Lecture 16: Heap Exploitation Practice II: Use-after-free, Double-free, and Heap-Overflow with Fastbin

Sanchuan Chen

schen@auburn.edu

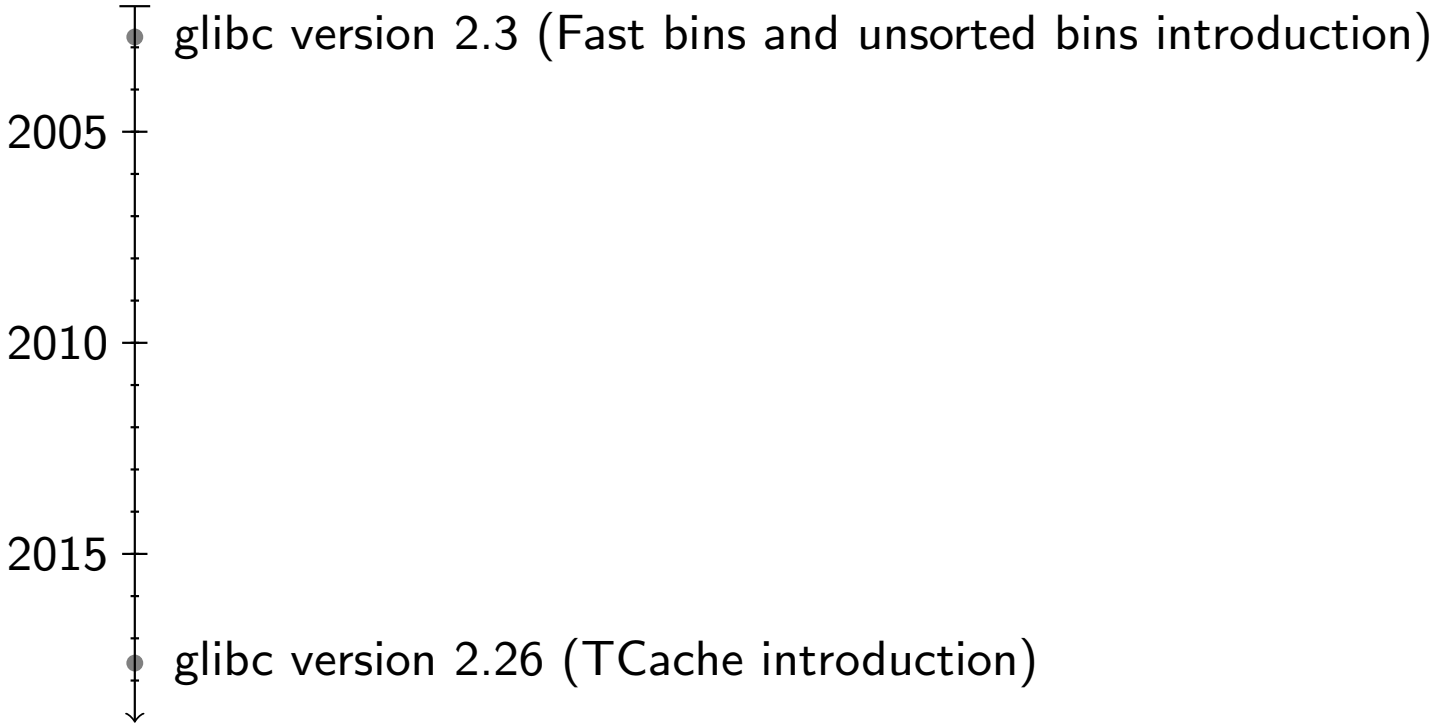
11/13/2023

Fast Bins, Small Bins, and Large Bins

ptmalloc2 [1, 2] has an arena for each thread and the arena for main thread is called main arena. Main arena stores in data segment of libc.so. The following is an example arena. The fields include `fastbinsY` for fast bins, `bins` for bins, and `system_mem`, etc. In the older version of glibc: dlmalloc, there is only one arena and will be synchronized when multiple threads access the arena.

```
pwndbg> arena
{
  mutex = 0,
  flags = 0,
  have_fastchunks = 0,
  fastbinsY = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  top = 0x22a9270,
  last_remainder = 0x0,
  bins = {0x7fc20030fca0 <main_arena+96>, 0x7fc20030fca0 <main_arena+96>...},
  binmap = {0, 0, 0, 0},
  next = 0x7fc20030fc40 <main_arena>,
  next_free = 0x0,
  attached_threads = 1,
  system_mem = 135168,
  max_system_mem = 135168
}
```

In glibc 2.27, there are generally five types of bins: TCache bins, fast bins, unsorted bins, small bins, and large bins.



Except for TCache bins, the other four types of bins are stored in data structure `arena`. In the data structure, `fastbinsY` is for fast bins, `bins` is for unsorted bins, small bins, and large bins.

TCache Bins

We can see in the following source code snippet, the maximum number of TCache bins is 64 and for each bin, the maximum number of chunks is 7. The same size of chunks will be placed in the same TCache bin.

```
malloc/malloc.c
303 /* We want 64 entries. This is an arbitrary limit, which tunables can reduce.*/
304 # define TCACHE_MAX_BINS        64
305 # define MAX_TCACHE_SIZE        tidx2usize (TCACHE_MAX_BINS-1)
```

Fast Bins

Fast bins are similar with TCache bins, in a sense that chunks of same size will be put in the same fast bins.

We can see from the following `pwntools` output that the fastbins has 7 bins by default.

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
```

Unsorted Bins, Small Bins, and Large Bins

There are 126 bins in arena→bins.

The 1st bin is **unsorted bin** and there is only one unsorted bin.

Unsorted bin is a transition point as chunks will be put in unsorted bin and later put in small bins or large bins.

Bin 2-63 (62 bins) are **small bins**, which are for chunks with size smaller than 1024 bytes. Similarly, small bins also stores chunks with same size in the same bin.

Bin Number	2	3	...	63
Chunk Size	0x20	0x30		0x3f0 (1008)

Bin 64-126 (63 bins) are **large bins**, which are for chunks with size over 1024 bytes.

Bin Number	64	65	...
Chunk Size Range	[0x400,0x440)	[0x440,0x480)	

Bin Number	95	96	...
Chunk Size Range	[0xbc0,0xc00)	[0xc00,0xe00)	

The following source code describes the size of each large bins:

```
1436 /*
1437    Indexing
1438
1439     Bins for sizes < 512 bytes contain chunks of all the same size, spaced
1440     8 bytes apart. Larger bins are approximately logarithmically spaced:
1441
1442     64 bins of size       8
1443     32 bins of size      64
1444     16 bins of size     512
1445      8 bins of size    4096
1446      4 bins of size   32768
1447      2 bins of size  262144
1448      1 bin  of size what's left
1449
1450     There is actually a little bit of slop in the numbers in bin_index
1451     for the sake of speed. This makes no difference elsewhere.
1452
1453     The bins top out around 1MB because we expect to service large
1454     requests via mmap.
1455
1456     Bin 0 does not exist.  Bin 1 is the unordered list; if that would be
1457     a valid chunk size the small bins are bumped up one.
1458  */
```

The allocated and free chunks overlay the following data structure `malloc_chunk`, as shown in the following source code. For different kinds of bins, some fields exist and some do not.

