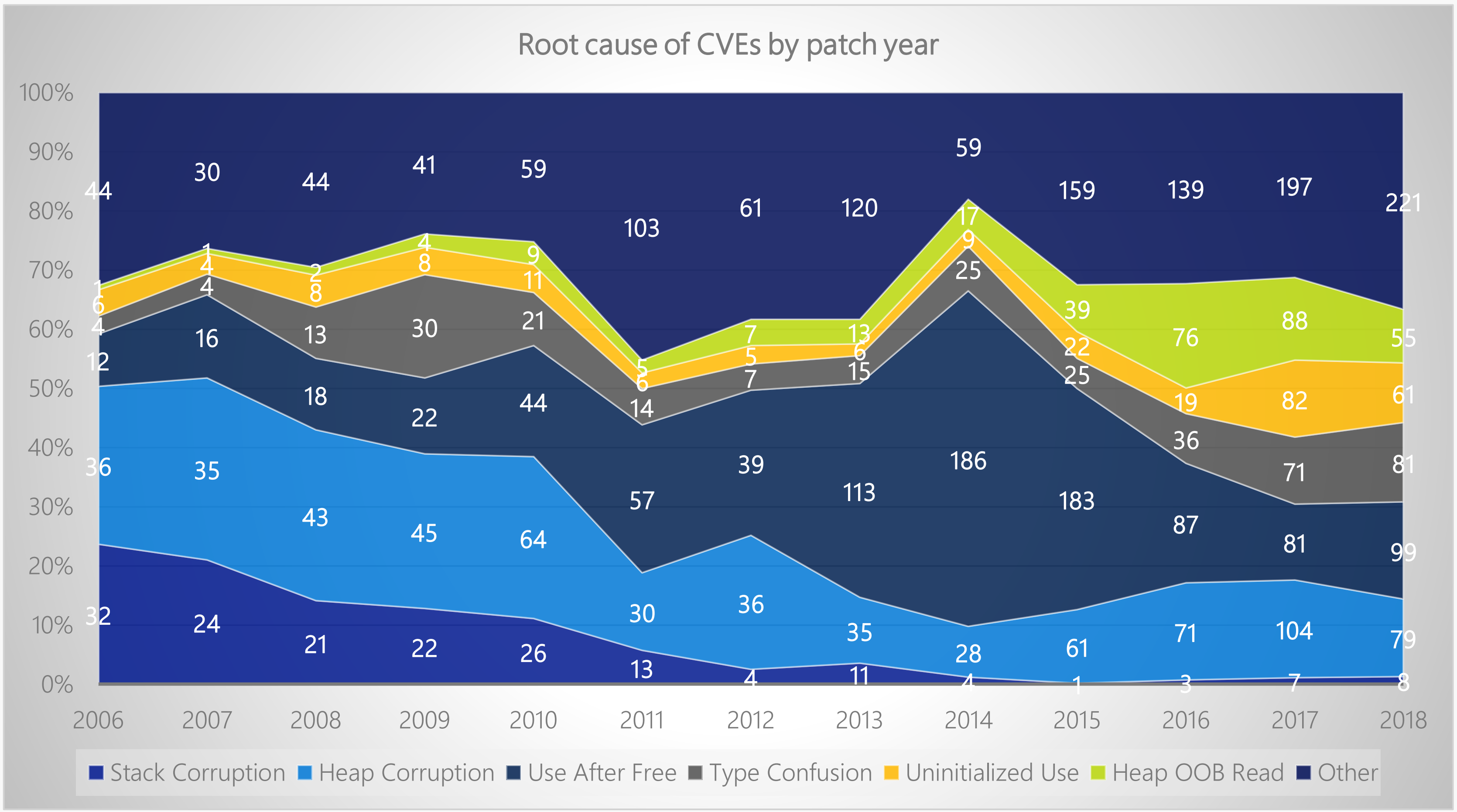
Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape

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)

