COMP 737011 - Memory Safety and Programming Language Design

Lecture 3: Heap Attack and Protection

Hui Xu

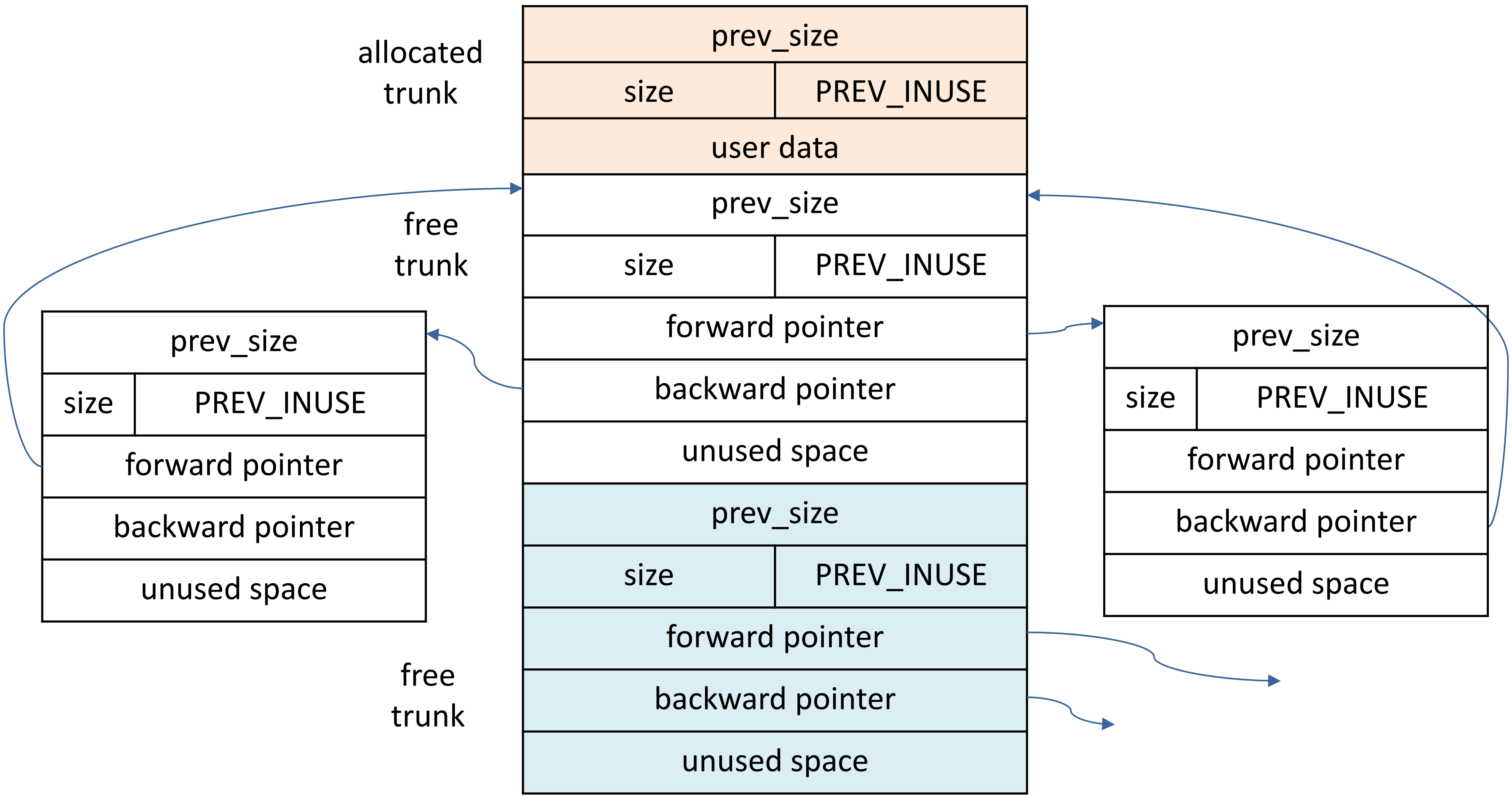
xuh@fudan.edu.cn

Outline

- 1. Heap Analysis
- 2. Heap Attack
- 3. Protection Techniques

1. Heap Analysis

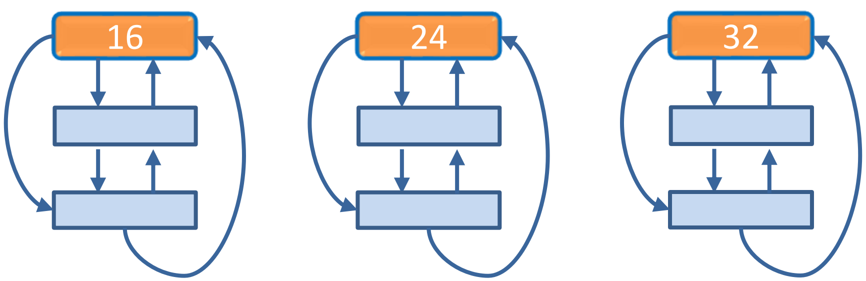
Recall: Chunk Structure



Recall: Doug Lea's Allocator

- Freed memory chunks are managed as bins
  - Regular bins for sizes < 512 bytes are spaced 8 bytes apart
  - Larger bins are approximately logarithmically spaced
- The detailed implementations could vary among allocators

	list	**coalesce**	**data**
Fast bin	single-linked	no	small
Regular bin	double-linked	may	could be large



Analyze The Program with GEF

- How many chunks will be allocated?
- What happens to the bins?
- Use the GEF (GDB Enhanced Features) tool for analysis
  - https://hugsy.github.io/gef/

break 1

break 2

```
int main(int argc, char** argv) {
    char *p[10];
    for(int i=0; i<10; i++){
        p[i] = malloc (10 * (i+1));
        strcpy(p[i], "nowar!!!");
    }

    for(int i=0; i<10; i++){
        free(p[i]);
    }
    return 0;
}
```

Disassemble

break 1

break 2

```
gef➤  disass main
Dump of assembler code for function main:
...
0x..1189 <+41>:  movsxd rdi,eax
0x..118c <+44>:  call   0x1050 <malloc@plt>
0x..1191 <+49>:  mov   rcx,rax
0x..1194 <+52>:  movsxd rax,DWORD PTR [rbp-0x64]
