Memory Corruption Attacks and Defenses: Advanced Topics

Introduction to Security (192.019)

Pedro **Bernardo**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**


R13 0x49be90 (__preinit_array_start) —▸ 0x4016a0 ◂— endbr64
R14 0x1
R15 0x1
RBP 0x7fffffffc190 ◂— 0x1
RSP 0x7fffffffc190 ◂— 0x1
RIP 0x4016fb (main+4) ◂— mov eax, 0
[DISASM]
► 0x4016fb <main+4> mov eax, 0 <0x4016f7>
0x401700 <main+9> call vuln <vuln>
0x401705 <main+14> mov edi, 0x473004
0x40170a <main+19> call system <system>
0x401715 <main+30> ret
0x401716 nop word ptr cs:[rax + rax]
0x401720 <call_fini> endbr64
0x401724 <call_fini+4> push rbp
0x401725 <call_fini+5> lea rax, [rip + 0x9a76c] <0x49be98>
[STACK]
00:0000 rbp rsp 0x7fffffffc190 ◂— 0x1
01:0008 0x7fffffffc198 —▸ 0x4018ea (__libc_start_call_main+106) ◂— mov edi, eax
02:0010 0x7fffffffc1a0 ◂— 0x3188
03:0018 0x7fffffffc1a8 —▸ 0x4016f7 (main) ◂— push rbp
04 0x7fffffffc1b0 ◂— 0x100000018

GLIBC Heap

What is the heap?

The heap is a pool of memory used for dynamic allocation at runtime

- **malloc** - allocates a memory chunk of the requested size
- **free** - deallocates a given chunk
- **calloc** - allocates memory and returns a pointer to it
- **realloc** - attempts to resize a memory chunk

```c
int main() {
  char * buffer = NULL;
  buffer = (char*) malloc(
          sizeof(char)*0x100);

  fgets(stdin, buffer, 0x100);
  printf("%s", buffer);

  /* release the allocated memory */
  free(buffer);

  return 0;
}
```

Allocates a chunk of size 0x100 chars and stores a pointer to it in the local variable **buffer**

Deallocates the chunk pointed to by **buffer**

A program can't always know how much memory it will need at runtime. Dynamic memory allows the program to request and free memory on demand, according to its needs

Where is the heap?

```
Start              End                Offset             Perm Path
0x00005600bac8e000 0x00005600bac8f000 0x0000000000000000 r-- /home/student5/lecture3/demo/malloc/a.out
0x00005600bac8f000 0x00005600bac90000 0x0000000000001000 r-x /home/student5/lecture3/demo/malloc/a.out
0x00005600bac90000 0x00005600bac91000 0x0000000000002000 r-- /home/student5/lecture3/demo/malloc/a.out
0x00005600bac91000 0x00005600bac92000 0x0000000000002000 r-- /home/student5/lecture3/demo/malloc/a.out
0x00005600bac92000 0x00005600bac93000 0x0000000000003000 rw- /home/student5/lecture3/demo/malloc/a.out
0x00005600bc8a5000 0x00005600bc8c6000 0x0000000000000000 rw- [heap]
0x00007faf7ddac000 0x00007faf7ddce000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007faf7ddce000 0x00007faf7df46000 0x0000000000022000 r-x /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x00007faf7df46000 0x00007faf7df94000 0x000000000019a000 r-- /usr/lib/x86_64-linux-gnu
0x00007faf7df94000 0x00007faf7df98000 0x00000000001e7000 r-- /usr/lib/x86_64-linux-gnu
0x00007faf7df98000 0x00007faf7df9a000 0x00000000001eb000 rw- /usr/lib/x86_64-linux-gnu
0x00007faf7df9a000 0x00007faf7dfa0000 0x0000000000000000 rw-
0x00007faf7dfa6000 0x00007faf7dfa7000 0x0000000000000000 r-- /usr/lib/x86_64-linux-gnu
0x00007faf7dfa7000 0x00007faf7dfca000 0x0000000000001000 r-x /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007faf7dfca000 0x00007faf7dfd2000 0x0000000000024000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007faf7dfd3000 0x00007faf7dfd4000 0x000000000002c000 r-- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007faf7dfd4000 0x00007faf7dfd5000 0x000000000002d000 rw- /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007faf7dfd5000 0x00007faf7dfd6000 0x0000000000000000 rw-
0x00007ffdd3931000 0x00007ffdd3952000 0x0000000000000000 rw- [stack]
0x00007ffdd39ca000 0x00007ffdd39cd000 0x0000000000000000 r-- [vvar]
0x00007ffdd39cd000 0x00007ffdd39ce000 0x0000000000000000 r-x [vdso]
0xffffffffff600000 0xffffffffff601000 0x0000000000000000 --x [vsyscall]
```

Between the ELF binary and the libraries

GLIBC Heap

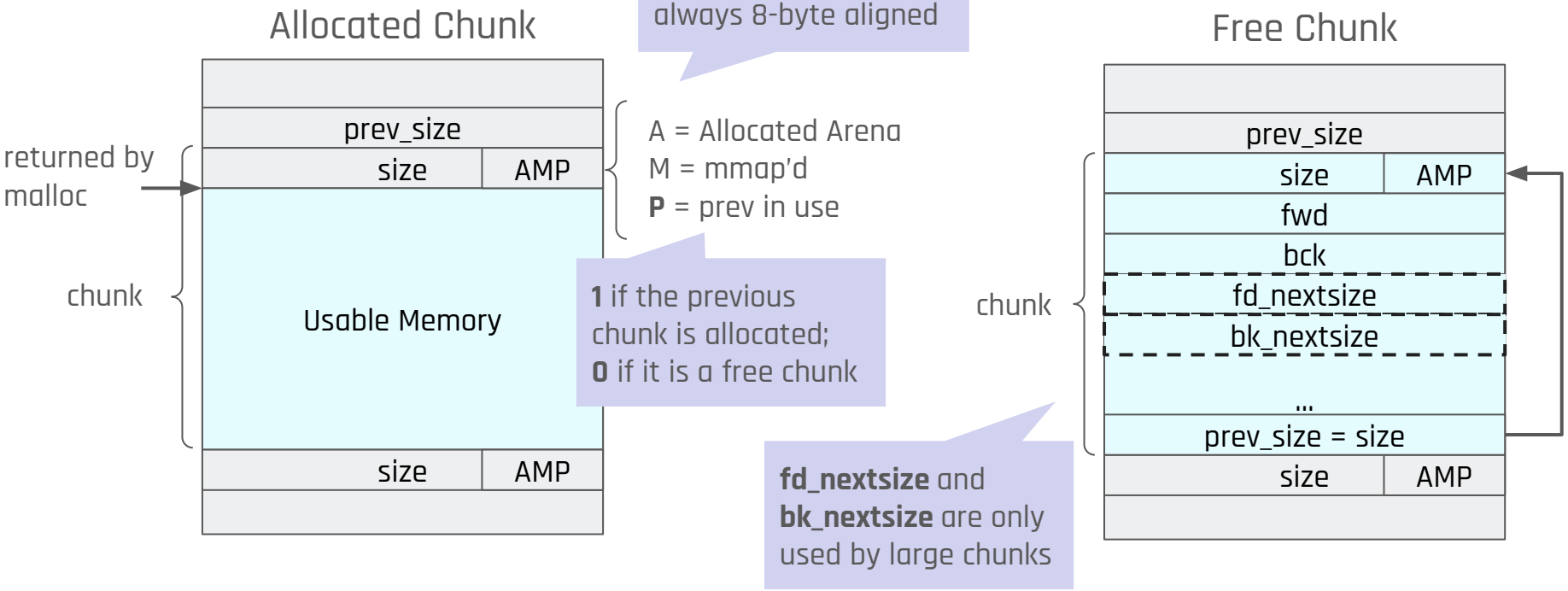
- There are several heap implementations, depending on the platform (windows, linux, etc.) and the library used (GNU LibC, FreeBSD, etc.)
- GLIBC implements *ptmalloc2*, based on *dlmalloc*
  - *ptmalloc2* supports multi-threading, improving performance
  - two different threads can manage their own heap (aka, per-thread arena)
- You can write your own heap implementation according to the needs of your application!

