# Memory Corruption

Roy Xu
OSIRIS Lab Hack Night

OSIRIS

# What is the heap?

- Dynamically allocated memory
  - Request memory
  - Release memory

- Allocated at runtime

- Pointers used to reference data

# Why use the heap?

- Stack variables are local (can't be accessed between functions)
  - Heap is persistent

- Allocate as much as you want
  - Unsure at runtime how much space needed

- Commonly used for larger objects

# Heap Implementations

- Many implementations have slightly different behavior
  - avrlibc
  - dlmalloc
  - tcmalloc

- Different implementations have different tradeoffs

- For class we're focusing on the glibc implementation of malloc

# How to use the heap?

- malloc(size_t n)
  - Returns a pointer to newly allocated chunk of at least n bytes
  - Should align to 16 bytes (implementation specific)
    - Last nibble (4 bits) of malloc addresses should be 0
    - Includes heap metadata
    - Simplifies malloc internals

# Malloc Chunk
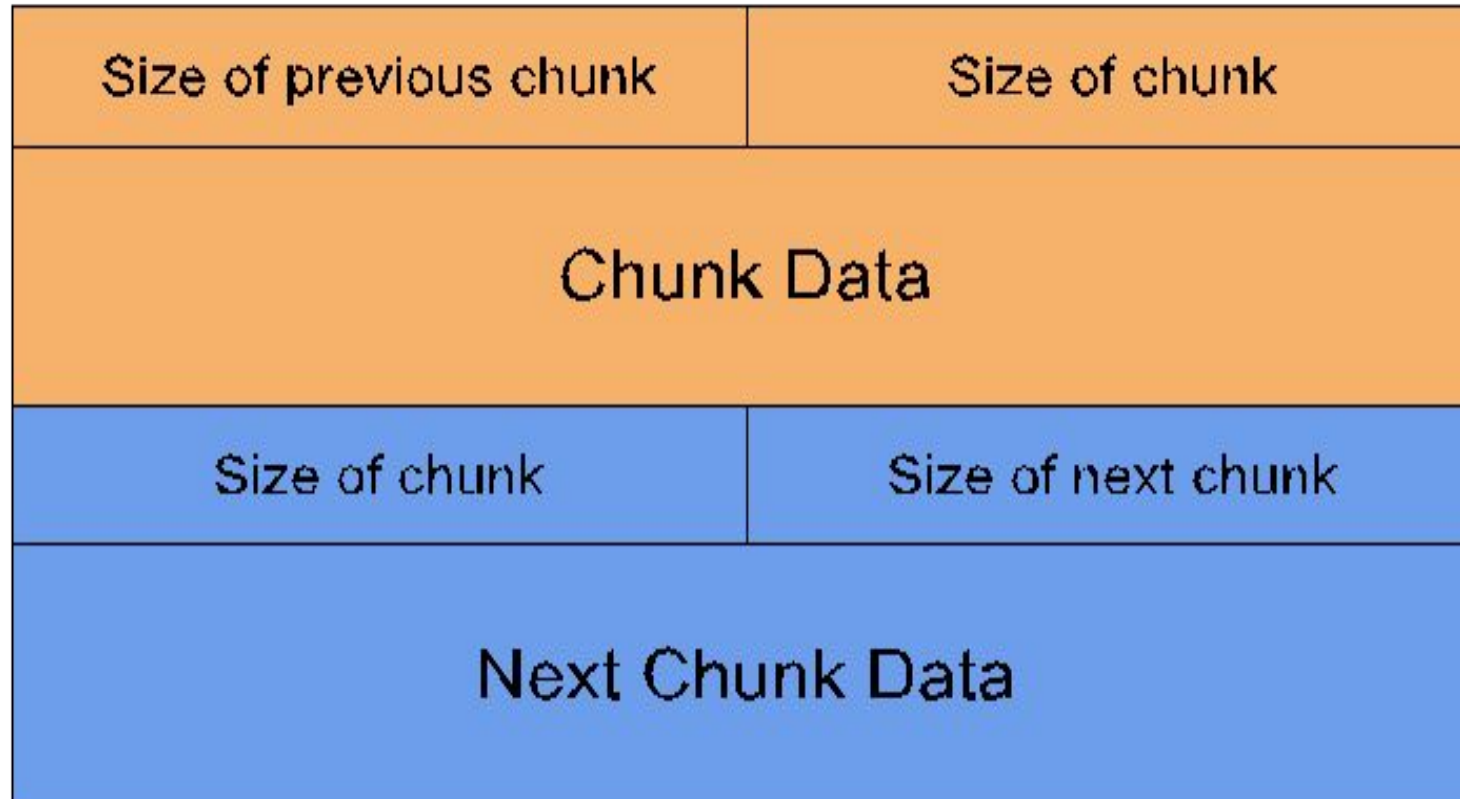
# Malloc Metadata

- Last 3 bits of size contain flags

  ○ PREV_INUSE – Set when previous chunk is allocated

  ○ IS_MMAPPED – Set when chunk is mmap'd (for larger allocations)

  ○ NON_MAIN_ARENA – When using a thread specific arena

- 0x21 – Size is 0x20 and previous chunk is allocated

a = malloc(0x8)

b = malloc(0x28)

c = malloc(0x20)

d = malloc(0x14)

| prev size 0x0 | size 0x21 |
|---|---|