0x..1198 <+56>:  mov   QWORD PTR [rbp+rax*8-0x60],rcx
0x..119d <+61>:  movsxd rax,DWORD PTR [rbp-0x64]
0x..11a1 <+65>:  mov   rdi,QWORD PTR [rbp+rax*8-0x60]
0x..11a6 <+70>:  lea   rsi,[rip+0xe57]    # 0x2004
0x..11ad <+77>:  call   0x1040 <strcpy@plt>
0x..11b2 <+82>:  mov   eax,DWORD PTR [rbp-0x64]
...
0x..11c0 <+96>:  mov   DWORD PTR [rbp-0x68],0x0
0x..11c7 <+103>:   cmp   DWORD PTR [rbp-0x68],0xa
0x..11cb <+107>:   jge   0x11ed <main+141>
0x..11d1 <+113>:   movsxd rax,DWORD PTR [rbp-0x68]
0x..11d5 <+117>:   mov   rdi,QWORD PTR [rbp+rax*8-0x60]
0x..11da <+122>:   call   0x1030 <free@plt>
...
```

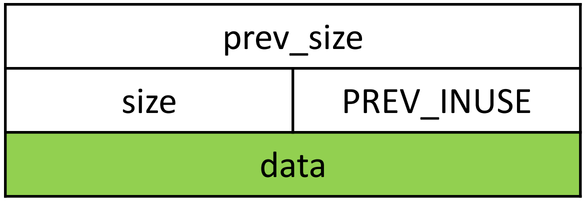
Check the Allocated Chunk

```
gef➤  break *main+82
Breakpoint 1 at 0x401191
gef➤  r
gef➤  search-pattern nowar
[+] Searching 'nowar' in memory
[+] In '/home/aisr/memory_safety/3-
heapattack/a.out'(0x555555556000-0x555555557000), permission=r--
  0x555555556004 - 0x55555555600c  →   "nowar!!!"
[+] In '/home/aisr/memory_safety/3-
heapattack/a.out'(0x555555557000-0x555555558000), permission=r--
  0x555555557004 - 0x55555555700c  →   "nowar!!!"
[+] In '[heap]'(0x555555559000-0x55555557a000), permission=rw-
  0x5555555592a0 - 0x5555555592a8  →   "nowar!!!"
gef➤  n
gef➤  search-pattern nowar
...
[+] In '[heap]'(0x555555559000-0x55555557a000), permission=rw-
  0x5555555592a0 - 0x5555555592a8  →   "nowar!!!"
  0x5555555592c0 - 0x5555555592c8  →   "nowar!!!"
```

