Lec13: Heap Exploitation

Taesoo Kim

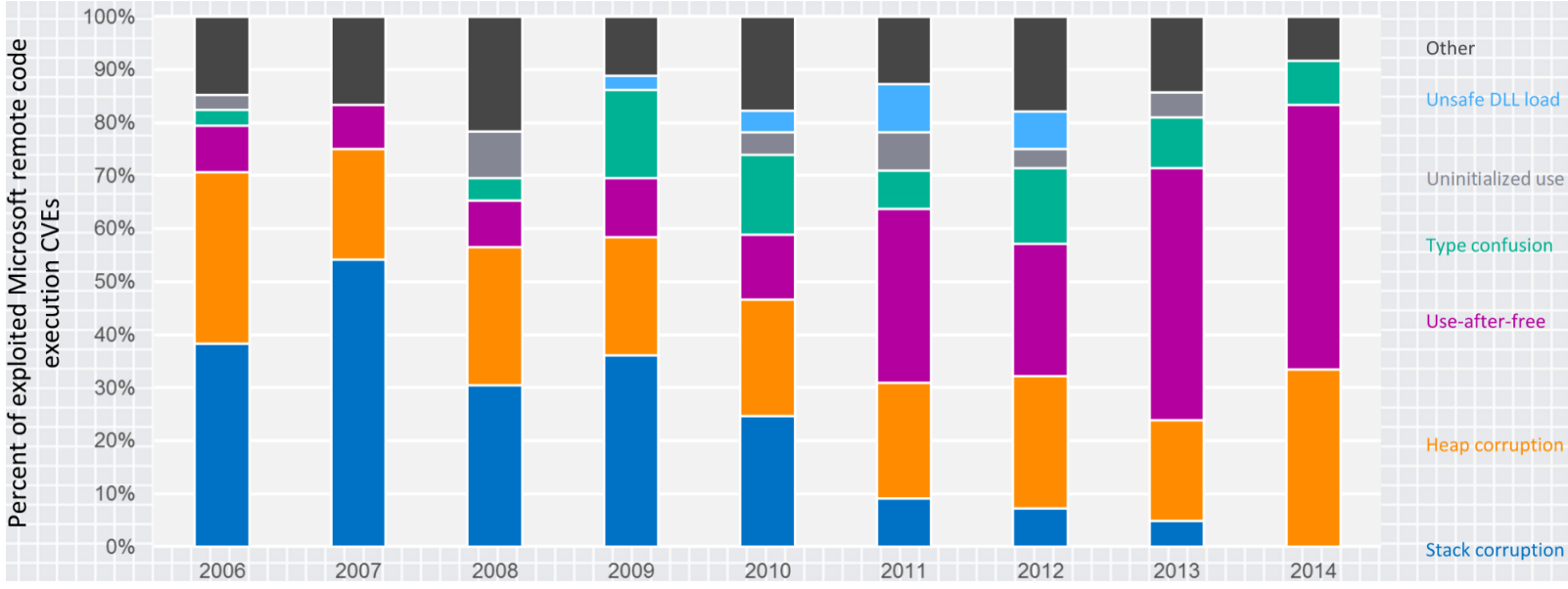
Administrivia

- In-class CTF on Nov 22-23 → Gathering at Klaus 1116 E/W
- We will give each team our feedback by Sat
- Revise/resubmit your challenge by Nov 20 (Wed)

NSA Codebreaker Challenge

- NSA Codebreaker Challenge → Due: Dec 6

Trends of Vulnerability Classes



Ref. Exploitation Trends: From Potential Risk to Actual Risk, RSA 2015

Classifying Heap Vulnerabilities

- Common: buffer overflow/underflow, out-of-bound read
  - *Much prevalent* (i.e., quality, complexity)
  - *Much critical* (i.e., larger attack surface)
- Heap-specific issues:
  - **Use-after-free** (e.g., dangled pointers)
  - Incorrect uses (e.g., double frees)

