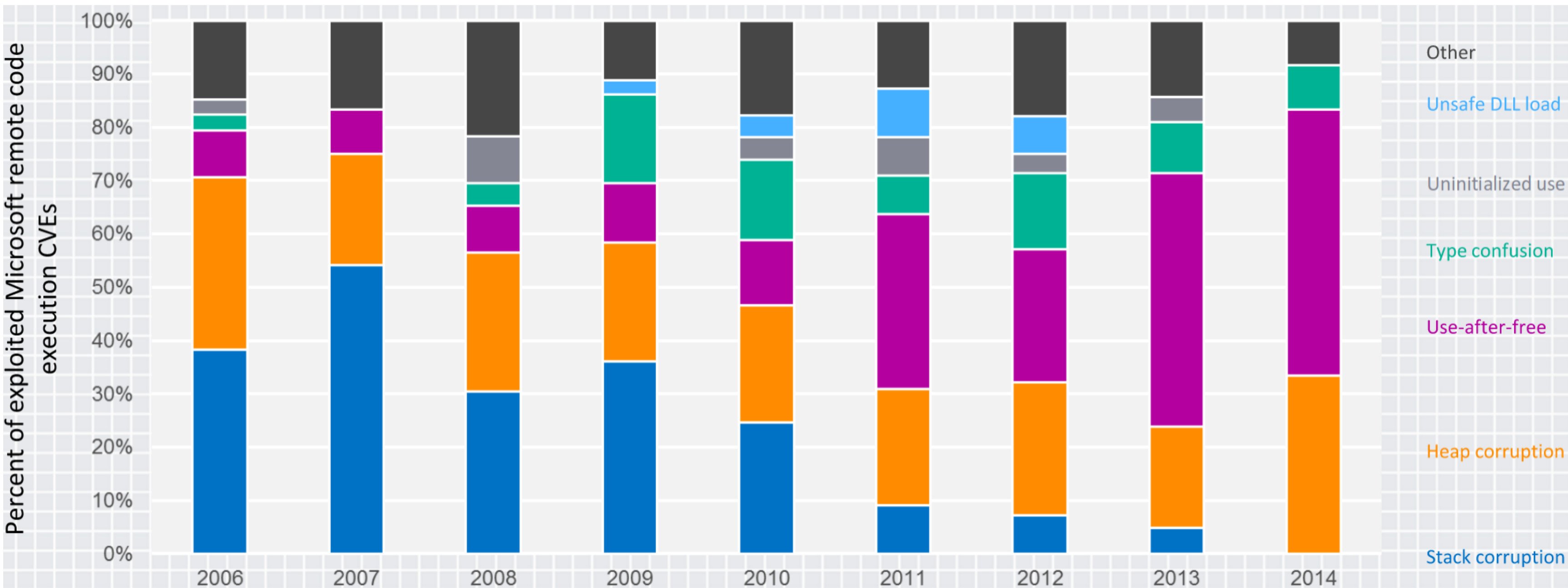
Lec04: Heap-related Vulnerabilities

Taesoo Kim

Goals and Lessons

- Learn about the **heap**-related vulnerabilities
  - Buffer overflow/underflow, out-of-bound read
  - **Use-after-free**, including double frees
- Understand their security implications
- Learn them from the real-world examples

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

```c
// 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?
Q8. ptr = calloc(nmemb, size); *ptr?
```

