COMP 737011 - Memory Safety and Programming Language Design

Lecture 3: Heap Attack and Protection

徐 辉

xuh@fudan.edu.cn

Outline

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

1. Heap Analysis

Target Program

- How many chunks will be allocated?
- What happens to the bins?

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

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

Analyze the Program with GEF

- Install GEF (GDB Enhanced Features):
  - https://gef.readthedocs.io/en/master/
- Add two break points

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

            for(int i=0; i<10; i++)
break 2         free(p[i]);
        }
```

Checkout What Happens

- Recall the structure of chunks

prev_size	
size	PREV_INUSE
forward pointer	
unused space	
size	

```
gef➤  break *main+65
Breakpoint 1 at 0x401191
gef➤  r
gef➤  search-pattern nowar
[+] Searching 'nowar' in memory
…
[+] In '[heap]'(0x405000-0x426000),
  0x4052a0 - 0x4052a8  →   "nowar!!

gef➤  x/10g 0x405290
0x405290:       0x0     0x21
0x4052a0:       0x212121726177 6f6e      0x0
0x4052b0:       0x0     0x20d51
0x4052c0:       0x0     0x0
0x4052d0:       0x0     0x0
```

10-byte header for x86-64:
Header content: 0x21,
- chunk size: 0x20 byte
- previous in use: 1 (last byte)

More Chunks

- After several iterations…

```
gef➤  heap chunks
Chunk(addr=0x405010, size=0x290, flags=PREV_INUSE)
    [0x0000000000405010     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x4052a0, size=0x20, flags=PREV_INUSE)
    [0x00000000004052a0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x4052c0, size=0x20, flags=PREV_INUSE)
    [0x00000000004052c0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x4052e0, size=0x20, flags=PREV_INUSE)
    [0x00000000004052e0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405300, size=0x30, flags=PREV_INUSE)
    [0x0000000000405300     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405330, size=0x30, flags=PREV_INUSE)
    [0x0000000000405330     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405360, size=0x40, flags=PREV_INUSE)
    [0x0000000000405360     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x4053a0, size=0x50, flags=PREV_INUSE)
    [0x00000000004053a0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x4053f0, size=0x50, flags=PREV_INUSE)
    [0x00000000004053f0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405440, size=0x20bd0, flags=PREV_INUSE)
    [0x0000000000405440     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x405440, size=0x20bd0, flags=PREV_INUSE)  ←  top chunk
```

View the Bins (tcachebins)

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

```
gef➤  heap bins
───────── Tcachebins for thread 1 ──────────────────────
Tcachebins[idx=0, size=0x20, count=3] ←  Chunk(addr=0x4052e0, size=0x20, flags=PREV_INUSE)
←  Chunk(addr=0x4052c0, size=0x20, flags=PREV_INUSE)  ←  Chunk(addr=0x4052a0, size=0x20,
flags=PREV_INUSE)
Tcachebins[idx=1, size=0x30, count=2] ←  Chunk(addr=0x405330, size=0x30, flags=PREV_INUSE)
←  Chunk(addr=0x405300, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40, count=1] ←  Chunk(addr=0x405360, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x50, count=2] ←  Chunk(addr=0x4053f0, size=0x50, flags=PREV_INUSE)
←  Chunk(addr=0x4053a0, size=0x50, flags=PREV_INUSE)
Tcachebins[idx=4, size=0x60, count=1] ←  Chunk(addr=0x405440, size=0x60, flags=PREV_INUSE)
Tcachebins[idx=5, size=0x70, count=1] ←  Chunk(addr=0x4054a0, size=0x70, flags=PREV_INUSE)

