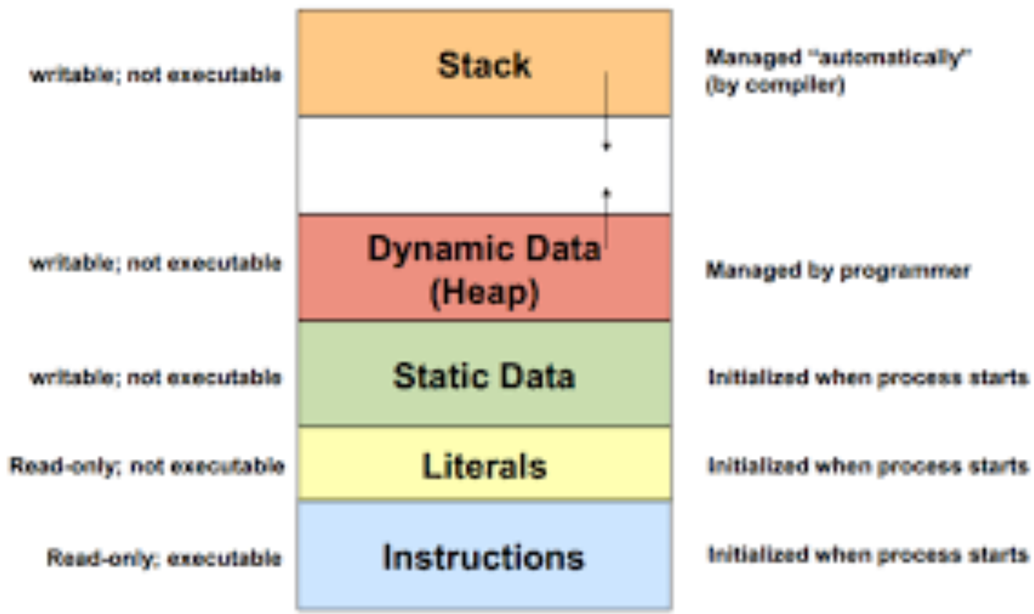
CS165 – Computer Security

Heap Attacks

February 6, 2024

Heap Memory

- What is heap memory?



Heap Memory

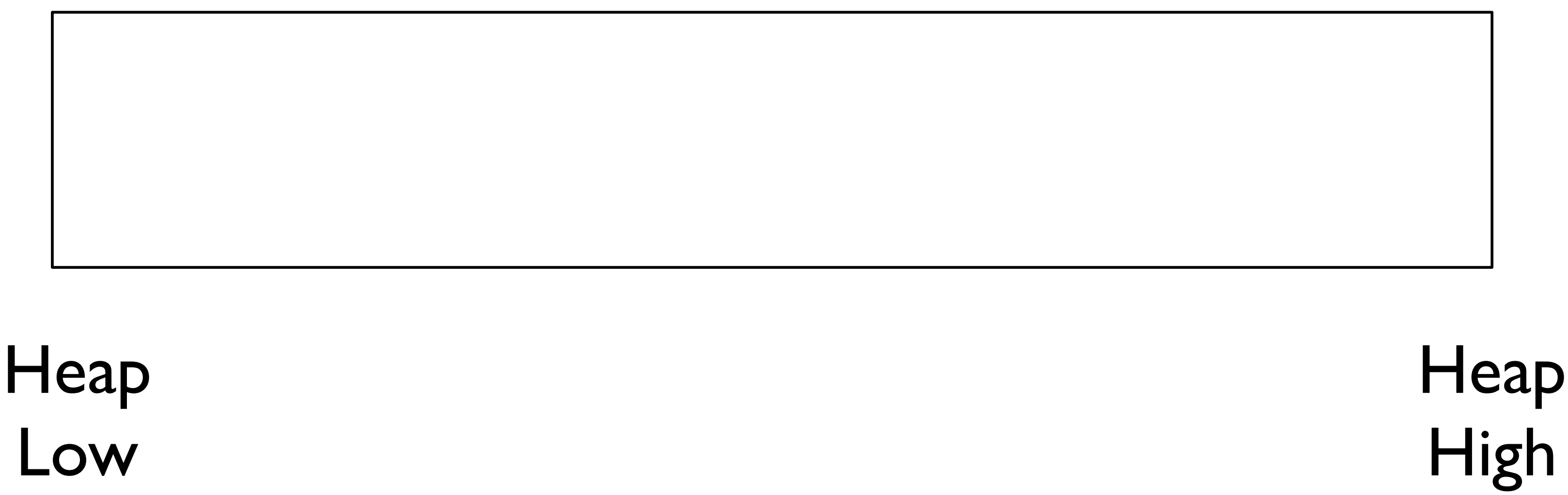
- Another region of memory that may be vulnerable to attacks is heap memory
  - Attacks similar to those on stack memory, such as buffer overflows, are possible
    - Although the attack techniques differ somewhat
      - Target metadata – kinds of similar, but different effect
      - Target data – data may include code pointers