Check the Allocated Chunk

```
gef➤  x/30b 0x555555559290
0x555555559290: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x555555559298: 0x21   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x5555555592a0: 0x6e   0x6f   0x77   0x61   0x72   0x21   0x21   0x21
0x5555555592a8: 0x00   0x00   0x00   0x00   0x00   0x00
```

- chunk size: 0x20
- previous in use: 1



- The chunk size is 32 bytes, including the header fields.
- If the previous chunk is in use, the prev_size filed can be used to store data of the previous trunk

View The Chunks

```
gef➤  heap chunks
Chunk(addr=0x555555559010, size=0x290, flags=PREV_INUSE)
    [0x0000555555559010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)
    [0x00005555555592a0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)
    [0x00005555555592c0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555592e0, size=0x20d30, flags=PREV_INUSE)
    [0x00005555555592e0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x5555555592e0, size=0x20d30, flags=PREV_INUSE)  ←  top chunk
```

- The chunk sizes are both 0x20 for the first two malloc
- 16 bytes spaced apart

After Several Iterations

```
gef➤  heap chunks
Chunk(addr=0x555555559010, size=0x290, flags=PREV_INUSE)
    [0x0000555555559010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)
    [0x00005555555592a0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)
    [0x00005555555592c0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555592e0, size=0x30, flags=PREV_INUSE)
    [0x00005555555592e0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559310, size=0x30, flags=PREV_INUSE)
    [0x0000555555559310     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559340, size=0x40, flags=PREV_INUSE)
    [0x0000555555559340     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559380, size=0x50, flags=PREV_INUSE)
    [0x0000555555559380     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555593d0, size=0x50, flags=PREV_INUSE)
    [0x00005555555593d0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559420, size=0x60, flags=PREV_INUSE)
    [0x0000555555559420     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559480, size=0x70, flags=PREV_INUSE)
    [0x0000555555559480     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x5555555594f0, size=0x70, flags=PREV_INUSE)
    [0x00005555555594f0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x555555559560, size=0x20ab0, flags=PREV_INUSE)
    [0x0000555555559560     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x555555559560, size=0x20ab0, flags=PREV_INUSE)  ←  top chunk
```

View The Bins (tcachebins)

```
gef➤  heap bins
────────────────────────────────────── Tcachebins for thread 1 ──────────────────────────────────────
All tcachebins are empty
─────────────────────────────── Fastbins for arena at 0x7ffff7facb80 ───────────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70] 0x00
Fastbins[idx=6, size=0x80] 0x00
───────────────────────────── Unsorted Bin for arena at 0x7ffff7facb80 ─────────────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────────────── Small Bins for arena at 0x7ffff7facb80 ──────────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────────────── Large Bins for arena at 0x7ffff7facb80 ──────────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

- Freed chunks will be added to tcachebins (new in libc 2.6)

View The Bins (tcachebins)

- Freed chunks after several iterations.

```
gef➤  heap bins
───────── Tcachebins for thread 1 ──────────────────────
Tcachebins[idx=0, size=0x20, count=2]
←  Chunk(addr=0x5555555592c0, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x5555555592a0, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=1, size=0x30, count=2]
←  Chunk(addr=0x555555559310, size=0x30, flags=PREV_INUSE)
←  Chunk(addr=0x5555555592e0, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40, count=1]
←  Chunk(addr=0x555555559340, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x50, count=2]
←  Chunk(addr=0x5555555593d0, size=0x50, flags=PREV_INUSE)
←  Chunk(addr=0x555555559380, size=0x50, flags=PREV_INUSE)
Tcachebins[idx=4, size=0x60, count=1]
←  Chunk(addr=0x555555559420, size=0x60, flags=PREV_INUSE)
```

<table>
<tr><td colspan="2">prev_size</td></tr>
<tr><td>size</td><td>PREV_INUSE</td></tr>
<tr><td colspan="2">forward pointer (data)</td></tr>
<tr><td colspan="2">backward pointer (optional)</td></tr>
<tr><td colspan="2">unused space</td></tr>
</table>

View The Freed Chunks in tcachebins

