COMP 737011 - Memory Safety and Programming Language Design

Lecture 3: Heap Attack and Protection

XU, Hui

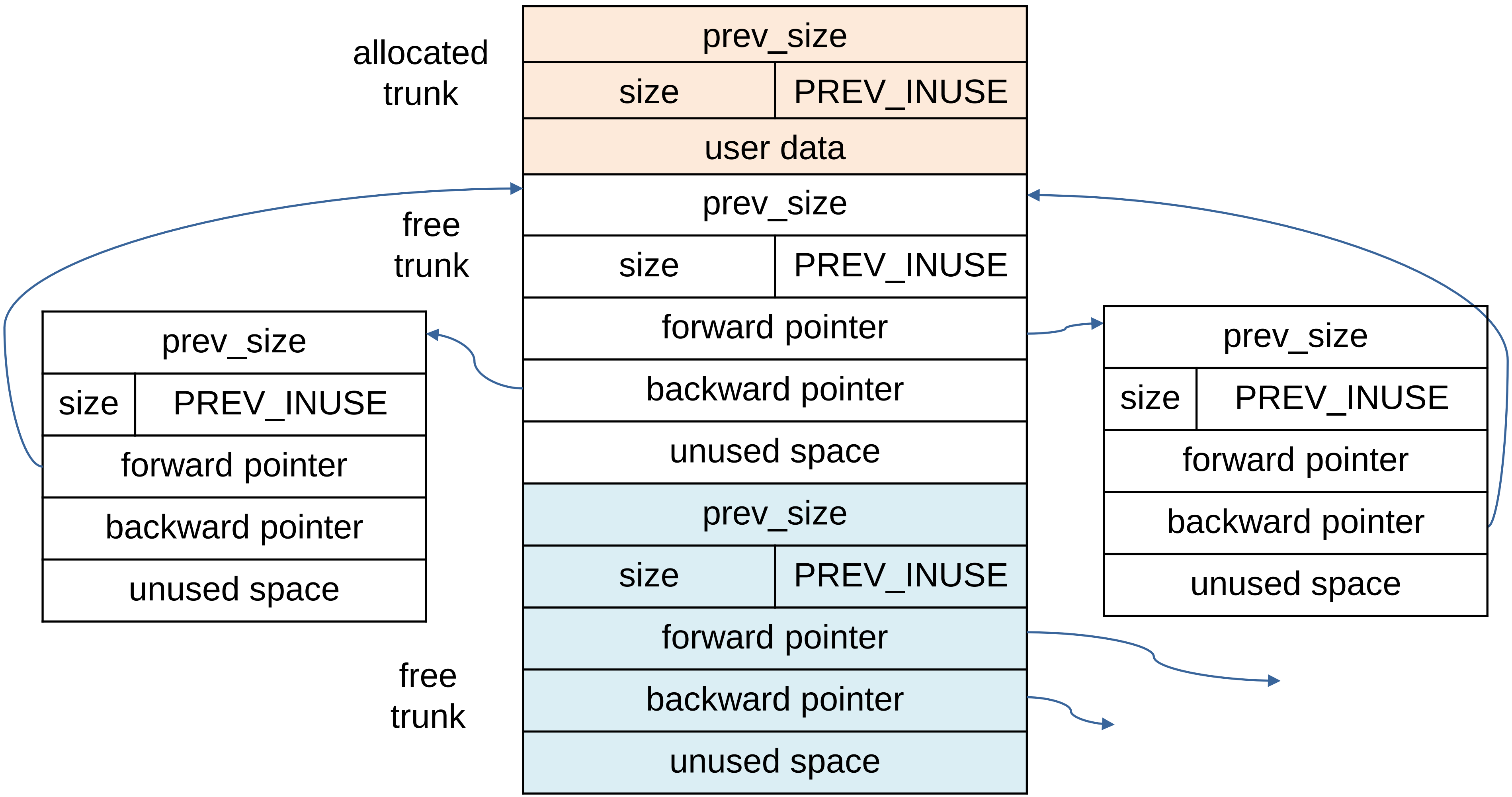
xuh@fudan.edu.cn

Outline

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

1. Heap Analysis

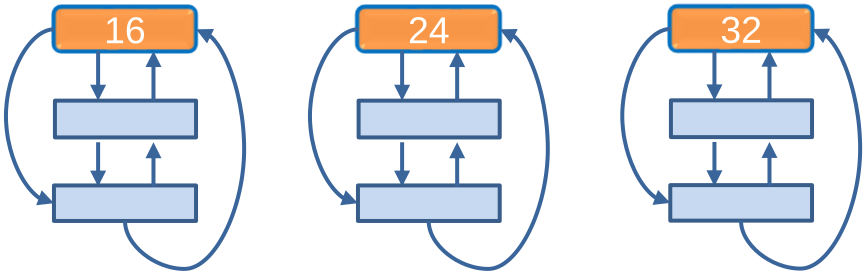
Recall: Chunk Structure



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 implemtations 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/

```
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;
}
```

break 1

break 2

Disassemble

```
gef➤  disass main
Dump of assembler code for function main:
   ...
   0x0000000000401179 <+41>:    movsxd rdi,eax
   0x000000000040117c <+44>:    call   0x401050 <malloc@plt>
   0x0000000000401181 <+49>:    movsxd rcx,DWORD PTR [rbp-0x64]
   0x0000000000401185 <+53>:    mov    QWORD PTR [rbp+rcx*8-0x60],rax
   0x000000000040118a <+58>:    movsxd rax,DWORD PTR [rbp-0x64]
   0x000000000040118e <+62>:    mov    rdi,QWORD PTR [rbp+rax*8-0x60]
   0x0000000000401193 <+67>:    mov    esi,0x402004
   0x0000000000401198 <+72>:    call   0x401040 <strcpy@plt>
break 1   0x000000000040119d <+77>:    mov    eax,DWORD PTR [rbp-0x64]
   0x00000000004011a0 <+80>:    add    eax,0x1
   0x00000000004011a3 <+83>:    mov    DWORD PTR [rbp-0x64],eax
   0x00000000004011a6 <+86>:    jmp    0x401166 <main+22>
   0x00000000004011ab <+91>:    mov    DWORD PTR [rbp-0x68],0x0
   0x00000000004011b2 <+98>:    cmp    DWORD PTR [rbp-0x68],0xa
   0x00000000004011b6 <+102>:   jge    0x4011d8 <main+136>
   0x00000000004011bc <+108>:   movsxd rax,DWORD PTR [rbp-0x68]
   0x00000000004011c0 <+112>:   mov    rdi,QWORD PTR [rbp+rax*8-0x60]
   0x00000000004011c5 <+117>:   call   0x401030 <free@plt>
