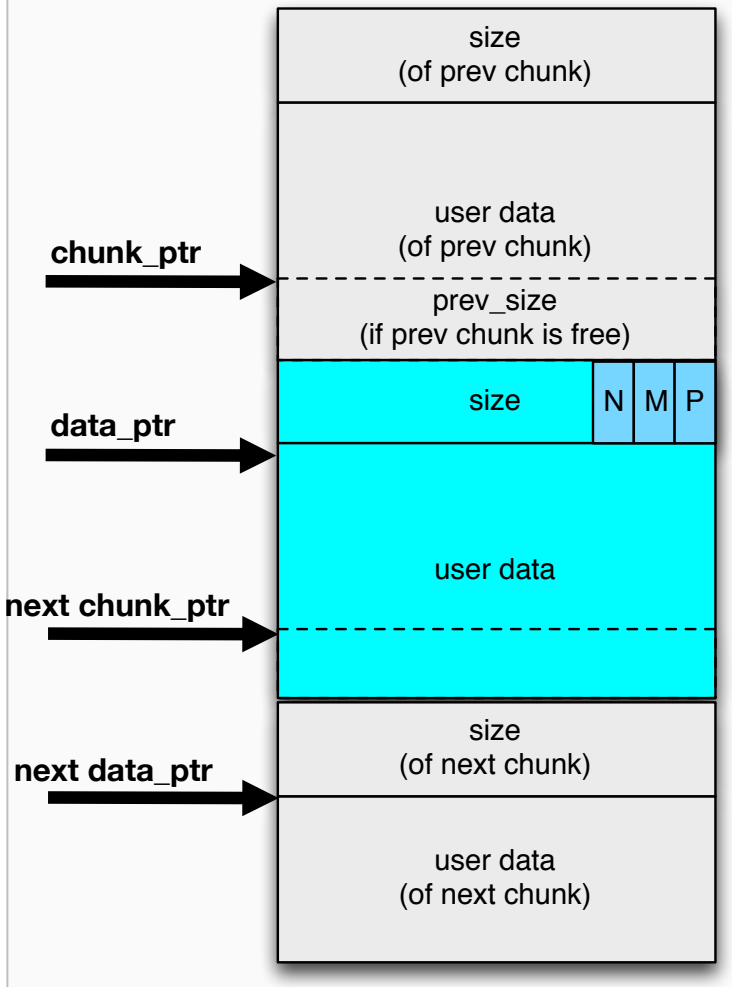
Heap Exploits

Chao Zhang
Tsinghua University

Review: Memory Allocator's Goals

- Efficiency
  - fast to get memory
  - fast to free memory
- Memory fragments
  - very few wasted memory
  - very few fragments

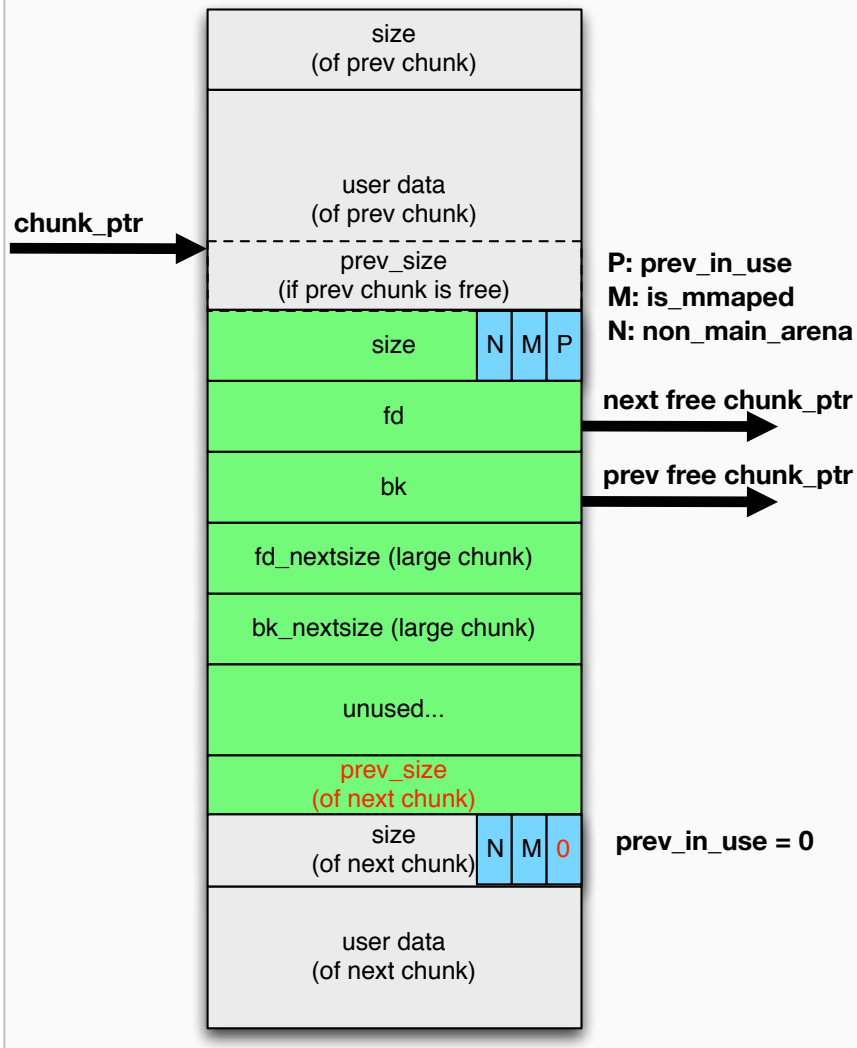
Glibc: Chunk In Use



P: prev_in_use
M: is_mmaped
N: non_main_arena

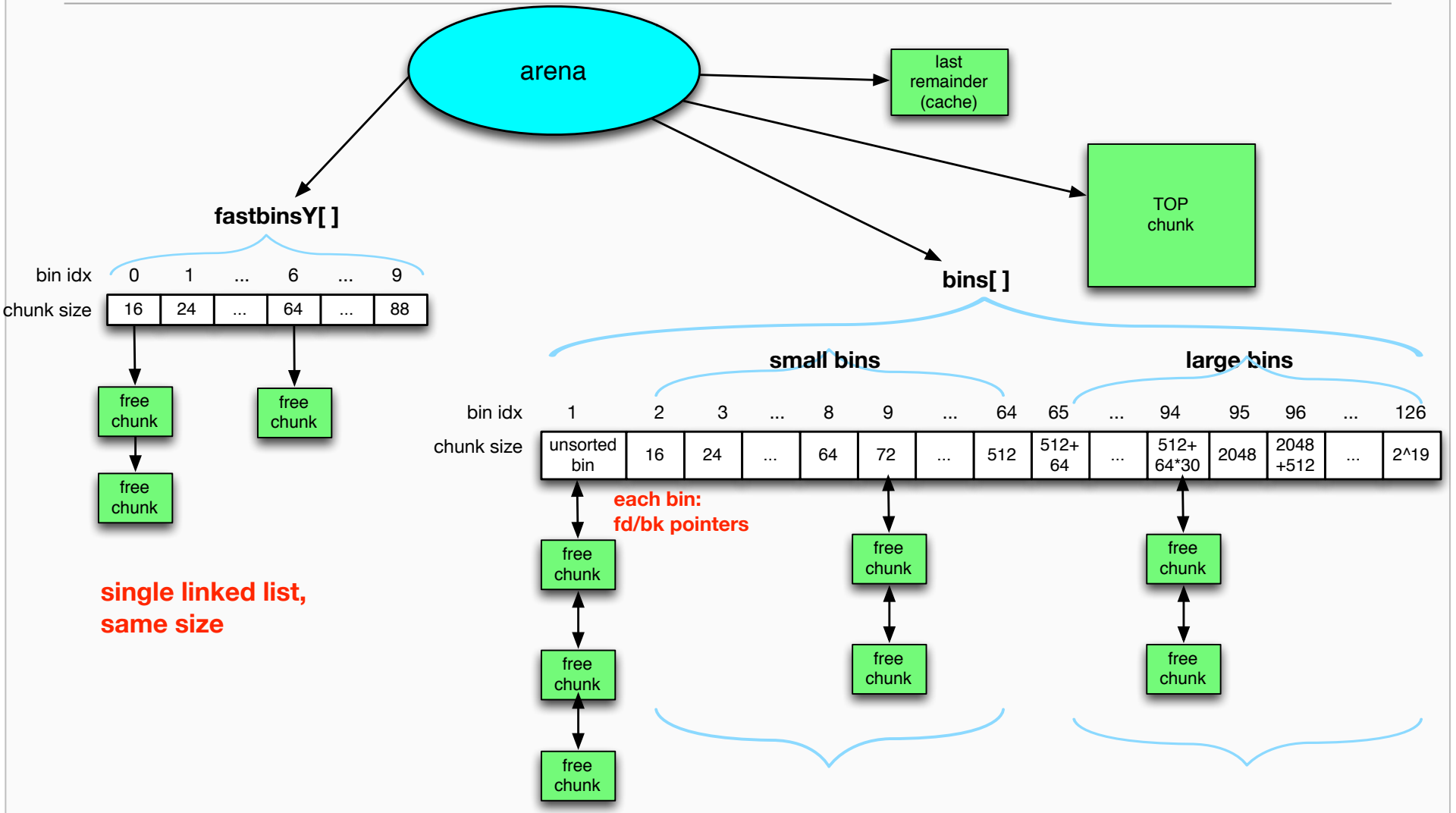
- prev_size field
  - belongs to previous chunk
  - is set when previous chunk is free *(and not in fastbins)*
- prev_in_use
  - is cleared when previous chunk is free *(and not in fastbins)*
- is_mmaped
  - this memory is from mmap()
- non_main_arena
  - this chunk is allocated by non-main thread