```
gef➤  x/200xb 0x555555559290
0x555555559290: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559298: 0x21    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592a0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592a8: 0x10    0x90    0x55    0x55    0x55    0x55    0x00    0x00
0x5555555592b0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592b8: 0x21    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592c0: 0xa0    0x92    0x55    0x55    0x55    0x55    0x00    0x00
0x5555555592c8: 0x10    0x90    0x55    0x55    0x55    0x55    0x00    0x00
0x5555555592d0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592d8: 0x31    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592e0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592e8: 0x10    0x90    0x55    0x55    0x55    0x55    0x00    0x00
0x5555555592f0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x5555555592f8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559300: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559308: 0x31    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559310: 0xe0    0x92    0x55    0x55    0x55    0x55    0x00    0x00
0x555555559318: 0x10    0x90    0x55    0x55    0x55    0x55    0x00    0x00
0x555555559320: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559328: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559330: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559338: 0x41    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559340: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x555555559348: 0x10    0x90    0x55    0x55    0x55    0x55    0x00    0x00
0x555555559350: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Summarization of Allocation Behaviors

- The first malloc reserves a large chunk (32KB)
  - The first 0x290 bytes used for bin management
  - The following mallocs obtain trunks from the reserved trunk.
- Freed chunks are added to tcachebins
  - Single-linked list, first-in-last-out
  - Max length of the list in each bin: 7
- Exceeding chunks will be put into fastbins