break 2   0x00000000004011ca <+122>:   mov    eax,DWORD PTR [rbp-0x68]
   ...
```

Check the Allocate Trunk

```
gef➤  break *main+77
Breakpoint 1 at 0x401191
gef➤  r
gef➤  search-pattern nowar
[+] Searching 'nowar' in memory
…
[+] In '[heap]'(0x405000-0x426000), permission=rw-
  0x4052a0 - 0x4052a8  →  "nowar!!!

gef➤  x/10xb 0x405290
0x405290:   0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x405298:   0x21  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x4052a0:   0x6e  0x6f  0x77  0x61  0x72  0x21  0x21  0x21
0x4052a8:   0x00  0x00  0x00  0x00
```

prev_size	
size	PREV_INUSE
forward pointer (data)	
backward pointer (optional)	
unused space	

Header size: 16 bytes
- chunk size: 0x20
- previous in use: 1

- Minimal trunk size is 20
- If the previous chunk is in use, the prev_size filed can be used to store data of the previous trunk

View The Chunks

- Trunks created after several iteration.

```
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=0x30, flags=PREV_INUSE)
    [0x00000000004052e0     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405310, size=0x30, flags=PREV_INUSE)
    [0x0000000000405310     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405340, size=0x40, flags=PREV_INUSE)
    [0x0000000000405340     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x405380, size=0x50, flags=PREV_INUSE)
    [0x0000000000405380     6e 6f 77 61 72 21 21 21 00 00 00 00 00 00 00 00    nowar!!!........]
Chunk(addr=0x4053d0, size=0x20c40, flags=PREV_INUSE)
    [0x00000000004053d0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................]
Chunk(addr=0x4053d0, size=0x20c40, flags=PREV_INUSE)  ←  top chunk
```