| data | alignment |
| prev size 0x0 | size 0x31 |
| data | data |
| data | data |
| data | size 0x31 |
| data | data |
| data | data |
| prev size 0x0 | size 0x21 |
| data | data |
| alignment / data | |

# How to use the heap? (cont.)

- free(void* p)
  - ○ Release the chunk of memory pointed to by p
  - ○ Can have unintended effects if p has already been freed
  - ○ Expected that you null out p after
    - ■ p = 0;

# Freed Chunk

| Size of previous chunk | Size of chunk |
|---|---|
| Forward Pointer | Back Pointer |
| | |
| Size of chunk | Size of next chunk |

# Free

free(a)

free(b)

free(c)

free(d)

# Malloc First Fit

- malloc tries to reuse space efficiently

- freed chunks are placed in a linked list according to size

- pointers in freed chunks point to forward and back chunk
  - small chunks are stored in singly-linked list (no back pointer)
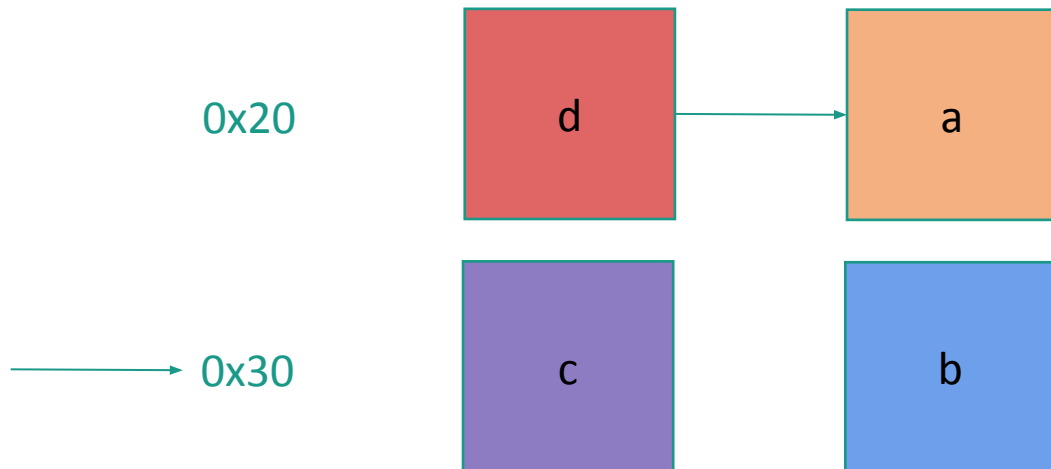
e = malloc(0x10)

f = malloc(0x10)

| | | |
|---|---|---|
| 0x602000 | prev size 0x0 | size 0x21 |
| 0x602010 | new data | new data |
| 0x602020 | prev size 0x0 | size 0x31 |
| 0x602030 | | data |
| 0x602040 | data | data |
| 0x602050 | data | size 0x31 |
| 0x602060 | 0x602020 | data |
| 0x602070 | data | data |
| 0x602080 | prev size 0x0 | size 0x21 |
| 0x602090 | new data | new data |
| 0x6020a0 | alignment | old data |

# Use After Free (UAF)

- What happens when pointers aren't nulled out after a free?

