Lec13: Heap Exploitation

Taesoo Kim

Administrivia

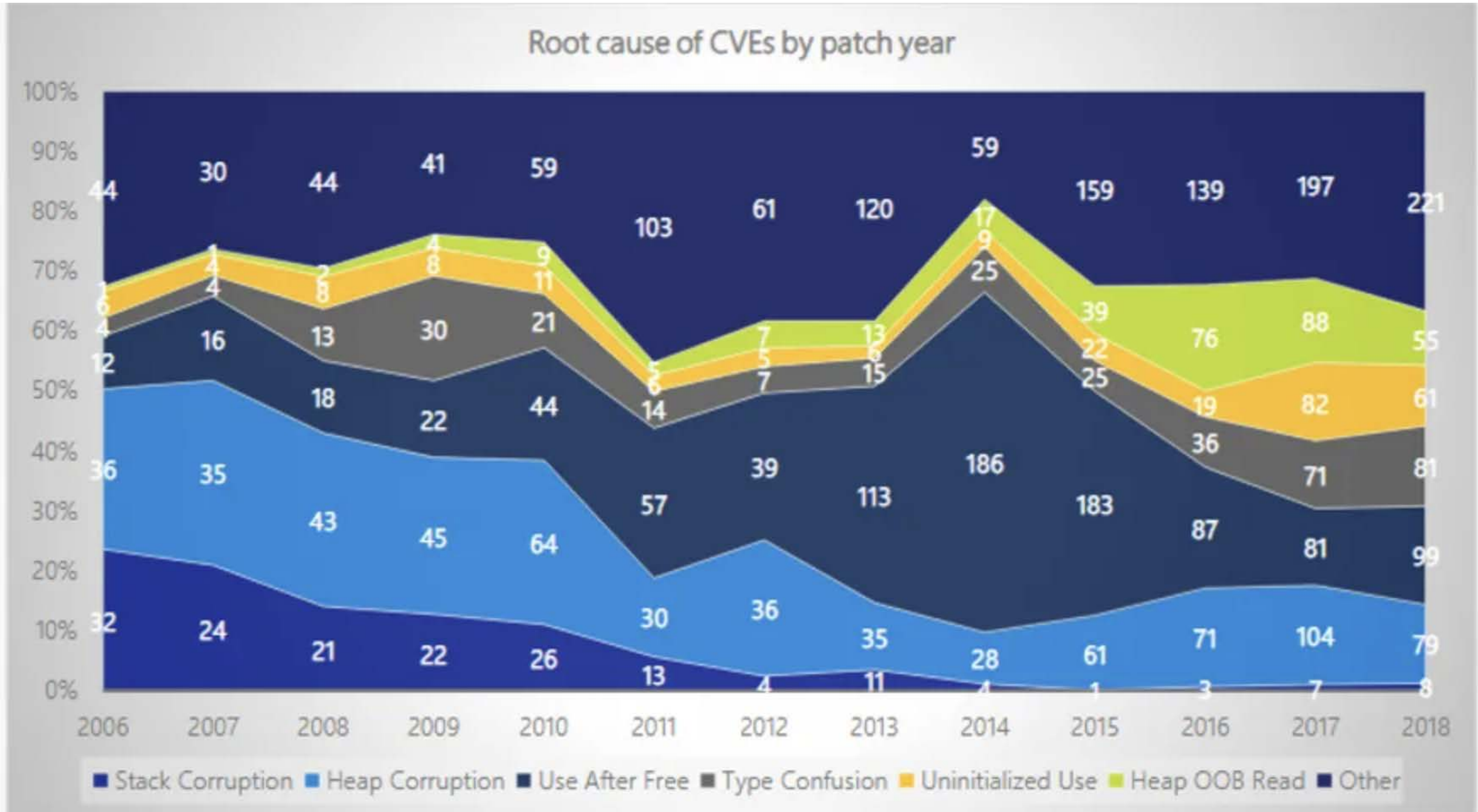
- In-class CTF on Dev 1 → Gathering at Coda 9th, Atrium
- Submit your team's challenge by Nov 27