- Trunk size is 0x20/0x30/0x40/0x50/0x60/0x70/0x80
- 16 bytes spaced apart

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=0x4052c0, size=0x20, flags=PREV_INUSE)  ←
Chunk(addr=0x4052a0, size=0x20, flags=PREV_INUSE)
Tcachebins[idx=1, size=0x30, count=2] ←  Chunk(addr=0x405310, size=0x30, flags=PREV_INUSE)  ←
Chunk(addr=0x4052e0, size=0x30, flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40, count=1] ←  Chunk(addr=0x405340, size=0x40, flags=PREV_INUSE)
Tcachebins[idx=3, size=0x50, count=2] ←  Chunk(addr=0x4053d0, size=0x50, flags=PREV_INUSE)  ←
Chunk(addr=0x405380, size=0x50, flags=PREV_INUSE)
Tcachebins[idx=4, size=0x60, count=1] ←  Chunk(addr=0x405420, size=0x60, flags=PREV_INUSE)
Tcachebins[idx=5, size=0x70, count=1] ←  Chunk(addr=0x405480, size=0x70, flags=PREV_INUSE)
─────────────────────────── Fastbins for arena at 0x7ffff7faeb80 ───────────────────────────
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 0x7ffff7faeb80 ─────────────────────────
[+] Found 0 chunks in unsorted bin.
────────────────────────── Small Bins for arena at 0x7ffff7faeb80 ──────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
────────────────────────── Large Bins for arena at 0x7ffff7faeb80 ──────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
```

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

View The Freed Chunks in tcachebins

- The previous size field is always 0 (not used)
- Only hold a forward pointer; no backward pointer needed
- Can you explain the reason?