- Dangling pointers – leftover references to freed chunks

- How can we abuse this?

# UAF (cont.)

- Structs are commonly stored on the heap

```
struct example {
    void (* toUpper)(char *);
    char buffer[16];
}
```

Our example struct converts our buffer to uppercase characters and prints them out

```
struct exploit{
    int number;
    int foo;
    char bar[8];
}
```

# UAF (cont.)

Create an example struct on the heap
     a = malloc(0x18)

| prev size | size 0x21 |
|-----------|-----------|
| function  | buffer    |
| buffer    |           |

| prev size | size 0x21 |
|-----------|-----------|
| number    | foo       |
| bar       |           |

# Exploit



What can we write into the exploit struct
- any arbitrary function (system?)
- /bin/sh;

What happens if we still have a reference to the example struct?
- Call system on /bin/sh;

# UAF (cont.)

- What if we don't have a convenient function pointer?

- Often there will be option to edit a value in the struct

- Overwrite GOT? Return address?

# UAF Info Leaks

- Often string pointers will be stored on the heap

- Overwrite with info you want to leak

# UAF Mitigations

- Don't leave references to freed chunks

# UAF in the wild

- Popular vulnerability especially in browser exploits

- Doesn't require memory corruption

- Can be used for info leaks

- Detecting UAF in complex applications can be very difficult
  - Just not leaving dangling pointers is easy to say, hard in practice

- Exist only in certain states of execution
  - Hard to detect statically

# Heap overflows

- Similar to buffer overflows but fewer protections
  - Canaries aren't a thing in the heap

- Similar to UAF, just need to write over function pointers or strings

# Things to take into account

- Heap is affected by ASLR

- Typically only Read and Write
  - No shellcode

# Heap Spray

- Not an exploit, more of a technique to make exploits more reliable

- Fill heap with large amount of data relevant to exploit

- Helps assist with ASLR

- On 32 bit systems, address space is 4GB
  - If we spray with 3GB of data, 75% chance that 0x23456789 (random address) exsits

- 64 bit system has $2^{64}$ bytes of address space
  - Not realistic for spraying
  - Still useful if you can narrow down target to spray

# Heap Grooming/Feng Shui

- Heap allocations are predictable

- Doing the same allocations and deallocations leads to same result
  - If malloc implementation is identical

- Arrange chunks in such a way that makes exploiting easier

# Metadata Exploits

- Exploit the metadata of the chunk

- Force malloc internal functions to give arbitrary write or read
  - Manipulate size of chunks to create overlaps
  - Overwrite forward and back pointers

- Very popular in CTFs

- Come to lab if you're interested in learning more

# Double Free

- Expected that a freed chunk will not be freed again

- What if we break the rules?

- Check exists to prevent double freeing a chunk

- What if we free another chunk in between a double free?

# Double Free (cont.)

Create two chunks

Free List


A


B
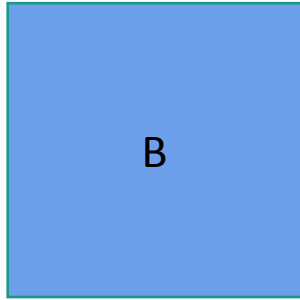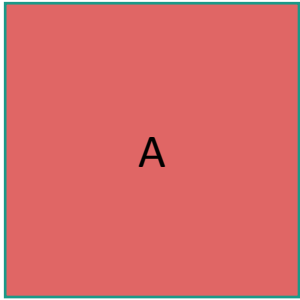
# Double Free (cont.)