CS101: Common Goals of Heap Allocators

1. Performance
2. Memory fragmentation
3. (sometime) 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)

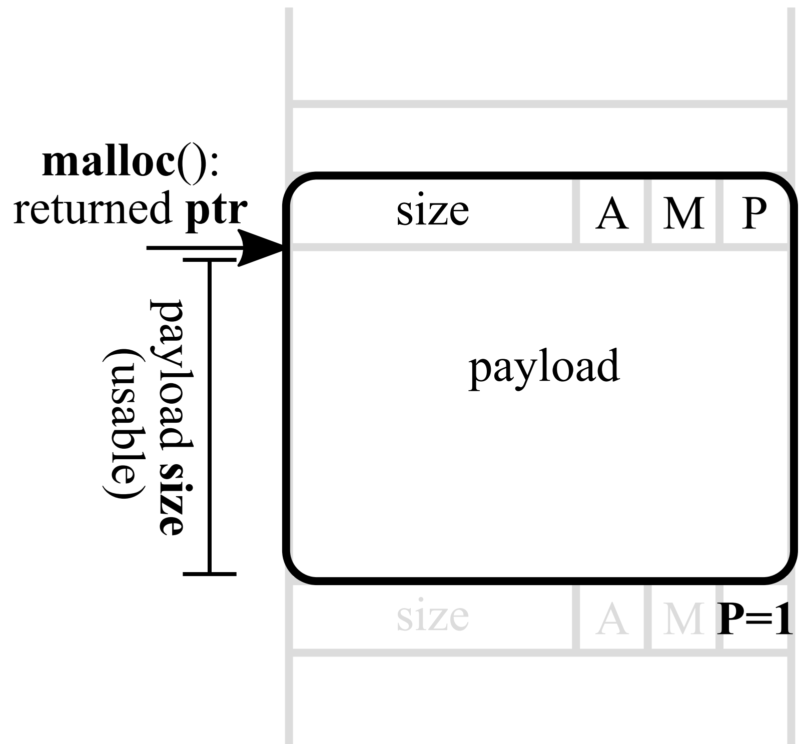
1. **B**inning: size-base groups/operations
   - e.g., caching the same size objects together
2. **I**n-place metadata: metadata before/after or even inside
   - e.g., putting metadata inside the freed region
3. **C**ardinal metadata: no encoding, direct pointers and sizes
   - e.g., using raw pointers 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);
```

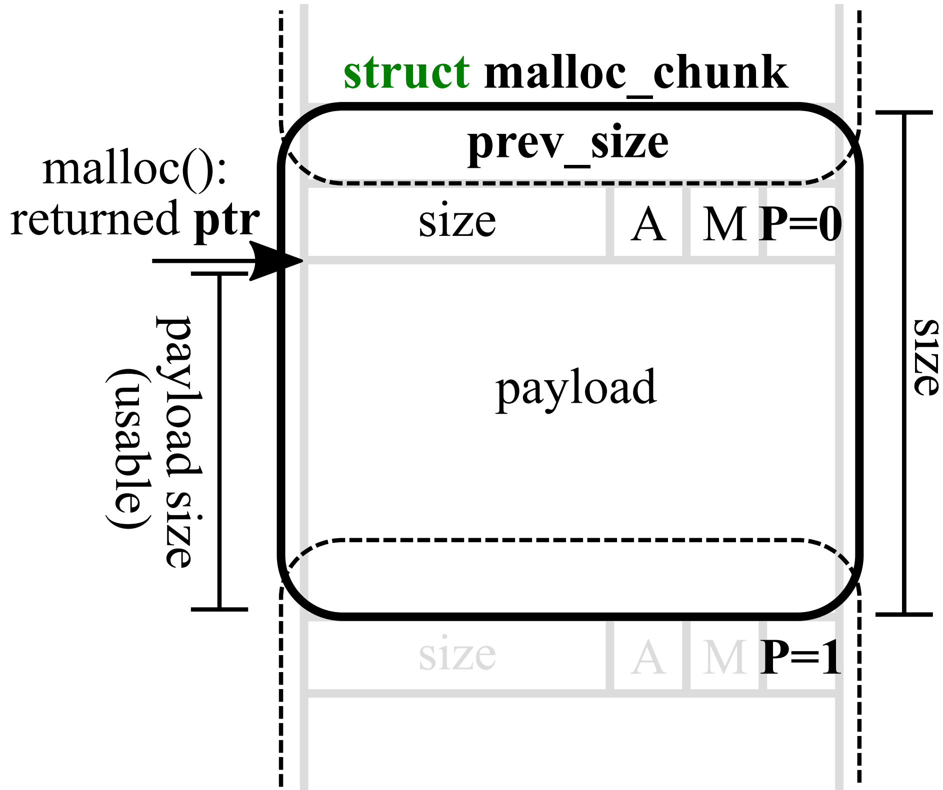


(a) allocated chunk