```
1060 struct malloc_chunk {
1061
1062   INTERNAL_SIZE_T  mchunk_prev_size; /* Size of previous chunk (if free). */
1063   INTERNAL_SIZE_T  mchunk_size;      /* Size in bytes, including overhead.*/
1064
1065   struct malloc_chunk* fd;           /* double links -- used only if free.*/
1066   struct malloc_chunk* bk;
1067
1068   /* Only used for large blocks: pointer to next larger size.  */
1069   struct malloc_chunk* fd_nextsize;  /* double links -- used only if free.*/
1070   struct malloc_chunk* bk_nextsize;
1071 };
```

Allocated and Freed Chunks in Different Bins

The following figure shows the allocated and freed chunk in **TCache bins**.

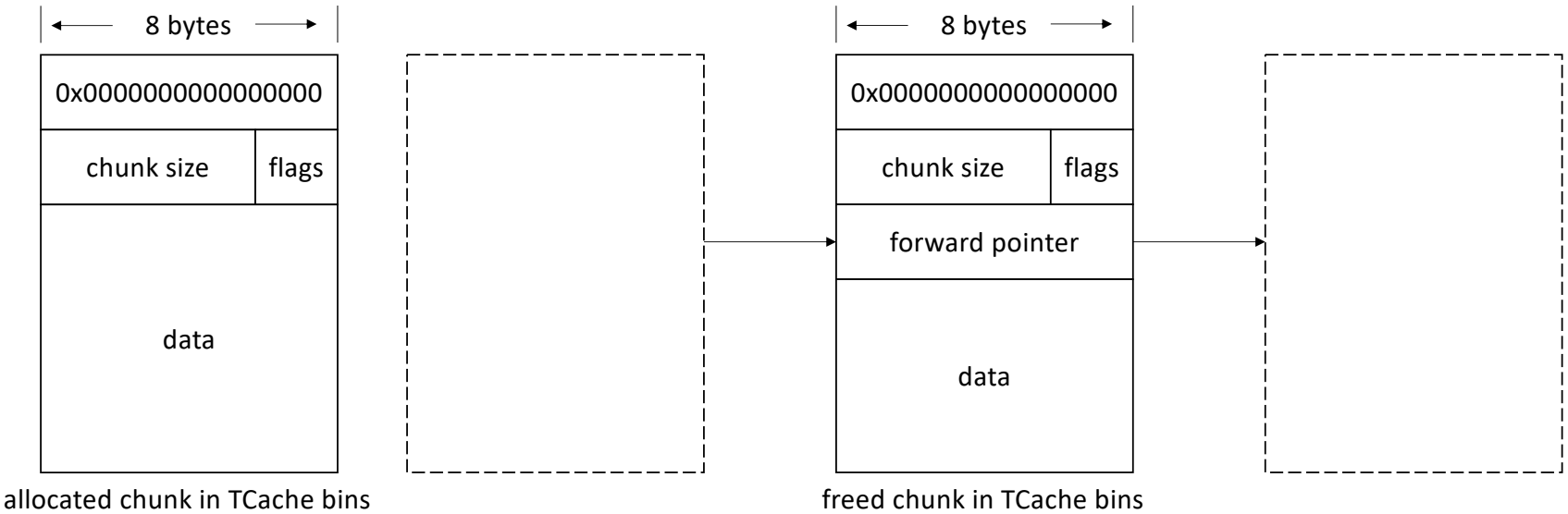


Figure: 1: Allocated chunks and freed chunks in TCache bins

Allocated and Freed Chunks in Different Bins

The following figure shows the allocated and freed chunk in **fast bins**.

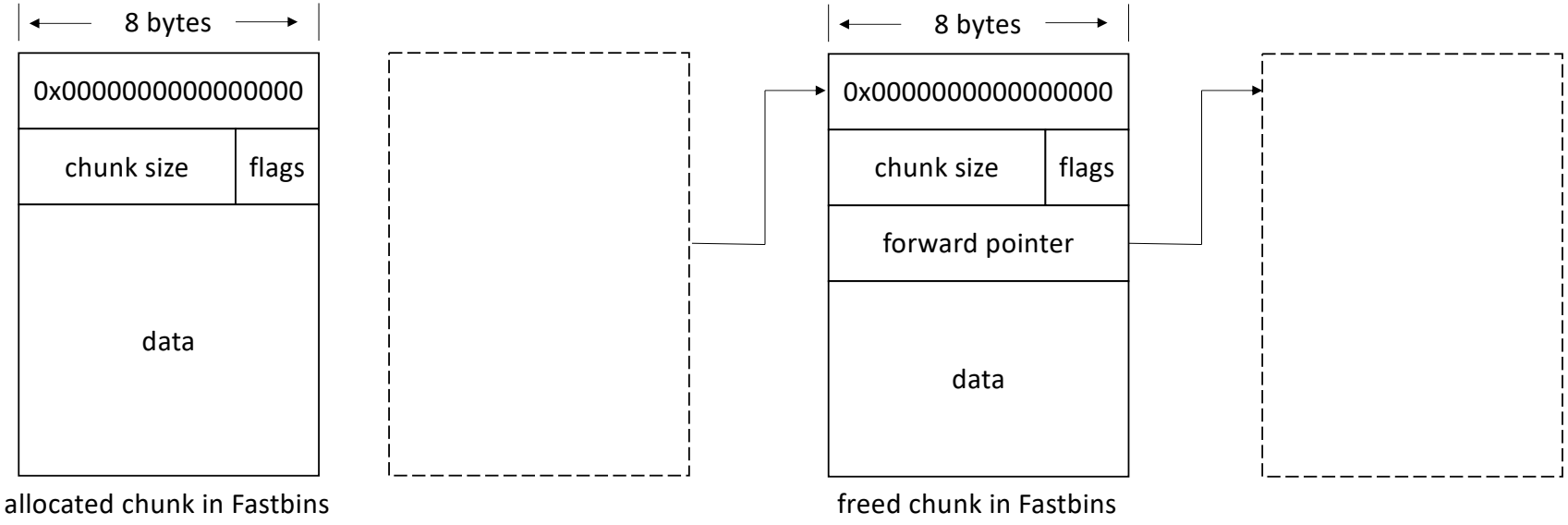


Figure: 2: Allocated chunks and freed chunks in fast bins

Allocated and Freed Chunks in Different Bins

The following figure shows the allocated and freed chunk in **unsorted bins** and **small bins**.

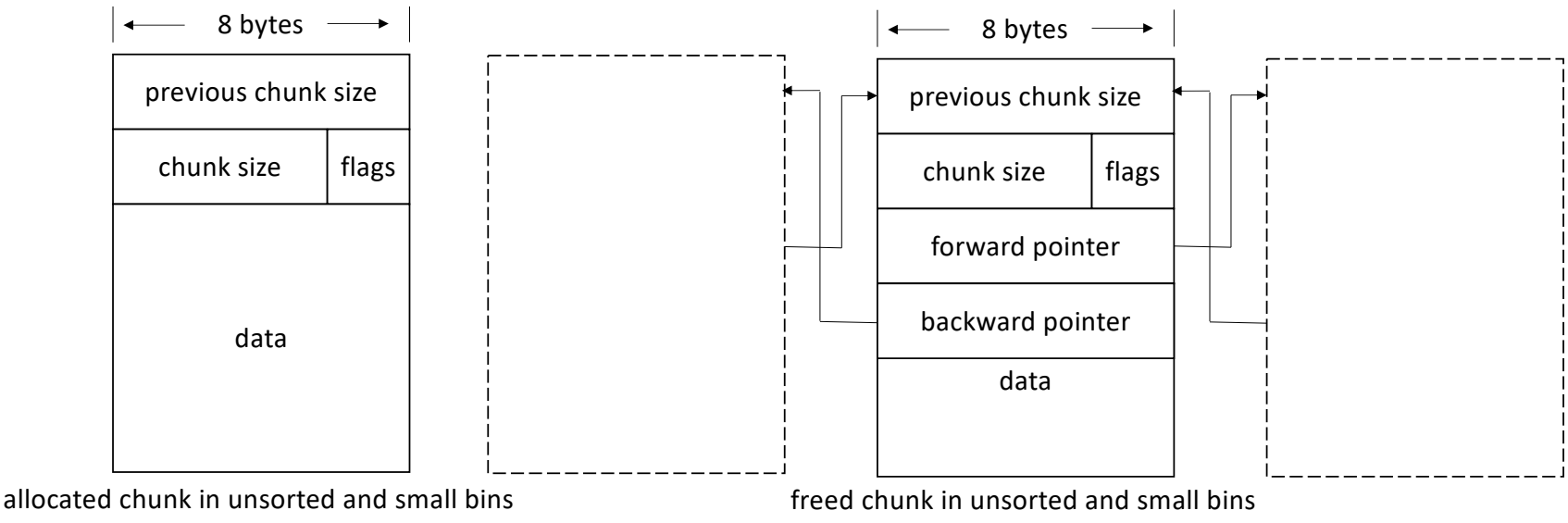


Figure: 3: Allocated chunks and freed chunks in unsorted bins and small bins

Allocated and Freed Chunks in Different Bins

The following figure shows the allocated and freed chunk in **large bins**.

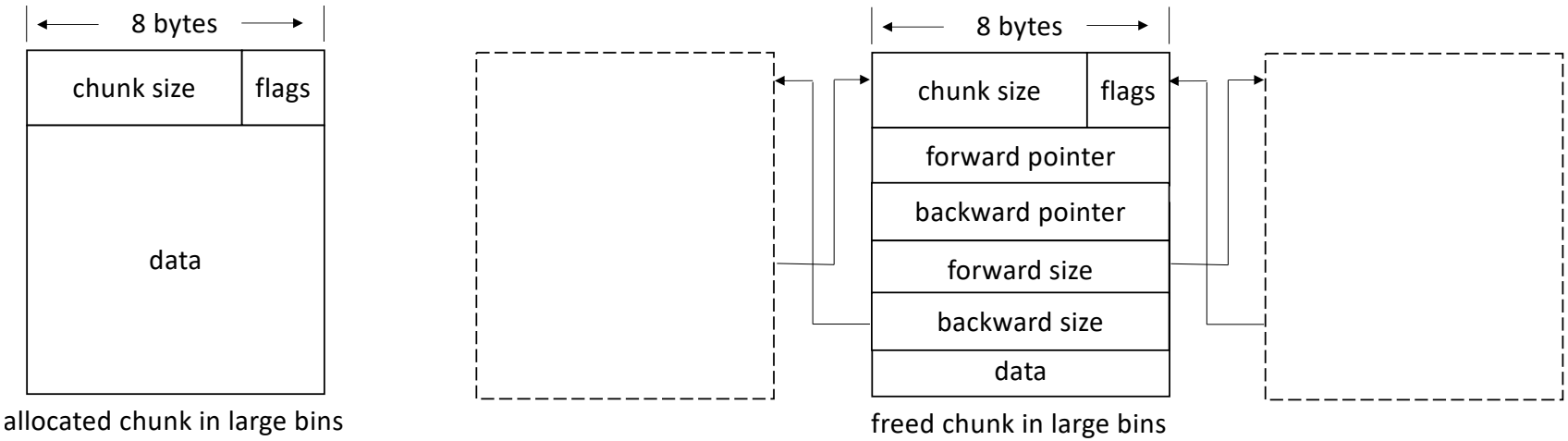


Figure: 4: Allocated chunks and freed chunks in large bins

The Details of `free` and `malloc`

`free`

- Recycle to TCache: If there is room in the tcache, store there.
- Recycle to fastbin: If the chunk is small enough, store to fastbin.
- munmap: If the chunk was mmap'd, munmap it.
- Coalesce: If this chunk is adjacent to another free chunk, coalesce.
- Put to unsorted bin: Store in the unsorted bin, unless it's the top chunk.
- Return to OS: If the chunk is large enough, coalesce fastbins and see if the top chunk is large enough to give some memory back to the system.

The Details of `free` and `malloc`

`malloc`

- Allocate from TCache.
- mmap: If the request is very large, use `mmap` system call to request memory directly from the operating system.
- Allocate from fastbin.
- Allocate from smallbin.
- Resolve all the deferred frees (fastbins $\rightarrow$ unsorted bin).
- Resolve all the deferred frees (unsorted bin $\rightarrow$ small/large bins).
- Find a large-enough chunk for large request.
- Split off part of the top chunk.

Leaking libc Base Address from Heap

When address space layout randomization (ASLR) is enable, which is quite normal nowadays, the heap starting address will be at a randomized address for each run. To perform JIT-ROPs, we often need the base address of either main executables, or libc. We can actually rely on heap attack to leak glibc base address from heap.

In this subsection, we will introduce one way to leak libc base address from *unsorted bin*.

Data Structures for Leaking the libc Baseaddr

As we can see in Figure. 5, the unsorted bins have a double linked list. Some of the chunks point to the head of the bin, which is in the arena. The head of the bin has a constant offset from the libc base address, and we can use this offset to calculate the base address each time the program runs.

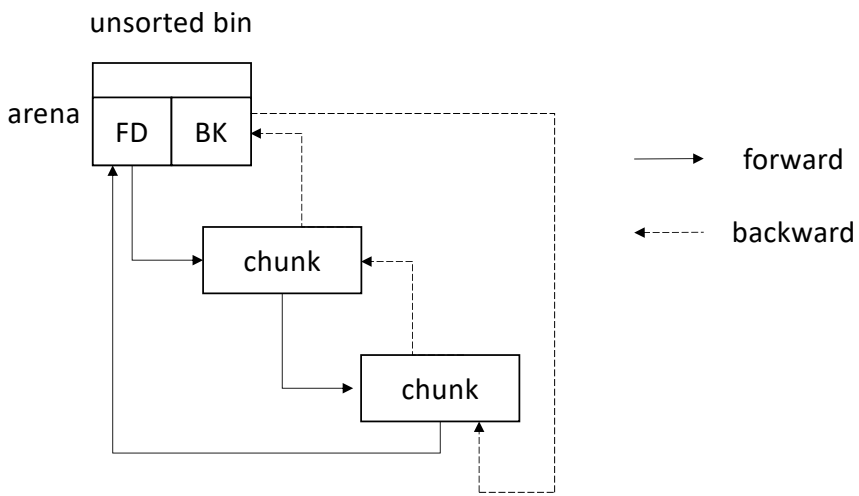


Figure: 5: Unsorted bins

Leaking libc base address from unsorted bins has the following steps:

- Set up the layout of unsorted bins.
- Read the content of the chunk in unsorted bins.
- Calculate the base address of libc from the value of the `fd` pointer.

Suppose we have a program and its interface is shown in the following. A user can allocate a chunk of heap memory, free the chunk, or edit the chunk.