───────── Fastbins for arena at 0x7ffff7fadb80 ───────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
...
```

Characteristics of Tcachebins

- Single-linked list
- First-in-last-out
- Max length of the list in each bin: 7
- Exceeding chunks will be put into fastbins

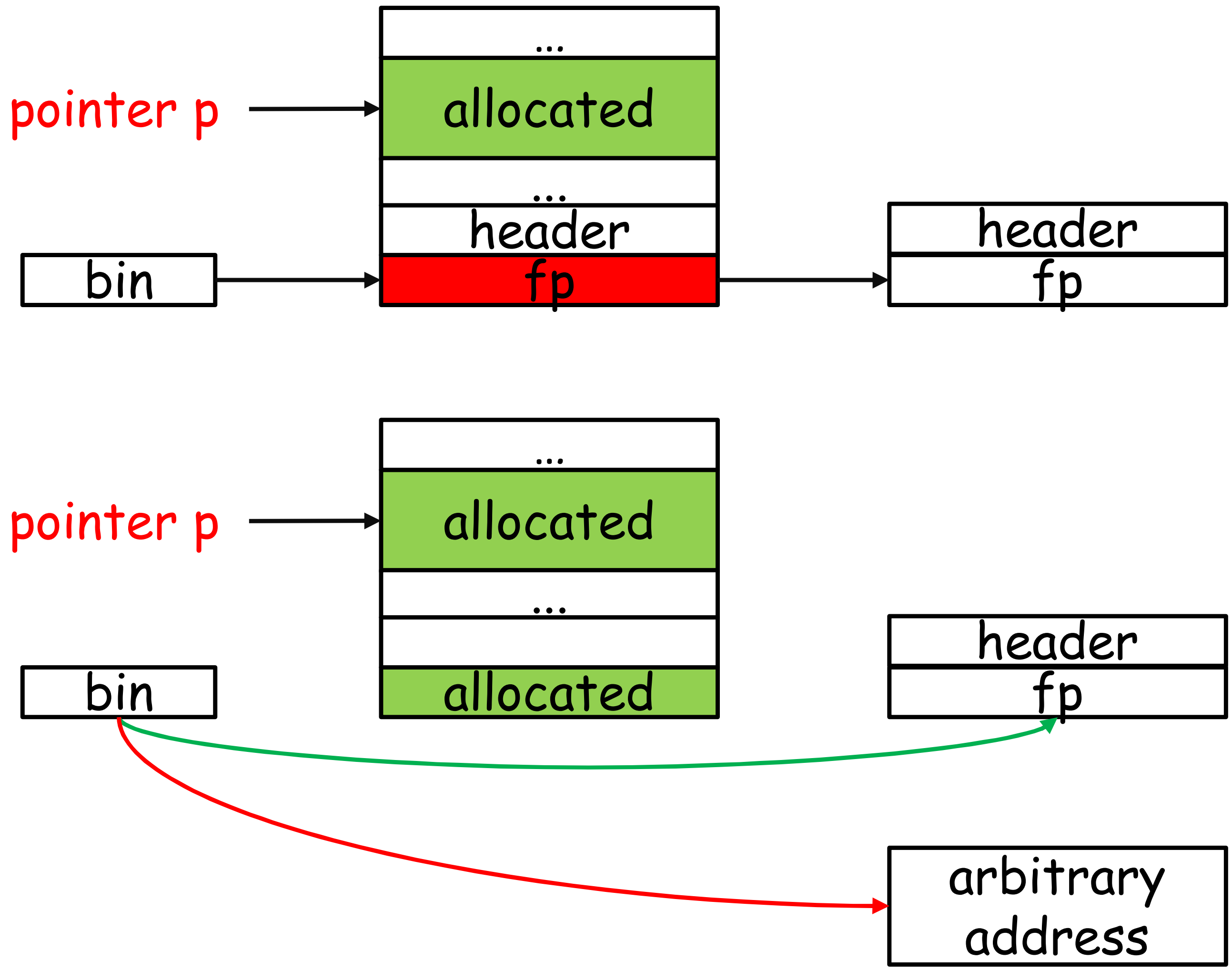
2. Heap Attack

Heap Vulnerablilities

- Heap overflow
- Use after free
- Double free

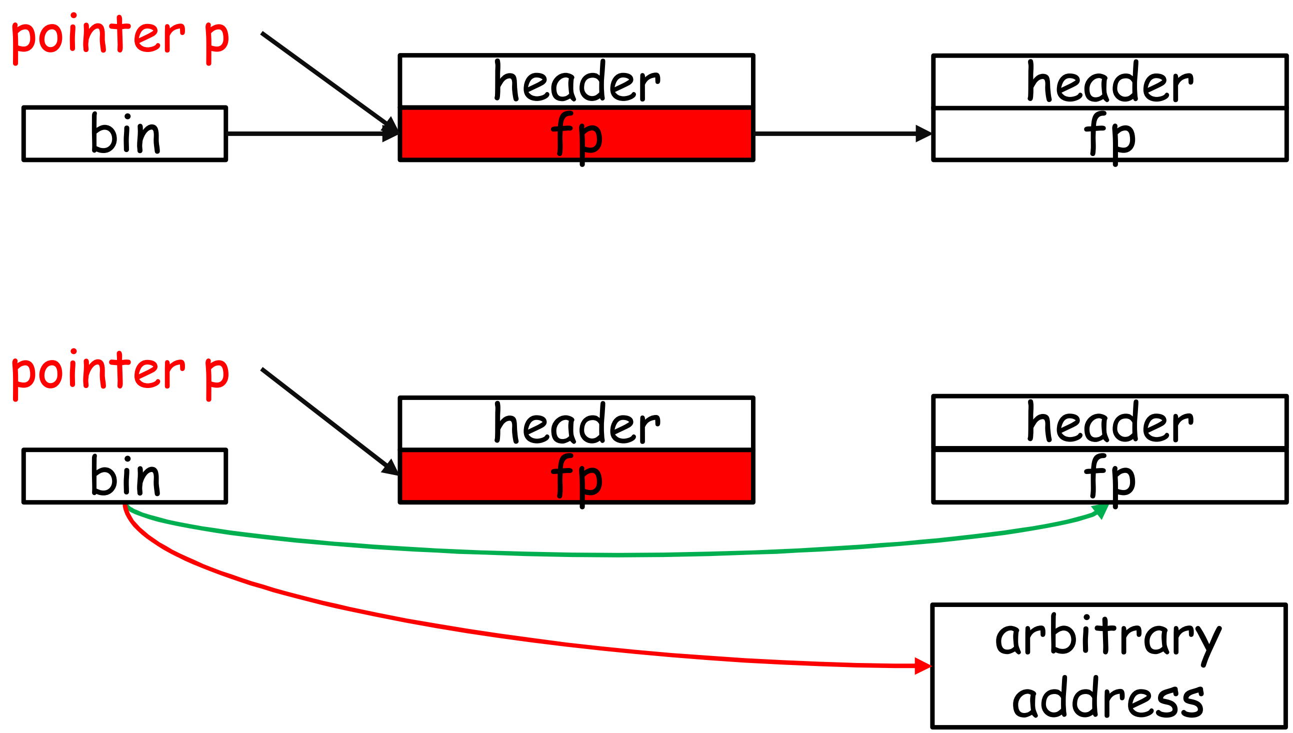
Heap Overflow