GLIBC Heap - Terminology

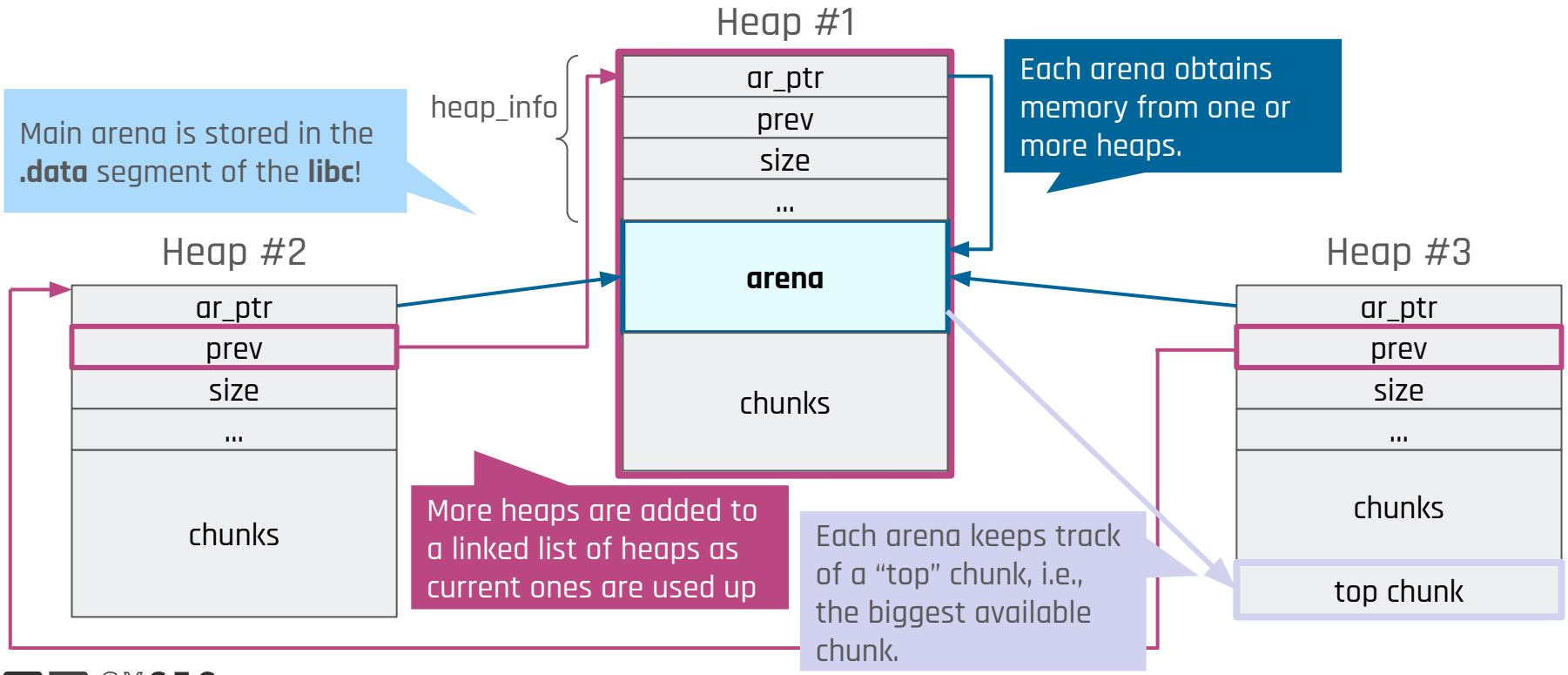
- **Arenas** are structures used to managed “the heap”
  - each thread gets its own arena (the main thread's arena is called the **main arena**)
  - contain pointers to different types of chunks and different heaps
- **Heaps** are contiguous regions of memory, subdivided into chunks
  - each heap belongs to one arena
- **Chunks** are small pieces of memory that can be allocated, freed, or combined with adjacent chunks (coalesced)
  - chunks are wrappers around a block of memory given to the application (via malloc)

GLIBC Heap - Chunks

Since the last 3 bits are reserved for the AMP flags, chunks are always 8-byte aligned



GLIBC Heap - Arenas and Heaps

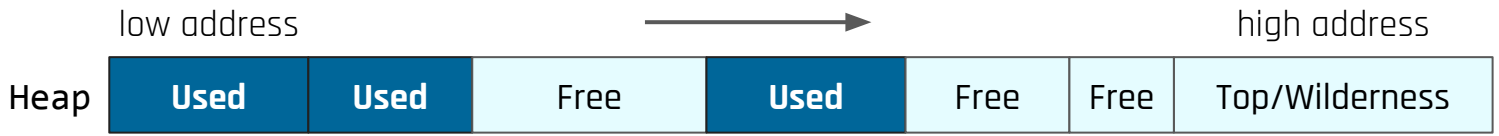


GLIBC Heap - Chunks

- **Allocated Chunk** - A chunk that is in use, i.e., owned by the application
- **Free Chunk** - A chunk that has been deallocated, i.e., owned by glibc
- **Top Chunk** - The largest available chunk, used to service new allocation requests
  - if `top_chunk->size > requested->size`, split in two
    - User chunk (requested size)
    - Remainder chunk (of remaining size)
  - else top chunk is extended using `sbrk` or `mmap`
- **Last Remainder Chunk** - The chunk of the remaining size when the top chunk is split