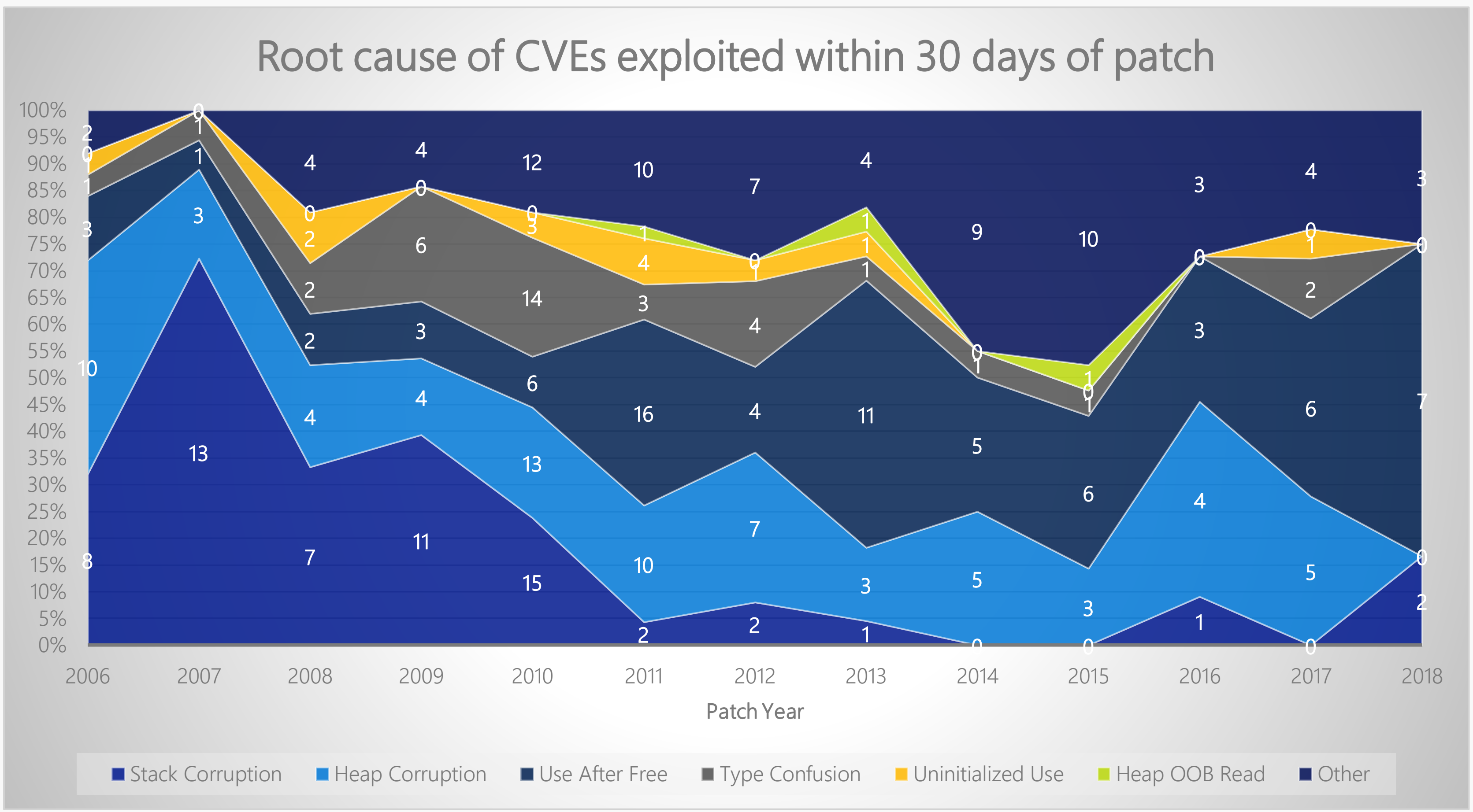
Top root causes since 2016: #1: heap out-of-bounds | #2: use after free | #3: type confusion | #4: uninitialized use

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

Why talk about these attacks?