Heap Memory

- Another region of memory that may be vulnerable to attacks is heap memory
  - However, the complexity of managing heap memory brings other attacks into consideration
  - Some are rather complex (e.g., heap spraying)
- Today, we will look at the new attack types and attack techniques for the heap
  - Just a couple of simpler ones

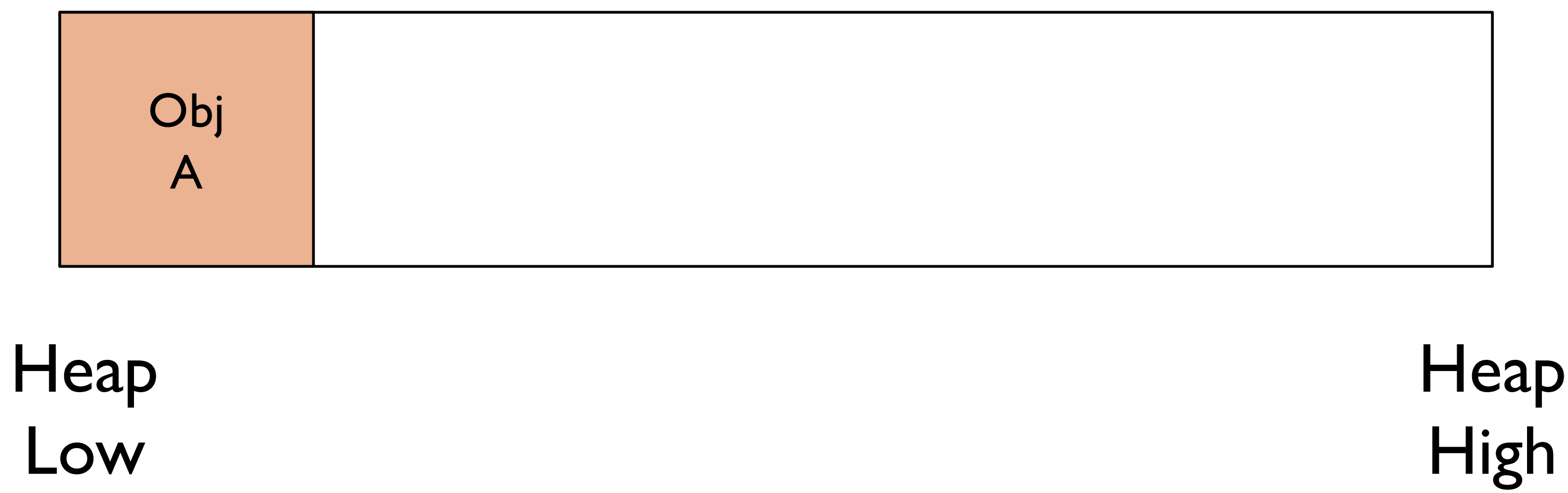
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - It is a contiguous region of virtual memory (can expand)



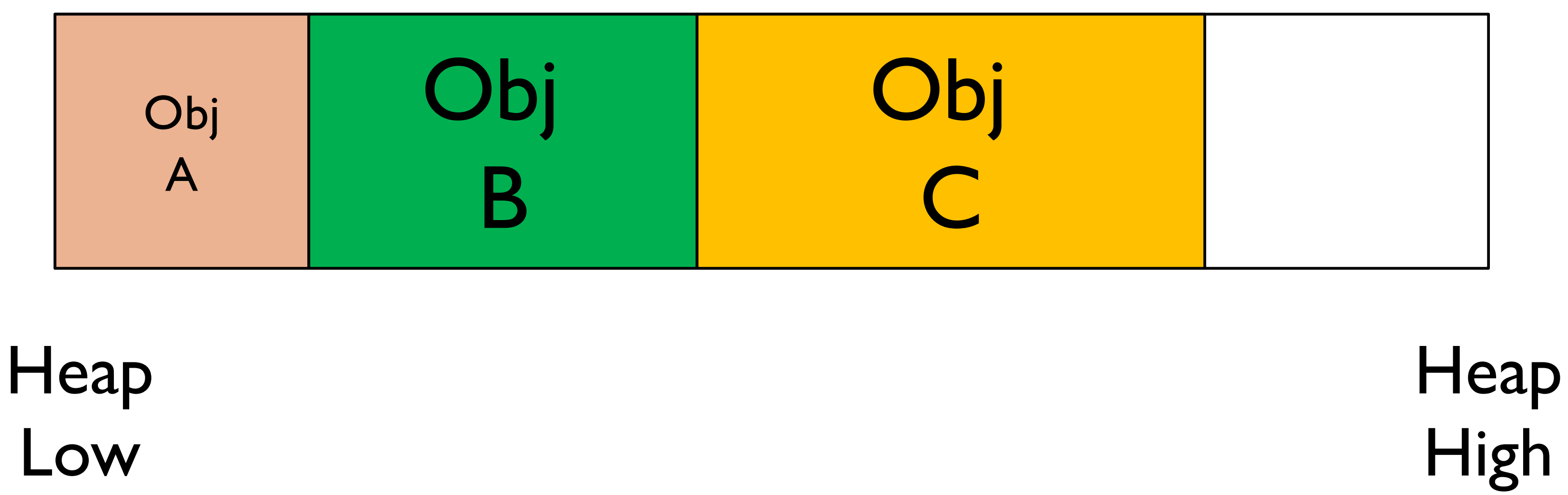
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - An allocation is assigned a contiguous range of virtual memory within the heap (e.g., on malloc)