```
pwndbg> r
Starting program: /home/schen/comp6700/lec22/use_after_free
Choose from the following menu:
0. [exit]
1. [m]alloc with size, e.g. m 2
2. [f]ree with index, e.g, f 1
3. [e]dit allocated chunk's content, e.g, e 2
4. [l]ist all pointers, e.g, l
5. [r]ead content by index, e.g., r 1
```

An Attack Example

STEP 1: Set up the layout of unsorted bins.

In this step, we will fill the TCache bins and unsorted bins, e.g., malloc and then free for 7 times to fill in TCache bins, and malloc and free once for unsorted bins.

The following is the bins layout after we malloc and free 8 times. We choose to malloc memory chunk of 300 bytes, as it can be put in TCache bins but not fast bins. We can see that the TCache bins have 7 chunks and unsorted bins have 1 chunks, whose forward pointer points to an address with constant offset from the libc base address.

```
pwndbg> bins
tcachebins
0x310 [  7]: 0x1a9e4c0 --> 0x1a9e1b0 --> 0x1a9dea0 --> ... --> 0x1a9d260 <-- 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x1a9e7c0 --> 0x7fcd34083ca0 (main_arena+96) <-- 0x1a9e7c0
smallbins
empty
largebins
empty
```

An Attack Example

STEP 2: Read the content of the chunk in unsorted bins.

We read the content of the chunk (e.g., using the option 5 in this example) and calculate the offset of the this address (which is in arena) and the libc base address we have.

In this example, the content is 0x7fcd34083ca0. We run the following command and get the libc base address in this run and we can see the libc base address is 0x7fcd33c98000.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r--p     1000 0      ./use_after_free
          0x401000           0x402000 r-xp     1000 1000   ./use_after_free
          0x402000           0x403000 r--p     1000 2000   ./use_after_free
          0x403000           0x404000 r--p     1000 2000   ./use_after_free
          0x404000           0x405000 rw-p     1000 3000   ./use_after_free
         0x1de4000          0x1e05000 rw-p    21000 0      [heap]
    0x7fcd33c98000     0x7fcd33e7f000 r-xp   1e7000 0      ./libc-2.27.so
    0x7fcd33e7f000     0x7fcd3407f000 ---p   200000 1e7000 ./libc-2.27.so
    0x7fcd3407f000     0x7fcd34083000 r--p     4000 1e7000 ./libc-2.27.so
    0x7fcd34083000     0x7fcd34085000 rw-p     2000 1eb000 ./libc-2.27.so
    0x7fcd34085000     0x7fcd34089000 rw-p     4000 0
    0x7fcd34089000     0x7fcd340b0000 r-xp    27000 0      ./ld-2.27.so
    0x7fcd342ac000     0x7fcd342b0000 rw-p     4000 0
    0x7fcd342b0000     0x7fcd342b1000 r--p     1000 27000  ./ld-2.27.so
    0x7fcd342b1000     0x7fcd342b2000 rw-p     1000 28000  ./ld-2.27.so
    0x7fcd342b2000     0x7fcd342b3000 rw-p     1000 0
    0x7ffda9afb000     0x7ffda9b1c000 rw-p    21000 0      [stack]
    0x7ffda9b78000     0x7ffda9b7c000 r--p     4000 0      [vvar]
    0x7ffda9b7c000     0x7ffda9b7e000 r-xp     2000 0      [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsyscall]
```