- Change the forward pointer of the top free chunk.
- What happens when allocating the chunk?



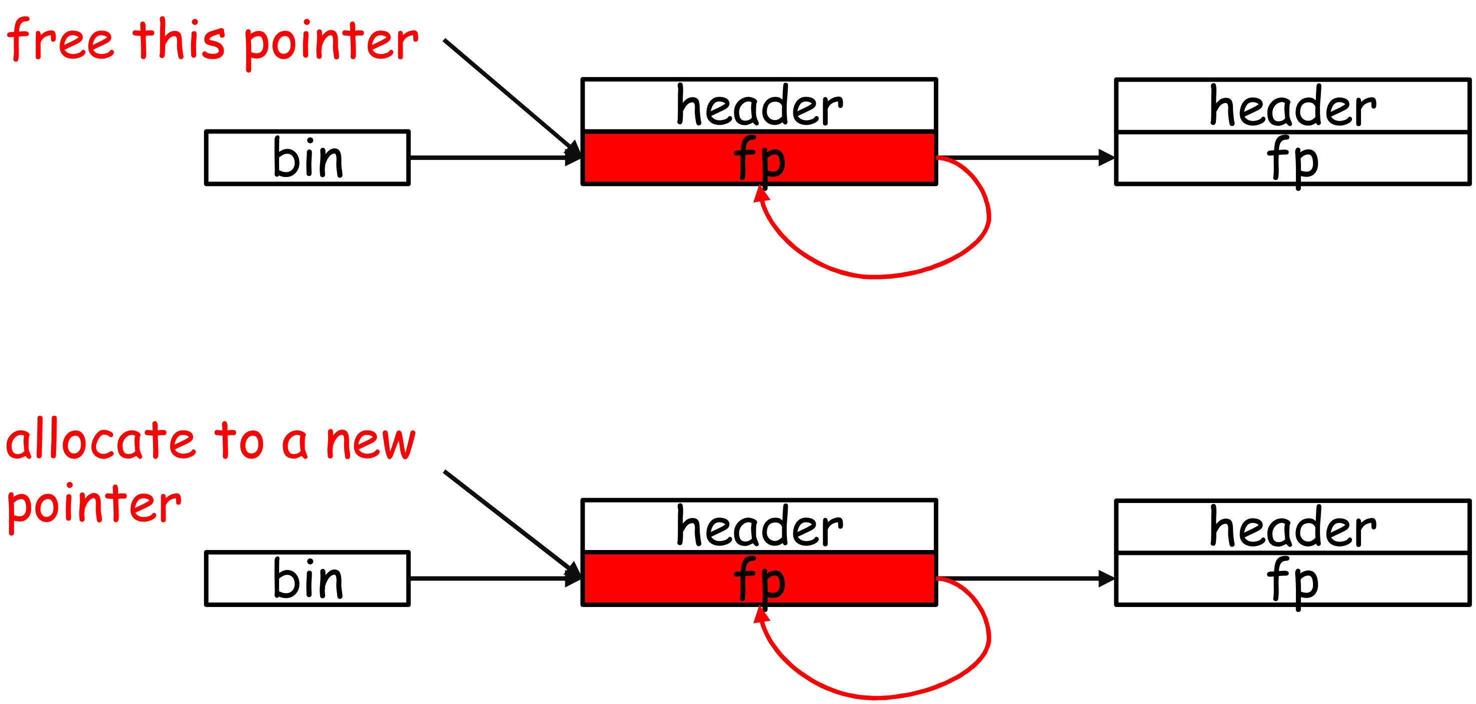
Use After Free Is Easier

- Directly change the forward pointer of the top free chunk



What About Double Free?

- The forward pointer points to the chunk itself
- After allocation, the chunk is still in the list



Address of Attacking Interest

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

In Class Practice

- Write a C program and demonstrate UAF
- Write a C program and demonstrate Double Free
  - You may encounter some detection techniques

3. Protection Techniques

Detect Bugs in Allocator?

- Detect invalid behaviors
  - invalid pointers in free()
  - double free in free()
  - read/write dangling/invalid pointers
  - broken free lists
- Prevent attack?
  - Increase the difficulty of heap manipulation

Invalid Pointer: Alignment Check

- Trade off between efficiency and resillience