Glibc: Chunk Freed

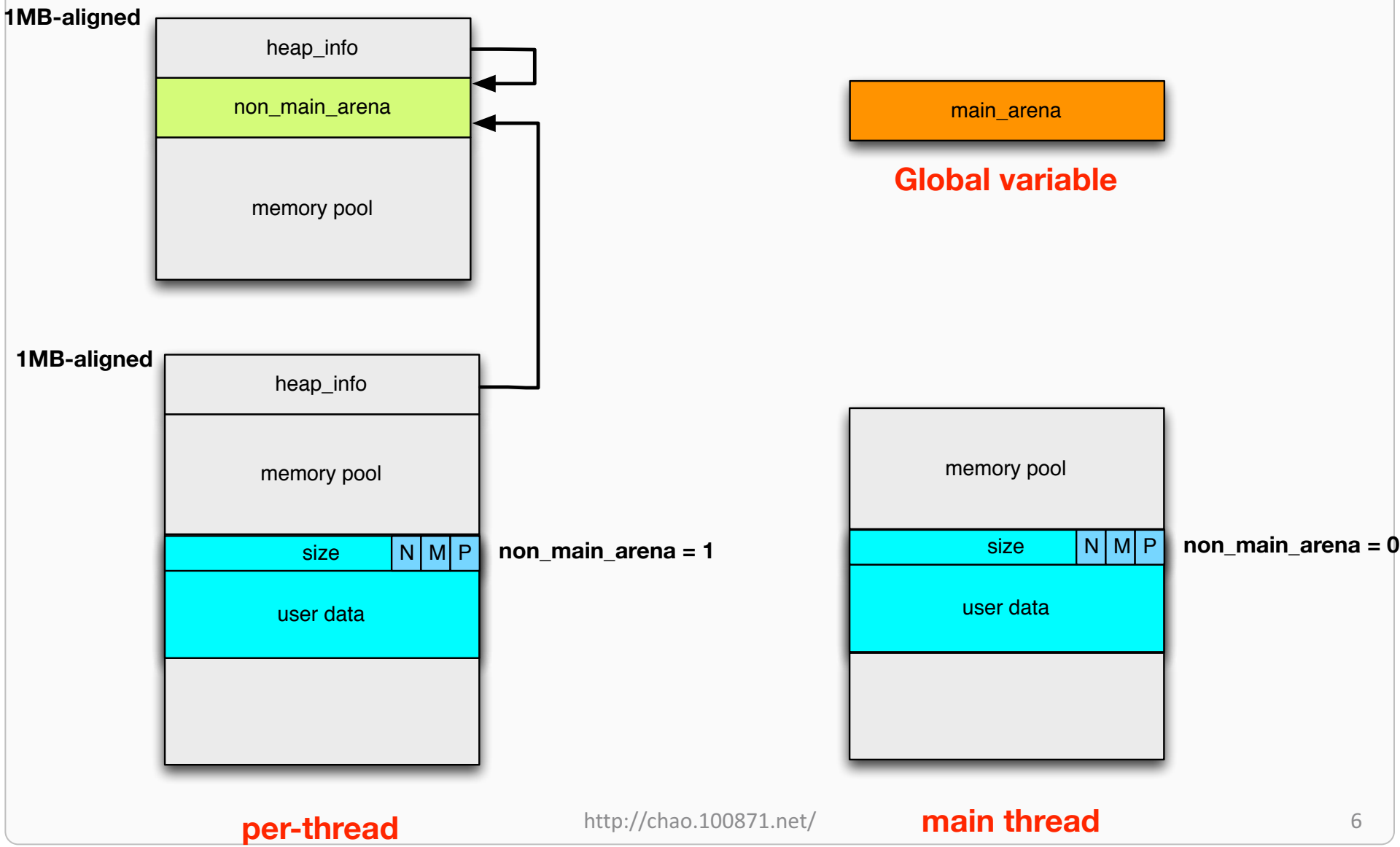


- prev_size of next chunk
  - this free chunk's size
- prev_in_use of next chunk
  - is cleared
- But,
  - they are not set if this free chunk is kept in fastbins

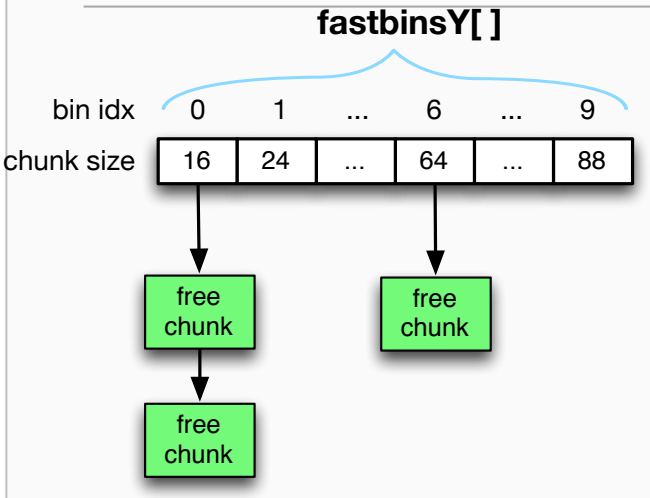
Review: Arena (32-bit allocators)



Get Arena


1MB-aligned
heap_info
non_main_arena
memory pool
main_arena
Global variable
1MB-aligned
heap_info
memory pool
size
N
M
P
non_main_arena = 1
user data
memory pool
size
N
M
P
non_main_arena = 0
user data
per-thread
main thread

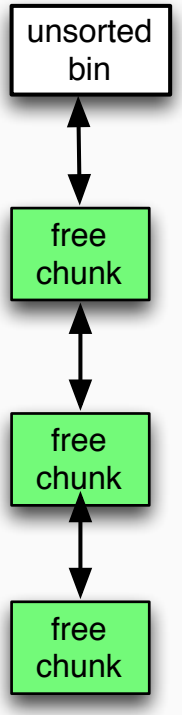
Fast-bins



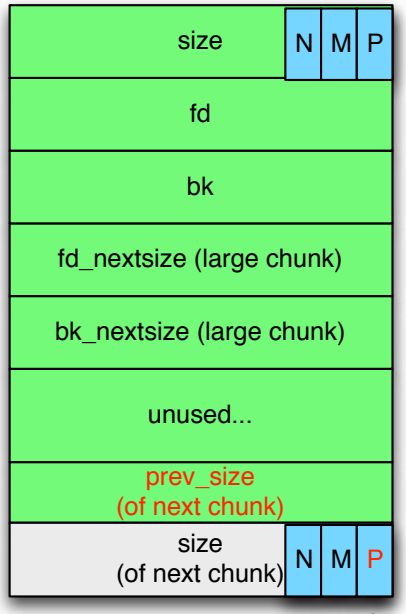
- Keyword: fast
- malloc(fast_size)
  - get one chunk from the specific fast-bin if available
- free(fast_size)
  - place into the specific fast-bin
  - only set fd of current chunk
  - not set
    - bk of current chunk
    - prev_size of next chunk
    - P bit of next chunk
  - not collapse prev/next free chunk

single linked list, same size

unsorted_bin

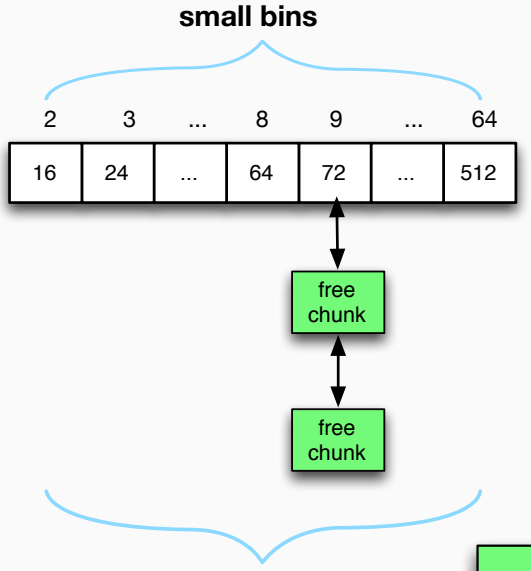


double linked list, unsorted size



- malloc()
  - if unsorted_bin is searched, its chunks will be iterated one by one
    - returns the chunk if its size fits and stops iterating, otherwise
    - puts the chunk into small/large bin
      - the only place to insert small/large bin
- free(non_fast_size)
  - put it into unsorted_bin
    - no need to select small/large bin
  - collapse previous free chunk
    - check P bit of current chunk
  - collapse next free chunk
    - check P bit of the next next chunk
  - set
    - fd/bk of current chunk
    - prev_size, P bit of next chunk

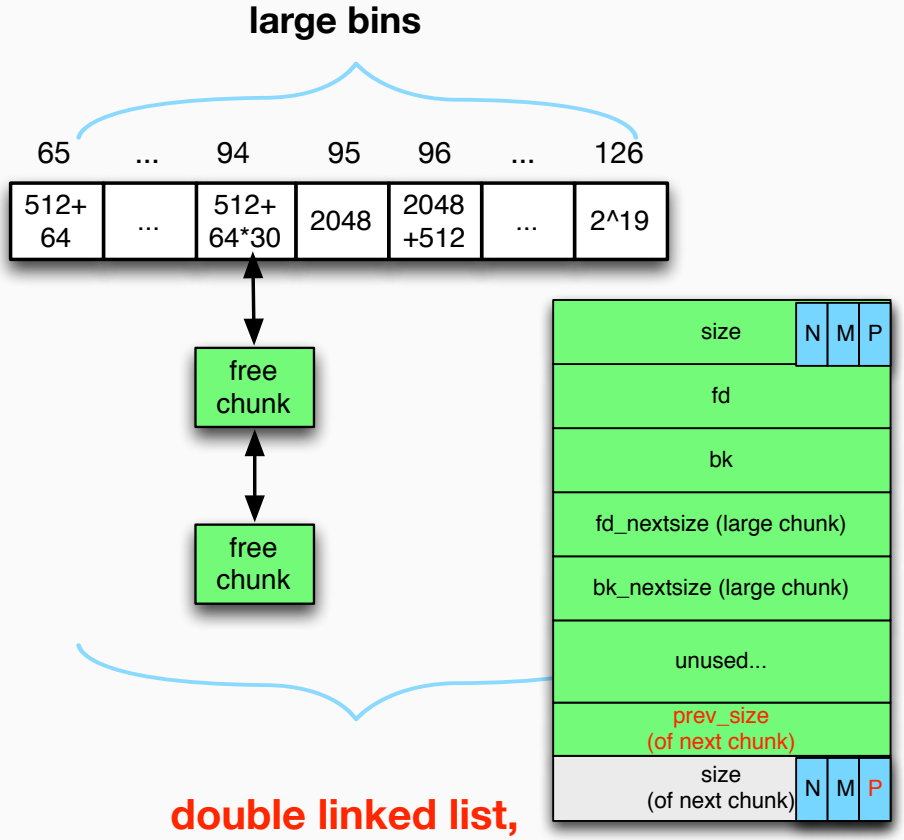
Small-bins



double linked list, same size

- malloc()
  - if small-bins are searched
    - return the chunk if its size fits
- free()
  - doesn't put into small-bin
- Insertion
  - unsorted_bin is searched for allocation, chunk size doesn't satisfy requirement
- Deletion (unlink)
  - it is allocated to objects, or
  - it is merged with neighbor chunks being freed