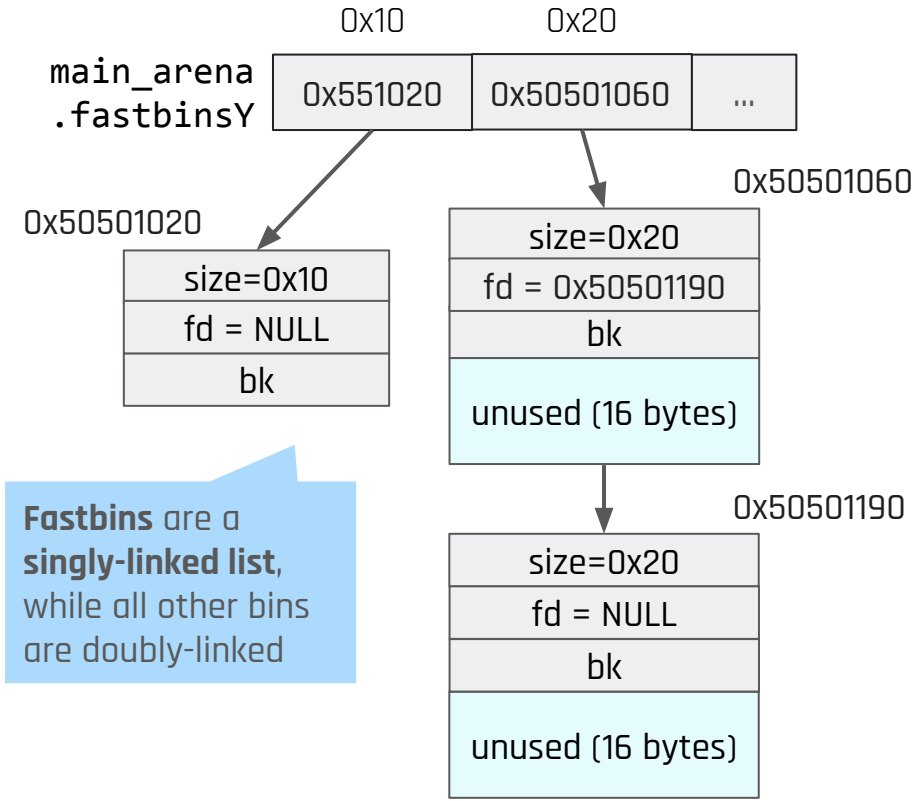
GLIBC Heap - Arena Bins

- Within each arena, chunks are either in use, or they are free.
- In-use chunks are not tracked by the arena
- Free chunks are stored in “free lists” called bins
- Bins are categorized based on the chunk sizes they hold



Main Arena - Bins

- **Fast bins** - 10 buckets (7 used by default) that store chunks from 32-128 bytes by size, 16 bytes apart.
- **Small bins** - 62 buckets, with chunks of up to 1024 bytes in size.
- **Large bins** - 63 buckets that store chunks of 1024+ bytes.
- **Unsorted bins** - Only 1 bucket. Small and large chunks go to this bin when freed.
  - Acts as a cache layer to speed up allocation and deallocation requests.
  - Unsorted bin is sorted when it is iterated over in the next call to malloc



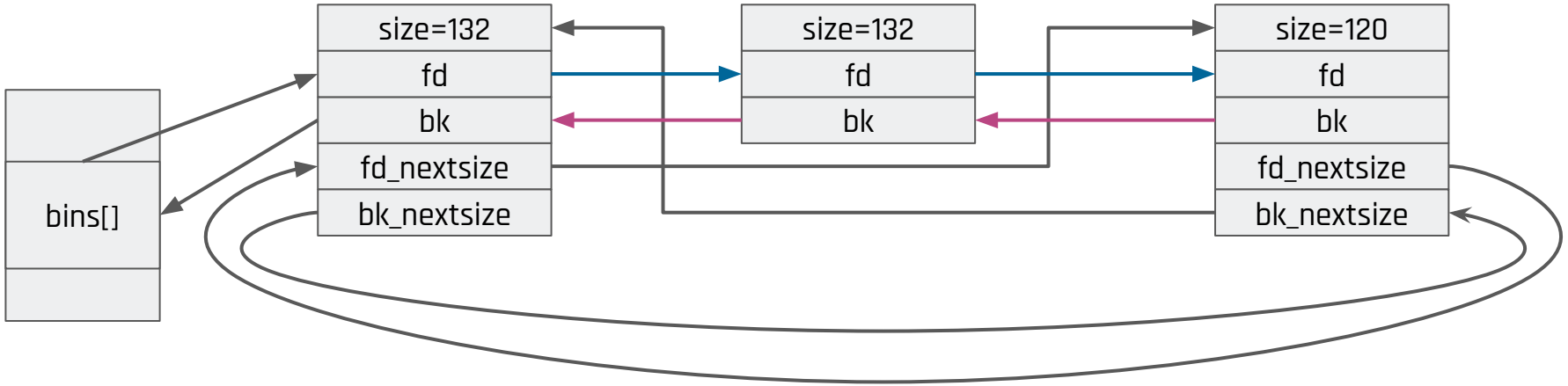
Coalescence

- Two free chunks cannot be adjacent
  - If so, they should be **combined into a single free chunk**
- This merging of adjacent free chunks is called **coalescence**
- Coalescence reduces fragmentation, but makes free slower
- However, not all chunks can be coalesced:
  - Fastbin-sized chunks are not coalesced

Large Bin

Large bins are extra special:

- They use the size field of chunks to keep an ordered doubly-linked list of chunks
- `malloc` can quickly search through the Large bins to find the first *big-enough* chunk

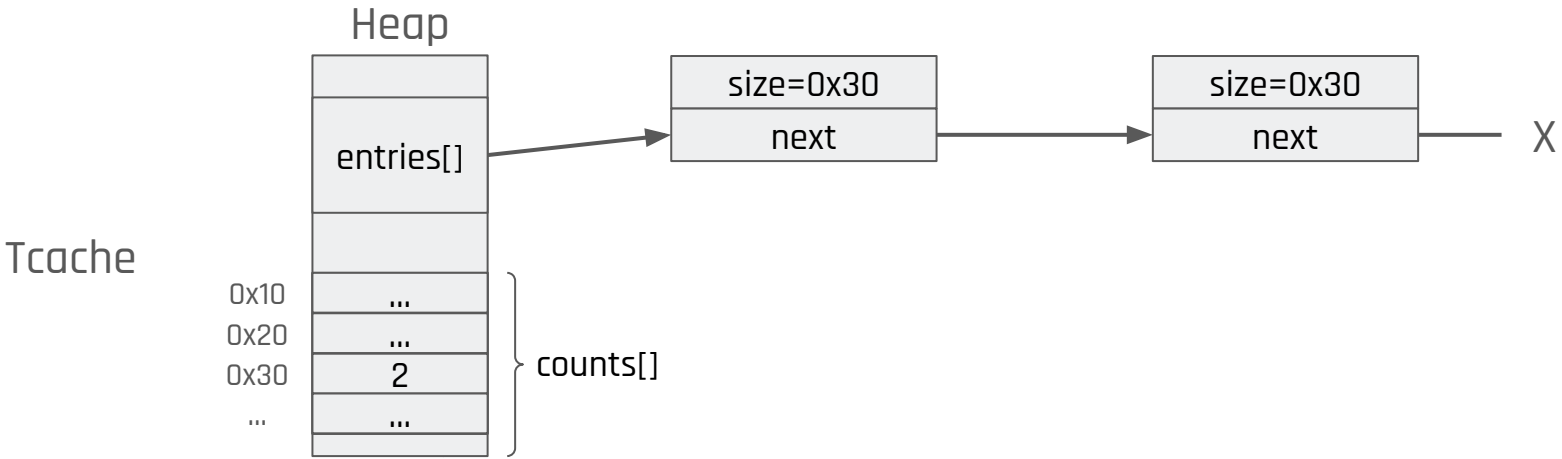


Bins Summary

Bin	Linked-list Type	Chunk Size Range	Coalescing
Fast	Singly-linked	0x10-0x80	❌
Small	Doubly-linked	0x80-0x400	✅
Large	Doubly-linked	0x400+	✅
Unsorted	Doubly-linked	Small and Large chunks	❌