An Attack Example

STEP 3: Caculate the base of libc by the content in fd pointer.

```
Next time we perform the attack, the address of libc base address will be different, but we can get the co

In a new run, the content leaked is 0x7fa660387ca0. We can calculate libc base address:

0x7fa660387ca0 - (0x7fcd34083ca0 - 0x7fcd33c98000) = 0x7fa65ff9c000
```

An Attack Example

STEP 3: Caculate the base of libc by the content in fd pointer.

We can verify this via the following command:

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r--p     1000 0      ./use_after_free
          0x401000           0x402000 r-xp     1000 1000   ./use_after_free
          0x402000           0x403000 r--p     1000 2000   ./use_after_free
          0x403000           0x404000 r--p     1000 2000   ./use_after_free
          0x404000           0x405000 rw-p     1000 3000   ./use_after_free
         0x21df000          0x2200000 rw-p    21000 0      [heap]
    0x7fa65ff9c000     0x7fa660183000 r-xp   1e7000 0      ./libc-2.27.so
    0x7fa660183000     0x7fa660383000 ---p   200000 1e7000 ./libc-2.27.so
    0x7fa660383000     0x7fa660387000 r--p     4000 1e7000 ./libc-2.27.so
    0x7fa660387000     0x7fa660389000 rw-p     2000 1eb000 ./libc-2.27.so
    0x7fa660389000     0x7fa66038d000 rw-p     4000 0
    0x7fa66038d000     0x7fa6603b4000 r-xp    27000 0      ./ld-2.27.so
    0x7fa6605b0000     0x7fa6605b4000 rw-p     4000 0
    0x7fa6605b4000     0x7fa6605b5000 r--p     1000 27000  ./ld-2.27.so
    0x7fa6605b5000     0x7fa6605b6000 rw-p     1000 28000  ./ld-2.27.so
    0x7fa6605b6000     0x7fa6605b7000 rw-p     1000 0
    0x7ffd19551000     0x7ffd19572000 rw-p    21000 0      [stack]
    0x7ffd19582000     0x7ffd19586000 r--p     4000 0      [vvar]
    0x7ffd19586000     0x7ffd19588000 r-xp     2000 0      [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsyscall]
```

We can see the libc base address is 0x7fa65ff9c000, which matches our calculation result.

Use After Free Attack on Fast Bins

If an already freed memory chunk is used, this may corrupt the heap metadata and lead to use after free attacks (UAF).

In fastbin UAF attack, we can edit a chunk after it is freed.

Figure. 6 shows the a fast bin with three chunks. With a UAF vulnerability, we can edit the second chunk and make its forward pointer points to a victim chunk (fake chunk).

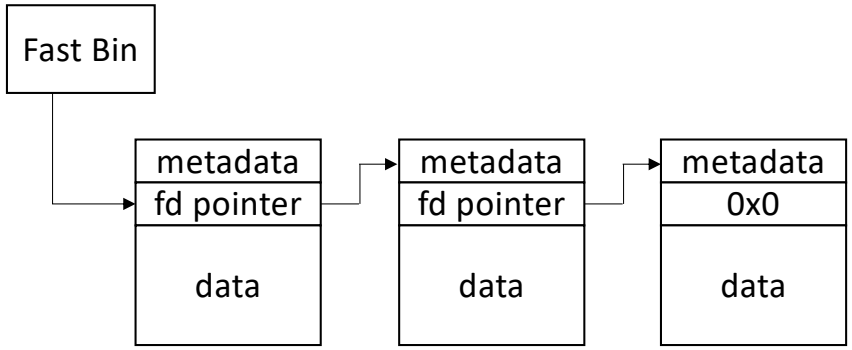


Figure: 6: A fast bin with three chunks

Use After Free Attack on Fast Bins

Figure. 7 shows the layout of a fast bin when the second chunk is edited to point to a fake chunk. The victim chunk (fake chunk) should satisfy the following criteria otherwise glibc will trigger an error: the fake chunk should have a forward pointer of 0 or transitively point to a chunk that has a forward pointer of 0.

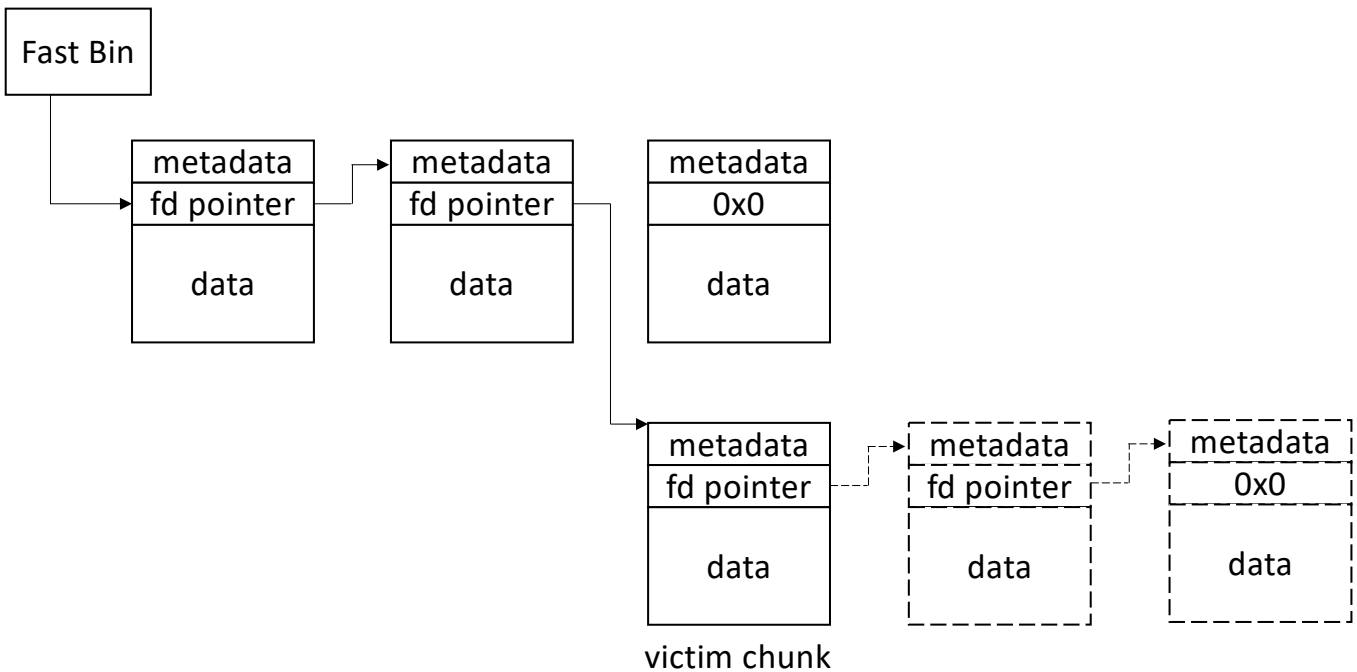


Figure: 7: A fast bin with a chunk pointing to a fake chunk

Use After Free Attack on Fast Bins

An example UAF attack on fast bins has the following steps:

- Prepare TCache bins and Fast bins.
- Identify a fake chunk.
- Link the fake chunk to fast bin free list.
- Emptify the TCache bin.
- Tcache refill.
- malloc to get fake chunk.
- Overwrite victim memory.
- Trigger the vulnerability.

A Use-after-free Vulnerability Example

Suppose we have a program and its interface is shown in the following. A user can allocate a chunk of heap memory, free the chunk, or edit the chunk.