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

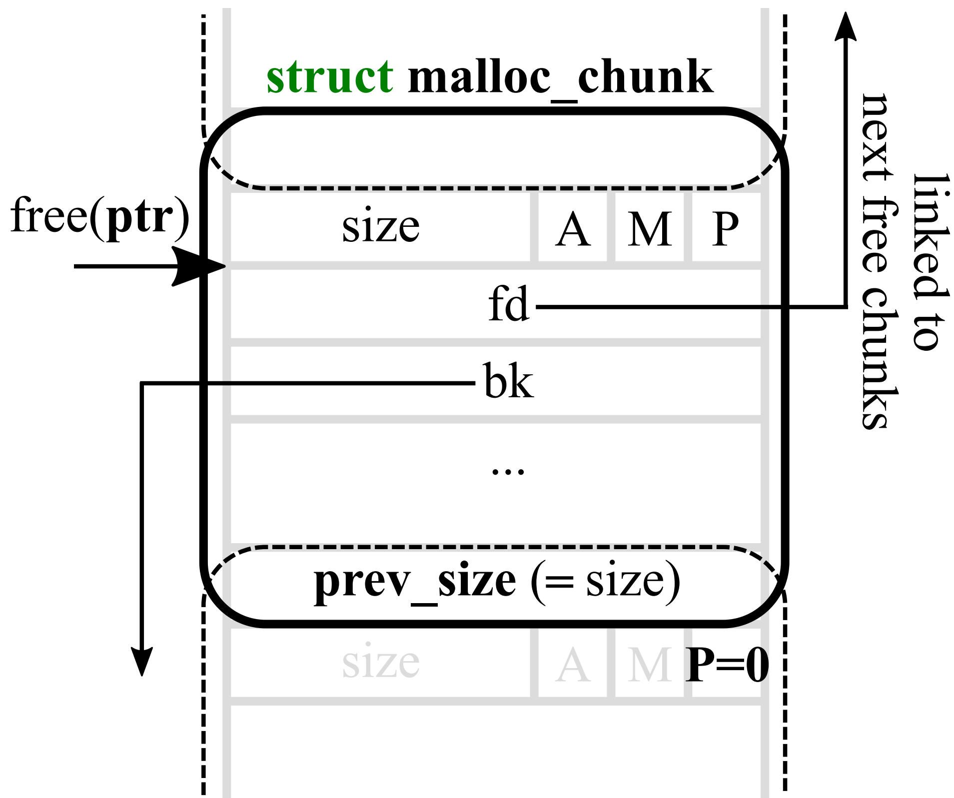
Remarks: Memory Allocation

- Given a 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



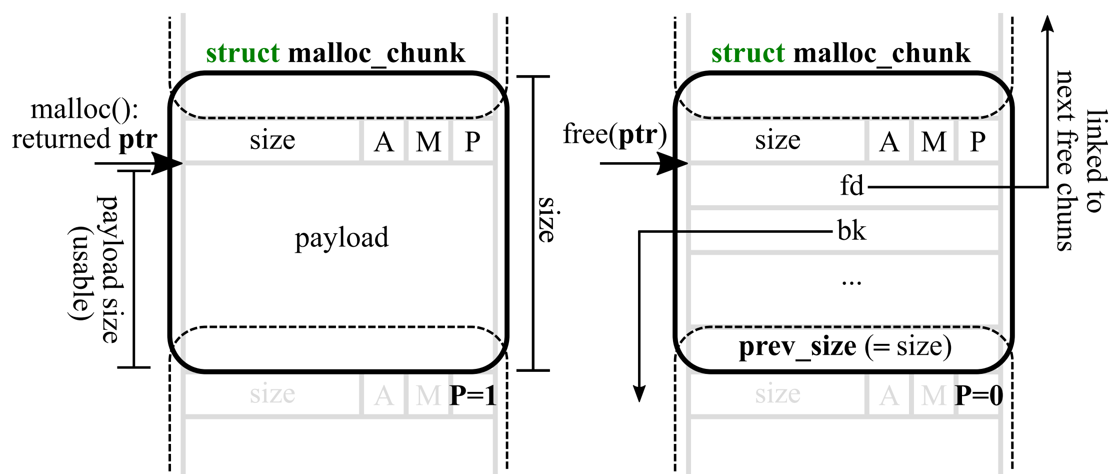
(b) free chunk
(e.g., small bin)

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)



(a) allocated chunk

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

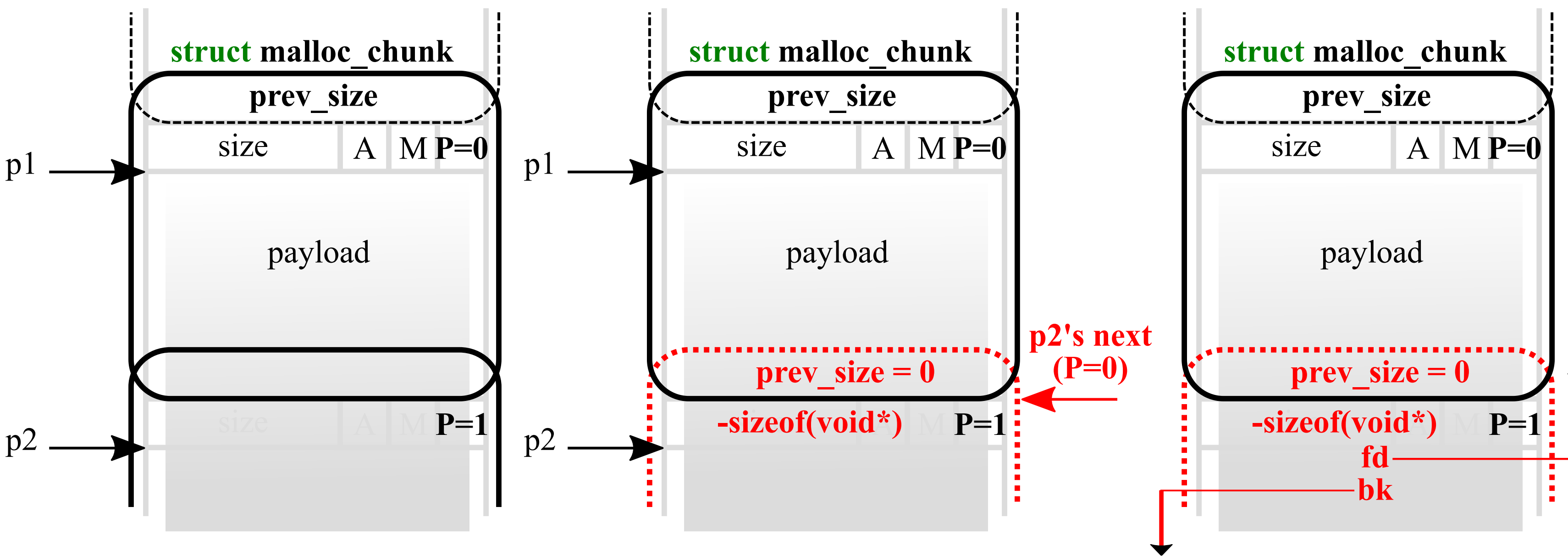
Security Implication of Heap Overflows

- 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 unlike 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?