Simple High-level Interfaces

```
// allocate a memory region (an object)
void *malloc(size_t size);
// free a memory region
void free(void *ptr);

// allocate a memory region for an array
void *calloc(size_t nmemb, size_t size);
// resize/reallocate a memory region
void *realloc(void *ptr, size_t size);

// new Type == malloc(sizeof(Type))
// new Type[size] == malloc(sizeof(Type)*size)
```

CS101: Heap Allocators

```
Q0. ptr = malloc(size); *ptr?
Q1. ptr = malloc(0); ptr == NULL?
Q2. ptr = malloc(-1); ptr == NULL?
Q3. ptr = malloc(size); ptr == NULL but valid? /* vaddr = 0 */

Q4. free(ptr); ptr == NULL?
Q5. free(ptr); *ptr?
Q6. free(NULL)?

Q7. realloc(ptr, size); ptr valid after?
Q8. realloc(NULL, size)?
Q9. ptr = calloc(nmemb, size); *ptr?
```

CS101: Common Goals of Heap Allocators

1. Performance
2. Memory fragmentation
3. Security

```
// either fast, secure, (external) fragmentation!
1. malloc() -> mmap()                          & free() -> unmap()
2. malloc() -> brk()                           & free() -> nop
3. malloc() -> base += size; return base       & free() -> nop
```

Memory Allocators

Allocators	B	I	C	Description (applications)
ptmalloc	✓	✓	✓	A default allocator in Linux
dlmalloc	✓	✓	✓	An allocator that ptmalloc is based on
jemalloc	✓		✓	A default allocator in FreeBSD
tcmalloc	✓	✓	✓	A high-performance allocator from Google
PartitionAlloc	✓		✓	A default allocator in Chromium
libumem	✓		✓	A default allocator in Solaris

Common Design Choices (Security-Related)

1. **Binning**: size-base groups/operations
    - e.g., caching the same size objects together
2. **In-place metadata**: metadata before/after or even inside
    - e.g., putting metadata inside the freed region
3. **Cardinal metadata**: no encoding, direct pointers and sizes
    - e.g., using raw pointers for linked lists

Memory Allocators

Allocators	B	I	C	Description (applications)
ptmalloc	✓	✓	✓	A default allocator in Linux
dlmalloc	✓	✓	✓	An allocator that ptmalloc is based on
jemalloc	✓		✓	A default allocator in FreeBSD
tcmalloc	✓	✓	✓	A high-performance allocator from Google
PartitionAlloc	✓		✓	A default allocator in Chromium
libumem	✓		✓	A default allocator in Solaris

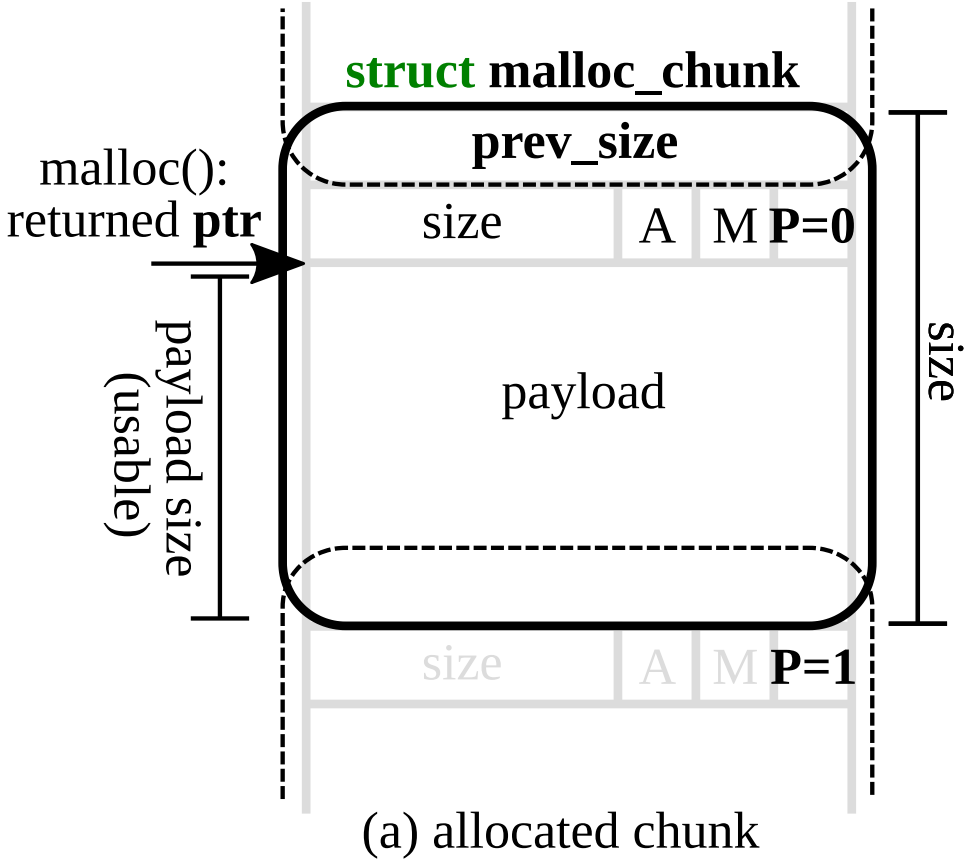
ptmalloc in Linux: Memory Allocation

```
ptr = malloc(size);
```