NSA Codebreaker Challenge

- NSA Codebreaker Challenge → Due: Dec 08

Rank	School	Task 0	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Score
1	Georgia Institute of Technology	196	141	124	49	29	11	8	3	3	3	80,806.00
2	University of North Georgia	215	157	129	61	28	23	17	1	0	0	67,185.00
3	University of California, Santa Cruz	138	116	104	16	11	7	4	3	3	3	57,698.00
4	Dakota State University	158	99	82	22	13	10	8	2	1	1	43,048.00
5	Strayer University	142	65	49	20	14	12	8	1	0	0	30,292.00
6	SANS Technology Institute	63	51	41	19	16	10	6	2	0	0	28,673.00
7	Carnegie Mellon University	78	45	38	11	4	2	1	1	1	1	20,528.00

Trends of Vulnerability Classes



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);

// in C++
// new Type == malloc(sizeof(Type))
// new Type[size] == malloc(sizeof(Type)*size) -- Q. problem?
```

Review: Heap Allocation APIs

```
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. free(ptr); free(ptr)?

Q8. realloc(ptr, size); *ptr?
Q9. realloc(NULL, size)?
Q10. 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

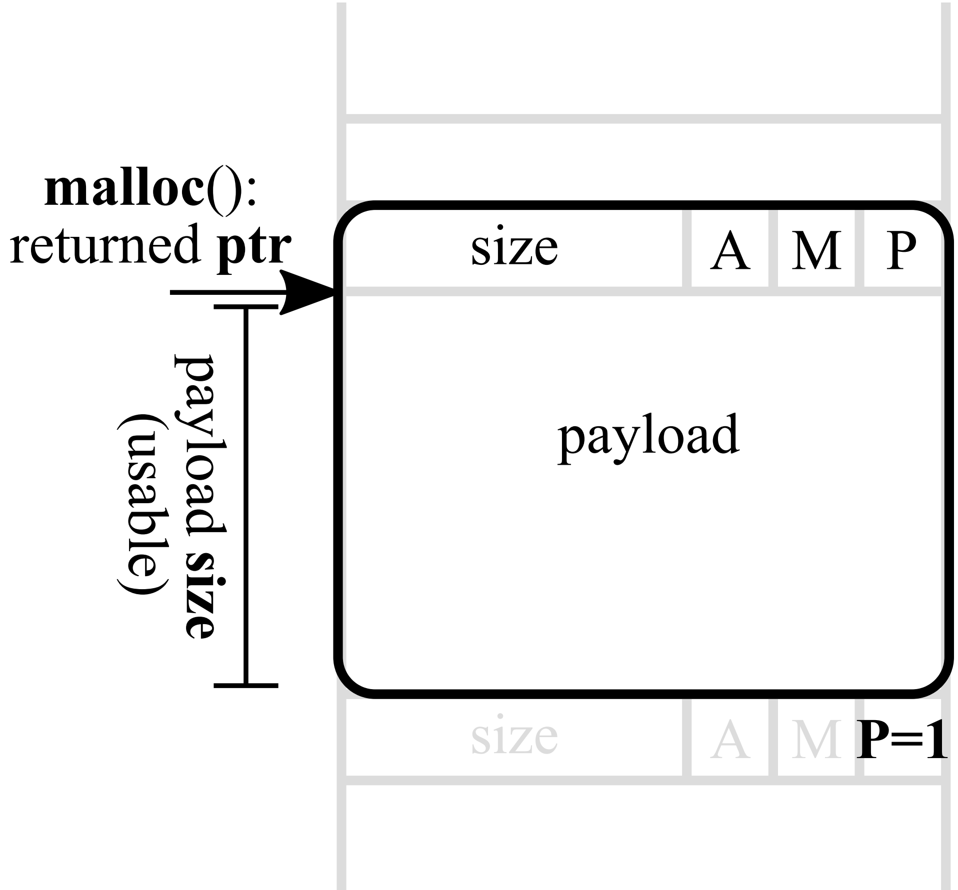
- Recently, Mimalloc by Microsoft and Scudo by LLVM