```
pwndbg> r
Starting program: /home/schen/comp6700/lec22/use_after_free
Choose from the following menu:
0. [exit]
1. [m]alloc with size, e.g. m 2
2. [f]ree with index, e.g, f 1
3. [e]dit allocated chunk's content, e.g, e 2
4. [l]ist all pointers, e.g, l
```

An Attack Example

STEP 1: Prepare TCache bins and Fast bins.

We first allocate 10 small size of chunks (e.g., 0x20 bytes) and then free these 10 chunks. Since by default, there are at most 7 chunks in each TCache bin, then fast bin would have 3 chunks, as shown in Figure 8.

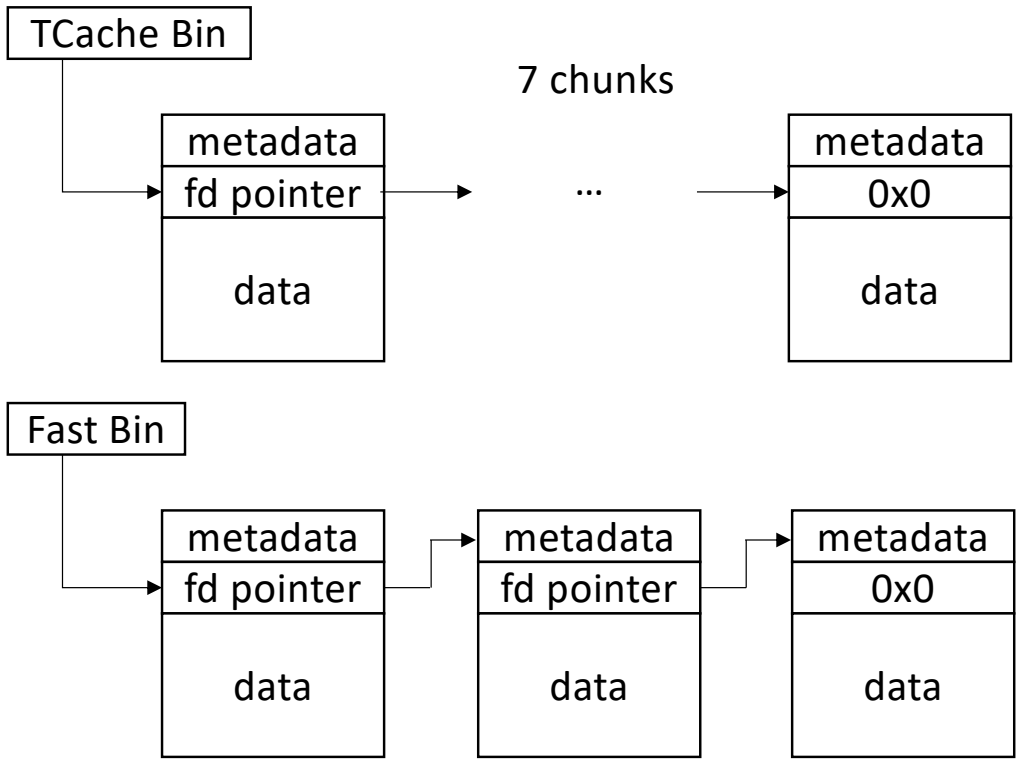


Figure: 8: UAF attack on Fast Bins STEP 1

An Attack Example

STEP 2: Identify a fake chunk.

As we have seen the fake chunk should have forward pointer pointing to address 0 or transitively point to a chunk that has a forward pointer of 0. The malloc GOT entry we used in the past does not satisfy this requirement. We will use __malloc_hook instead in this example, and __malloc_hook has a default value of 0, which satisfies our requirements. The fake chunk is shown in Figure 9.

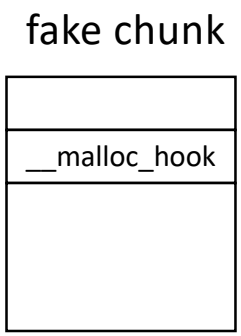


Figure: 9: UAF attack on Fast Bins STEP 2

An Attack Example

STEP 3: Link the fake chunk to fast bin free list.

In this step, we use the use-after-free vulnerability to edit the chunk at the beginning of the fast bin free list and make it point to our fake chunk, as shown in Figure 10.

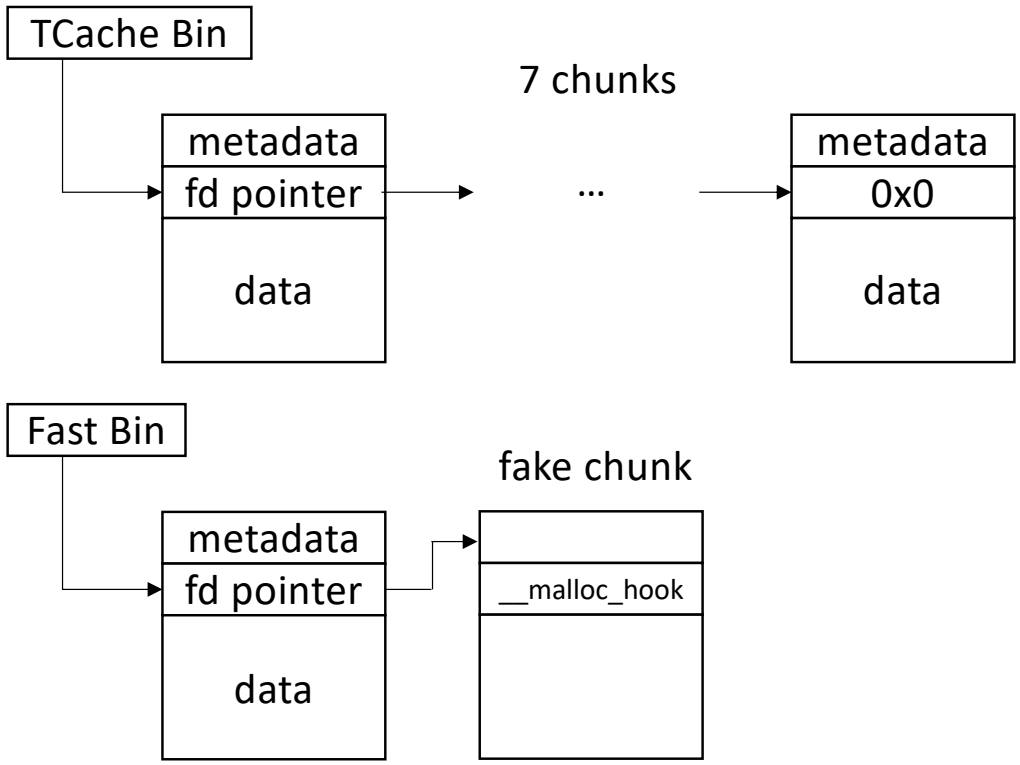


Figure: 10: UAF attack on Fast Bins STEP 3

An Attack Example

STEP 4: Emptify the TCache bin.

We malloc 7 times which will emptify the TCache bins, as shown in Figure 11.

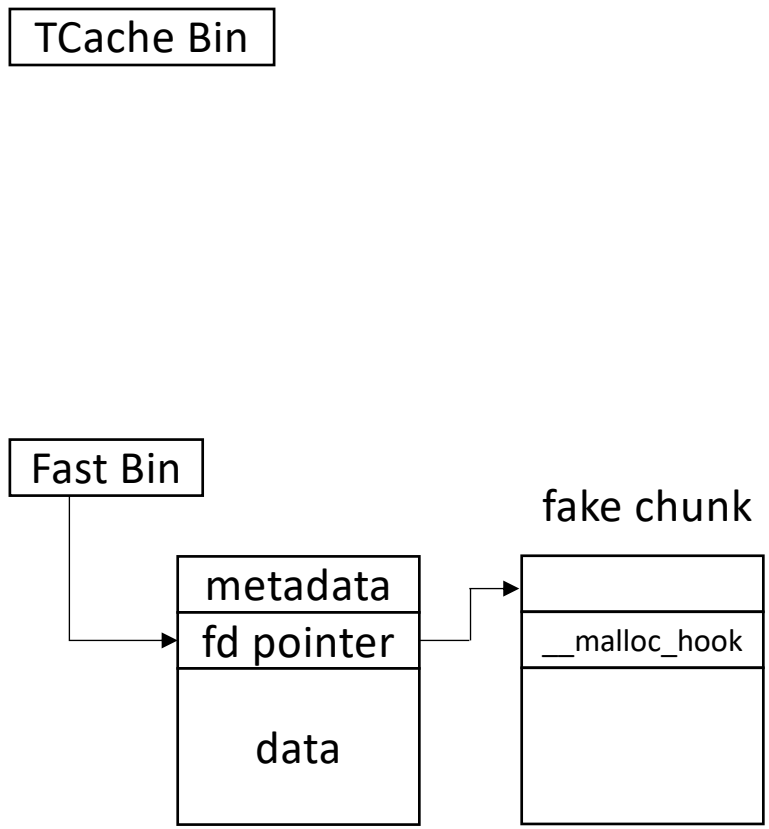


Figure: 11: UAF attack on Fast Bins STEP 4

An Attack Example

STEP 5: Tcache refill.

In this step, we malloc once, which will return the first chunk of fast bin and put the rest in fast bin to TCache bin, as shown in Figure 12.

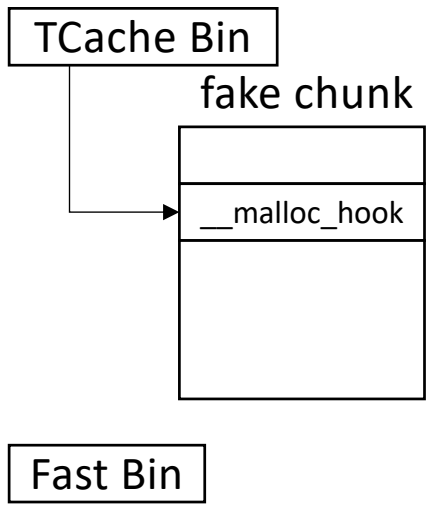


Figure: 12: UAF attack on Fast Bins STEP 5

An Attack Example

STEP 6: malloc to get fake chunk.

In this step, we malloc once and get the fake chunk as return address, and the TCache bins and fast bins are empty as shown in Figure 13.

TCache Bin

Fast Bin

Figure: 13: UAF attack on Fast Bins STEP 6

An Attack Example

STEP 7: Overwrite victim memory.

In this step, we overwrite the victim memory: __malloc_hook to win function address, as shown in Figure 14.

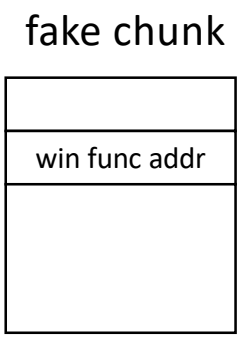


Figure: 14: UAF attack on Fast Bins STEP 7

An Attack Example

STEP 8: Trigger the vulnerability.

We call malloc again, and the control flow is directed to __malloc_hook, which is actually win function address. The control flow is hijacked to win function and we get the following from the terminal:

```
You win!
```

Double Free Attack on Fast Bins

If an allocated memory chunk is freed twice, this may corrupt the heap content and lead to double free attacks.

However, in fast bins, double free vulnerability is different from TCache bins. As we can see in Figure 15, in TCache bins double free vulnearbility, a chunk can point to itself in TCache bins, while in fast bins, a chunk can point to another chunk and again another chunk to the original chunk, which we will exploit in this attack.

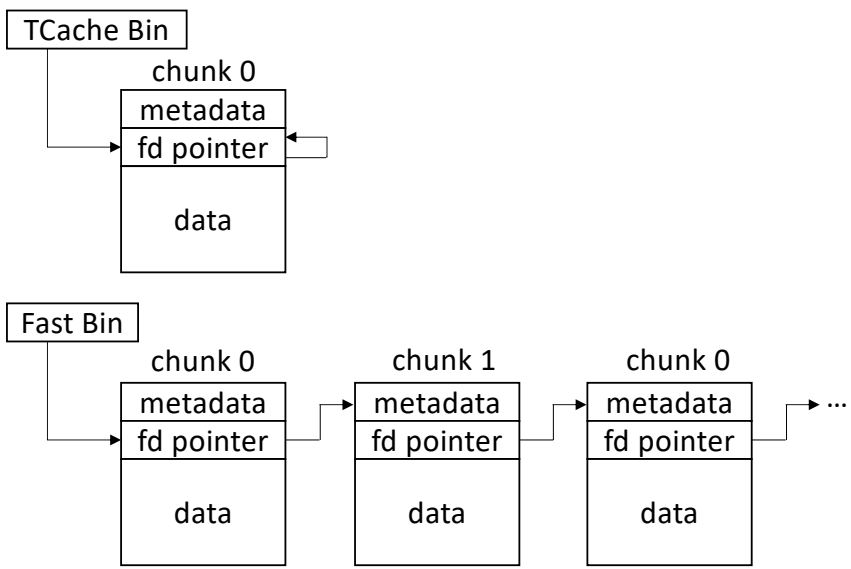


Figure: 15: Double free vulnerability in fast bins

Double Free Attack on Fast Bins

An example of double free attack on fast bins has the following steps:

- Prepare the TCache bins and fast bins layout (one chunk in fast bin is freed twice)
- Empty the TCache bins
- malloc once (we will get the vulnerable chunk)
- Link the vulnerable chunk to a fake chunk
- malloc three times to get the fake chunk
- Edit the fake chunk and perform the attack

A Double Free Vulnerability Example

Suppose we have a program and its interface is shown in the following.