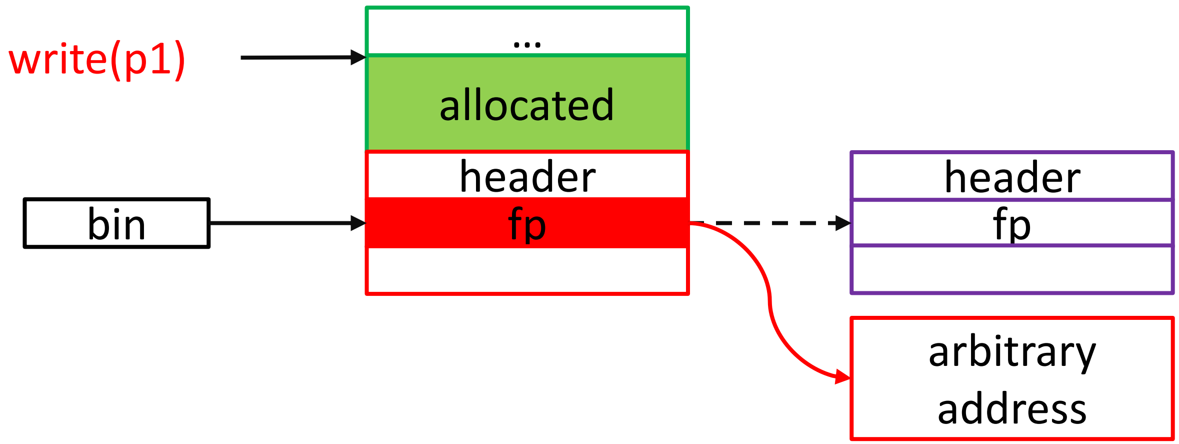
2. Heap Attack

Heap Vulnerabilities

- Heap overflow
- Use after free
- Double free

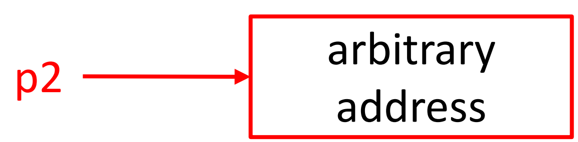
Heap Overflow

Step1: modify the fp of the next chunk to an arbitrary address



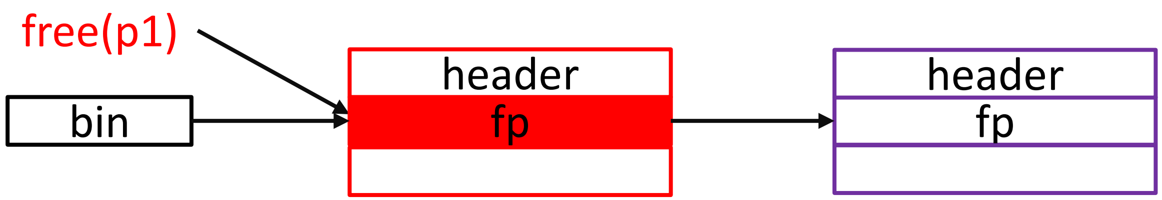
Step2: allocate the next chunk via malloc()

Step3: call malloc() again



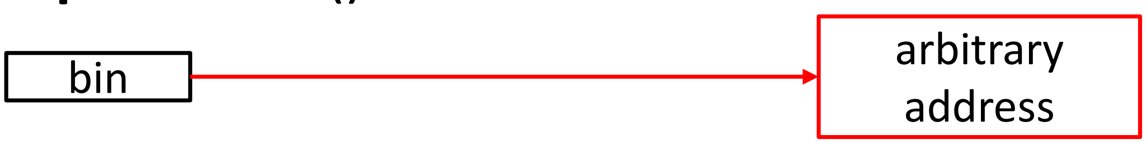
Use After Free

Step1: free(p1)

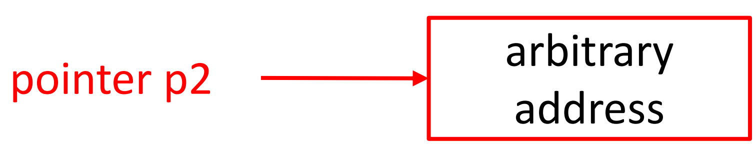


Step2: modify fp to an arbitrary address

Step3: malloc()



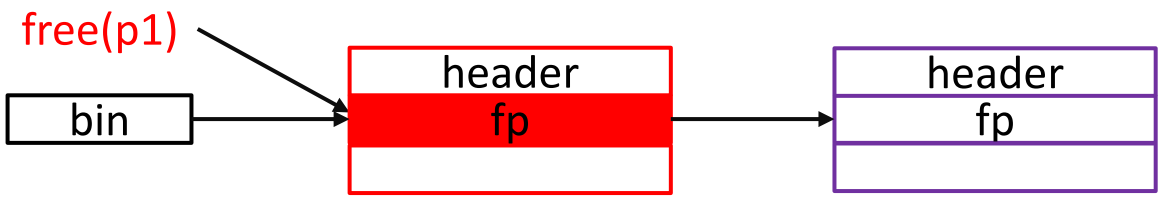
Step4: malloc() again to obtain a pointer to the arbitrary address



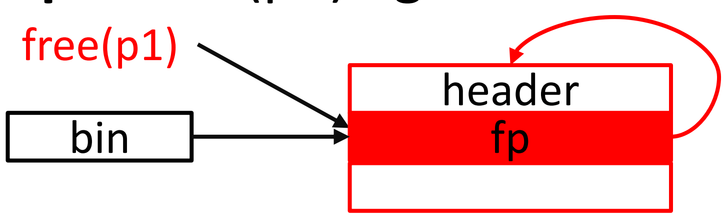
Double Free

```
//free(new):
first = header->next
header->next = new
new->next = first
```

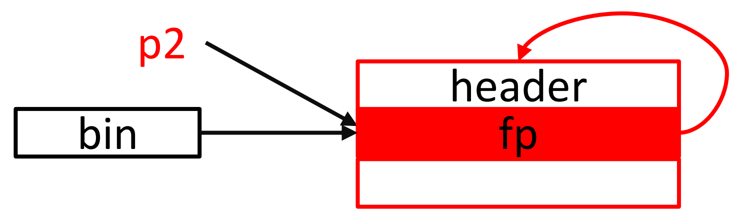
Step1: free(p1)