```
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)?

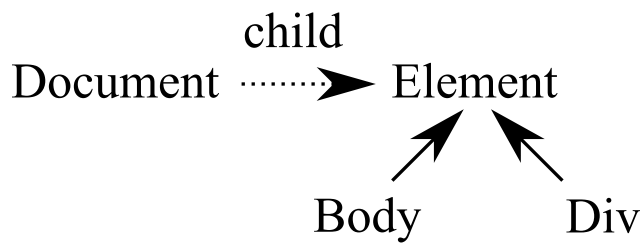
```
int *ptr = malloc(size);
free(ptr);

*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)

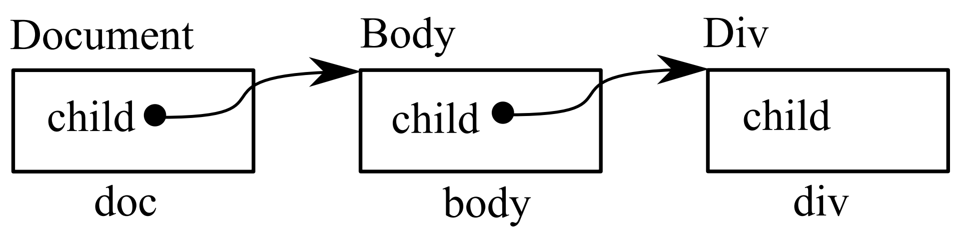
```
struct msg { void (*handler)(); };

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

Use-after-free with Application Context



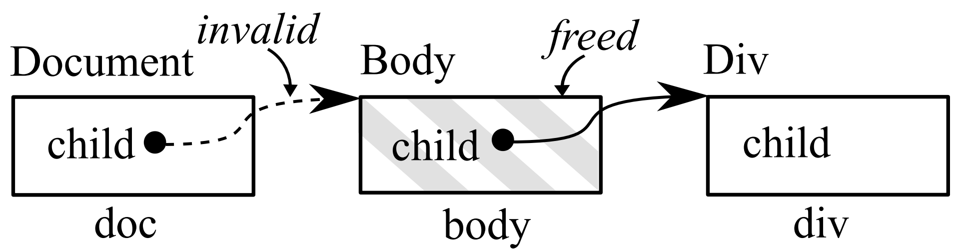

```
class Div: Element;
class Body: Element;
class Document { Element* child; };
```