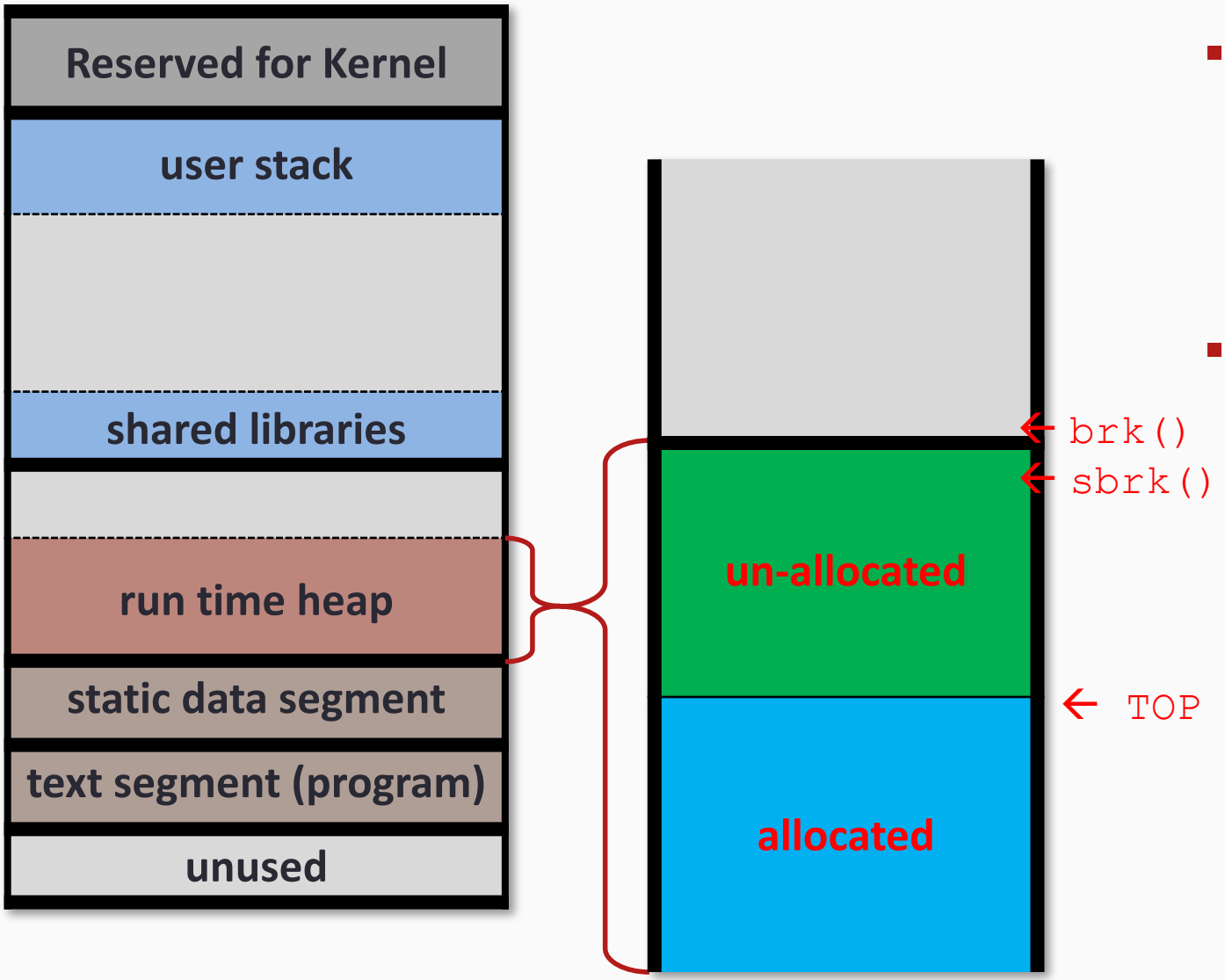
Large-bins



double linked list,
next chunk size is larger

- malloc()
  - if large-bins are searched
    - find the smallest chunk that is big enough
    - split this large chunk
      - return one to user
      - put another in unsorted_bin
- free()
  - doesn't put into large-bin
- Insertion
  - unsorted_bin is searched for allocation, chunk size doesn't satisfy requirement
- Deletion (unlink)
  - it is allocated to objects, or
  - it is merged with neighbor chunks being freed

Top

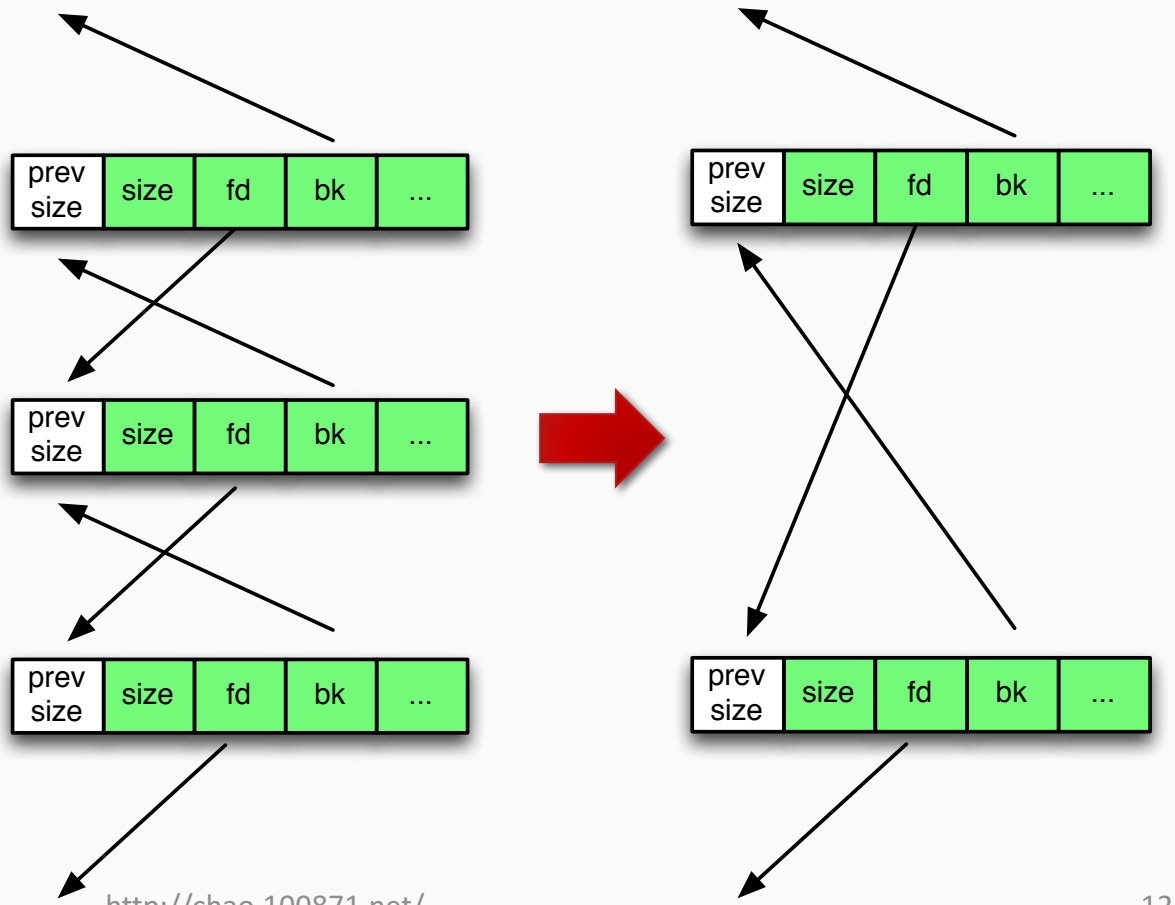


- memory pool allocated by kernel
- glibc allocate objects inside the pool

Review: unlink

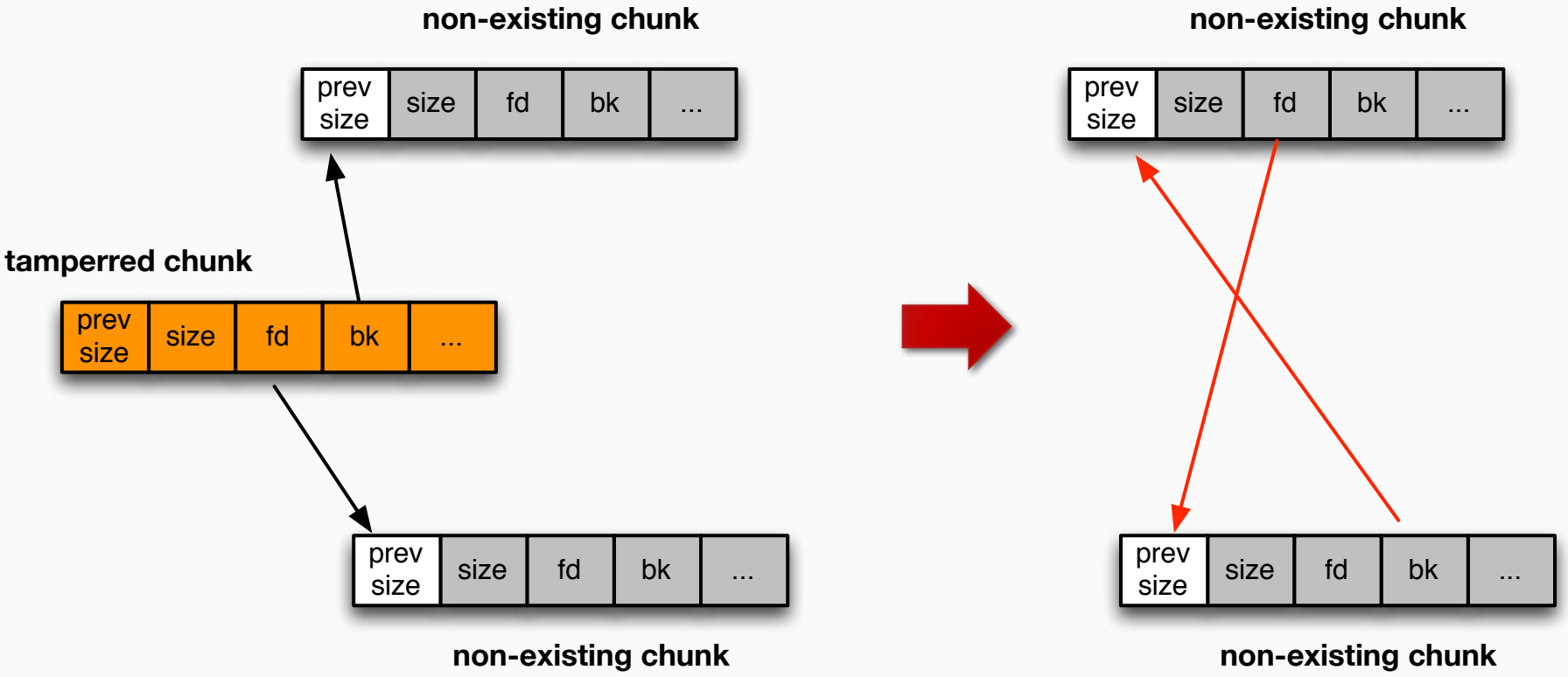
- remove node, and update BK/FD,

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



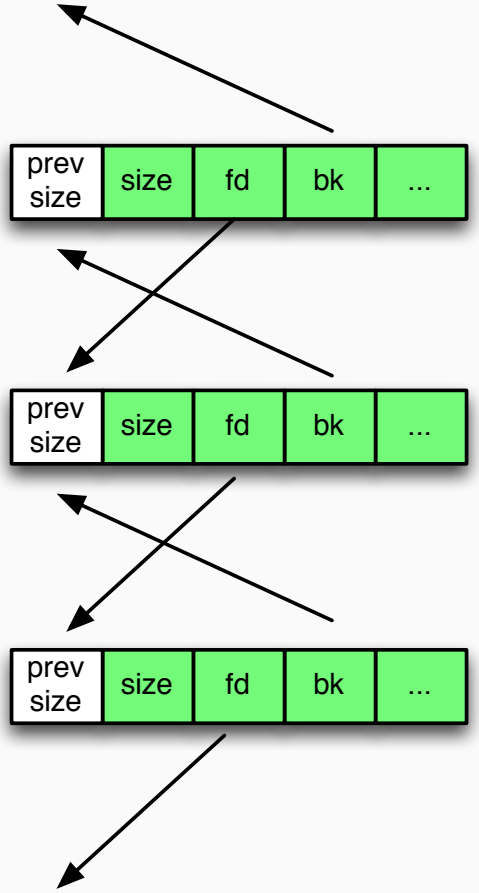
Compromised unlink

- When a chunk to be unlinked (e.g., malloc, free) is compromised, …



Write arbitrary content to arbitrary address!

How to trigger unlink?

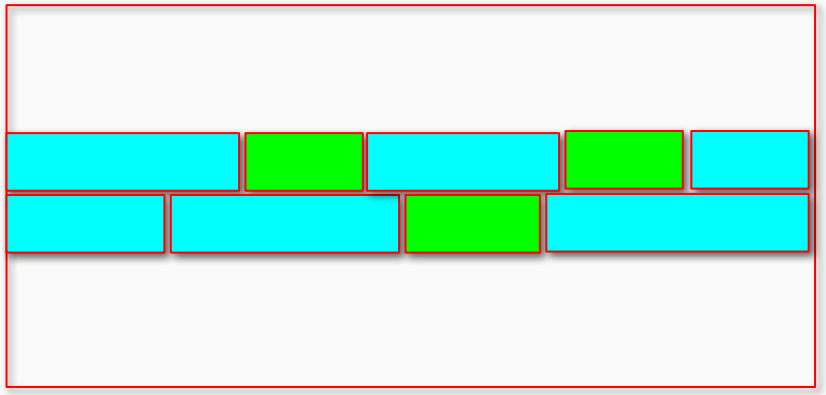


- Case 1:
  - The free chunk is merged with previous/following neighbor free chunk
- Case 2:
  - The free chunk is picked for allocation.