Thread-Local Cache (tcache)

- Introduced in libc 2.26 to improve heap performance
- Similar to fastbins, but with less restrictions/security checks
  - 64 buckets, 0x10-0x400 sizes, up to 7 chunks per bucket
- It is allocated with malloc, so it resides on the heap, instead of the libc



Thread-Local Cache (tcache)

Tcache Put

```
static void tcache_put
     (mchunkptr chunk, size_t tc_idx) {
     tcache_entry *e = (tcache_entry *)
          chunk2mem (chunk);

     e->next = tcache->entries[tc_idx];
     tcache->entries[tc_idx] = e;
     ++(tcache->counts[tc_idx]);
}
```

Tcache Get

```
void *tcache_get (size_t tc_idx){
     tcache_entry *e =
          tcache->entries[tc_idx];

     tcache->entries[tc_idx] = e->next;
     --(tcache->counts[tc_idx]);
     return (void *) e;
}
```

glibc-2.27

Heap Vulnerabilities: Use After Free

Use After Free (UAF)

- UAFs are a kind of vulnerability where a given chunk is used (read or written to) after it was deallocated (returned back to the libc)
- UAFs can be achieved by using dangling pointers
  - dangling pointers are references to data that has been `free`'d
  - There are no guarantees on data referenced by a dangling pointer.
  - In fact, they likely contain heap or libc addresses (fd and bk pointers)

```
char *x = malloc(0x40);
printf("%p\n", x); // 0x505020
free(x);
fgets(x, 0x40, stdin); // UAF
```

Use After Free - Security Concerns

- Memory dereferenced through UAF is **usually valid memory**, so there are no mechanisms that prevent this from happening (like segmentation faults)
- **Free chunks contain heap and libc pointers**, which can be leaked through I/O functions
- It is **possible to corrupt** these structures by **overwriting metadata** (pointers), leading to arbitrary write primitives
  - Arbitrary write is a very strong primitive that can be escalated to full control-flow hijacking attacks
- Hard to avoid, since memory management in C is manual

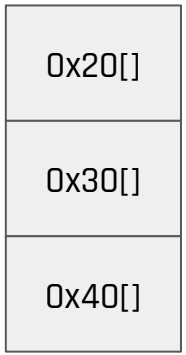
Use After Free - Double-Free

- A subclass of UAF, where the same chunk is `free`'d twice
- Simple to exploit a few libc versions ago when the security checks on the tcache and the fastbin were lackluster
- Idea:
  a. Free a chunk of a given size twice. It will be inserted in a bin (or tcache) twice, so the **fd** pointer will point to itself
  b. Allocate a chunk of the same size. The chunk that was just free'd twice will be returned
  c. The chunk will now be allocated, but still present in a free list
  d. Overwrite the **fd** pointer to the desired location
  e. Allocate two more chunks of the same size. The second chunk will be located in an attacker-specified location

Tcache Dup Attack

a = malloc(0x20)

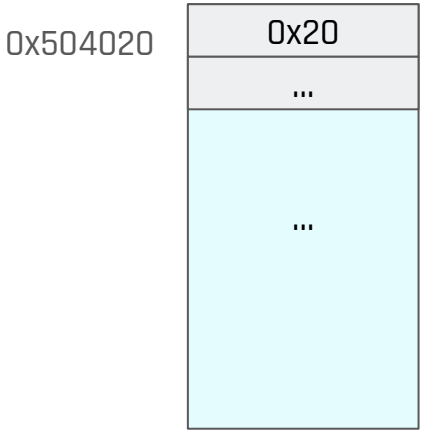
Tcache



Variables:

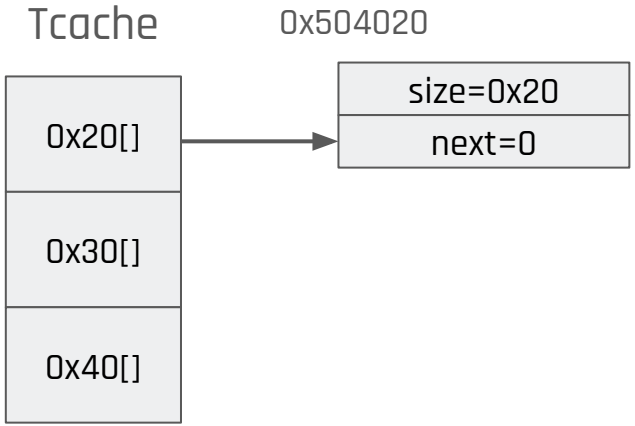
a = 0x504028

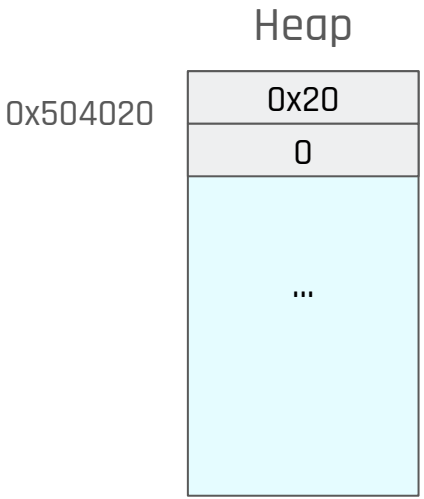
Heap



Tcache Dup Attack

free(a)

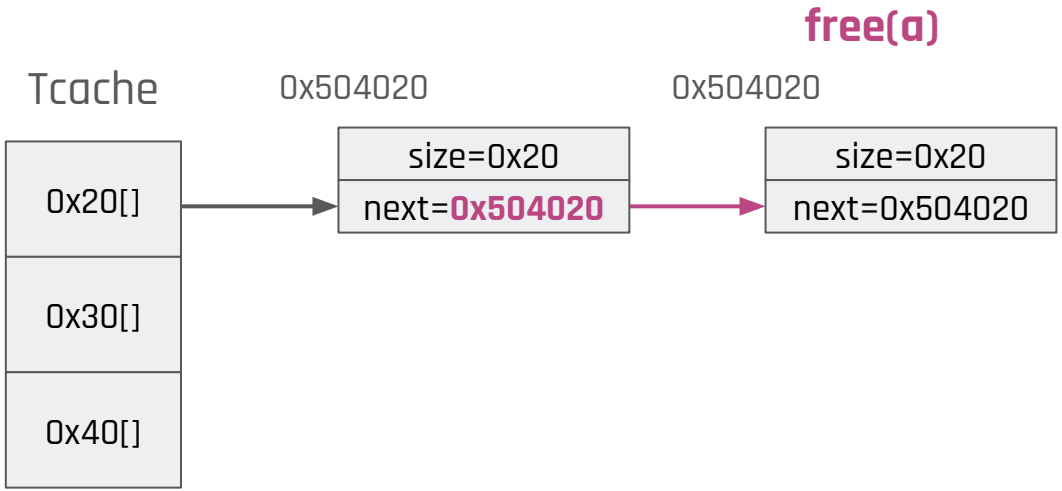


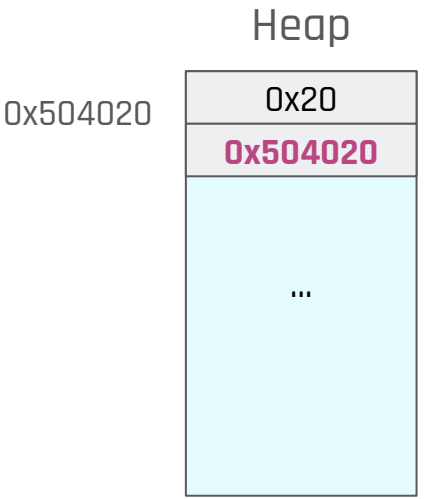


Variables:

a = 0x504028

Tcache Dup Attack



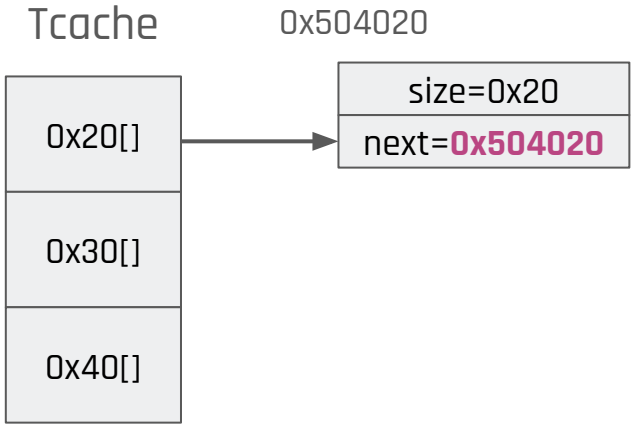


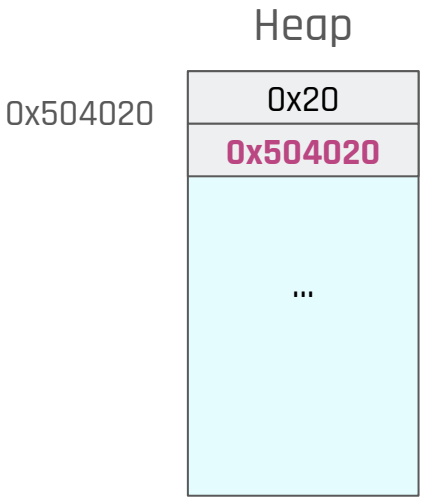
Variables:

a = 0x504028

Tcache Dup Attack

b = malloc(0x20)

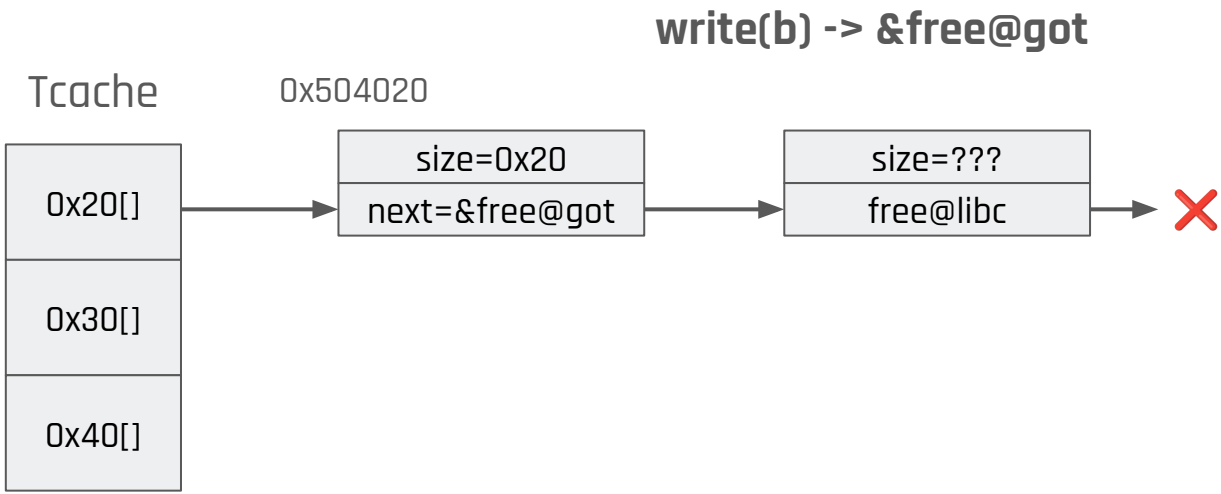


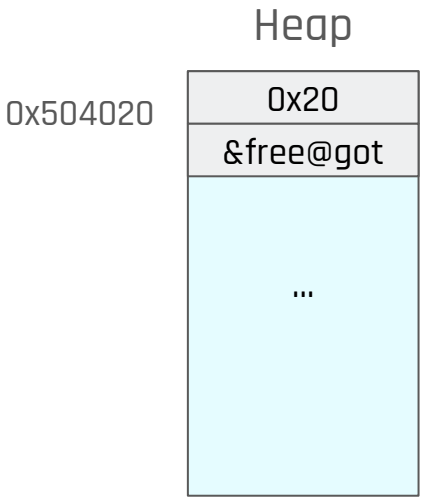


Variables:

a = 0x504028
b = 0x504028

Tcache Dup Attack





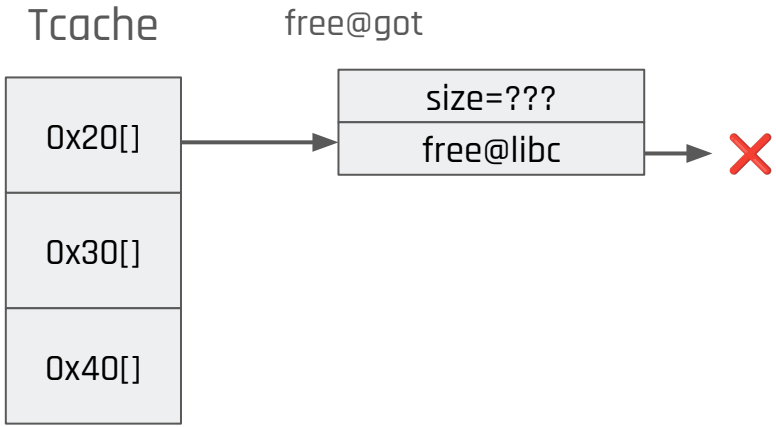
Variables:

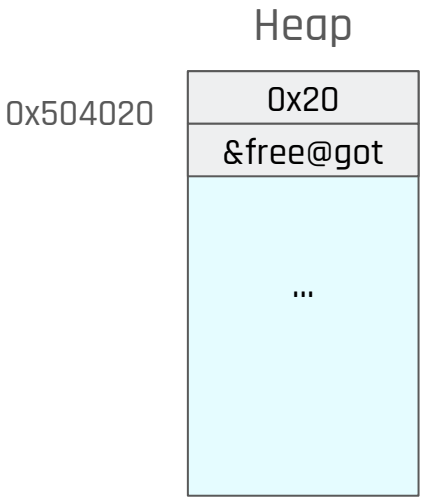
a = 0x504028

b = 0x504028

Tcache Dup Attack

c = malloc(0x20)