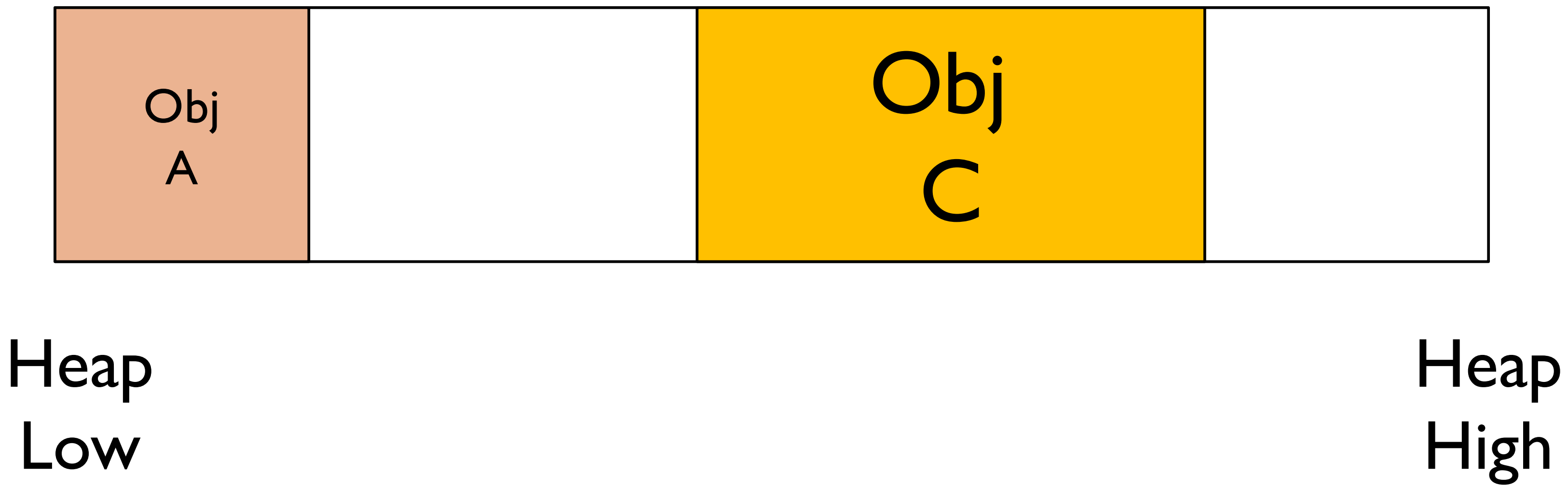
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - An allocation is assigned a contiguous range of virtual memory within the heap (e.g., on malloc)