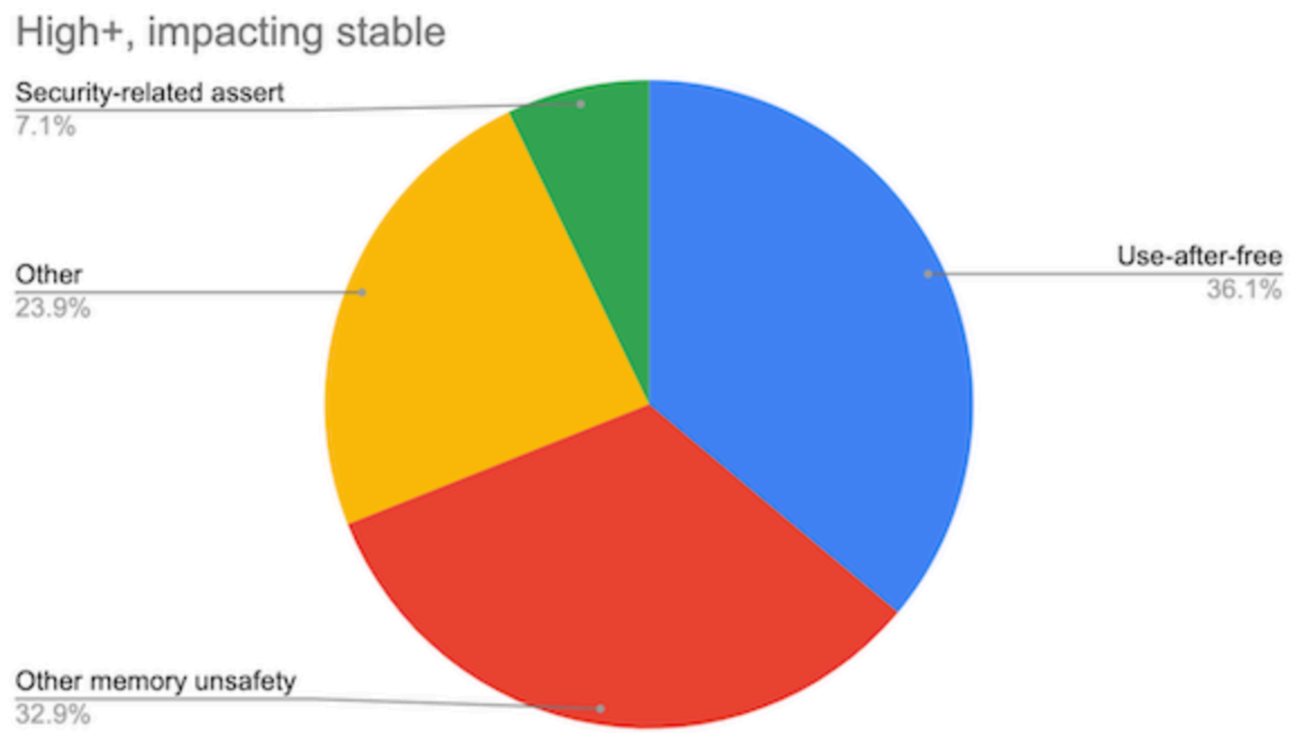
Chromium > Chromium Security >

Memory safety

The Chromium project finds that around 70% of our serious security bugs are memory safety problems. Our next major project is to prevent such bugs at source.

The problem

Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of *those* are use-after-free bugs.



(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

Why talk about these attacks?

CVE-2019-7286: iOS use-after-free in cfprefsd

CVE-2020-15999: FreeType Heap Buffer Overflow in Load_SBit_Png

Entry added September 20, 2021

WebKit

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: Processing maliciously crafted web content may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

Description: A use after free issue was addressed with improved memory management.

CVE-2021-30858: an anonymous researcher

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

Is this program safe?

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

Is this program safe?

```c
void foo(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

```c
#include <string.h>

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

Is this program safe?

```
void foo(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);
```

Is this program safe?

```
void foo(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:

```c
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

Are these exploited in the wild?

Entry added September 20, 2021

CoreGraphics

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: Processing a maliciously crafted PDF may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

Description: An integer overflow was addressed with improved input validation.

CVE-2021-30860: The Citizen Lab

Today

- Heap corruption
- Integer overflows

What does this all tell us?

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