ptmalloc in Linux: Data Structure

```
struct malloc_chunk {
  // size of "previous" chunk
  //  (only valid when the previous chunk is freed, P=0)
  size_t prev_size;

  // size in bytes (aligned by double words): lower bits
  // indicate various states of the current/previous chunk
  //   A: alloced in a non-main arena
  //   M: mmapped
  //   P: "previous" in use (i.e., P=0 means freed)
  size_t size;

  [...]
};
```

ptmalloc in Linux: Memory Allocation



(a) allocated chunk

Remarks: Memory Allocation

- Given an alloced ptr,
  1. Immediately lookup its size!
  2. Check if the previous object is alloced/freed (P = 0 or 1)
  3. Iterate to the next object (not previous object if alloced)
  4. Check if the next object is alloced/freed (the next, next one's P)

ptmalloc in Linux: Data Structure

```
struct malloc_chunk {
  [...]
  // double links for free chunks in small/large bins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;

  // double links for next larger/smaller size in largebins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

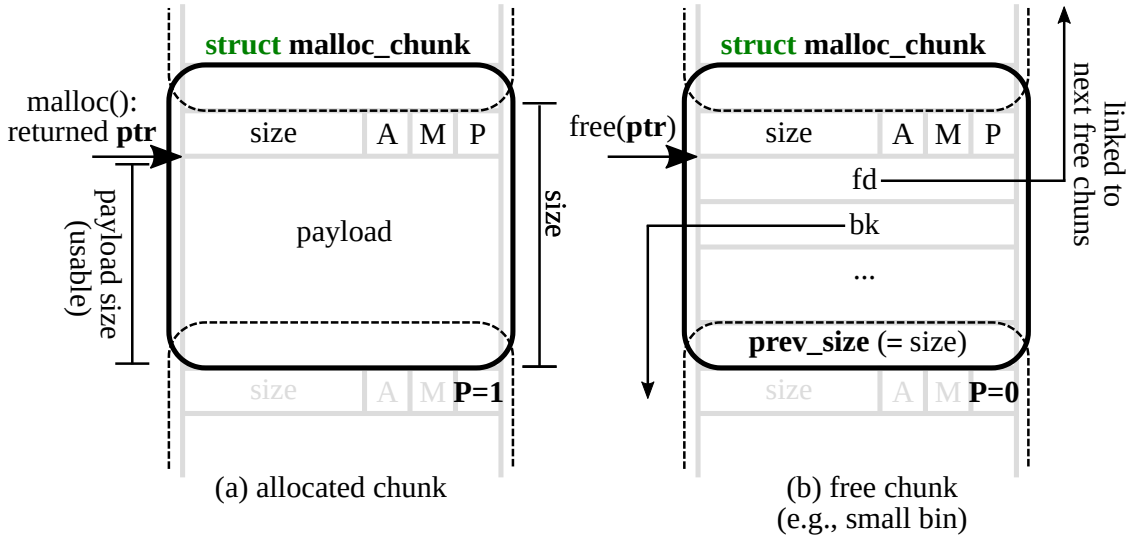
ptmalloc in Linux: Memory Free

Remarks: Memory Free

- Given a free-ed ptr,
    1. All benefits as an alloced ptr (previous remarks)
    2. Iterate to previous/next free objects via fd/bk links
- Invariant: **no two adjacent** free objects (P = 0)
    1. When free(), check if previous/next objects are free and consolidate!

Understanding Modern Heap Allocators

- Maximize memory usage: using free memory regions!