In a Word

- free() and malloc() rely on metadata to
  - get arena, traverse bins, lists, chunks
  - update fd/bk/size/prev_size etc. of related chunks
  - return memory to users

- But these metadata are not well-protected
  - e.g., heap overflow, use-after-free



1. Heap Overflows

Heap Overflows

- Buffer overflows are basically the same on the heap as they are on the stack
- Overflow targets on the stack:
  - return address,
  - saved stack frame pointers,
  - local function pointers
  - exception handlers
  - other sensitive data
- Overflow targets on the heap:
  - heap metadata
  - function pointers in objects
  - vtable pointers in objects
  - other sensitive data

Example

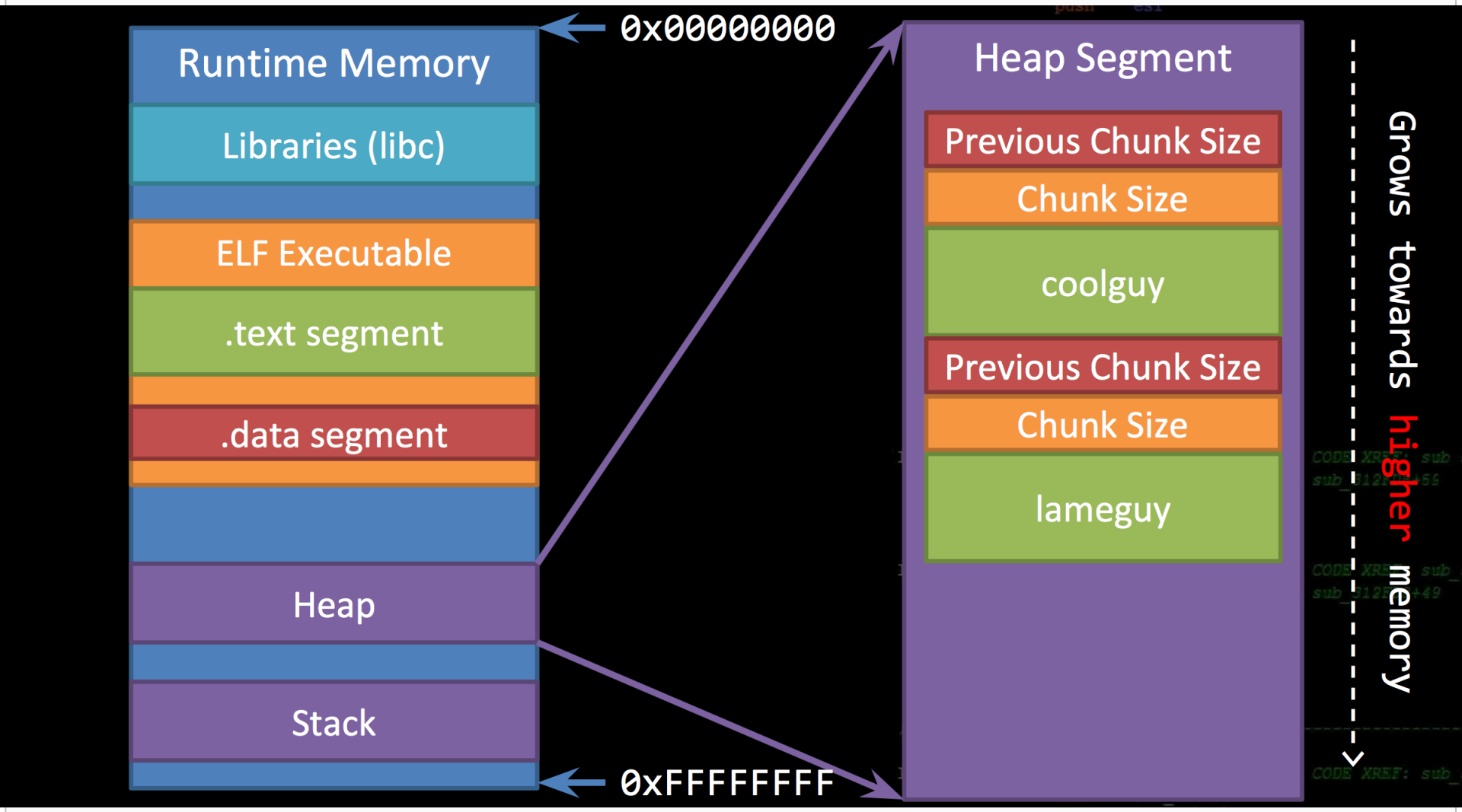
```
struct toystr {
    void (* message)(char *);
    char buffer[20];
};
```

Example

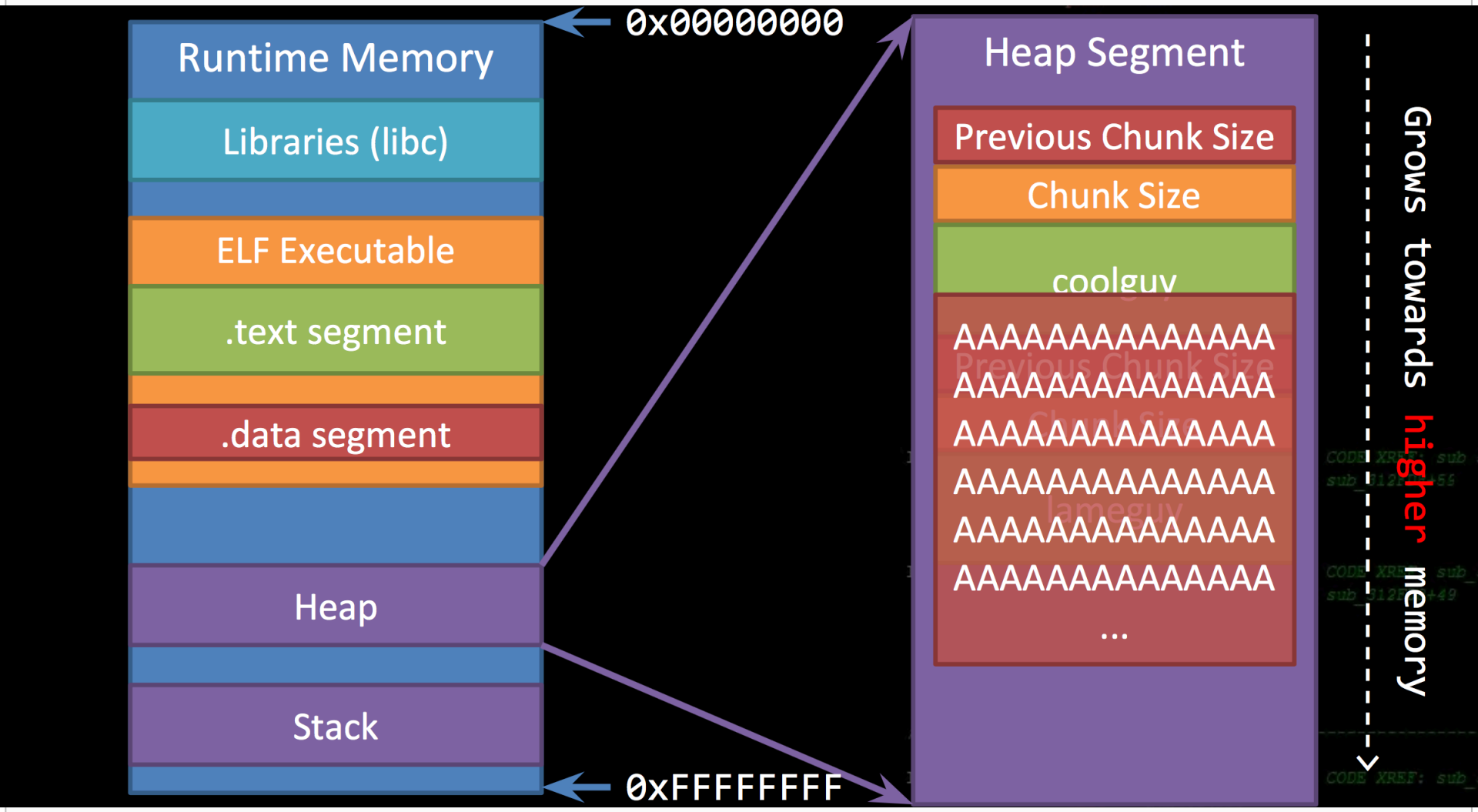
```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Silly heap overflow

Example


0x00000000
Runtime Memory
Libraries (libc)
ELF Executable
.text segment
.data segment
Heap
Stack
0xFFFFFFFF
Heap Segment
Previous Chunk Size
Chunk Size
coolguy
Previous Chunk Size
Chunk Size
lameguy
Grows towards higher memory

Example


0x00000000
Runtime Memory
Libraries (libc)
ELF Executable
.text segment
.data segment
Heap
Stack
0xFFFFFFFF
Heap Segment
Previous Chunk Size
Chunk Size
coolguy
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
AAAAAAAAAAAAAA
...
Grows towards higher memory

Example

```
coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));

coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 200, stdin);  // oopz...
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

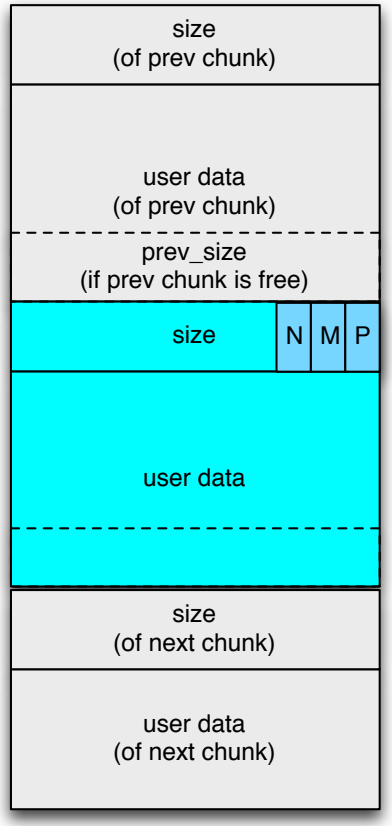
Silly heap overflow

Overwritten function pointer!

Think

- What else can heap overflow corrupt?
  - ~~function pointers in the objects~~
  - heap management metadata
  - VTable pointers
- How could we exploit them?

Exploit 2-1: overflow heap metadata



P: prev_in_use
M: is_mmaped
N: non_main_arena

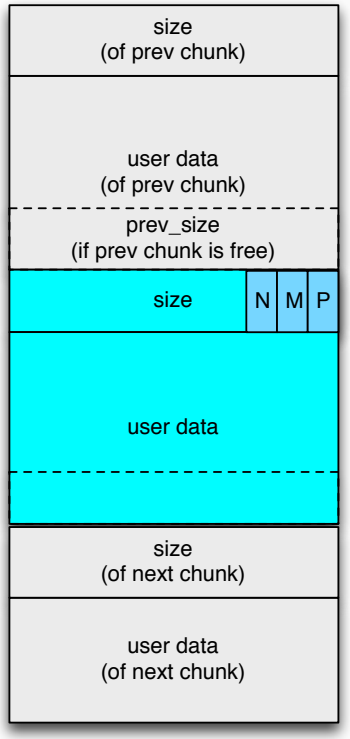
- overflow a chunk in-use
  - overflow P bit? (prev_in_use)
    - change it from 0 to 1?
      - when this chunk is freed, previous (free) chunk will not be merged.
        - ➢NOTE: merge operation only happens if the chunk being freed is larger than fast-bin-size
    - change it from 1 to 0?
      - when this chunk is freed, previous (in-use) chunk will be merged,
        - ➢Q1: what is the size of previous "freed" chunk?
        - ➢Q2: what happen after merging?
      - if the merged free chunk is then allocated to another object, then
        - ➢two different objects will share a same memory region.