```
gef➤  x/50xg 0x405290
0x405290:	0x0000000000000000	0x0000000000000021
0x4052a0:	0x0000000000000000	0x0000000000405010
0x4052b0:	0x0000000000000000	0x0000000000000021
0x4052c0:	0x00000000004052a0	0x0000000000405010
0x4052d0:	0x0000000000000000	0x0000000000000031
0x4052e0:	0x0000000000000000	0x0000000000405010
0x4052f0:	0x0000000000000000	0x0000000000000000
0x405300:	0x0000000000000000	0x0000000000000031
0x405310:	0x00000000004052e0	0x0000000000405010
0x405320:	0x0000000000000000	0x0000000000000000
0x405330:	0x0000000000000000	0x0000000000000041
0x405340:	0x0000000000000000	0x0000000000405010
0x405350:	0x0000000000000000	0x0000000000000000
0x405360:	0x0000000000000000	0x0000000000000000
0x405370:	0x0000000000000000	0x0000000000000051
0x405380:	0x0000000000000000	0x0000000000405010
0x405390:	0x0000000000000000	0x0000000000000000
0x4053a0:	0x0000000000000000	0x0000000000000000
0x4053b0:	0x0000000000000000	0x0000000000000000
0x4053c0:	0x0000000000000000	0x0000000000000051
0x4053d0:	0x0000000000405380	0x0000000000405010
0x4053e0:	0x0000000000000000	0x0000000000000000
0x4053f0:	0x0000000000000000	0x0000000000000000
0x405400:	0x0000000000000000	0x0000000000000000
0x405410:	0x0000000000000000	0x0000000000000061
```

Summarization of Allocation Behaviors