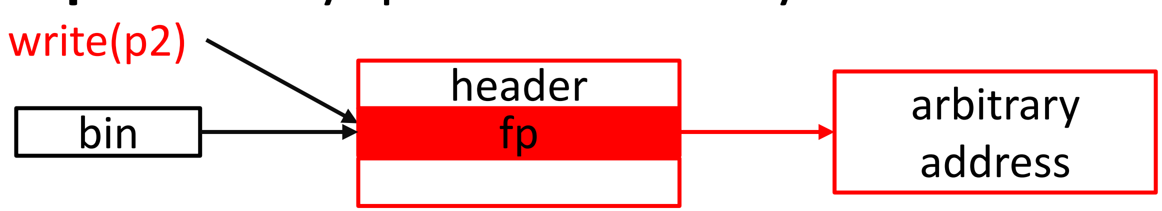
Step2: free(p1) again



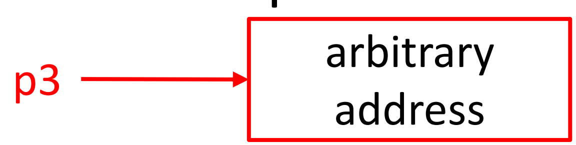
Step3: call malloc()



Step4: modify fp to an arbitrary address



Step5: malloc() twice to obtain a pointer to the arbitrary address



Address of Attacking Interest

- Return Address:
  - similar as buffer overflow
- Global Offset Table (GOT):
  - a table for dynamic linkage or position-independent code
  - change the table entries, *e.g.,* address of strcpy()
- Virtual Method Table (vtable):
  - abstract functions of C++/Rust

3. Protection Techniques

Detect Bugs in Allocator?

- Use static analysis or dynamic analysis?
- Detect invalid behaviors during malloc/free?
  - Chunk addresses should within the valid range?
  - A free chunk should not be freed again?
  - More fine-grained strategies?
- Detect invalid behaviors during read/write?
  - Overhead issues
- Increase the difficulty of heap attack?

Static Analysis Is Hard

- The fundamental point-to/alias analysis is NP-hard
- Several typical performance issues to consider
  - Flow-sensitivity: consider the order of statements
  - Path-sensitivity: analyze the result for each path
  - Context-sensitivity: inter-procedural issues
  - Field-sensitivity: how to model the members of objects
- Related papers:
  - Lee, *et al*. "Preventing Use-after-free with Dangling Pointers Nullification." NDSS 2015.
  - Van Der Kouwe, *et al*. "Dangsan: Scalable use-after-free detection." EuroSys 2017.
- We will have a class for the topic

Dynamic Approach Is Expensive

- Runtime detection mechanisms are needed
  - E.g., offset could be used => boundary check
- Trade-off between security and efficiency
- Mechanisms used in current allocators
  - alignment check
  - fasttop
  - canary

Alignment Check: Invalid Pointer Detection

- The following code is used within the function _int_free()
- Free a misaligned chunk is invalid

```
#define CHUNK_HDR_SZ (2 * SIZE_SZ) // 2 * size_t, 16 byte in x86-64
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define misaligned_chunk(p) \
    ((uintptr_t)(MALLOC_ALIGNMENT == CHUNK_HDR_SZ ?
         (p) : chunk2mem (p)) & MALLOC_ALIGN_MASK)

/* Little security check which won't hurt performance: the
   allocator never wrapps around at the end of the address space.
   Therefore we can exclude some size values which might appear
   here by accident or by "design" from some intruder.  */
  if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
    || __builtin_expect (misaligned_chunk (p), 0))
  malloc_printerr ("free(): invalid pointer");
```

https://github.com/bminor/glibc/blob/master/malloc/malloc.c

Fasttop: Double Free Detection

- Fasttop: pointer address should not be just freed
- Also used in the function of _int_free()

```
unsigned int idx = fastbin_index(size);
mfastbinptr fb = &fastbin (av, idx); //av is the malloc_state
mchunkptr old = *fb;
if (__builtin_expect (old == p, 0))
  malloc_printerr ("double free or corruption (fasttop)");
```

Canary (tcache_key): Double Free Detection

- Used only when USE_TCACHE is enabled
- Call tcache_put() in _init_malloc() to store the key