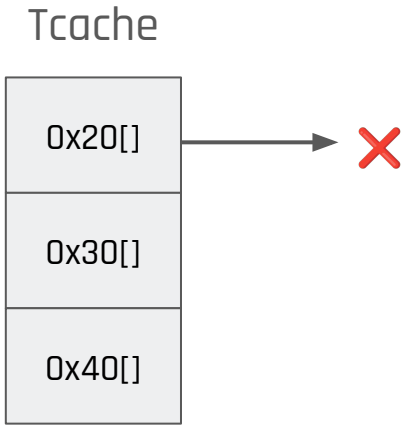


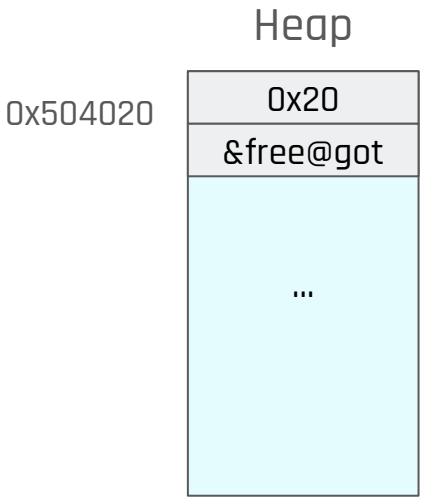
Variables:

a = 0x504028
b = 0x504028
c = 0x504028

Tcache Dup Attack

d = malloc(0x20)





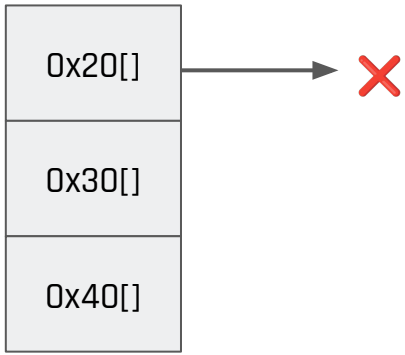
`malloc` returns a pointer to `free@got` from the tcache

Variables:

a = 0x504028
b = 0x504028
c = 0x504028
d = free@got

Tcache Dup Attack

Tcache



GOT entry of `free` is now overwritten with `&system`. Next time `free` is called, `system` will execute instead

write(d) = &system

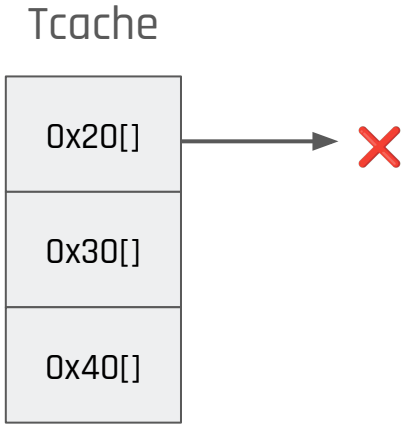
Heap

Variables:

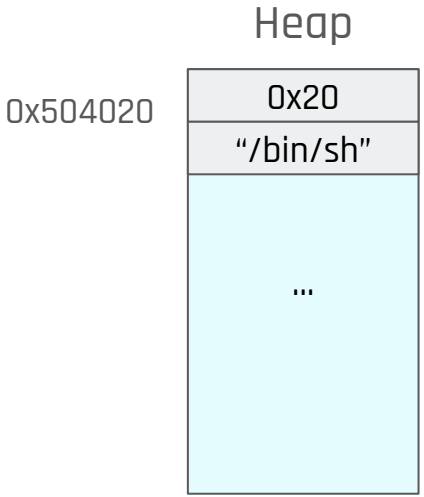
a = 0x504028
b = 0x504028
c = 0x504028

d = free@got

Tcache Dup Attack



write(c) = “/bin/sh”

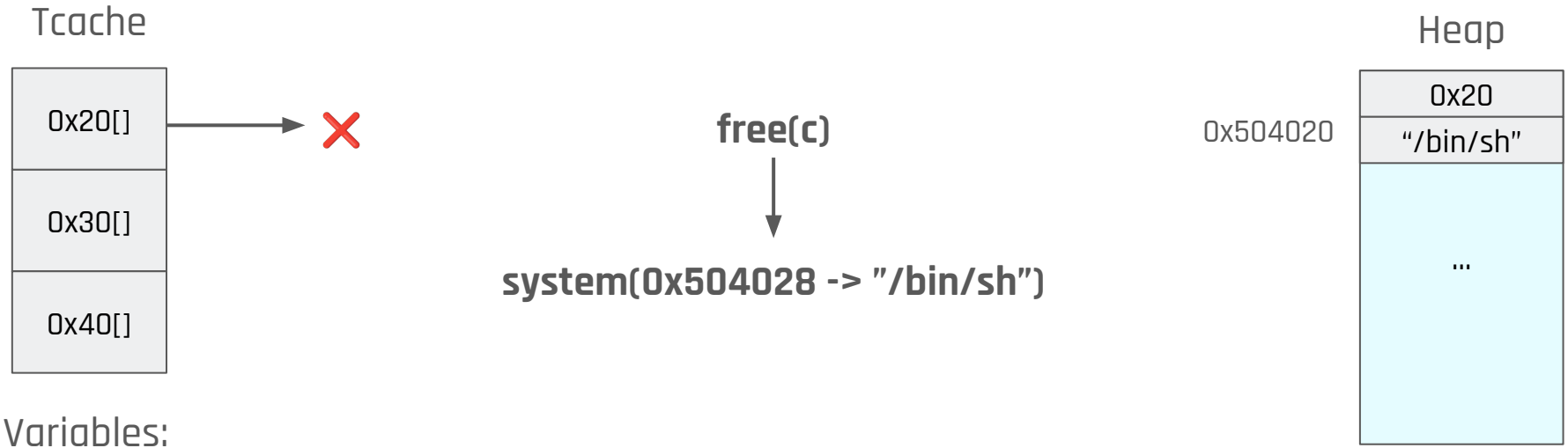


Variables:

a = 0x504028
b = 0x504028
c = 0x504028

d = free@got

Tcache Dup Attack



Variables:

a = 0x504028
b = 0x504028
c = 0x504028

d = free@got

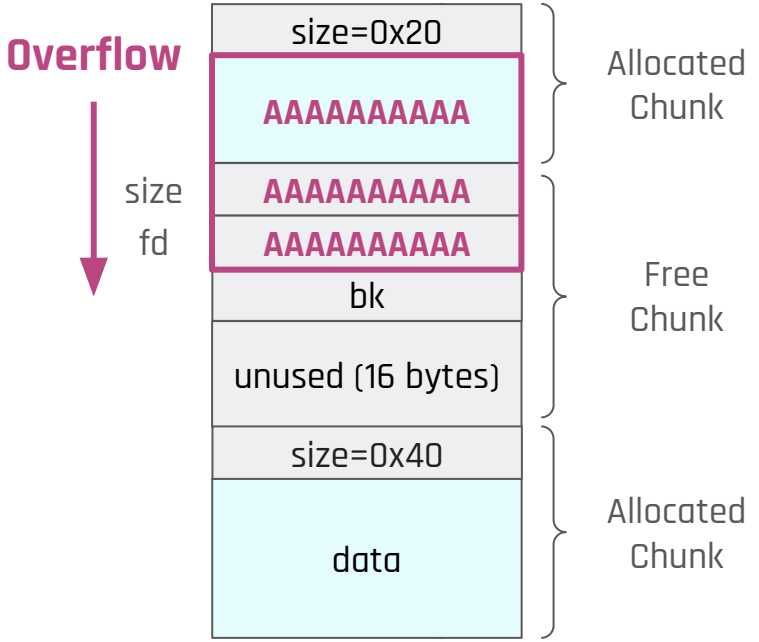
Tcache Dup Attack

- The tcache dup attack is no longer possible in recent versions of libc due to the mitigations introduced:
  - `tcache_entry->key` flag specifies whether a chunk is present in the tcache. It prevents double-free attacks
  - `tcache->next` pointers are now protected.
    - xor'ed with the address of the chunk itself,
    - if it is possible to poison the `next` pointer, a heap leak is required
  - There is also a check on the number of entries in a given tcache bin. If the counter is at 0, the tcache bin will be ignored