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();
```

Double Free

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

```c
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

```
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

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

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

```
    (fastbin)
      Bins
            +--------------+
            |              |
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()

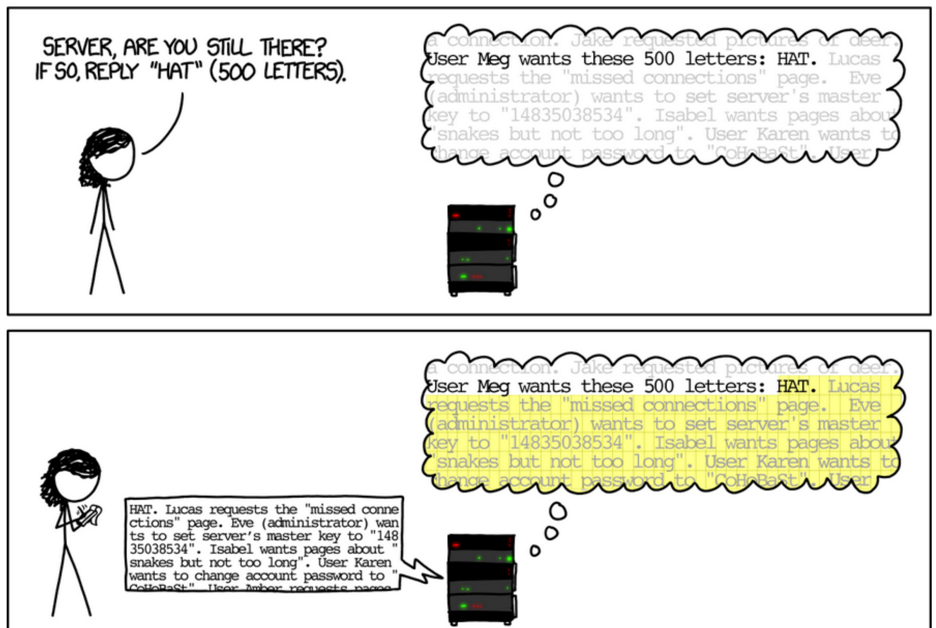
```
// @glibc/malloc/malloc.c

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

Exercise: Real-world Examples

- Ex1. OpenSSL (CVE-2014-0160)
- Ex2. Wireshark (CVE-2018-11360)
- Ex3. Linux vmcache (CVE-2018-17182)*

CVE-2014-0160: OpenSSL, Heartbleed

- Information leakage (i.e., private keys)



Ref. https://xkcd.com/1354/

CVE-2014-0160: OpenSSL, Heartbleed

- "Heartbeat" messages to ensure the connection is alive

```
                 |<---  len''  --->|
-> req: [REQ][len'][payload ....    ]
<- res: [RES][len'][payload ....    ][padding]

  len' == len''?
  what if len' < len''?
  what if len' > len''?
```

CVE-2014-0160: OpenSSL, Heartbleed

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

if (hbtype == TLS1_HB_REQUEST) {
  bp = OPENSSL_malloc(1 + 2 + payload + padding);

  /* Enter response type, length and copy payload */
  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);
```