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

ptmalloc in Linux: Memory Allocation

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



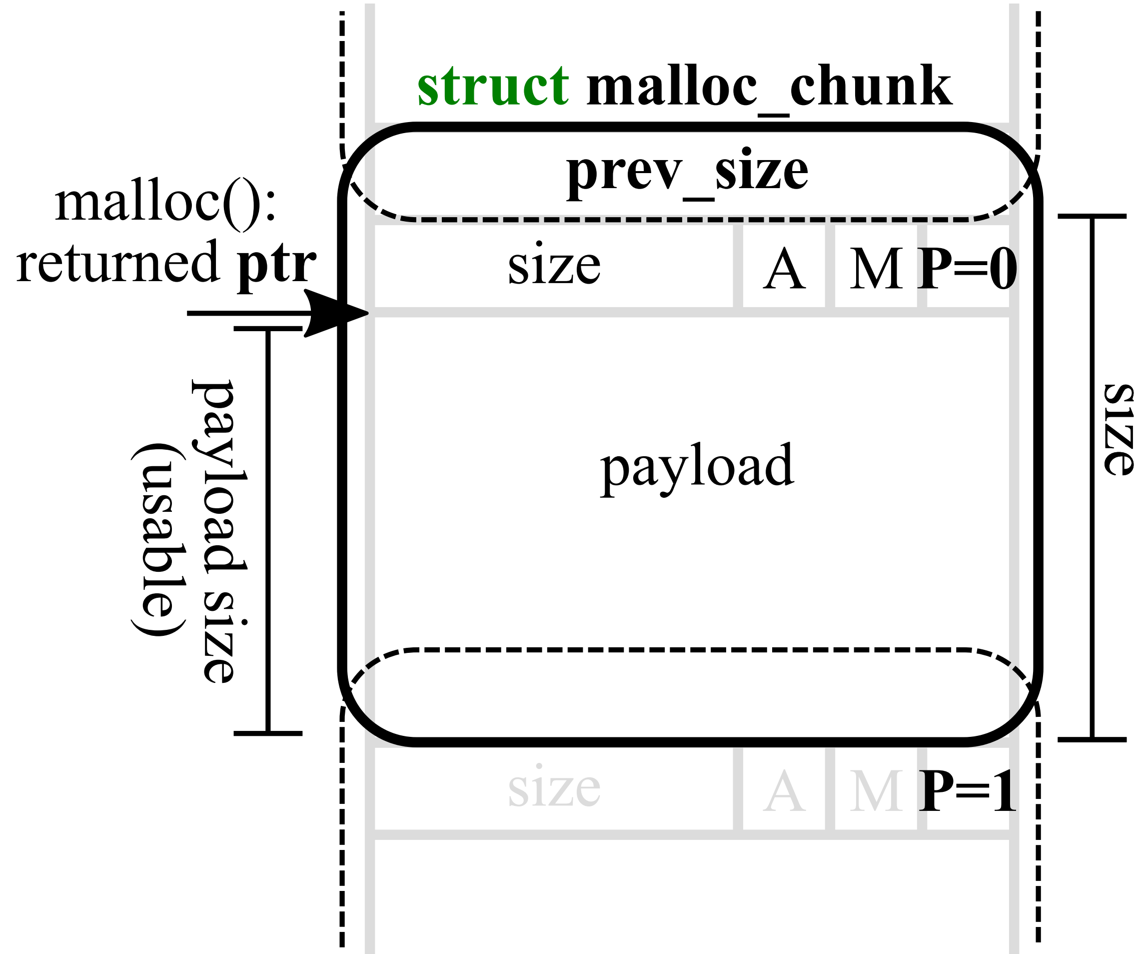
(a) allocated chunk

ptmalloc in Linux: Data Structure (glibc)

```
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;

  [...]
};
```

- Q. How to know if the current chunk is in-use or freed?