```
pwndbg> r
Starting program: /home/schen/comp6700/lec22/double_free
Choose from the following menu:
0. [exit]
1. [m]alloc with size, e.g. m 2
2. [f]ree with index, e.g, f 1
3. [e]dit allocated chunk's content, e.g, e 2
4. [l]ist all pointers, e.g, l
5. [r]ead content by index, e.g., r 1
```

An Attack Example

STEP 1: Prepare the TCache bins and fast bins layout

In this step, we malloc 9 times and free 9 times. The 7 chunks will be put in the TCache bins, and the other 2 will be put in the fast bins. We free the 8th chunk (numbered 7, as numbering starts from 0) again, then this chunk apears twice in fast bins. The bins layout is shown in Figure 16.

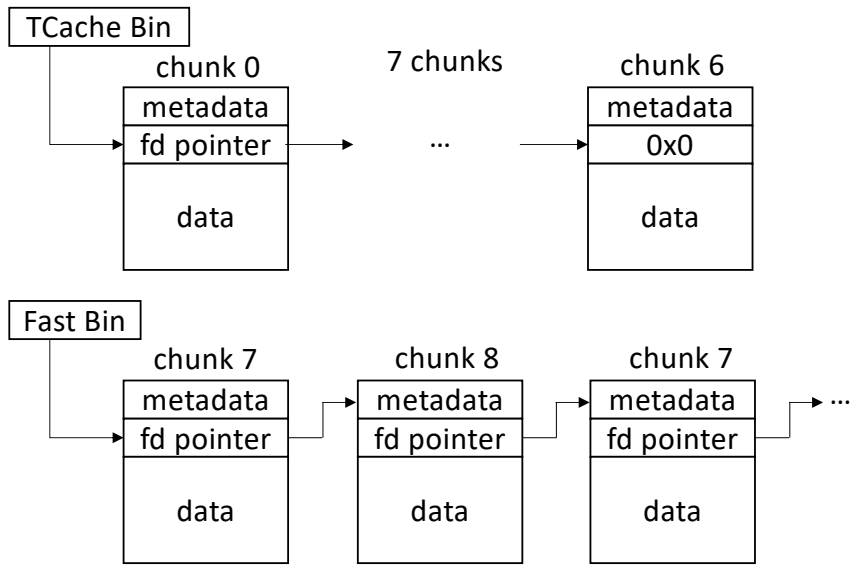


Figure: 16: Double free vulnerability in fast bins STEP 1

An Attack Example

STEP 2: Empty the TCache bins

We malloc 7 times to emptify the TCache bins, as shown in Figure 17.

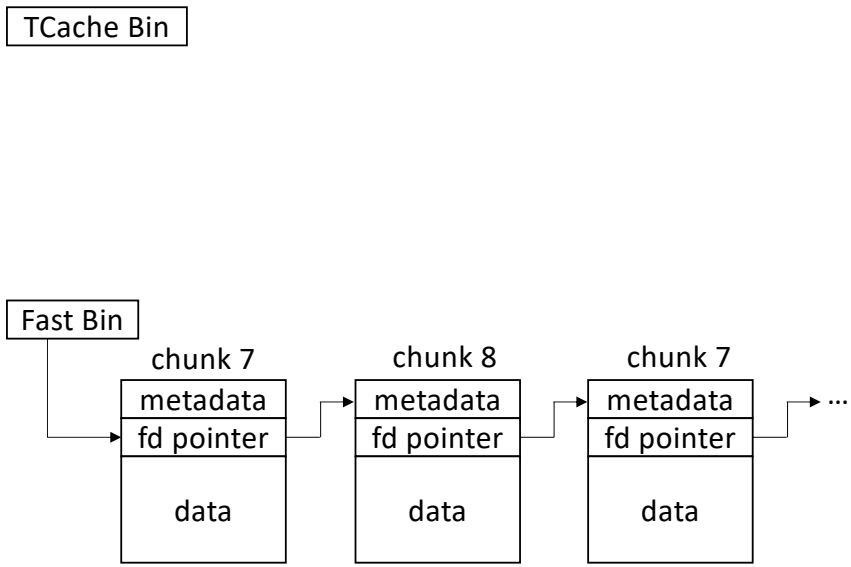


Figure: 17: Double free vulnerability in fast bins STEP 2

An Attack Example

STEP 3: malloc once

We malloc once and this time we get the vulnerable chunk, the rest two chunks in fast bins will be moved to the TCache bins and the bin's layout is shown as Figure 18.

Note that in the last step, chunk 7 is at the head of fast bin. The malloc in this step return chunk 7 and the rest of fast bin will be moved to TCache bin.

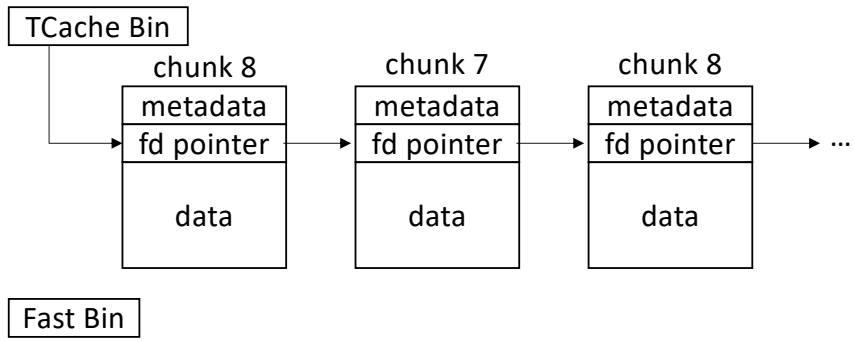


Figure: 18: Double free vulnerability in fast bins at STEP 3

An Attack Example

STEP 4: Link the vulnerable chunk to fake chunk

With the return address of the vulnerable chunk from last step, we edit the content of this chunk. Since the chunk is actually still in fast bins free list, we can make it to point to a fake chunk, which here we use malloc GOT entry.

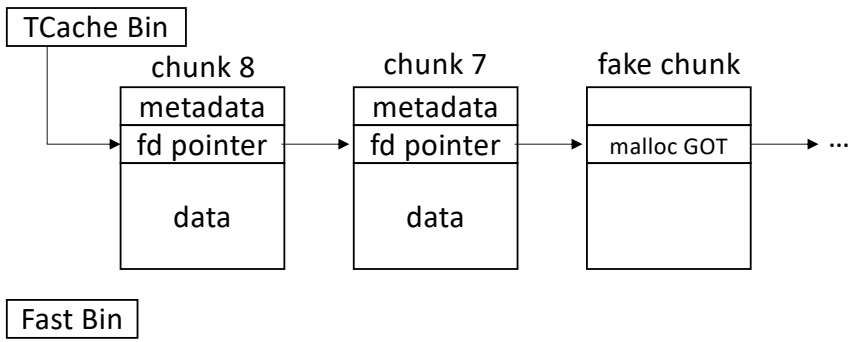


Figure: 19: Double free vulnerability in fast bins at STEP 4

An Attack Example

STEP 5: malloc three times to get the fake chunk

```
In this step, we malloc three times to get the fake chunk, which is chunk 7.
```

An Attack Example

STEP 6: Edit the fake chunk and perform the attack

```
We edit the fake chunk and change the malloc GOT entry to win function.

Finally, we allocate a new chunk, which will call function malloc and the control flow is hijacked
to function win.