Free A

B
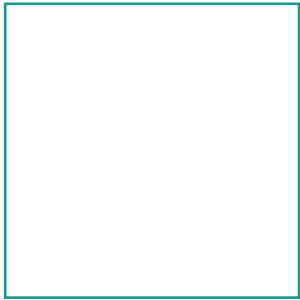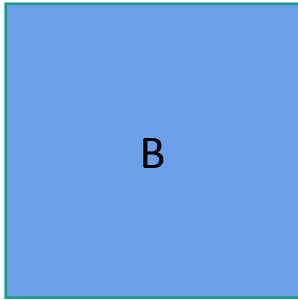
Free List

A
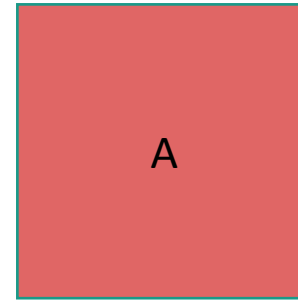
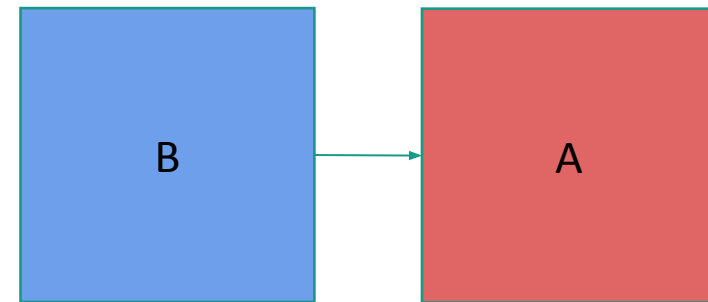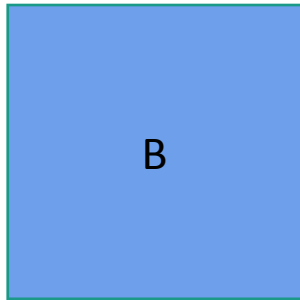# Double Free (cont.)