NOTE:

1. Allocated chunks are not tracked by arena/fastbin/..., but only an extra size field is attached.
2. Allocated chunks could be later freed and tracked by arena.

Exploit 2-2: overflow heap metadata

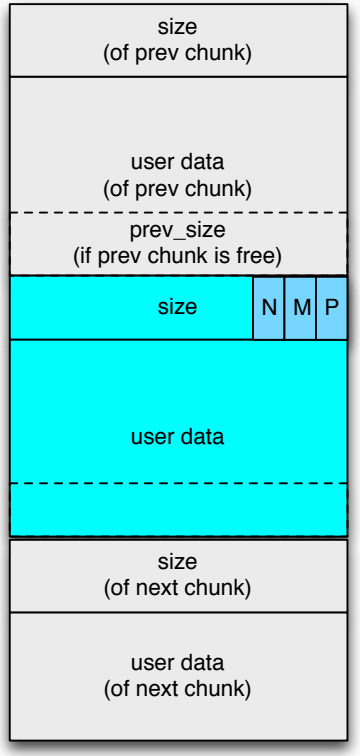
- overflow a chunk in-use
  - overflow N bit? (non_main_arena)
    - change it from 0 to 1?
      - will mark it as non_main_arena, switch the arena
        - ➢1MB alignment, heap_info, then per-thread arena
        - ➢if we control the heap_info, we control the arena
      - when this chunk is freed, it will be inserted into fastbin or unsorted_bin
        - ➢which are indexed by fake arena, and thus fake
      - the link list pointed by fastbin/unsorted_bin will be updated.
        - ➢e.g., if the fake unsorted_bin points to GOT entry, then GOT will be updated to point to the freed chunk we controlled.



P: prev_in_use
M: is_mmaped
N: non_main_arena

https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/

Exploit 2-3: overflow heap metadata



P: prev_in_use
M: is_mmaped
N: non_main_arena

- overflow a chunk in-use
  - overflow size field
    - enlarge the size field?
      - when this chunk is freed, a part of the next (maybe in-use) chunk is also freed
      - if this larger-than-expectation free chunk is allocated to another object, it will cause
        - ➢two different objects share a same memory.
    - shrink the size field?
      - waste a part of memory when freed, causing memory leak
        - ➢no one is aware of this wasted fragment, unless
        - ➢a full memory scan operation is performed

Exploit 2-4: overflow heap metadata

- overflow a free chunk
  - overflow P field (prev_in_use)
    - change it from 1 to 0
      - Q1: what if it is allocated to objects?
        - ➢will the P bit be corrected by the heap manager?
      - Q2: what if it is merged with other free chunks?
        - ➢e.g., the chunk after this free chunk is freed,
        - ➢it will try to merge the previous (in-use) chunk, and get a larger-than-expect free chunk
        - ➢this free chunk could be allocated to new objects later,
          - the new object will share memory with the previous (in-use) chunk
      - Q3: what if this chunk is moved between bins?
    - change it from 0 to 1
      - the previous (free) chunk will not be merged.

NOTE:

1. Freed chunks are tracked by arena/fastbin/...
2. Freed chunks could be later allocated to new objects, or be merged with new free chunks, or moved from fastbin → unsorted_bin → smallbin/largebin

Exploit 2-5: overflow heap metadata

- overflow a free chunk
  - overflow N field (non_main_arena)
    - change it from 0 to 1
      - will mark it as non_main_arena, switch the arena
        - 1MB alignment, heap_info, then per-thread arena
        - if we control the heap_info, we control the arena
      - Q1: what if this chunk is reallocated
        - NOTE: only the main_arena will allocate it to objects.
        - nothing? Will the heap manager correct this bit?
      - Q2: what if this chunk is merged with neighbor chunks?
        - i.e., when the next/previous chunk is being freed
        - NOTE: the next/previous chunk belongs to main_arena
      - Q3: what if this chunk is moved between bins?
        - nothing?
    - change it from 1 to 0?

Exploit 2-6: overflow heap metadata

- overflow a free chunk
  - overflow size field
    - what if this chunk is allocated?
      - when this chunk is in fastbin/smallbin?
        - ➢the size of chunks in these linked list is fixed (equals to allocation size), so
        - ➢this overwritten size field doesn't matter, right?
        - ➢will the heap manager correct this size field?
      - when this chunk is in unsorted_bin or largebin?
        - ➢the overwritten size field matters, right?

- what if this chunk is merged with neighbor chunks?
  - case 1: the next chunk is being freed
    - ➢this overwritten size field will be corrected, after the merge operation
  - case 2: the previous chunk is being freed
    - ➢it will try to merge this free chunk, by looking for the next chunk of this chunk (by checking P bit and prev_size)
    - ➢so, a wrong next chunk will be checked, and
    - ➢the heap manager will a wrong decision to merge the free chunk or not.

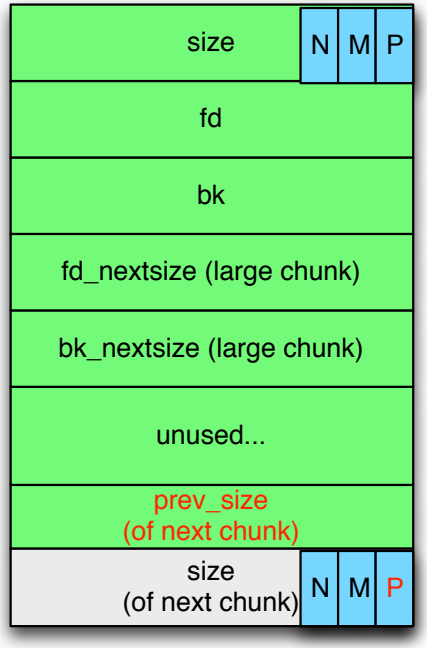
- What if this chunk is moved between bins?
  - it will be wrongly put into small/large bins, causing
  - unexpected memory allocation later

Exploit 2-7: overflow heap metadata

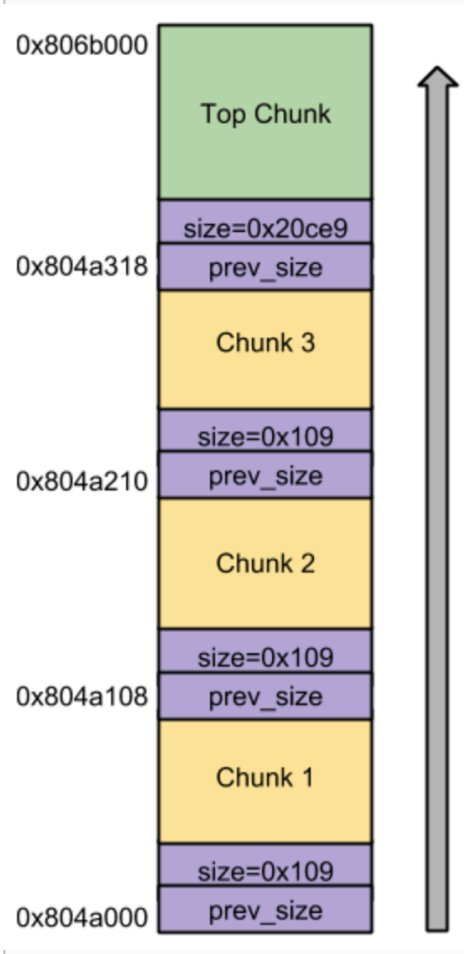
- overflow a free chunk
  - overflow fd/bk pointers
    - unlink issue

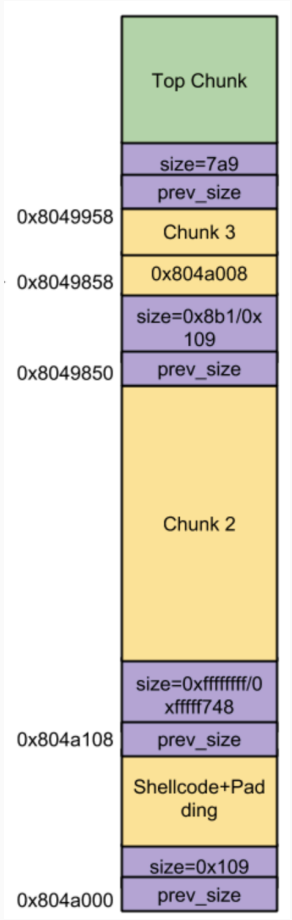
Exploit 2-8: overflow heap metadata

- overflow a free chunk
  - overflow prev_size field
    - what if shrink prev_size field?
      - when the next chunk is freed, what will happen?
        - ➢it will try to merge with the (part of) current chunk
        - ➢the (part of) current chunk will be unlink-ed, causing
          - unlink operation on fake fd/bk pointers, and
          - the current free chunk and the merged free chunk overlap
    - what if enlarge prev_size field?
      - when the next chunk is freed, what will happen?
        - ➢it will try to merge with the (larger-than-fact) current chunk
        - ➢the (larger-than-fact) current chunk will be unlink-ed, causing
          - unlink operation on fake fd/bk pointers, and
          - the previous (in-use) chunk will share memory with the new merged free chunk, which will be later allocated to new objects.



Exploit 2-9: overflow heap metadata





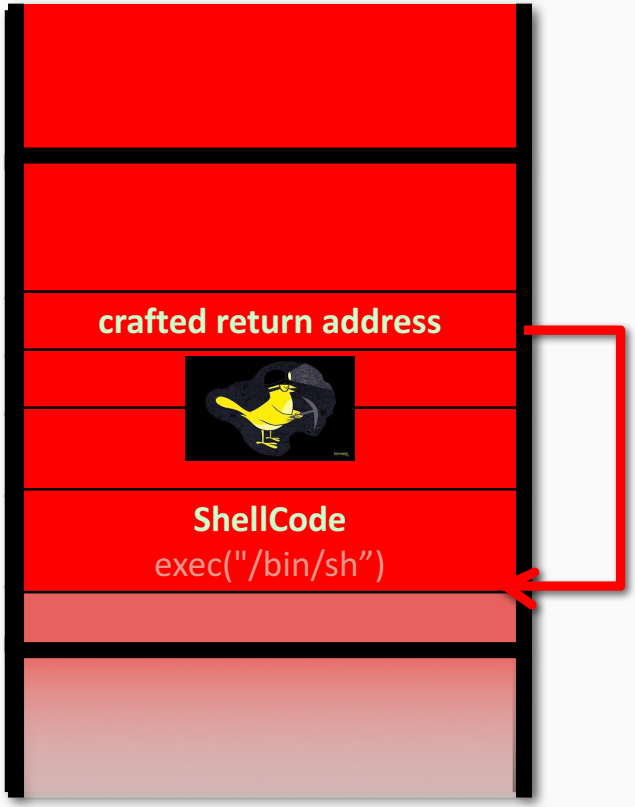
- overflow TOP chunk size field
  - shrink size?
    - waste memory
  - enlarge size (extremely large)
    - new allocations could always be fulfilled with the (larger-than-fact) TOP chunk
    - this (larger-than-fact) TOP chunk could even cover existing code/data, e.g., GOT table
    - so arbitrary address (e.g., GOT entries) could be returned for allocation

How to defense heap overflow?

Stack Canaries/Cookies

- Canaries for coal mine

- Canaries for return address?