ptmalloc in Linux: Memory Allocation



(a) allocated chunk

Remarks: Memory Allocation

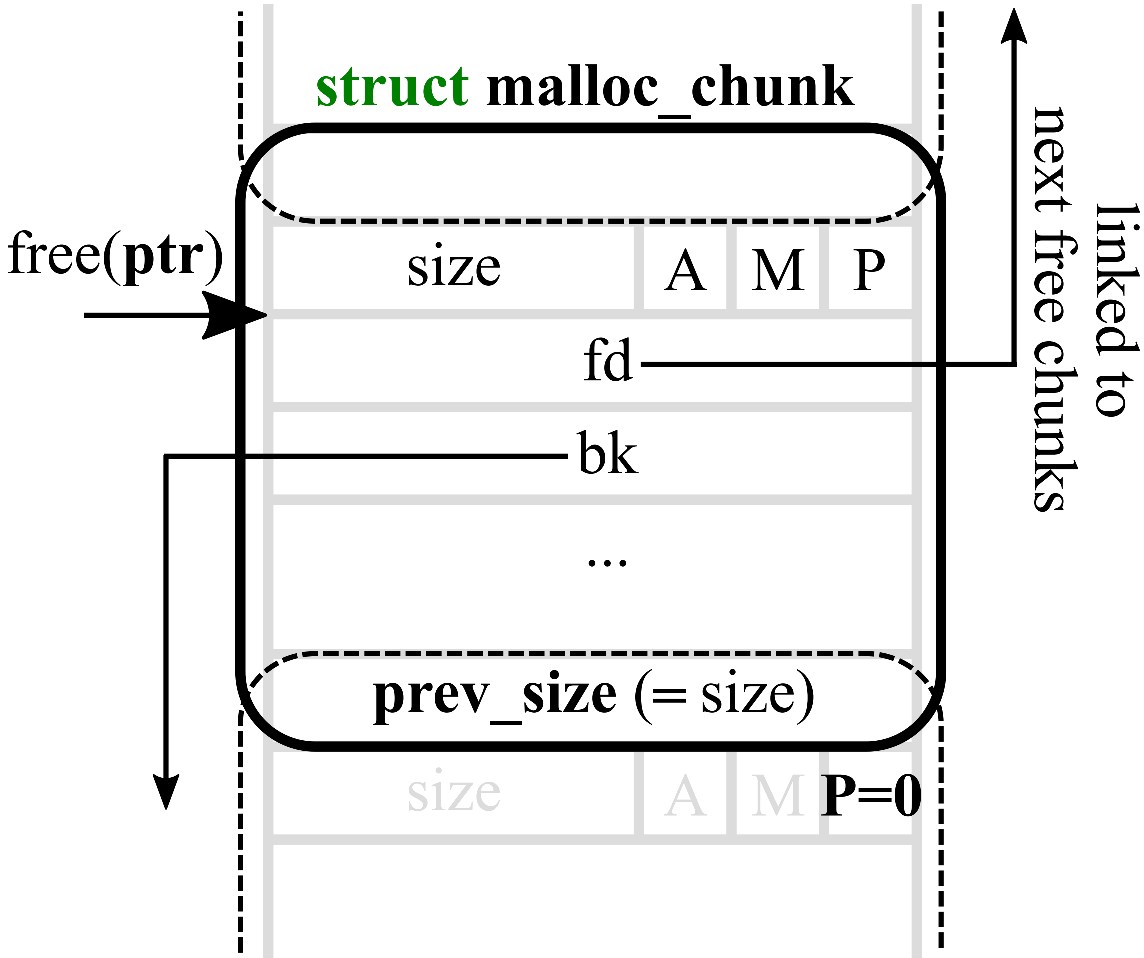
- Given an alloced ptr,
  1. Immediately lookup its size (SIZE)
  2. Check if the **previous** object is alloced/freed (P = 0 or 1)
  3. Check if the **next** object is alloced/freed (Q. how?)
  4. Iterate to the next object (ptr + SIZE)
  5. Iterate to the prev object **if freed** (ptr - PREV_SIZE)
  6. Not possible to iterate to the previous object if allocated

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 rgebins
  //  (only valid when this chunk is freed)
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

- Q. What if we access fd/bk of the allocated chunk?