You win!
```

Heap Overflow Attack on Fast Bins

If two chunks are next to each other, and there is no boundary check for using the chunk, the data of one chunk may corrupt the other chunk and lead to heap overflow attacks.

An example heap overflow attack on Fast bins has the following steps:

- compute the base of libc based on fd pointer in unsorted bin
- prepare TCache and Fast bins layout
- overflow fast bin chunk
- empty TCache bin for size 20
- TCache bin (of size 20) refill (reversely)
- malloc to get fake chunk
- edit the last allocated chunk and perform the attack

A Heap Overflow Vulnerability Example

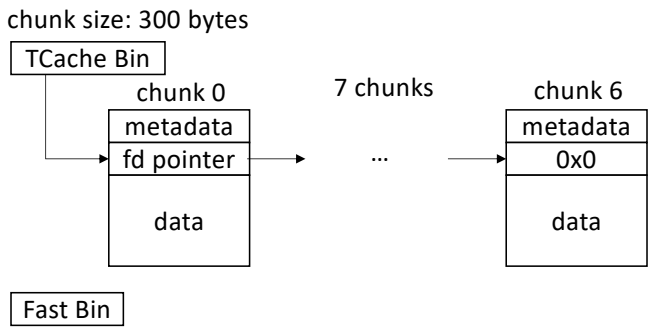
Suppose we have a program and its interface is shown in the following. A user can allocate a chunk of heap memory, free the chunk, edit the chunk, list the chunk, or read the content of the chunk.

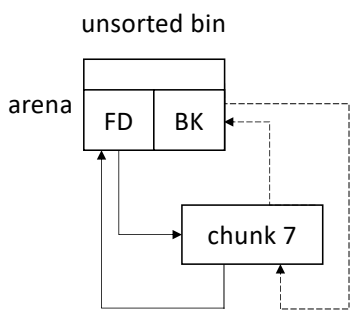
```
Choose from the following menu:
0. [exit]
1. [m]alloc with size, e.g. m 2
2. [f]ree with index, e.g, f 1
3. [e]dit allocated chunk's content, e.g, e 2
4. [l]ist all pointers, e.g, l
5. [r]ead content by index, e.g., r 1
```

An Attack Example

STEP 1: compute libc base address using unsorted bin

We allocate ten memory chunks of size 300 bytes, and then free eight memory chunks with index of 0-7. Since 300 bytes will not fill in fast bins, it will go to unsorted bins. And we calculate the libc base address as introduced in section 2. The bins layout is shown in Figure 20.





An Attack Example

STEP 2: prepare TCache and Fast bins layout

We fill tcache bin with 7 chunks and fastbin with chunk of index 18, and the bins layout of chunk size 20 bytes is shown in Figure 21.

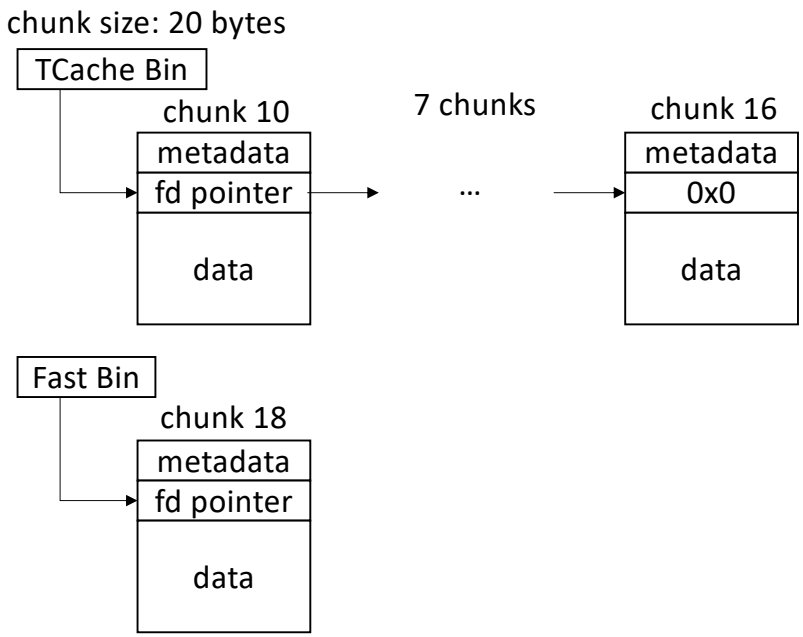


Figure: 21: Bins layout in STEP 2

An Attack Example

STEP 3: overflow fast bin chunk

We overflow the fast bin chunk with index of 17, and the bins layout of chunk size 20 bytes is shown in Figure 22.

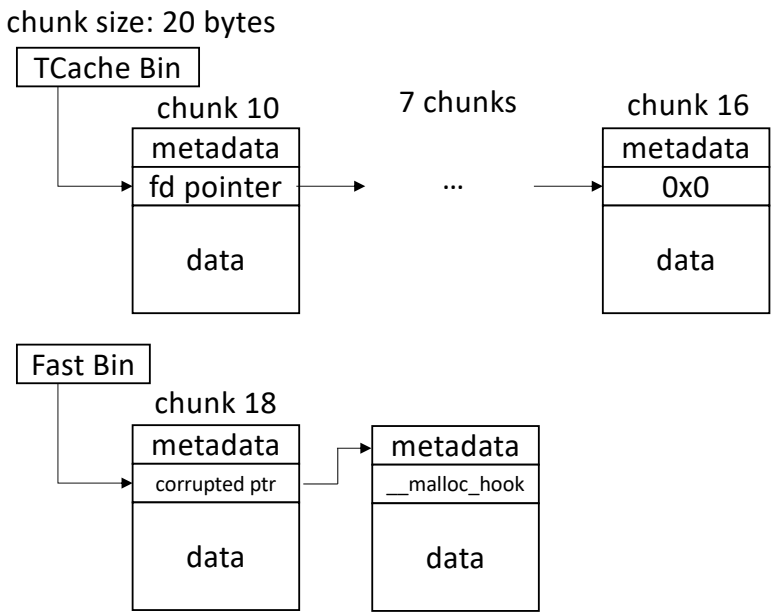


Figure: 22: Bins layout in STEP 3

An Attack Example

STEP 4: empty TCache bin for size 20

We allocate 7 chunks with size 20, and the bins of chunk size 20 bytes layout is shown in Figure 23.

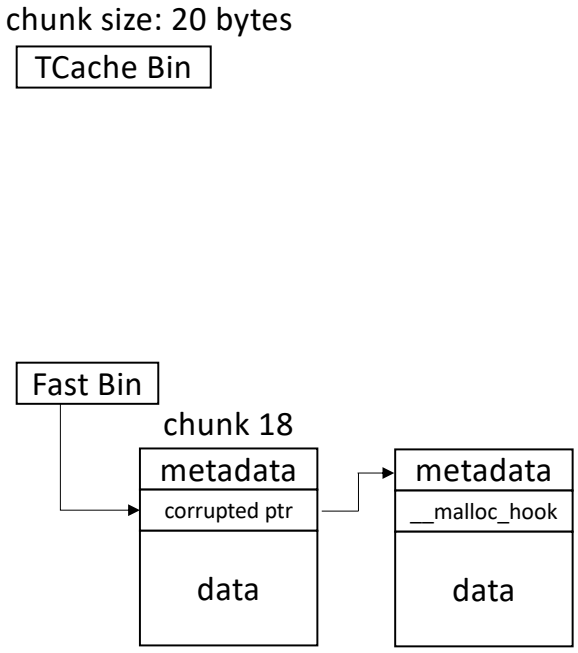


Figure: 23: Bins layout in STEP 4

An Attack Example

STEP 5: TCache bin (of size 20) refill (reversely)

We allocate one chunk with size 20, and the bins layout of chunk size 20 bytes is shown in Figure 24.

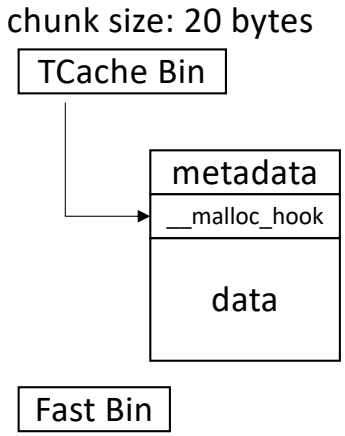


Figure: 24: ins layout in STEP 5

An Attack Example

STEP 6: malloc to get fake chunk

We allocate one chunk to get fake chunk, and the bins layout of chunk size 20 bytes is shown in Figure 25.

Figure: 25: Bins layout in STEP 6

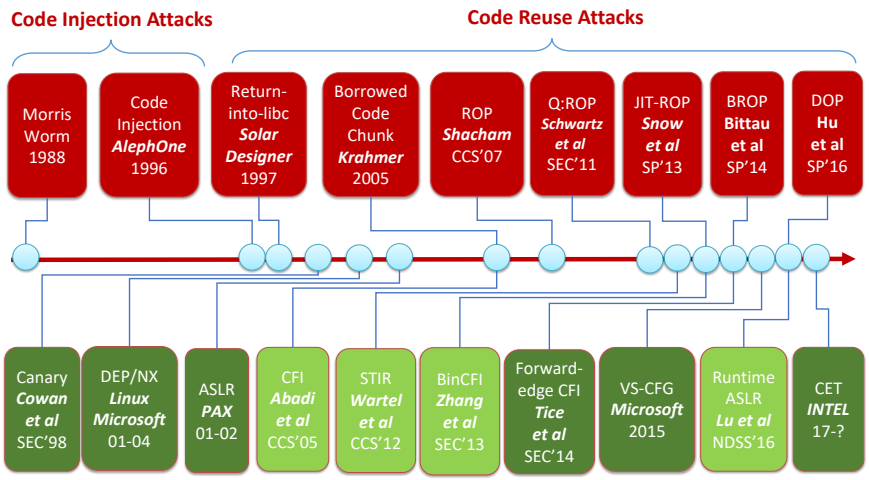
An Attack Example

STEP 7: edit the last allocated chunk and perform the attack

```
We edit the fake chunk and fill the __malloc_hook with win function address, and trigger the
vulnerability by allocate one new chunk, which will call function malloc and the control flow
is hijacked to function win.

You win!
```

Thank You



schen@auburn.edu
schuan.github.io

[1]Instructor appreciates the help from Prof. Zhiqiang Lin.

 GNU, "GNU C Library," https://sourceware.org/git/?p=glibc.git, (Accessed on 03/14/2021).

 ——, "GNU C Library Version 2.27," https://sourceware.org/git/?p=glibc.git;a=tree;h=ceb28de7aa2b986d1c5c53ec04ce8e6485b2eb8a;hb=23158b08a0908f381459f273a984c6fd328363cb, (Accessed on 03/14/2021).