Heap Vulnerabilities: Heap Overflow

Heap Overflow

- Just like stack-based buffer overflows, except on the heap
- Buffer over-reads can allow attackers to leak libc and heap pointers
- Exploitation is more nuanced than stack-based buffer overflows,
  - There is no clear exploitation target (saved rip)
  - Depends heavily on the application



Heap Overflow - What can they achieve?

- Overwrite data, changing the behavior of an application
  - I.e., changing control variables, pointers stored on the heap, etc.
- Manipulating chunk metadata
  - In some applications, a single null-byte overflow can be escalated to a full exploit that spawns a shell

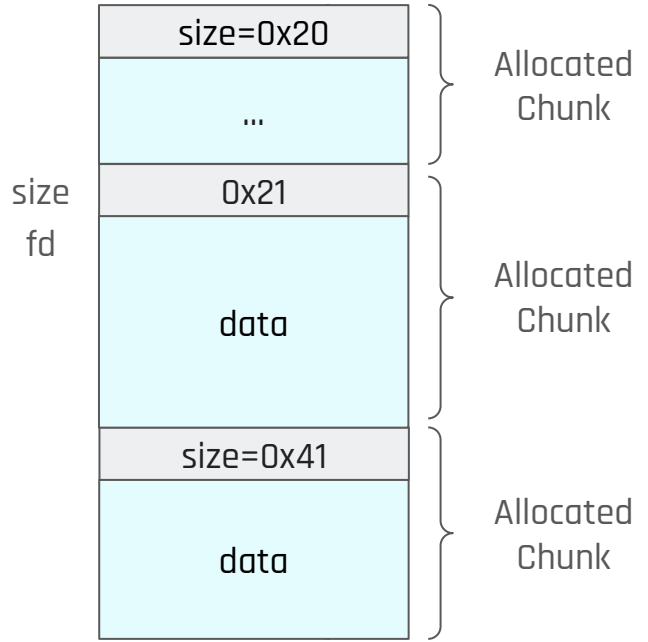
Heap Exploitation Techniques and Targets

Heap Attack Techniques - Glossary

- **Unsafe Unlink (historical)** - Abuse the UNLINK macro, called when moving chunks from the smallbin list, to achieve an arbitrary write primitive (requires UAF/BOF/etc.)
- **Tcache/Fastbin Poisoning** - Overwriting the **next/fd** pointer of a chunk in a fastbin or a tcache list to hijack the return value of `malloc`
- **Chunk Forging** - Create a "fake chunk" at an attacker-controlled memory location
- **Overlapping Chunks** - Corrupt chunk metadata to achieve a heap layout where a free chunk is located inside an allocated chunk
- **Heap Spraying** - In the absence of heap leaks, payloads can be placed repeatedly (sprayed) across the heap, increasing the odds of a correct guess, like in a NOP slide

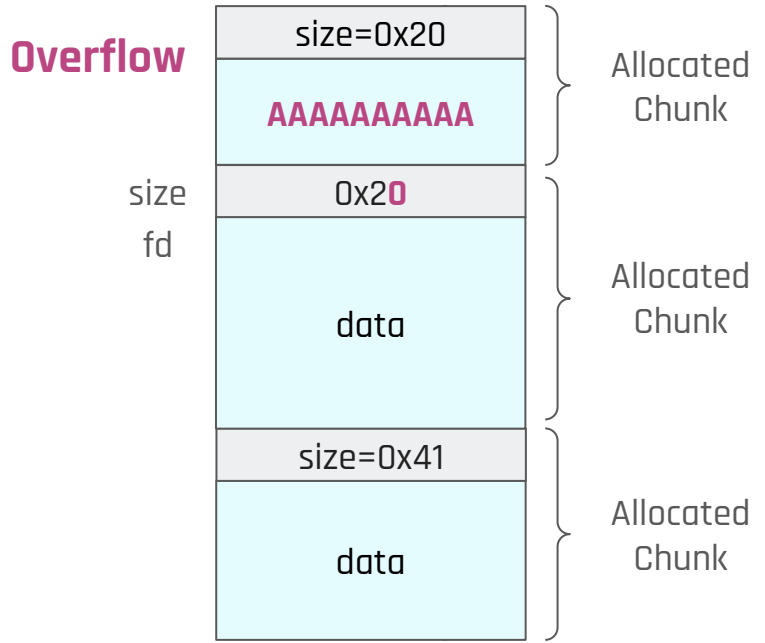
Heap Attack Techniques - Overlapping Chunks

- **A single null-byte overflow** can be used to overwrite the *prev in use* bit of an allocated chunk
- **When free**'d, if possible, it will be **coalesced with the previous chunk**, which is still allocated
- An attacker can then **request a new chunk** (malloc) and **obtain a reference to an already allocated chunk**
- After freeing it, the **attacker** just **obtained** a dangling **pointer to a free chunk**



Heap Attack Techniques - Overlapping Chunks

- **A single null-byte overflow** can be used to overwrite the *prev in use* bit of an allocated chunk
- **When free**'d, if possible, it will be **coalesced with the previous chunk**, which is still allocated
- An attacker can then **request a new chunk** (malloc) and **obtain a reference to an already allocated chunk**
- After freeing it, the **attacker** just **obtained** a dangling **pointer to a free chunk**



Attack Targets

- Common goal: Hijack Control-flow and achieve arbitrary code execution
- Arbitrary reads/writes are the most valuable primitives
  - Can be achieved through the attack techniques we've seen, and many more!
- There are several possible targets that allow hijacking control-flow:
  - malloc/free hooks
  - exit handlers
  - GOT overwrite
  - application specific points of failure (C++ vtables, global function pointers, etc.)

Attack Targets - Hooks

- Older versions of glibc allow registering hooks, i.e., functions that are called before certain functions, with the same arguments as the original
- `__malloc_hook`, `__free_hook`, `__realloc_hook` are examples of hooks that make for excellent exploitation targets
  - An attacker can overwrite thee `__free_hook` with `system`, and make the program free a chunk that contains the string "/bin/sh" to obtain a shell, since the hook (now system) is called with the argument of free ("/bin/sh")
- Hooks have been removed in recent versions of libc, since they were mostly used for exploitation instead of debugging...