ptmalloc in Linux: Memory Free



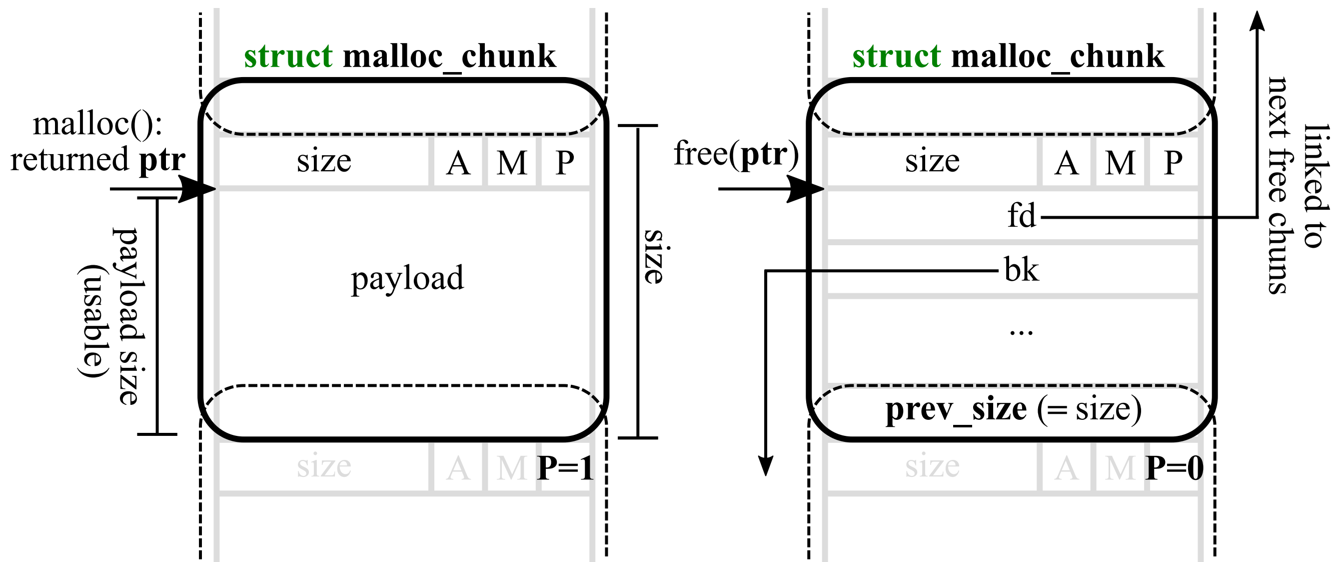
(b) free chunk

Remarks: Memory Free

- Given a free-ed ptr,
  1. All benefits as an alloced ptr (previous remarks)
  2. Iterate b/w **free** objects via fd/bk links
- Invariant: **no two adjacent** free objects
  1. When free() invoked, it is always consolidated to adjacent (i.e., fd/bk) objects!

Understanding Modern Heap Allocators

- Maximize memory usage: reusing 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)



(a) allocated chunk

(b) free chunk
(e.g., small bin)

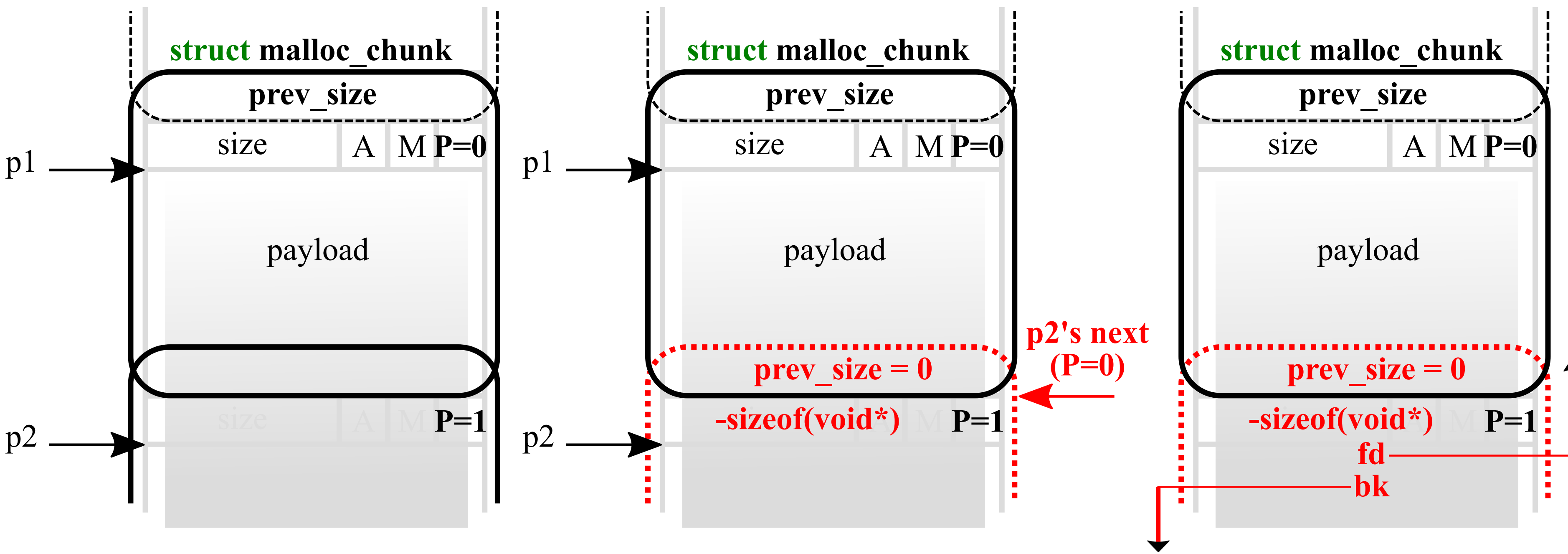
Security Implication of Heap Overflows

- A heap overflow can overwrite the heap metadata
- Incorrect API invocation would destroy the consistency of the metadata
- Allocated/freed objects can be easily crafted for benefits (and fun!)

```c
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!