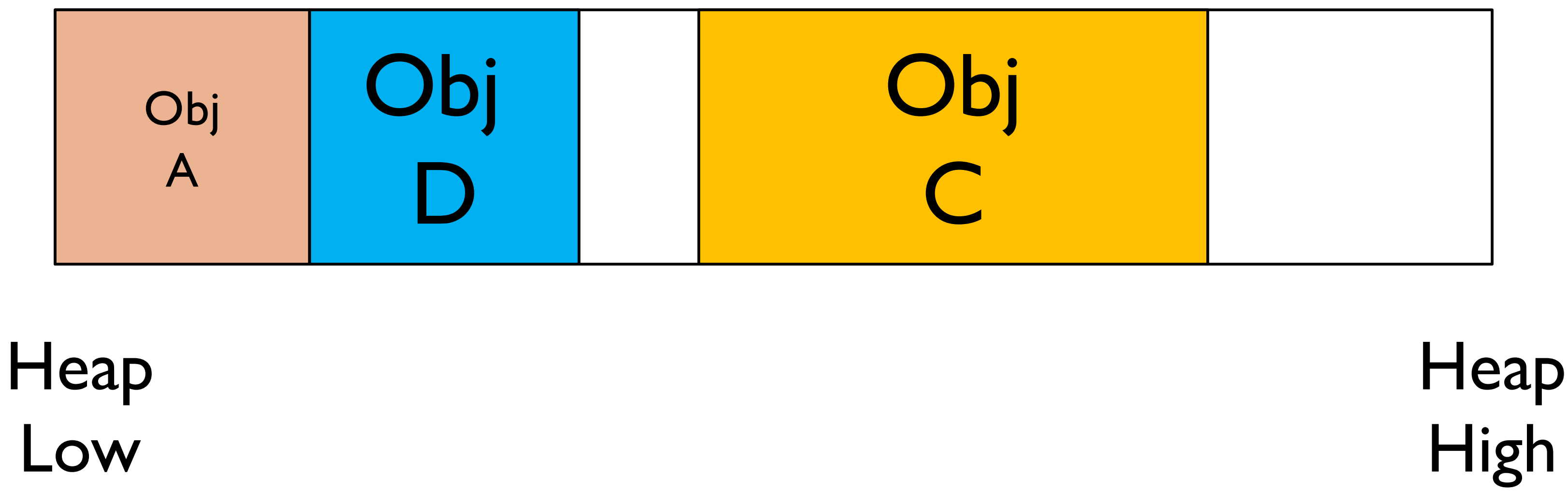
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - Memory from a specific allocation may be deallocated and reclaimed when no longer needed (e.g., on "free")



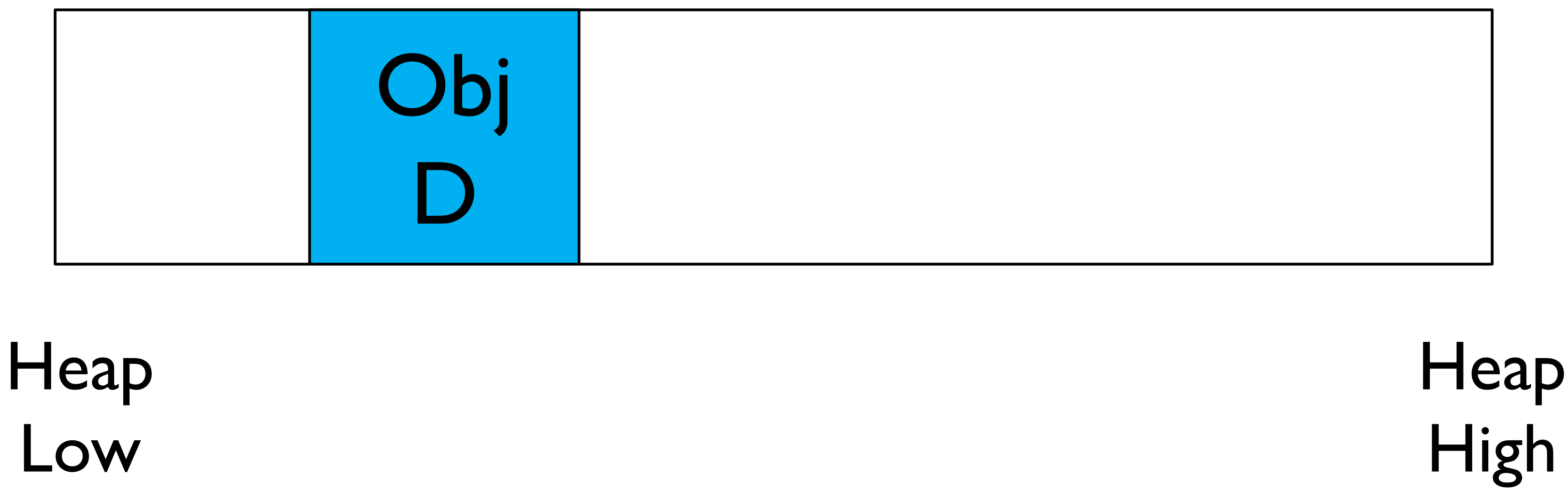
Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - Memory from a specific allocation may be reclaimed when no longer needed (e.g., on “free”) and reused



Heap Memory

- What is heap memory?
  - The heap memory region is where dynamic memory allocations take place
  - If you forget to reclaim memory no longer in use, that memory region is lost (i.e., memory leak)



Review: Stack Buffer Overflow

- Suppose that PacketRead causes an overflow on the memory region of the variable "packet" below
  - What is the potential impact?