Attack Targets - Exit Handlers

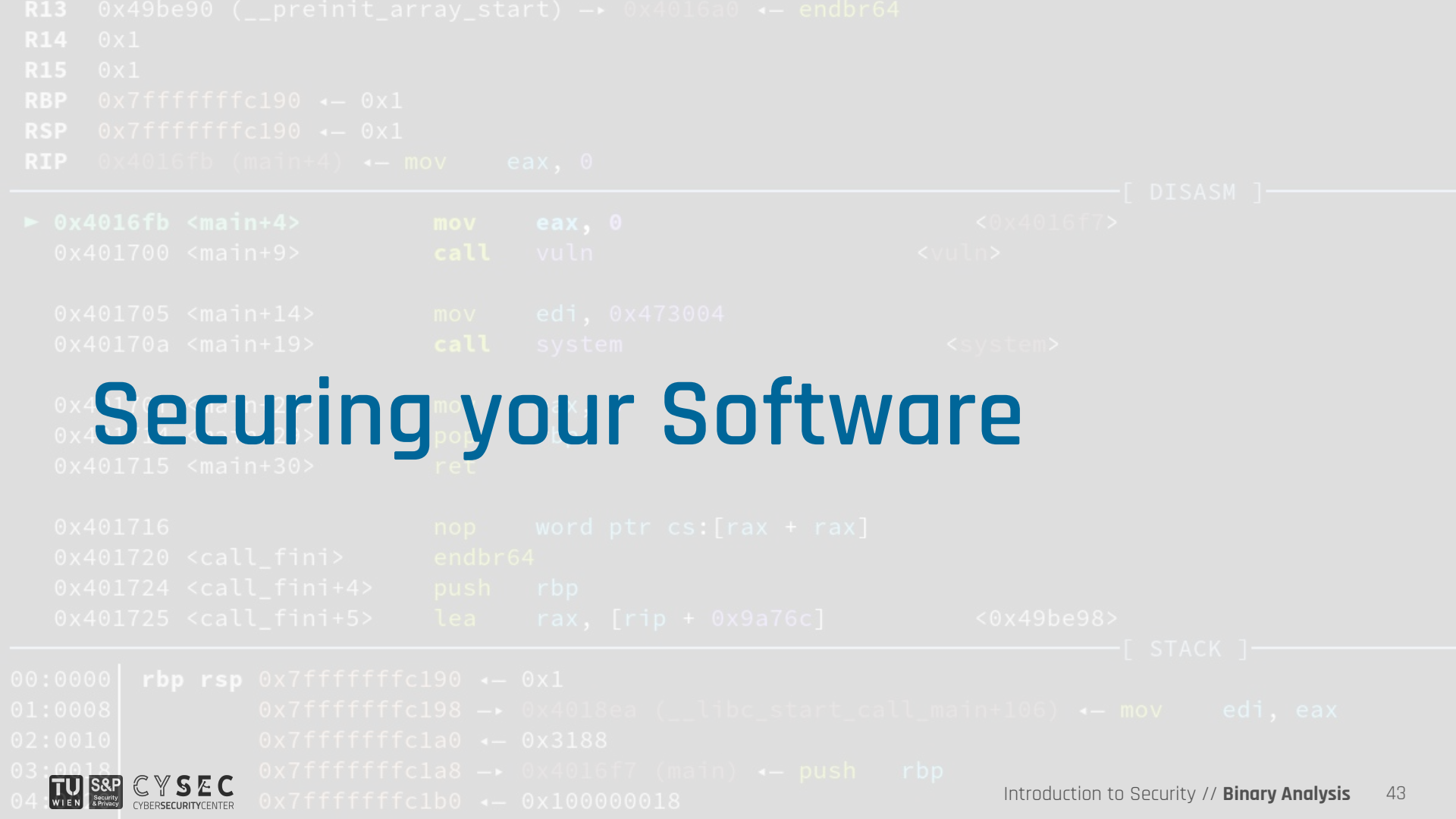
- When calling the libc `exit` function, all handlers registered through atexit() and on_exit() are called, before the call to the _exit() syscall
- An attacker can populate these lists with an arbitrary write primitive to hijack the instruction pointer
- Current versions of libc "*mangle*" (xor) these pointers with a global key

If an attacker can leak this key, they can forge "*mangled*" pointers.

Depending on the libc compilation, this global key could be writable. An attacker can overwrite it with 0 to effectively disable this mitigation.

Attack Targets - GOT Overwrite

- The Global Offset Table contains the addresses of libc functions used in the program
- It is referenced by the Procedure Linkage Table – a jump table present in the .plt section of the ELF binary that resolves library addresses at runtime
- By overwriting GOT entries, an attacker can hijack the instruction pointer when a given function is called
  - i.e., replacing **printf@got** with **system**
- Programs can be compiled with the **FULL RELRO** protection to make the GOT read-only
  - this is not a default compiler option since it greatly slows startup time, since all library symbols must be resolved before the program starts, every time it is executed.



Securing your Software

Safe Dynamic Memory

- Do not access freed memory
- Free memory when no longer needed
- Guarantee that enough memory is allocated for the given object
- Remove dangling pointers by setting them to NULL after freeing

```c
struct node {
 int value;
 struct node *next;
};

void insecure_free_list(struct node *head) {
 for (struct node *p = head; p != NULL;
           p = p->next) {
      free(p);
 }
}

void secure_free_list(struct node *head) {
 struct node *q;
 for (struct node *p = head; p != NULL;
           p = q) {
      q = p->next;
      free(p);
 }
}
```

p->next is accessed after p is freed

Safe String and I/O operations

- Make sure all strings are null terminated
- Guarantee that storage for strings has enough space for data and the null terminator
- Use the **n** version of string functions (strncpy, strncmp, strncat, etc.)
  - Ensure that the function you are using writes the string terminator character

```
void copy(size_t n, char src[n],
     char dest[n]) {
  size_t i;

  for (i=0; src[i] && (i<n-1); ++i) {
    dest[i] = src[i];
  }

  dest[i] = '\0';
}
```

Safe String and I/O operations

- Define maximum size macros and use them when declaring arrays, and as arguments to string functions
- Use “safe” I/O functions like `fgets`
- Check the result of I/O operations
  - return value informs if there was an error

```c
#define MAX_LEN 0x20

int main(){
  char buffer[MAX_LEN];
  if (fgets(buffer, MAX_LEN, stdin)) {
    p = strchr(buf, '\n');
      if (p) {
        *p = '\0';
      } else { /* Handle error */ }

    printf("%s\n", buffer);
    return 0;
}
```

Follow a security-focused coding standard!
https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf

Resources

- **Malloc security checks -** https://heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/security_checks
- **Malloc internals -** https://www.sourceware.org/glibc/wiki/MallocInternals
- **How2heap -** https://github.com/shellphish/how2heap
- **Glibc source code -** https://elixir.bootlin.com/glibc/latest/source
- **Temple of PWN -** https://www.youtube.com/playlist?list=PLiCcquURxSpbD9M0ha-Mvs-vLYt-VKlWt
- **LiveOverflow -** https://www.youtube.com/playlist?list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN
- **GEF gdb extension -** https://github.com/hugsy/gef