```
// unlink(P):
  FD = P->fd;
  BK = P->bk;
  FD->bk = BK;    // NOTE. let's abuse this write!
  BK->fd = FD;

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

  => *dst = val (arbitrary write!) // NOTE. any catch?
```

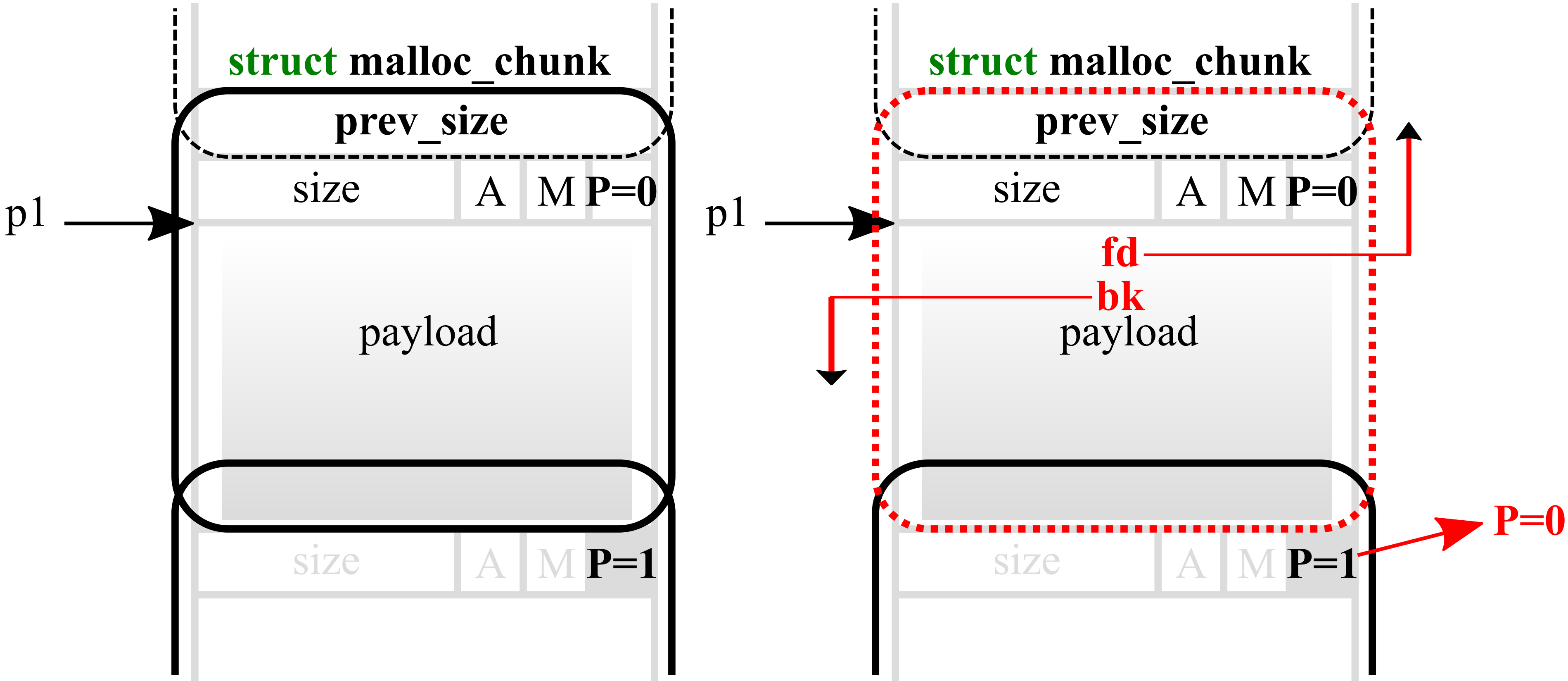
- Q. How to prevent this exploit technique?

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
*/
+     if (FD->bk != P || BK->fd != P)
+       malloc_printerr("corrupted double-linked list");
+     else {
        FD->bk = BK;
        BK->fd = FD;
        ...
+     }
```

- Q. Does it prevent the exploit completely?