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

Stack Buffer Overflow

- Suppose that PacketRead causes an overflow on the memory region of the variable "packet" below
  - What is the potential impact? "authenticated" may be set

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

Heap Buffer Overflow

- What happens if we allocate "packet" on the heap?
  - A buffer overflow of a buffer allocated on the heap is called a heap overflow – Impact?

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

Heap Buffer Overflow

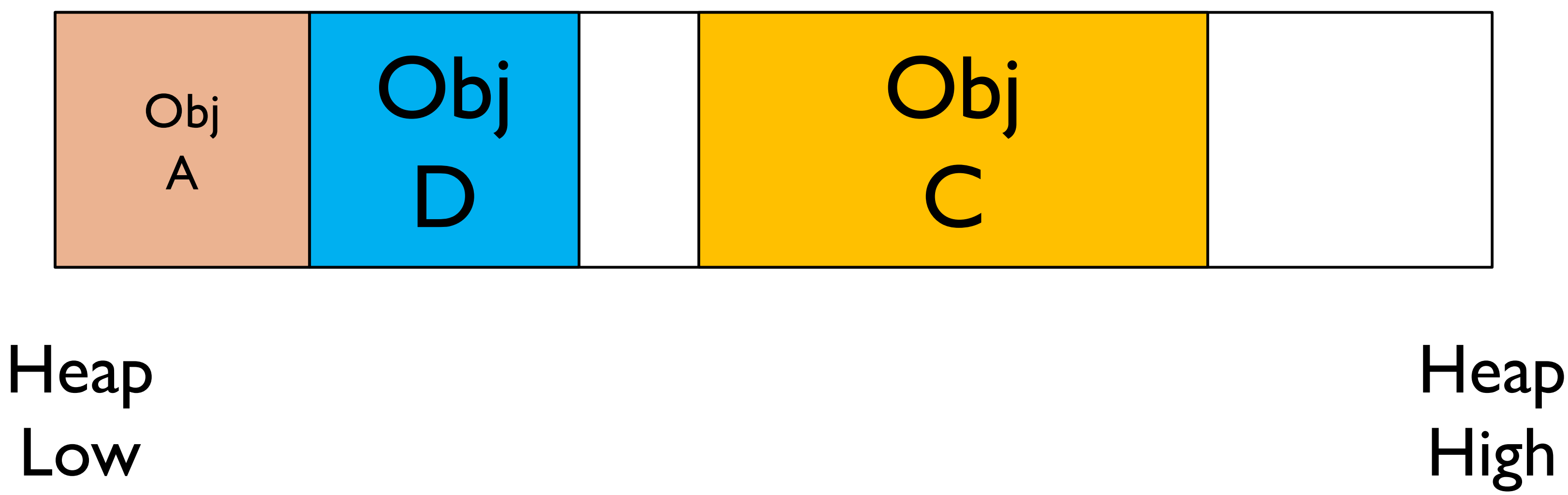
- While a heap overflow may impact heap memory regions, it won't impact stack memory (directly)
  - "authenticated" is unaffected, but something else may be affected