- Data structure to minimize fragmentation (i.e., fd/bk consolidation)
- Data structure to maximize performance (i.e., O(1) in free/malloc)



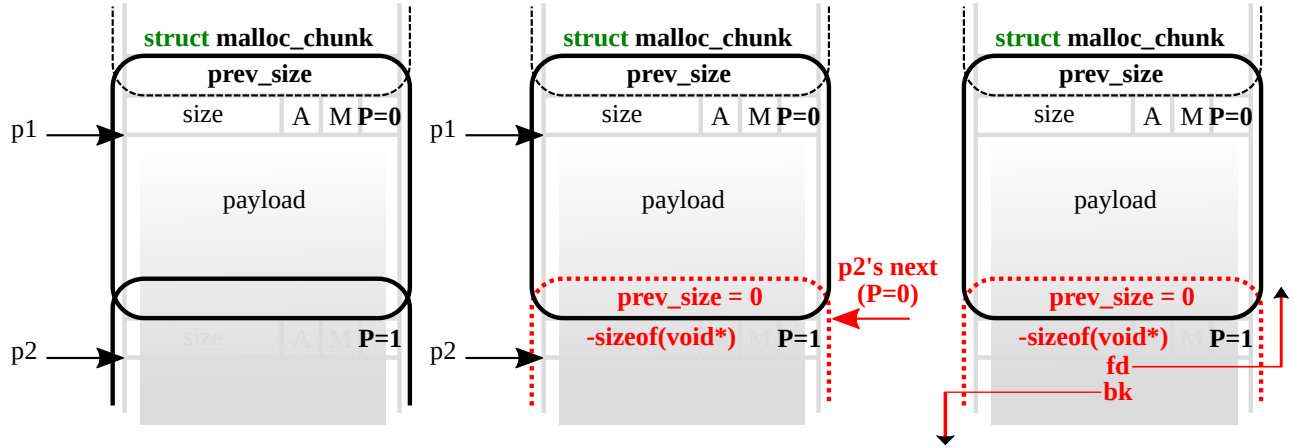
Security Implication of Heap Overflows

- All metadata can be easily modified/crafted!
- Or even new alloc/free objects are created for benefits (and fun!)

```
void *p1 = malloc(sz);
void *p2 = malloc(sz);

/* overflow on p1 */

free(p1);
```

Example: Unsafe Unlink (< glibc 2.3.3)

1. Overwriting to p2's size to -sizeof(void*), treating now as if p2 is free
2. When free(p1), attempt to consolidate it with p2 as p2 is free



Example: Unsafe Unlink (< glibc 2.3.3)

- To consolidate, perform unlink on p2 (removing p2 from the linked list)
- Crafted fd/bk when unlink() result in an arbitrary write!

```
p2's fd = dst - offsetof(struct malloc_chunk, bk);
p2's bk = val;

-> *dst = val (arbitrary write!)
```

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

Example: Mitigation on Unlink (glibc 2.27)

```
#define unlink(AV, P, BK, FD)
    /* (1) checking if size == the next chunk's prev_size */
*   if (chunksize(P) != prev_size(next_chunk(P)))
*     malloc_printerr("corrupted size vs. prev_size");
    FD = P->fd;
    BK = P->bk;
    /* (2) checking if prev/next chunks correctly point to me */
*   if (FD->bk != P || BK->fd != P)
*     malloc_printerr("corrupted double-linked list");
*   else {
      FD->bk = BK;
      BK->fd = FD;
      ...
*    }
```

Heap Exploitation Techniques!