Security Implication of NULL Overflow in Heap



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!
- Q. Why is it so critical in terms of security?

```c
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)?

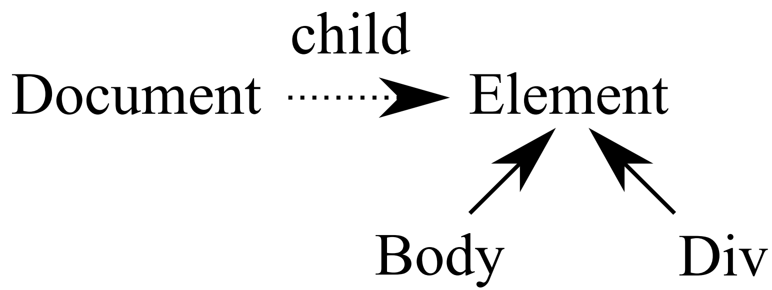
```c
int *ptr = malloc(size);
free(ptr);

*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)

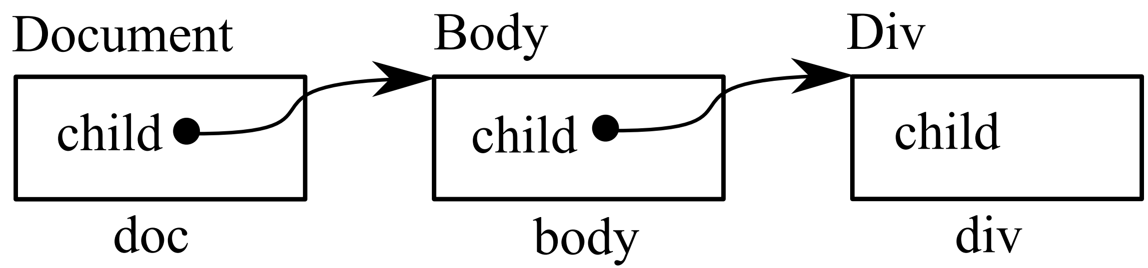
```
struct msg { ... void (*handler)(); ... };

struct msg *ptr = malloc(size);
free(ptr);

// later ...

ptr->handler(); // BUG. use-after-free!
```

Use-after-free with Application Context



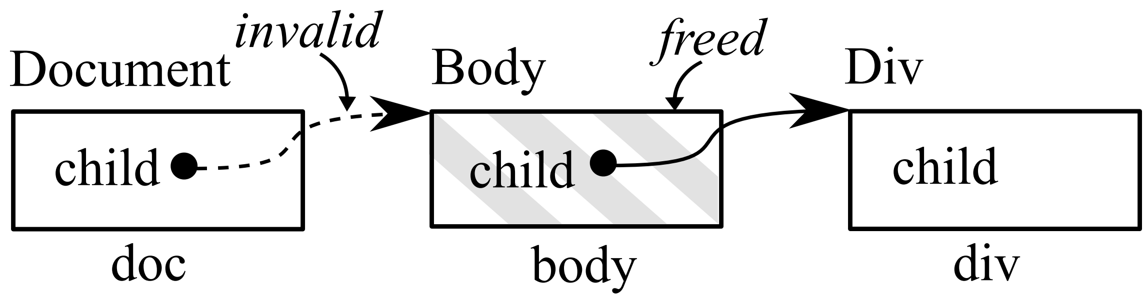

```
class Div: Element;
class Body: Element;
class Document { Element* child; };
```

Use-after-free with Application Context




```
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




```
// (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();
```

API Misuse: Double Free

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

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

malloc(size); // Q. what does it likely return?
malloc(size); // Q. what does it likely return?
```

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

```
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

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

ptr1 = malloc(sz=16) // hijacked!
ptr2 = malloc(sz=16) // hijacked! Q. why problematic?
```

```
    (fastbin)
    Bins           (1)  (2)
          +---------+----+
          |         v    v
sz=16 [ -]--+ [XX]--->[XX] +-->[fd]--->[fd]-->NULL
sz=24 [ -]--->[fd]--->NULL
sz=32 [ -]--->NULL
     ...
```

Double Free: Mitigation

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

```
// @glibc/malloc/malloc.c

    /* Check that the top of the bin is not the record we
e going to
       add (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      malloc_printerr ("double free or corruption
asttop)");
    ...
```

- Q. How to bypass?

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@54.88.195.85
Password: <password>

$ cd tut09-advheap
```

References

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