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

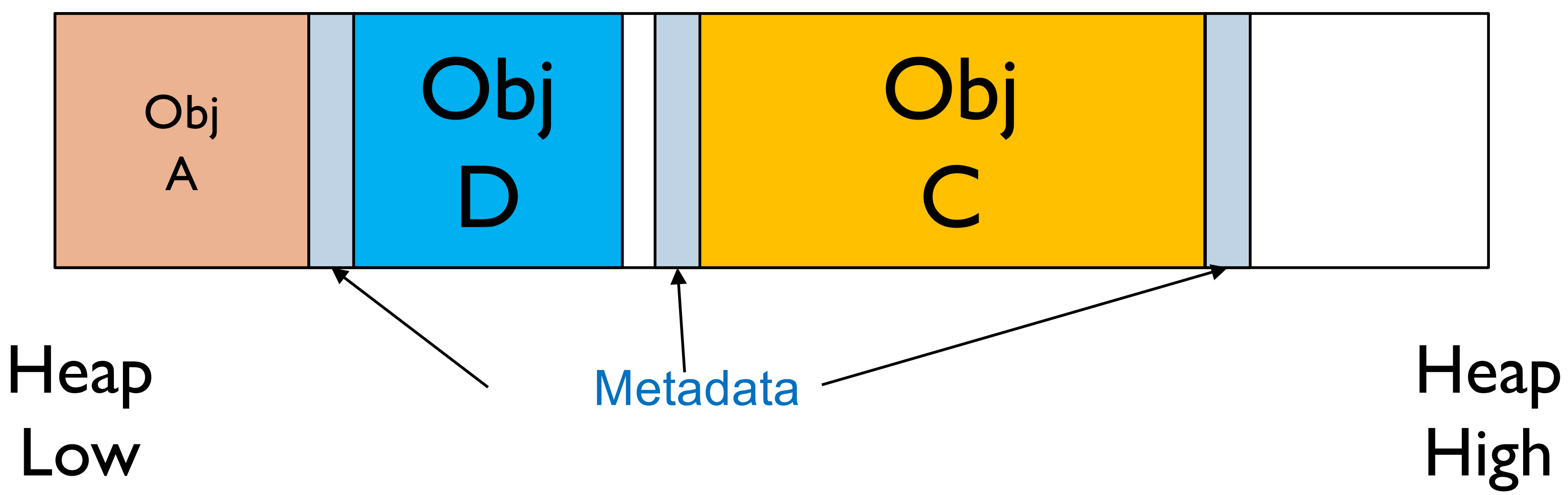
Heap Memory Layout

- The Heap Memory Layout below is idealized
  - Depends on the heap allocator
  - Many heap allocators store metadata with objects on the heap to manage the heap region



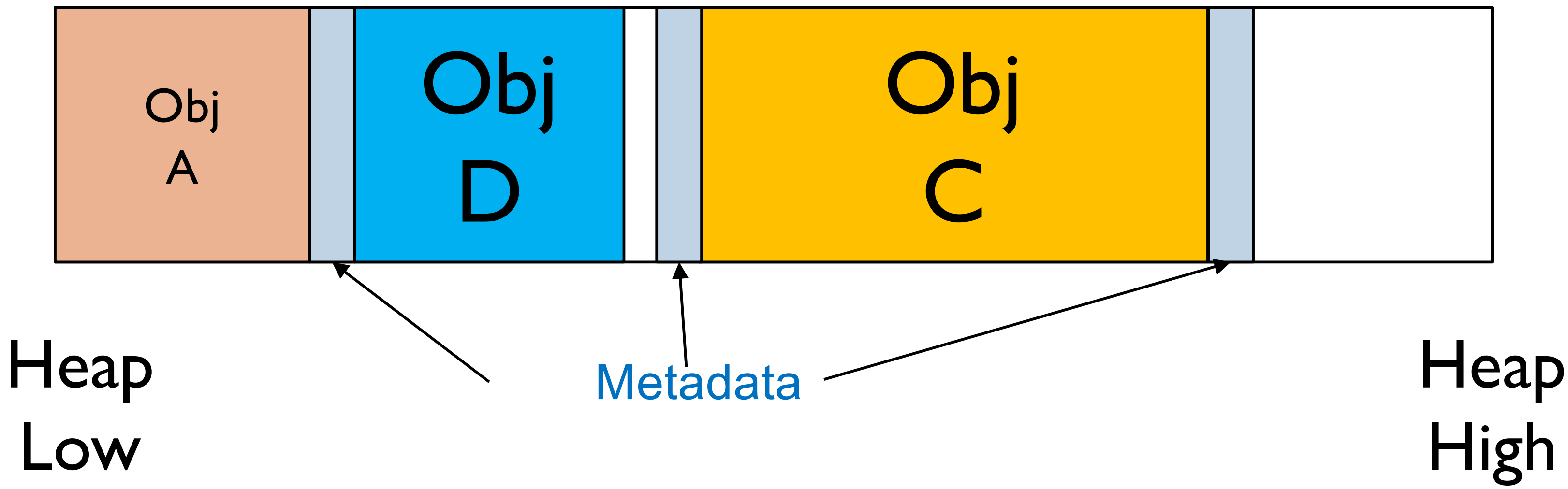
Heap Memory Layout

- The Heap Memory Layout often includes metadata
  - Depends on the heap allocator
  - Metadata is often placed between objects to store information needed to manage allocation state – e.g., sizes and status