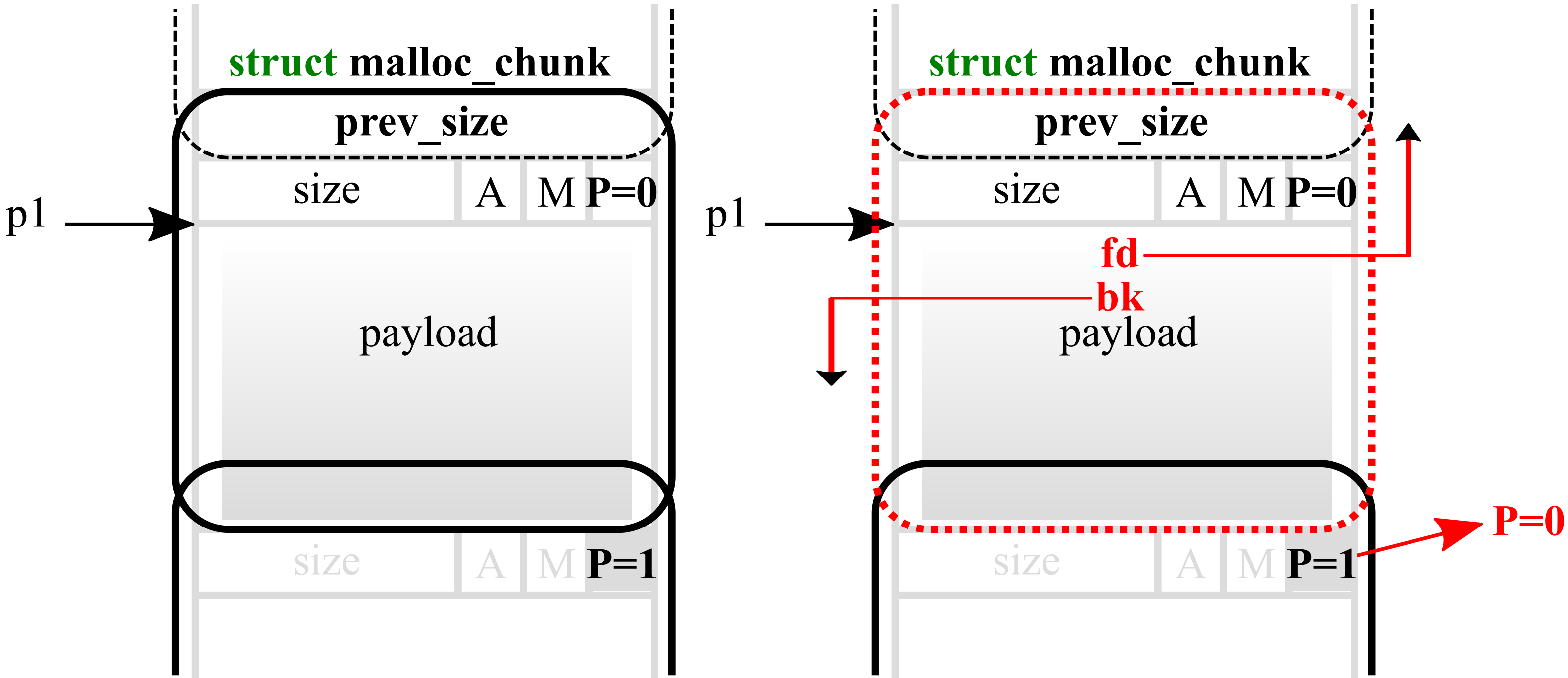
CVE-2014-0160: OpenSSL, Heartbleed

```
    unsigned int payload;
    ...
+   /* Read type and payload length first */
+   if (1 + 2 + 16 > s->s3->rrec.length)
+     return 0; /* silently discard */
+
+   hbtype = *p++;
+   n2s(p, payload);
+   // NOTE. int overflow?
+   if (1 + 2 + payload + 16 > s->s3->rrec.length)
+     return 0; /* silently discard per RFC 6520 sec. 4 */
+   pl = p;
+
    if (hbtype == TLS1_HB_REQUEST) { ... }
```

CVE-2018-11360: Wireshark

```c
// NOTE. What's the semantics of data/len?
void IA5_7BIT_decode(unsigned char *dest,
                     const unsigned char *src, int len) {
  int i, j;
  gunichar buf;

  for (i = 0, j = 0; j < len;  j++) {
    buf = char_def_ia5_alphabet_decode(src[j]);
    i += g_unichar_to_utf8(buf,&(dest[i]));
  }
  dest[i]=0;
  return;
}
```

Security Implication of off-byte-one (NULL)



CVE-2018-17182: Linux vmcache*

- An optimization path for the single thread
- mm→vmcache_seqnum wraps around by another thread
  - → Dangled pointers suddenly become valid!

```
void vmacache_flush_all(struct mm_struct *mm) {
  /* Single threaded tasks need not iterate the entire list
   * process. We can avoid the flushing as well since the mm
   * was increased and don't have to worry about other threa
   * seqnum. Current's flush will occur upon the next lookup
  if (atomic_read(&mm->mm_users) == 1)
    return;
  ...
}
```

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

References

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