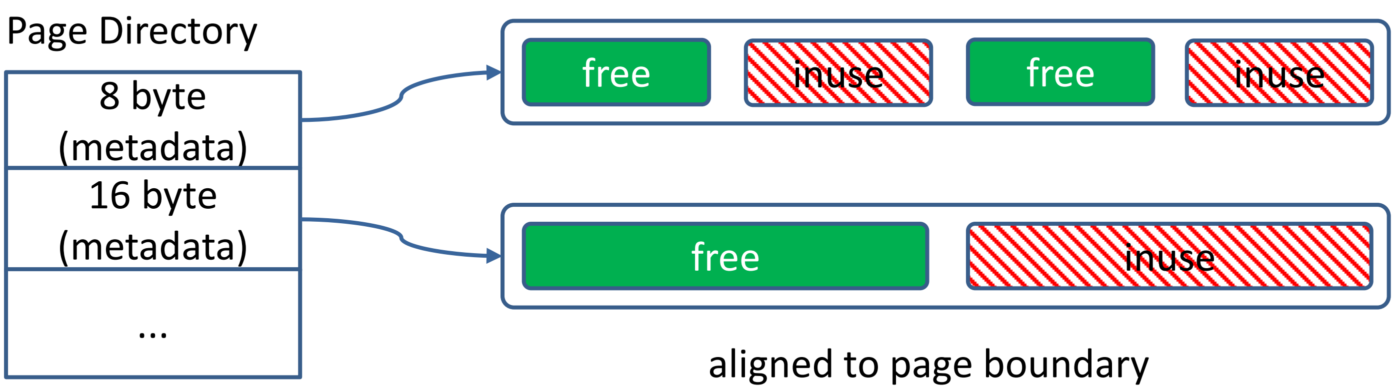
```
typedef struct tcache_entry {
  struct tcache_entry *next;
  uintptr_t key; //double free flag
} tcache_entry;

tcache_put (mchunkptr chunk, size_t tc_idx){
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  e->key = tcache_key;
  ...
}
```

- Check if content is still the key in the function of _int_free()

```
if (__glibc_unlikely (e->key == tcache_key)) {
   ...//probe the issue
}
```

More Approaches: BiBOP-Style Heap

- Big Bag of Pages:
  - contiguous areas of a multiple page size
  - each page has the same sized chunks
  - store heap metadata out-of-band (more secure)
- Originally proposed in PHKmalloc (OpenBSD)



https://papers.freebsd.org/1998/phk-malloc.files/phk-malloc-paper.pdf

More Papers to Read

- Berger, et al. "DieHard, Probabilistic memory safety for unsafe languages." *PLDI*, 2006.
- Novark, et al. "DieHarder: securing the heap." *CCS*, 2010.
- Akritidis. "Cling: A memory allocator to mitigate dangling pointers." *USENIX Security*, 2010.
- Sam, *et al*. "Freeguard: A faster secure heap allocator." *CCS,* 2017.

Programming Language Design

- Rust ownership-based mechanism
  - prohibit shared mutable aliases
  - no dangling pointer => preventing use after free, double free
- Shared mutable aliases should be wrapped with RC type
  - similar to shared_ptr in C++
- We will have a class for the topic

In Class Practice

- Write a C program with one of the following bugs and show how you can manipulate the free list with the bug.
  - Heap overflow
  - Use after free
  - Double free
- Hint:
  - Use the GEP tool to probe the trunks
  - You may encounter some detection techniques for double free

Solution

Solution: Use After Free

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
{
  char* p1 = malloc (22);
  char* p2 = malloc (22);
  free(p2);
  free(p1);
  *(int *) p1 = 0x411112;
  p1 = malloc(22);
  p2 = malloc(22);
  printf("Allocated memory address: %x\n", p2);
}
```

Solution: Double Free

```
void main(void)
{
  char* p1 = malloc (22);
  free(p1);
  p1[9] = 0x0; //overwrite e-key for double check
  free(p1);
  *(int *) p1 = 0x411112;
  p1 = malloc(22);
  p1 = malloc(22);
  printf("Allocated memory address: %x\n", p1);
}
```