```
Fast bin dup
Fast bin dup into stack
Fast bin dup consolidate
Unsafe unlink
House of spirit
Poison null byte
House of lore
Overlapping chunks 1
Overlapping chunks 2
House of force
Unsorted bin attack
House of einherjar
House of orange
Tcache dup
Tcache house of spirit
Tcache poisoning
Tcache overlapping chunks
*Unsorted bin into stack
*Fast bin into other bin
*Overlapping small chunks
*Unaligned double free
*House of unsorted einherjar
```

NOTE. * are what our group recently found and reported!

Use-after-free

- Simple in concept, but difficult to spot in practice!
- Why is it so critical in terms of security?

```
1 2 3 4 int *ptr = malloc(size);
free(ptr);

*ptr; // BUG. use-after-free!
```

Use-after-free

1. What would be the *ptr? if nothing happened?
2. What if another part of code invoked malloc(size)?

```
1 2 3 4
```

```
int *ptr = malloc(size);
free(ptr);

*ptr; // BUG. use-after-free!
```

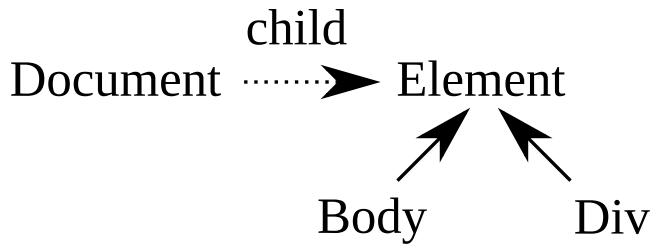
Use-after-free: Security Implication

1. What would be the *ptr? if nothing happened?
    - → Heap pointer leakage (e.g., fd/bk)
2. What if another part of code invoked malloc(size)?
    - → Hijacking function pointers (e.g., handler)

```
1 2 3 4 5 6
```

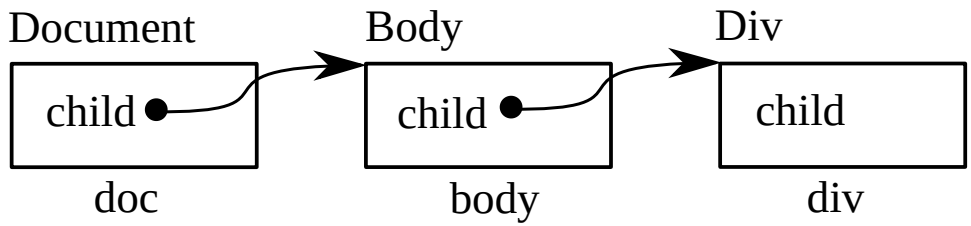
```c
struct msg { void (*handler)(); };

struct msg *ptr = malloc(size);
free(ptr);
// ...?
ptr->handler(); // BUG. use-after-free!
```

Use-after-free with Application Context




```
class Div: Element;
class Body: Element;
class Document { Element* child; };
```

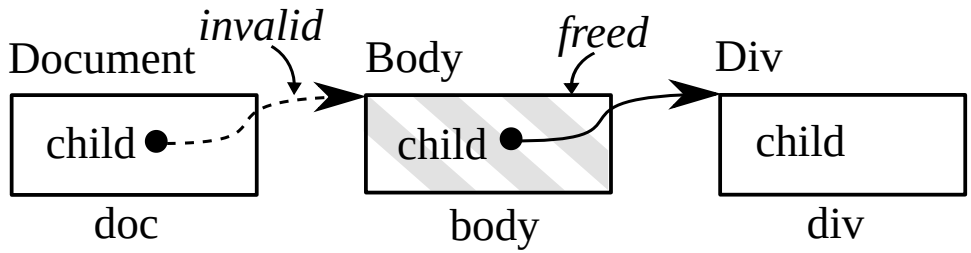
Use-after-free with Application Context