```
#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)

#define chunksize(p) (chunksize_nomask (p) & ~(SIZE_BITS))
#define chunksize_nomask(p) ((p)->mchunk_size)
size = chunksize (p);

/* 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

Double Free

- Fasttop: pointer address should not be “just” freed
- key in tcachebin: double free flag

```
if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
```

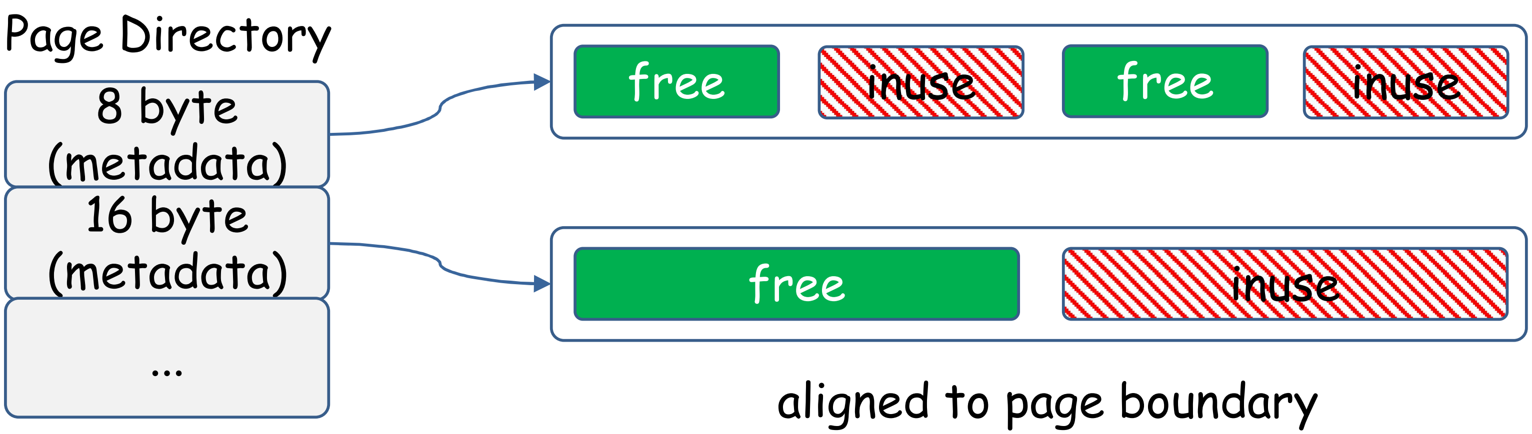
```
typedef struct tcache_entry {
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  uintptr_t key;
} tcache_entry;

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

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

RW Dangling/Invalid Pointer

- Detecting dangling/invalid pointer is difficult
  - offset could be used
  - difficult to determine whether an address is valid
- Classical approach to prevent attacks
  - PHKmalloc use in OpenBSD:
    - BiBOP-style heap or Big Bag of Pages
  - Sample 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.

BiBOP-Style Heap

- Store heap metadata out-of-band
- Use a bitmap to indicate free chunks
- Randminzed allocation: 1/M chance for each free chunk



Novark, et al. "DieHarder: securing the heap." CCS 2010.

Static Analysis

- Analyze whether a pointer being used is dangling
- Should infer the alias of pointers
  - The chunk could be freed via other variables
- General alise analysis problem 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
- We will have one class for the topic:
  - Sample papers to read
    - 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*.

Programming Language Design

- Rust ownership-based mechanism
- Preventing shared mutable aliases
- Shared mutable aliases should be wrapped with RC type
  - similar to shared_ptr in C++

```
let b = B:new();
let r1: &B = &b;
let r2: &B = &b;
```

Immutable borrow

b owns the object
r1 borrows the ownership immutably
r2 borrows the ownership immutably

```
let b = B:new();
let r1: &mut B = &mut b;
```

Mutable borrow

b owns the object
r1 borrows the ownership mutably
b temporily lost the ownership

More Reference

- https://guyinatuxedo.github.io
- https://doc.rust-lang.org/book/ch15-04-rc.html
- 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.
- 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.