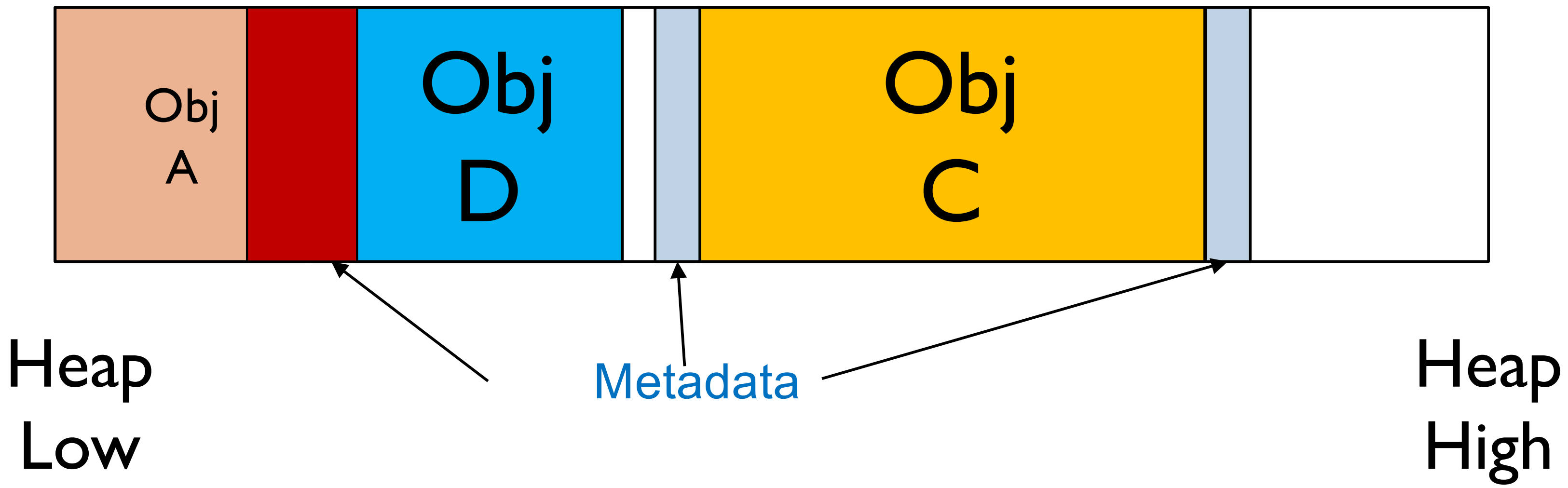
Attacks on Heap Metadata

- Overflow heap memory to modify metadata



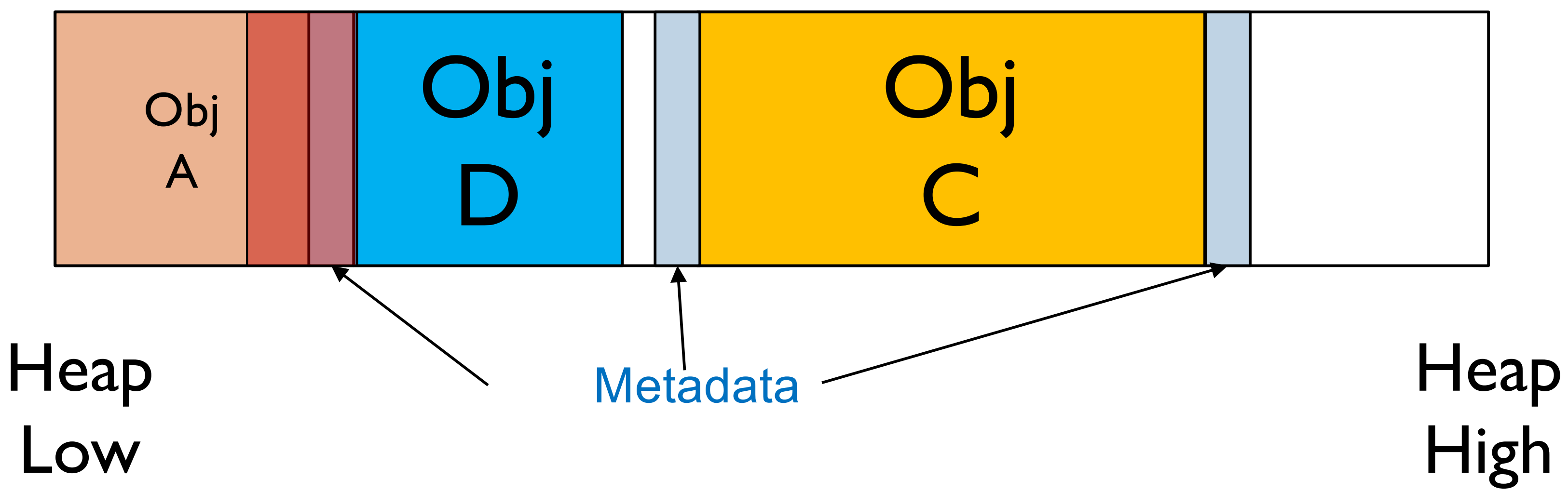
Attacks on Heap Metadata

- Overflow heap memory to modify metadata



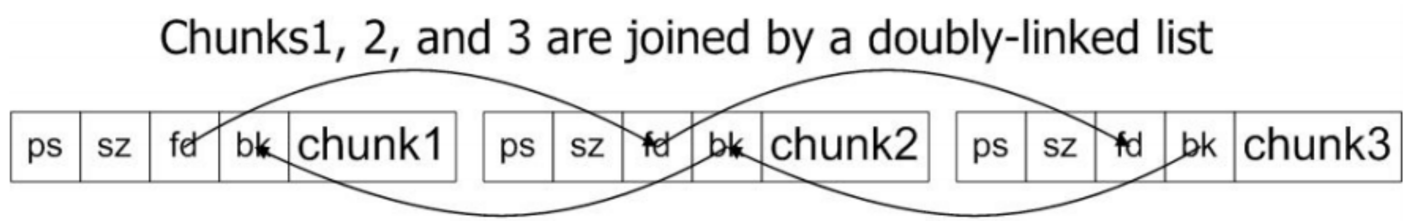
Attacks on Heap Metadata

- Overflow heap memory to modify metadata



Heap Metadata Maintains Chunks

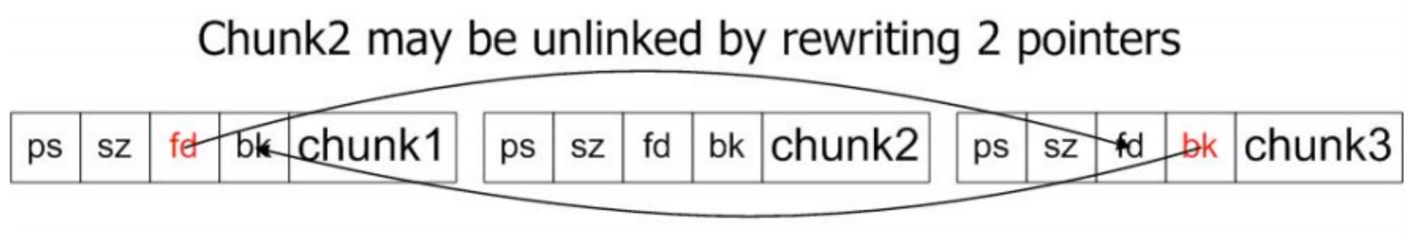
- Allocators maintain a doubly linked list of allocated and free "chunks"
  - Each allocated region (chunk) references…
    - The prior chunk (back)
    - The next chunk (forward)



    - Chunk 2 forward = address of Chunk 3
    - Chunk 2 back = address of Chunk 1