- The first malloc reserves a large trunk (32KB) at 0x405010
  - 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

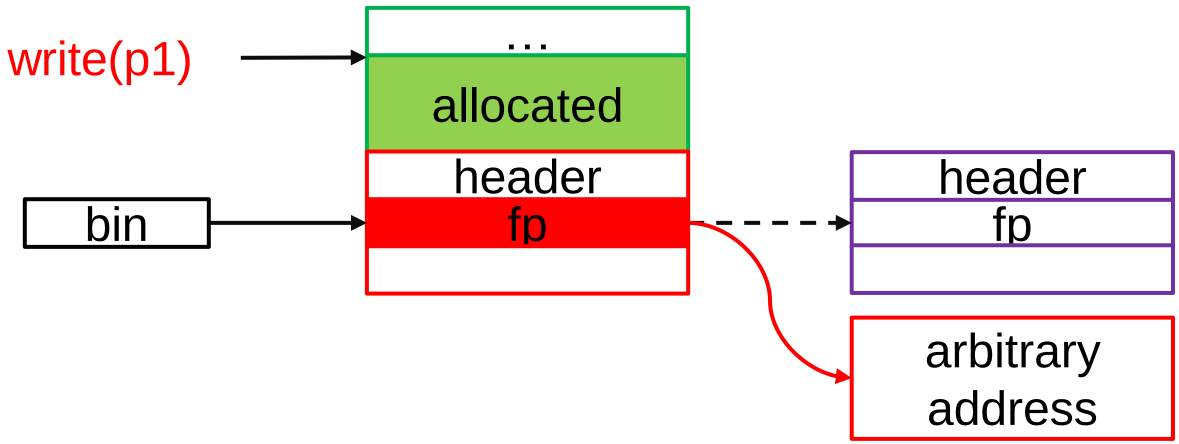
2. Heap Attack

Heap Vulnerablilities

- Heap overflow
- Use after free
- Double free

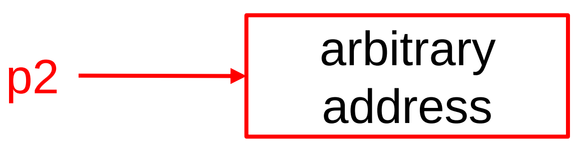
Heap Overflow

Step1: modify the fp of the nex trunk to an arbitrary address



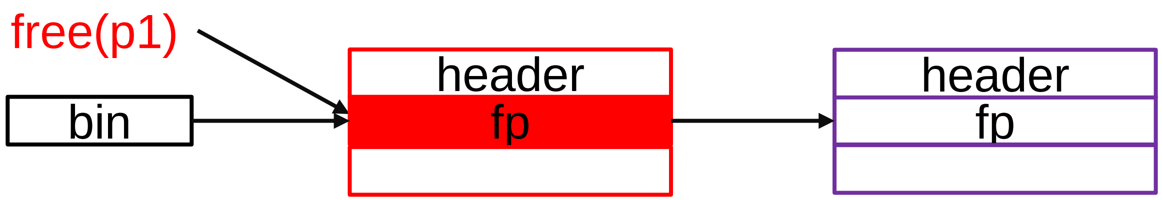
Step2: allocate the next trunk via malloc()

Step3: call malloc() again

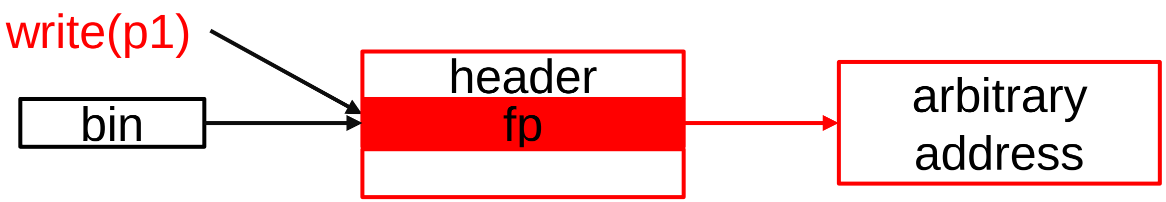


Use After Free

Step1: free(p1)

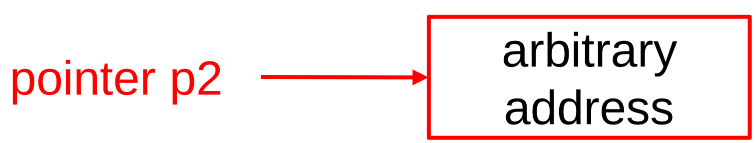


Step2: modify fp to an arbitrary address



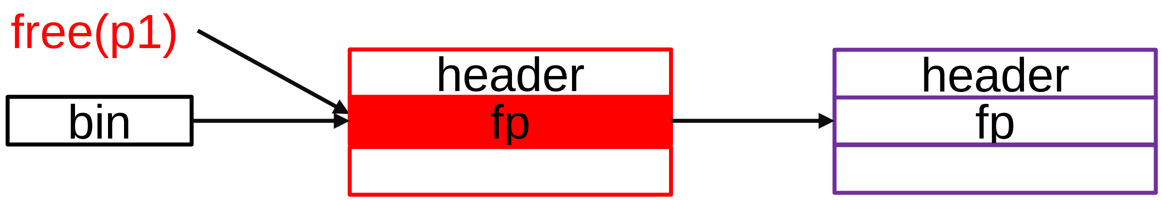
Step3: malloc()

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

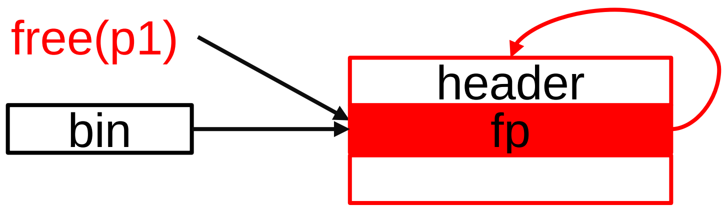


Double Free

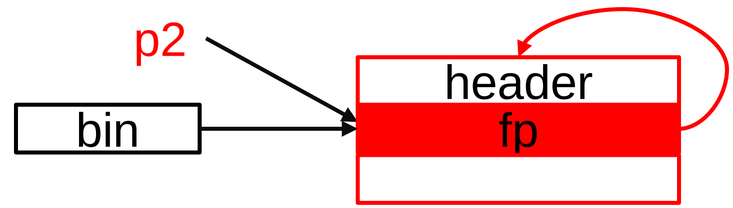
Step1: free(p1)



Step2: free(p1) again

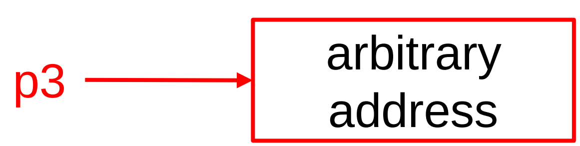


Step3: call malloc()



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-grined 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

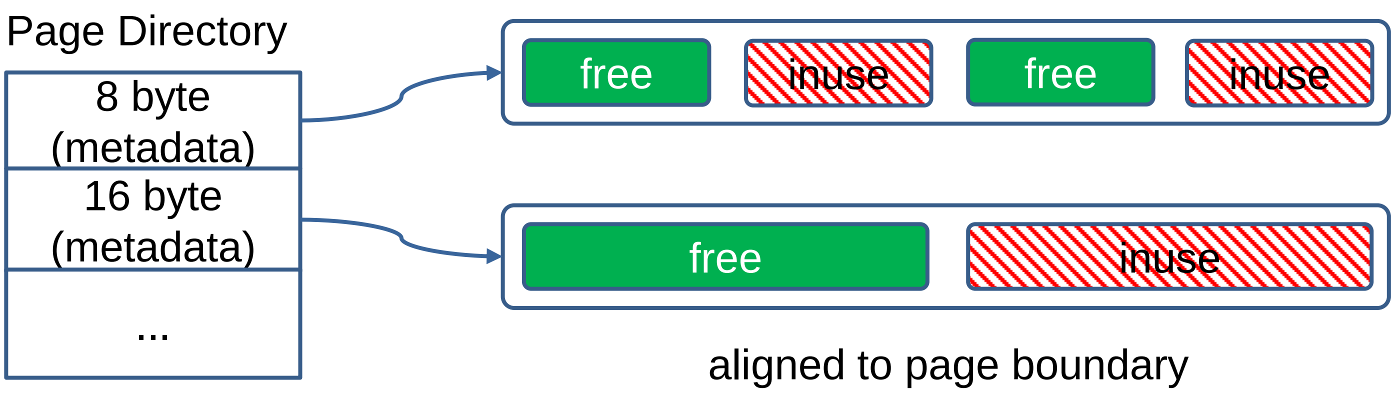
```
typedef struct tcache_entry {
  struct tcache_entry *next;
  uintptr_t key; //double free flag
} tcache_entry;

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);
}
```