Defenses against Heap Overflows

- Cookie is a good solution
  - Similar to StackGuard/GS, which places cookies/canaries near return addresses, we could place cookies near heap objects
- However,
  - the performance overhead is high
  - there is no good time for cookie check
    - for stack: check when function returns
    - for heap???
      - a reasonable one is when malloc/free is called
      - but they may not operate on the corrupted objects

2. Use-After-Free (UAF)

The problem

- The Cloud

- The Browsers

  - written in C++
  - BIG Targets

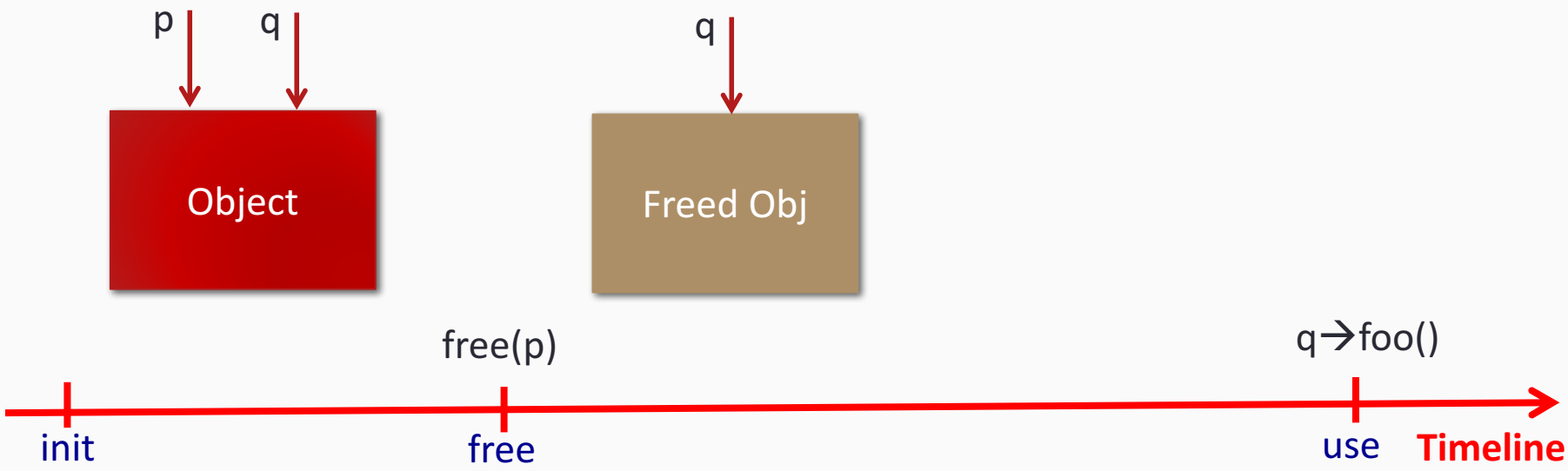
Google[1]:
"80% attacks exploit use-after-free..."

Microsoft[2]:
50% CVEs targeted Winows7 are UAF

1. https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf
2. http://download.microsoft.com/download/F/D/F/FDFBE532-91F2-4216-9916-2620967CEAF4/Software%20Vulnerability%20Exploitation%20Trends.pdf

Use-After-Free (UAF)

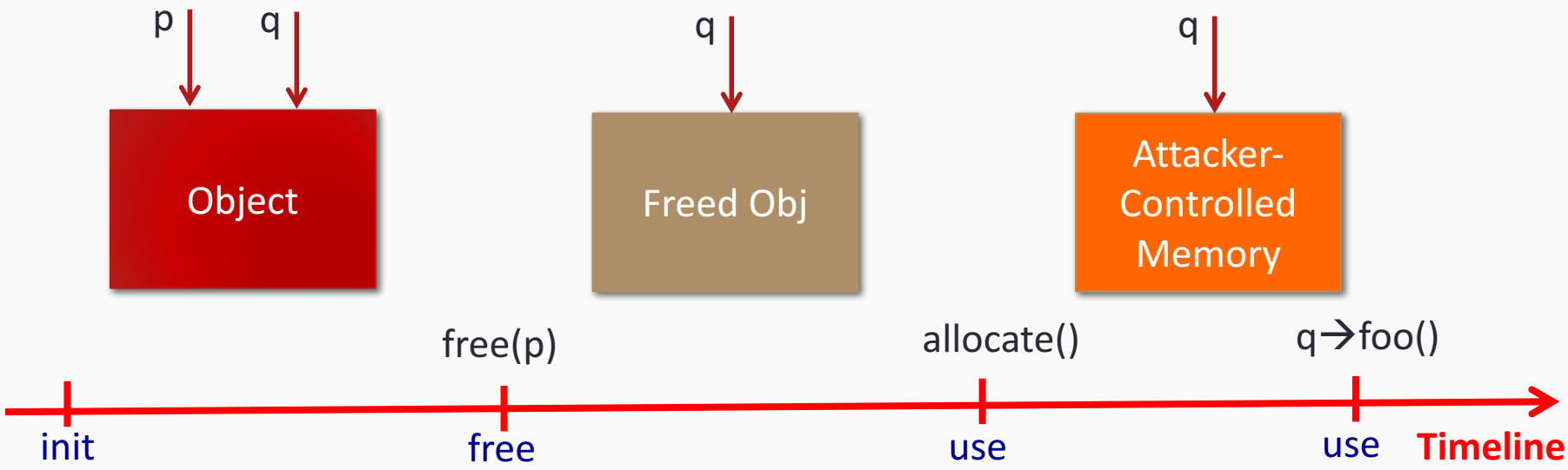
- The Vulnerability



- what if the freed memory is free?
- what if the freed memory is occupied?

Use-After-Free (UAF)

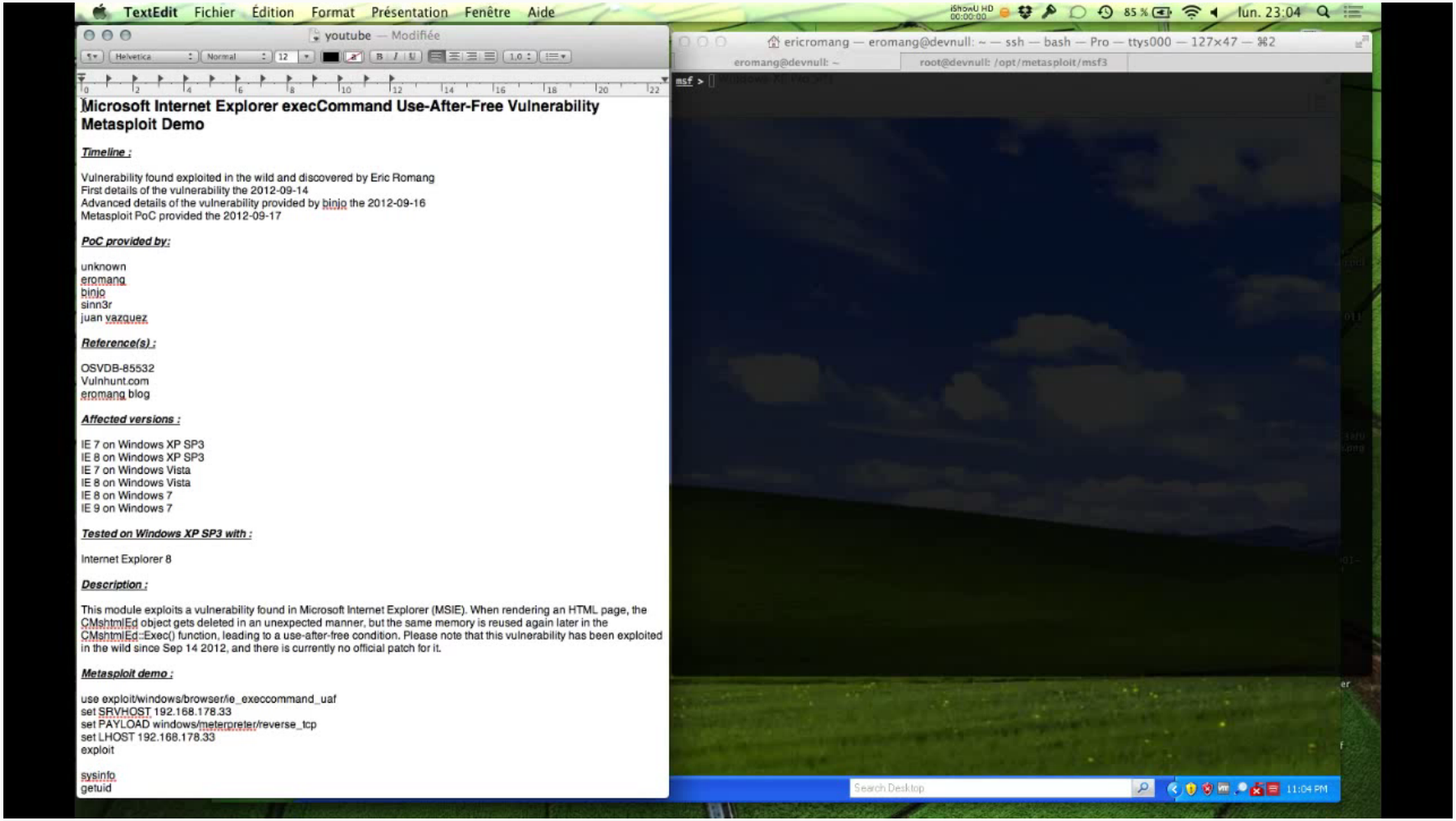
- The Exploit



The most common UAF exploit is VTable Hijacking

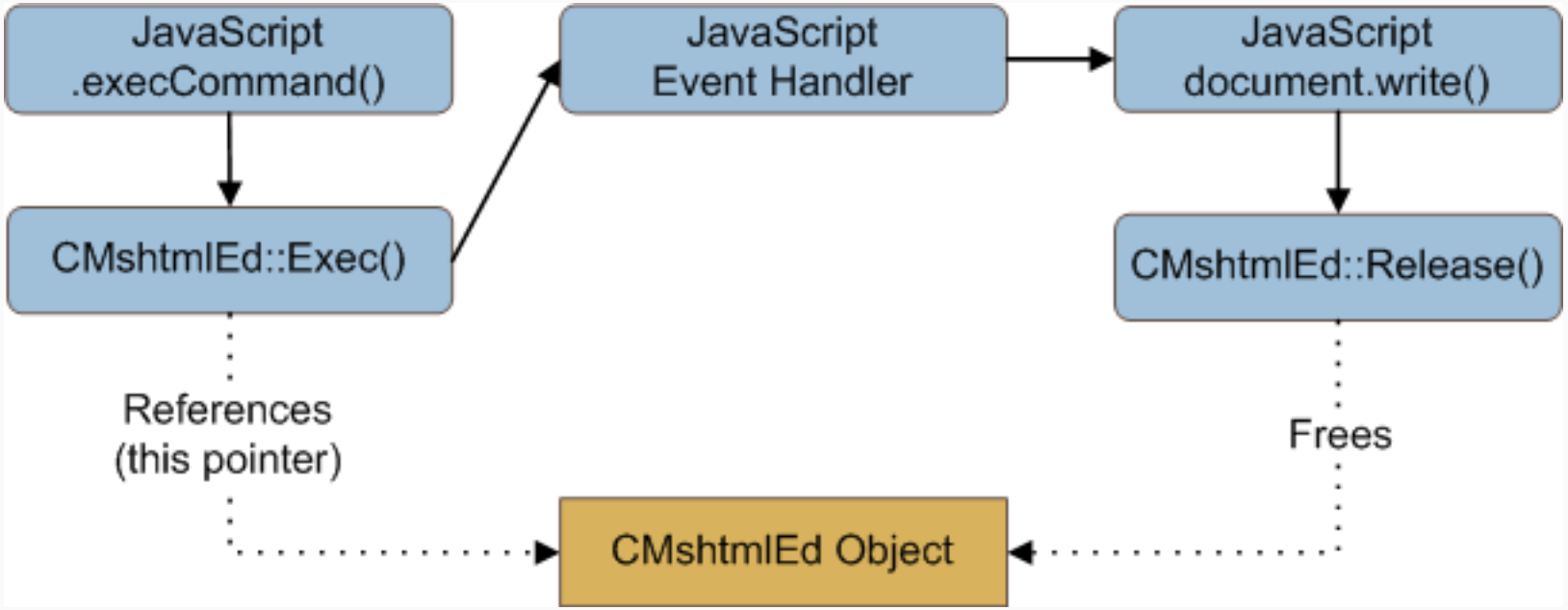
A popular way of exploiting UAF

MS12-063 Microsoft Internet Explorer execCommand Vulnerability Demo



Under the hood

- Use after free vulnerability (MS12-063)