```
1 2 3 4 5 6 7 8
```

```
class Div: Element;
class Body: Element;
class Document { Element* child; };

// (a) memory allocations
Document *doc = new Document();
Body *body = new Body();
Div *div = new Div();
```

Dangled Pointers and Use-after-free




```
1 2 3 4 5 6 7 8 9
10
```

```
// (b) using memory: propagating pointers
doc->child = body;
body->child = div;

// (c) memory free: doc->child is now dangled
delete body;

// (d) use-after-free: dereference the dangled pointer
if (doc->child)
    doc->child->getAlign();
```

Double Free

1. What happen when free two times?
2. What happen for following malloc()s?

```
1 2 3 char *ptr = malloc(size);
      free(ptr);
      free(ptr); // BUG!
```

Binning and Security Implication

- e.g., size-based caching (e.g., fastbin)

```
    (fastbin)
      Bins
sz=16 [ -]--->[fd]--->[fd]-->NULL
sz=24 [ -]--->[fd]--->NULL
sz=32 [ -]--->NULL
       ...
```

Double Free

- Bins after doing free() two times

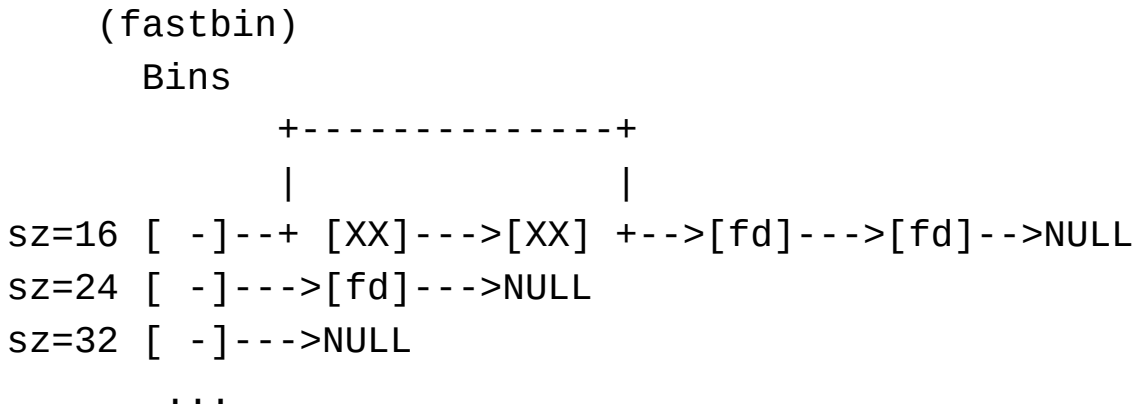
```
1 2 3 char *ptr = malloc(sz=16);
      free(ptr);
      free(ptr); // BUG!
```

```
    (fastbin)
      Bins  ptr      ptr
sz=16 [ -]--->[XX]--->[XX]--->[fd]--->[fd]-->NULL
sz=24 [ -]--->[fd]--->NULL
sz=32 [ -]--->NULL
       ...
```

Double Free: Security Implication

```
1 2 3 4 5 6
```

```c
char *ptr = malloc(sz=16);
free(ptr);
free(ptr); // BUG!