Free B
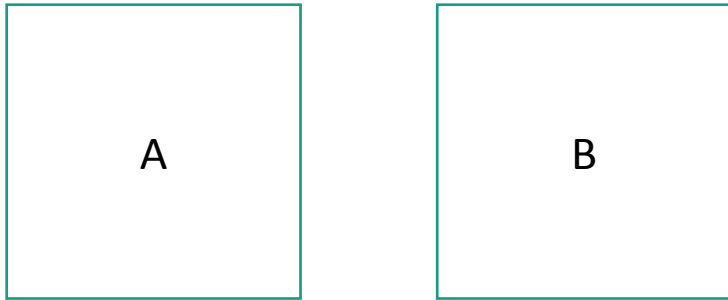
Free List

# Double Free (cont.)
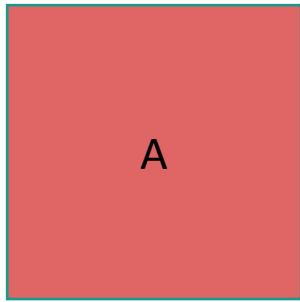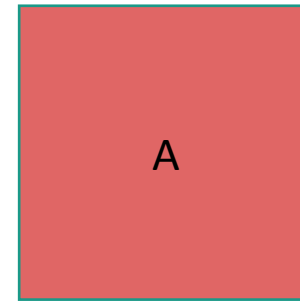
Free A

| A | B |
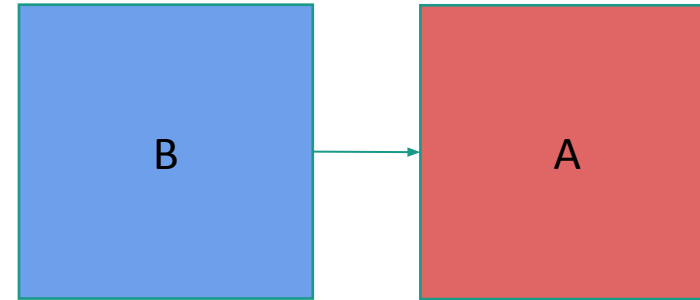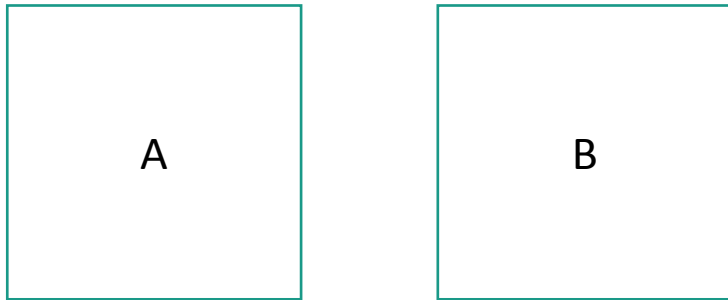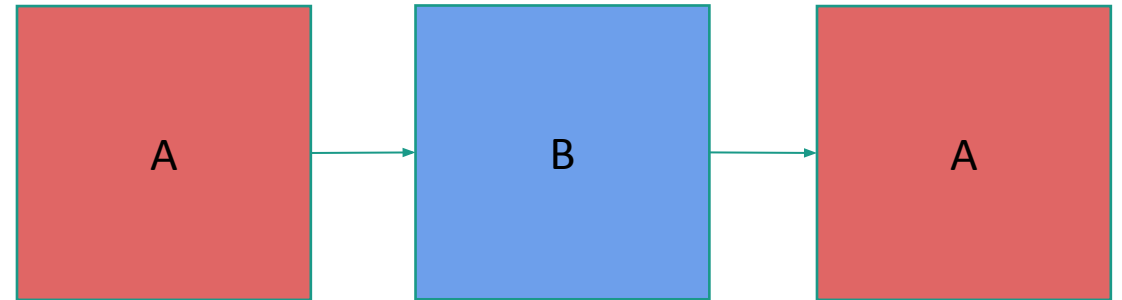
Free List

# Double Free (cont.)

Malloc new chunk
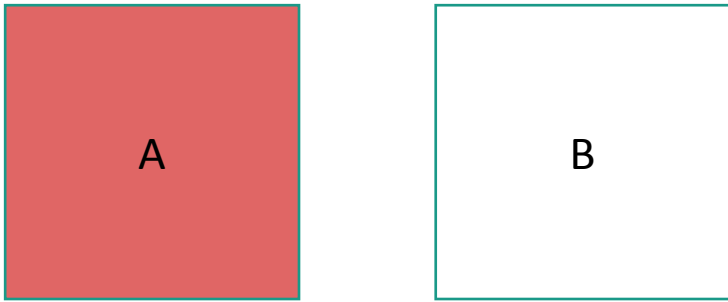
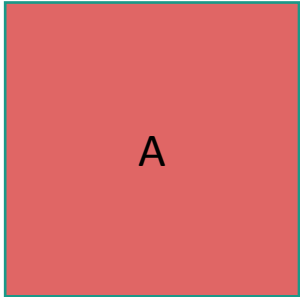Free List

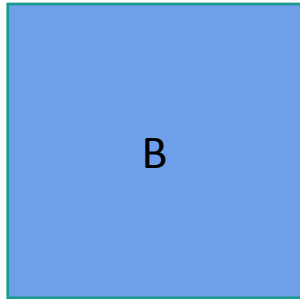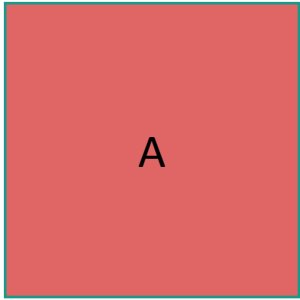# Double Free (cont.)

Malloc another chunk

A

B

Free List

A

# Double Free (cont.)

Malloc another chunk



Two references to the
same chunk

Free List

# Double Free Exploit

- We have two references to the same chunk

- How to exploit?

- Similar to UAF, you can write in an exploitable struct

- Freeing a chunk places forward pointers in the chunk
  - Leads to heap leak