Under the hood (2)

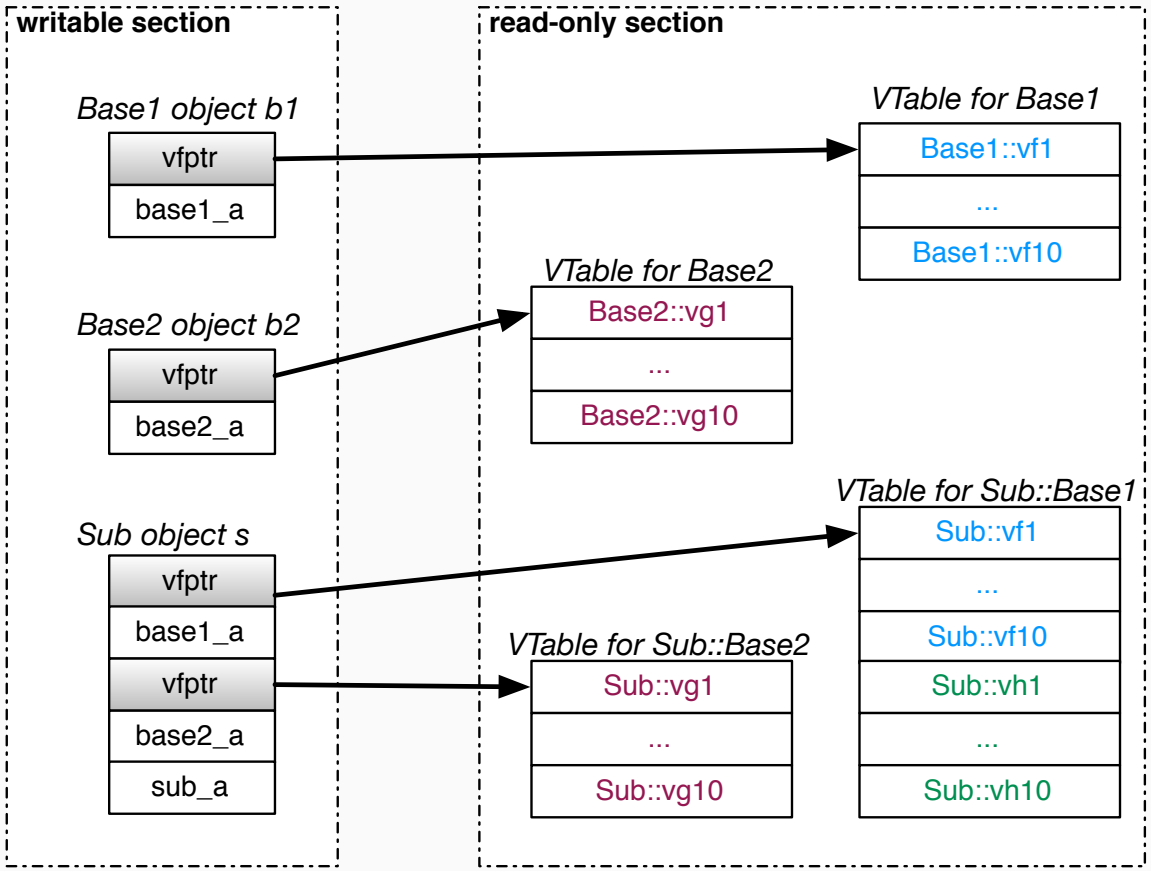
- Exploitable Point:

```
.text:74E7C4B3                  call    ?Exec@CCommand@@QAEJKPAUtagVARIANT@@0PAVCMshtmlEd@@@Z ; CCommand::Exec(
.text:74E7C4B8                  mov     esi, eax
.text:74E7C4BA
.text:74E7C4BA loc_74E7C4BA:                            ; CODE XREF: CMshtmlEd::Exec(_GUID const *,ulong,ulong,
.text:74E7C4BA                                          ; CMshtmlEd::Exec(_GUID const *,ulong,ulong,tagVARIANT
.text:74E7C4BA                  mov     edi, [edi+8]
.text:74E7C4BD                  mov     eax, [edi]
.text:74E7C4BF                  push    edi
.text:74E7C4C0                  call    dword ptr [eax+8]
.text:74E7C4C3                  mov     eax, esi
.text:74E7C4C5
.text:74E7C4C5 loc_74E7C4C5:                            ; CODE XREF: CMshtmlEd::Exec(_GUID const *,ulong,ulong,
.text:74E7C4C5                  pop     edi
.text:74E7C4C6                  pop     esi
.text:74E7C4C7                  pop     ebx
.text:74E7C4C8                  pop     ebp
.text:74E7C4C9                  retn    18h
.text:74E7C4CC ; ---------------------------------------------------------------------------
.text:74E7C4CC
.text:74E7C4CC loc_74E7C4CC:                            ; CODE XREF: CMshtmlEd::Exec(_GUID const *,ulong,ulong,
.text:74E7C4CC                  mov     esi, 80040100h
.text:74E7C4D1                  jmp     short loc_74E7C4BA
.text:74E7C4D1 ?Exec@CMshtmlEd@@UAGJPBU_GUID@@KKPAUtagVARIANT@@1@Z endp
.text:74E7C4D1
.text:74E7C4D1 ; ---------------------------------------------------------------------------
```

virtual call *CMshtmlEd::Exec()*

VTable for Dynamic Dispatch (C++)

class Sub: public Base1, Base2{…};



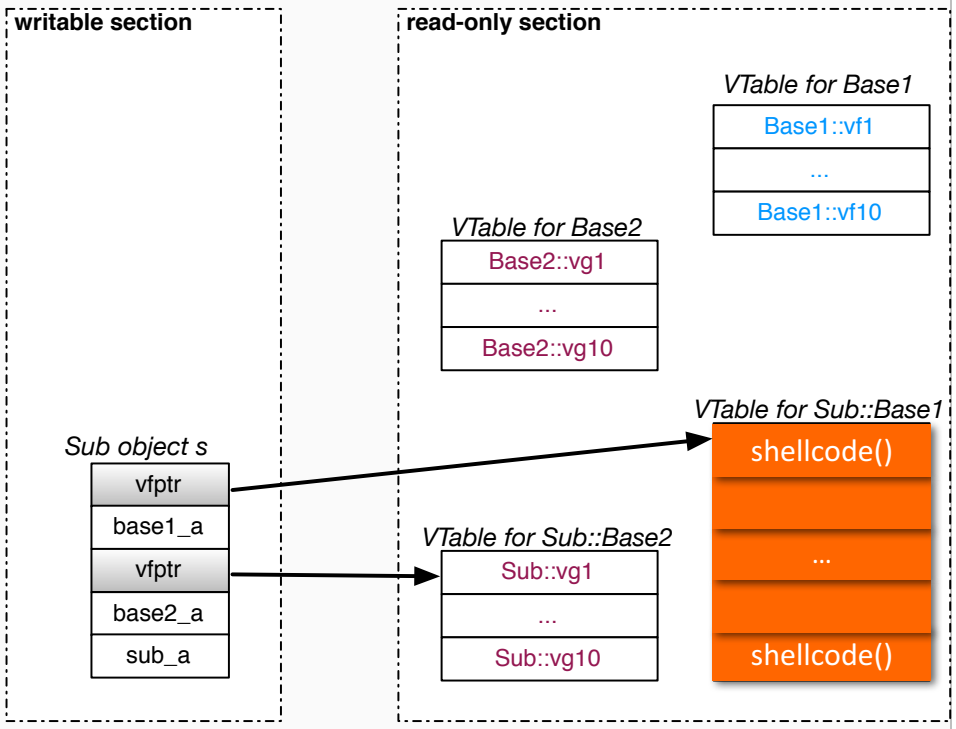

```
void foo(Base2* obj){
  obj→vg4();
}

void main(){
   Base2* obj = new Sub();
   foo(obj);
}
```

code section

```
; Function main()
push  SIZE
call    malloc()
mov   ecx, eax
call    Sub::Sub()
; now ECX points to the Sub object
add   ecx, 8
; now ECX points to the Sub::Base2 object
call    foo()
ret

; Function foo()
mov eax, [ecx]            ; read vfptr of Base2
mov edx, [eax+0x0C]   ; get vg4() from vtable
call  edx                     ; call Base2::vg4()
ret
```