ptr1 = malloc(sz=16) // hijacked!
ptr2 = malloc(sz=16) // hijacked!
```

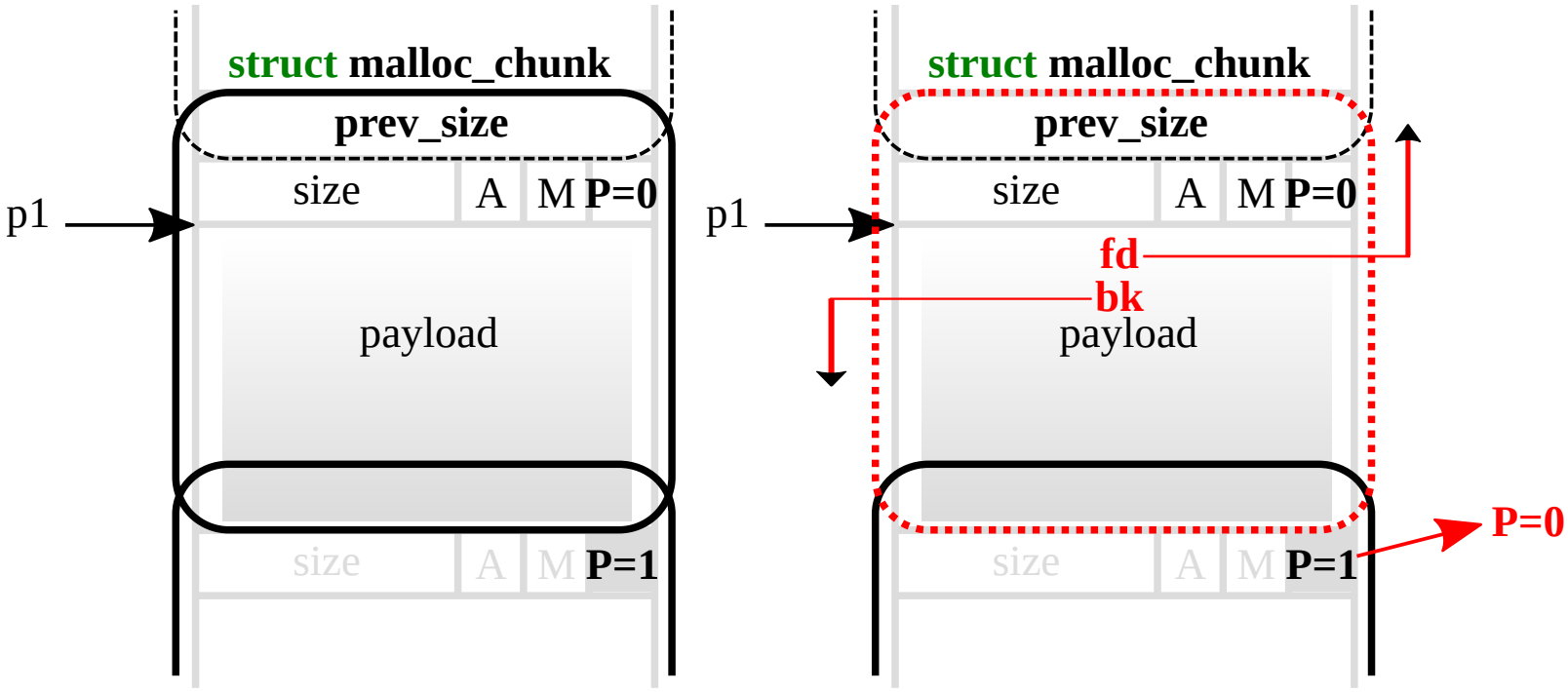


Double Free: Mitigation

- Check if the bin contains the pointer that we'd like to free()

```
1 // @glibc/malloc/malloc.c
2
3     /* Check that the top of the bin is not the record we are going t
4        add (i.e., double free).  */
5     if (__builtin_expect (old == p, 0))
6       malloc_printerr ("double free or corruption (fasttop)");
7     ...
```

Security Implication of off-byte-one (NULL)



Summary

- Two classes of **heap**-related vulnerabilities
  - Traditional: buffer overflow/underflow, out-of-bound read
  - Specific: **use-after-free**, **dangled pointers**, double free
- Understand why they are security critical and non-trivial to eliminate!
- Mitigation approaches taken by allocators

Today's Tutorial

- In-class tutorial:
  - Exploring common techniques
  - Exploiting tcache (simple binning)

```
$ ssh lab09@3.95.14.86
Password: <password>

$ cd tut09-advheap
```

References

- CVE-2014-0160
- CVE-2018-11360
- CVE-2018-17182
- Vudo - An object superstitiously believed to embody magical powers