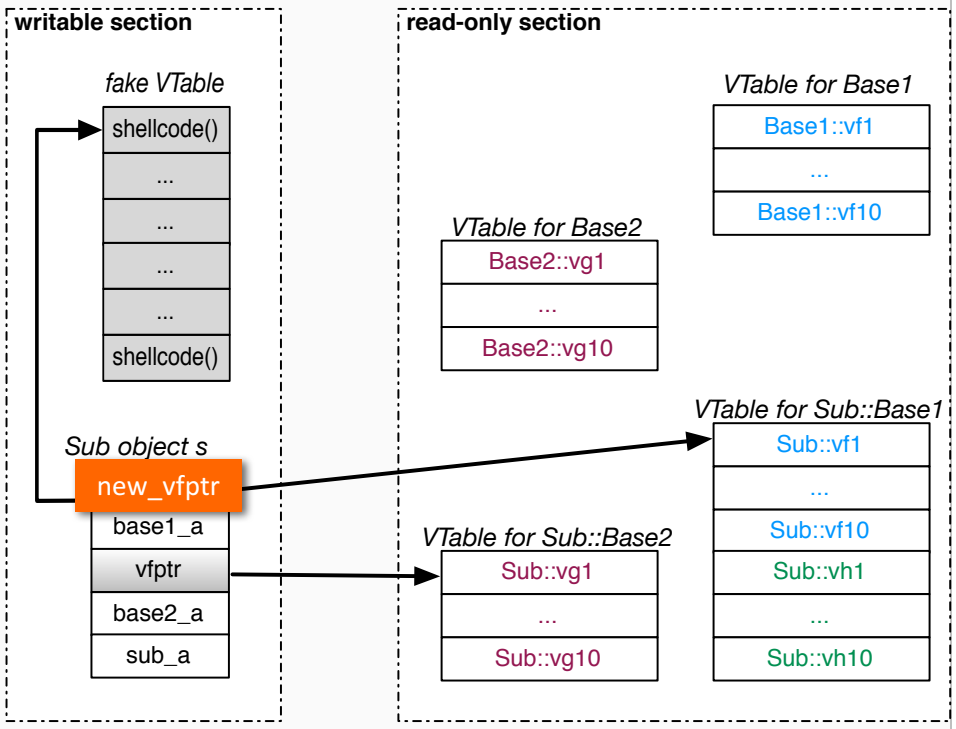
VTable Hijacking Classification

- VTable corruption
  - overwrite VTable
- VTable injection
- VTable reuse



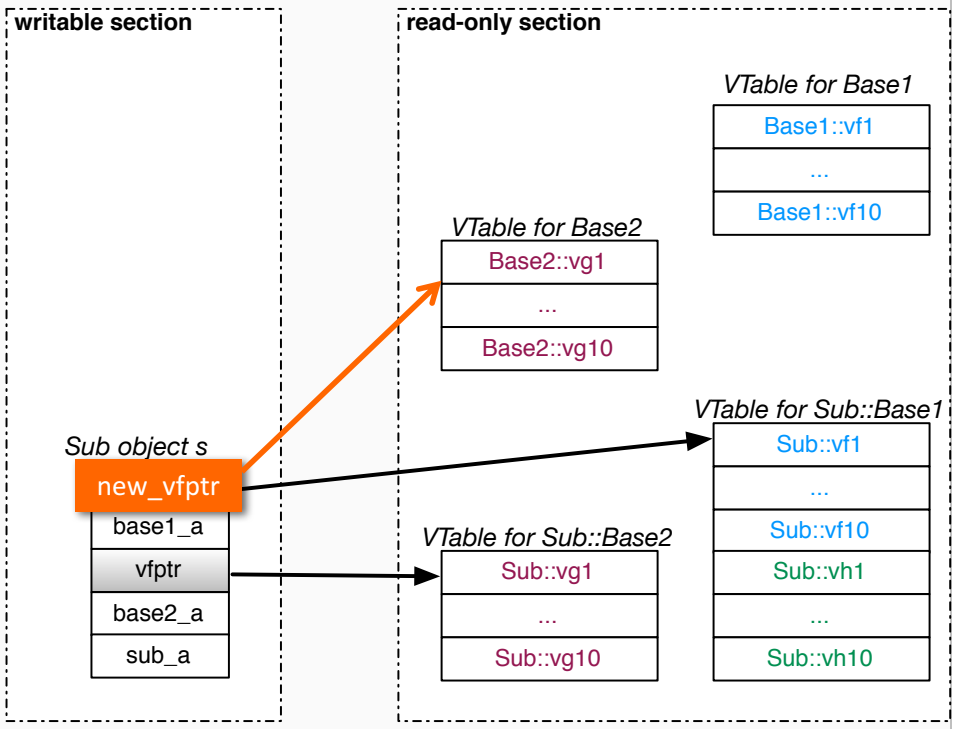
VTable Hijacking Classification

- VTable corruption
  - overwrite VTable
- VTable injection
  - overwrite vfptr
  - point to fake VTable
- VTable reuse



VTable Hijacking Classification

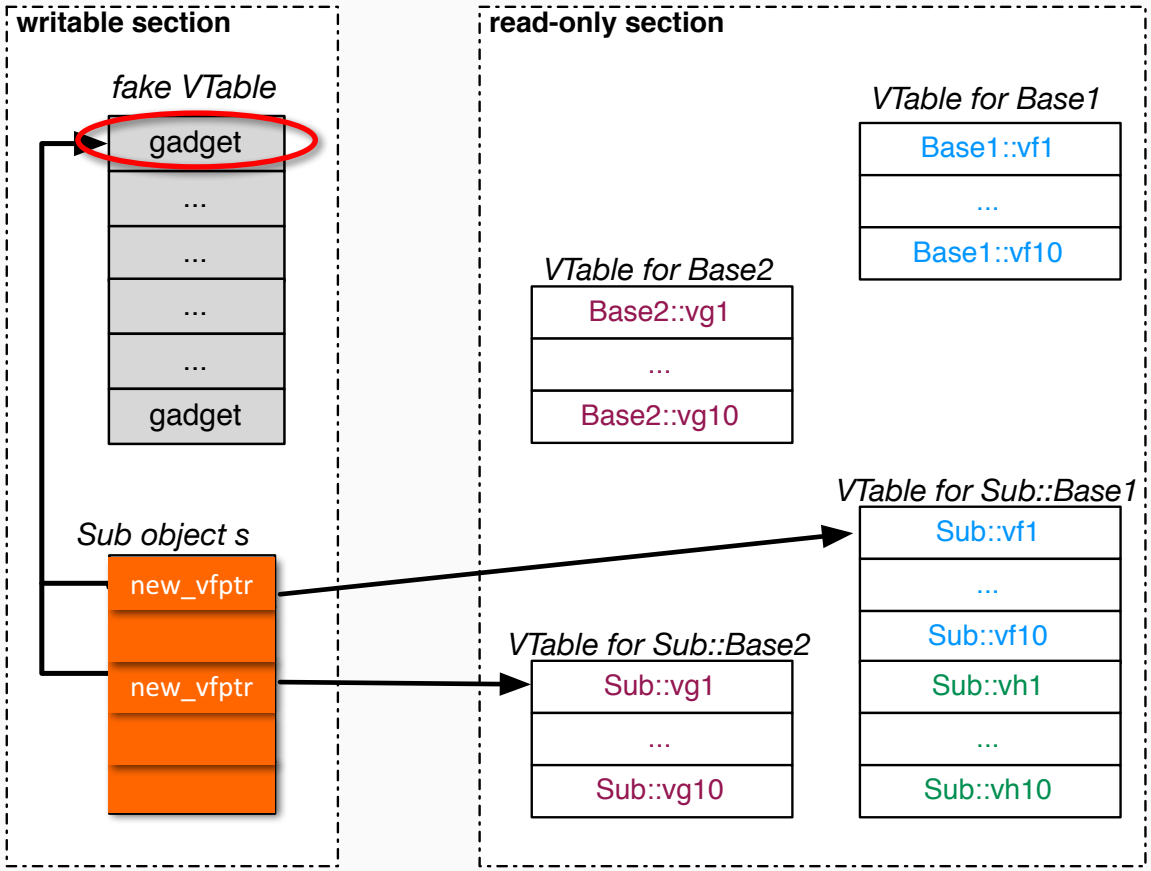
- VTable corruption
  - overwrite VTable
- VTable injection
  - overwrite vfptr
  - point to fake VTable
- VTable reuse
  - overwrite vfptr
  - point to existing VTable, data etc.



VTable Hijacking in real world

+ Vulnerabilities like use-after-free
+ VTable Injection
+ ROP gadgets

- Pwn2Own 2014 Firefox
- Pwn2Own 2014 Chrome
- CVE-2014-1772 IE



code section

```
; Function main()
push  SIZE
call    malloc()
mov   ecx, eax
call    Sub::Sub()
; now ECX points to the Sub object
add   ecx, 8
; now ECX points to the Sub::Base2 object
call    foo()
ret

; Function foo()
mov eax, [ecx]            ; read vfptr of Base2
mov edx, [eax+0x0C]   ; get vg4() from vtable
call  edx                     ; call Base2::vg4()
ret
```

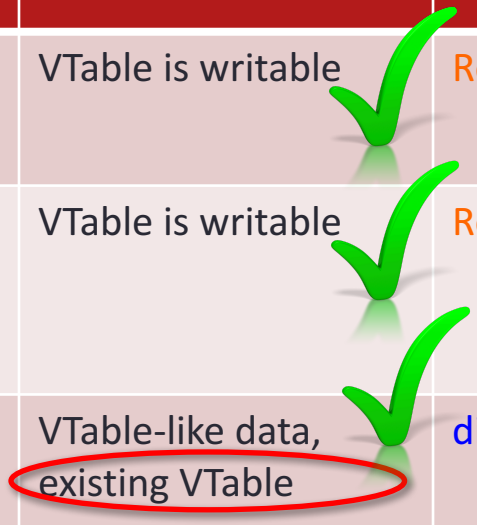
Defenses Against VTable Hijacking

Defense 1: VTint

	Attack	Requirement	
VTable Corruption	overwrite VTable	VTable is writable	
VTable Injection	overwrite vfptr, point to injected VTable	VTable is writable	
VTable Reuse	overwrite vfptr, point to existing VTable/data	VTable-like data, existing VTable	

Observation → Intuition

	Attack	Requirement	Countermeasure
VTable Corruption	overwrite VTable	VTable is writable	Read-only VTable
VTable Injection	overwrite vfptr, point to injected VTable	VTable is writable	Read-only VTable
VTable Reuse	overwrite vfptr, point to existing VTable/data	VTable-like data, existing VTable	different VTable/data



Need exact TYPE information

VTint vs. DEP

	VTint
VTable Corruption	Read-only VTable
VTable Injection	Read-only VTable
VTable Reuse	different VTable/data

	DEP
Code Corruption	Read-only Code Sec
Code Injection	Read-only Code Sec *(writable sections will not be executed)*
Code Reuse	NO

- Similar to DEP
  - lightweight, and can be binary-compatible
- Different from DEP
  - after hardening, the attack surface is smaller

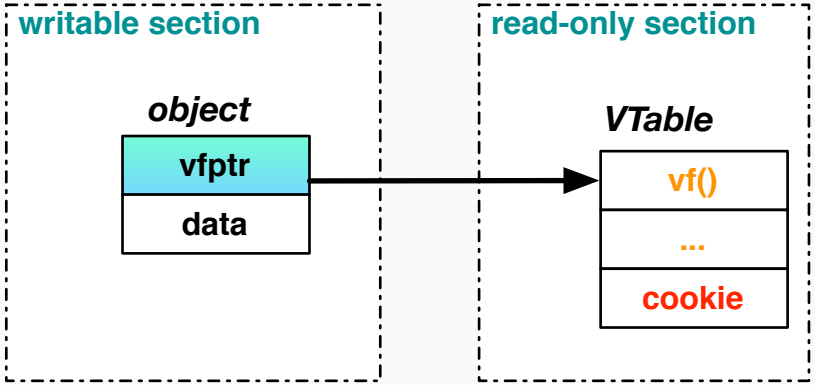
Binary Level Defense: vfGuard

- NDSS’ 15: Dynamic binary instrumentation
  - inspect virtual call instructions at runtime (using PIN)
  - enforce several policies on these instructions
- Pro
  - easy to extend, to enforce different policies
- Con
  - high overhead (PIN’s overhead * 118%)
  - could not defeat VTable reuse attacks.

Source Level Defense: **VTGuard**

- Microsoft IE10, Core Objects
  - special cookies at the end of Vtables

- Pro
  - lightweight, 0.5%
- Con
  - only core objects
  - VTable injection
  - Information leakage



Source Level Defense: **SafeDispatch**

- NDSS'14, LLVM-based
  - statically compute set of legitimate targets
  - dynamically validate against this set

- Policy 1:
  - VTable Check
  - Con:
    - heavy compile time analysis
    - high runtime overhead (~30%)

```
C *x = ...
ASSERT(VPTR(x) ∈ Valid(C));
x->foo();
```

- Policy 2:
  - method check
  - Con:
    - heavy compile time analysis
    - high runtime overhead (~7%)

```
C *x = ...
vptr = *((FPTR**)x);
//ASSERT(vptr ∈ {C::vptr, D::vptr});
f    = *(vptr + 0);
ASSERT(f ∈ ValidM(C,foo));
f(x)
```

Source Level Defense: **Forward Edge CFI**

- GCC-VTV [Usenix' 14], whitelist-based

  - compute **an incomplete set** of legitimate targets at **compile-time**
  - **merge** this set by using initializer functions **at load time**
  - **validate** runtime target against this set **at runtime**

- Pro:
  - support incremental building

- Con:
  - heavy runtime operation, i.e., hash table lookups

Source Level Defense: **RockJIT**

- CCS' 15, CFI-based
  - collect **type information** at **compile-time**
  - compute **equivalence classes** of transfer targets at **load time**, based on the collected type information.
  - update the **CFI checks** to only allow indirect transfers (including virtual calls) to one equivalence class at **load-time**
- Pro:
  - support incremental building
- Con:
  - heavy load time operations

? &#