Remove a Chunk

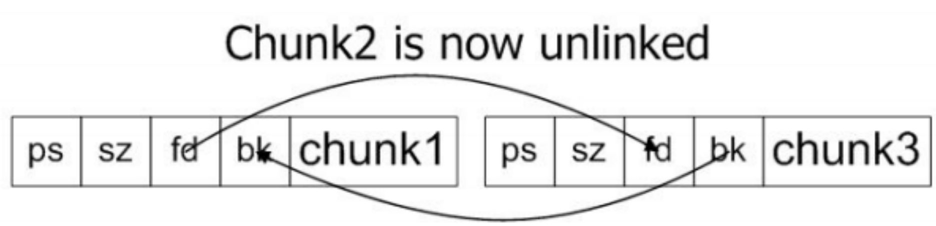
- Allocators maintain a doubly linked list of allocated and free “chunks”
  - Free a chunk by resetting the forward pointer of the back chunk and the back pointer of the forward chunk



    - Chunk 1 forward = address of Chunk 3
    - Chunk 3 back = address of Chunk 1

Remove a Chunk

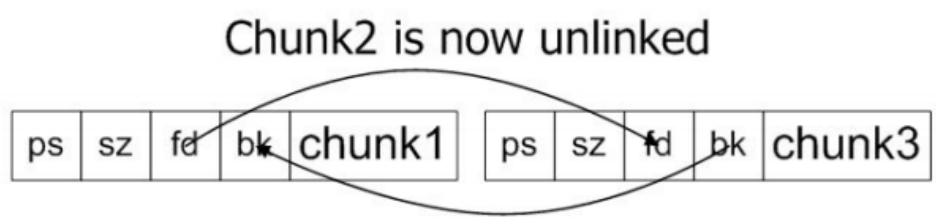
- Allocators maintain a doubly linked list of allocated and free "chunks"
  - Free a chunk by resetting the forward pointer of the back chunk and the back pointer of the forward chunk



- Chunk 1 forward = address of Chunk 3
- Chunk 3 back = address of Chunk 1

Remove a Chunk

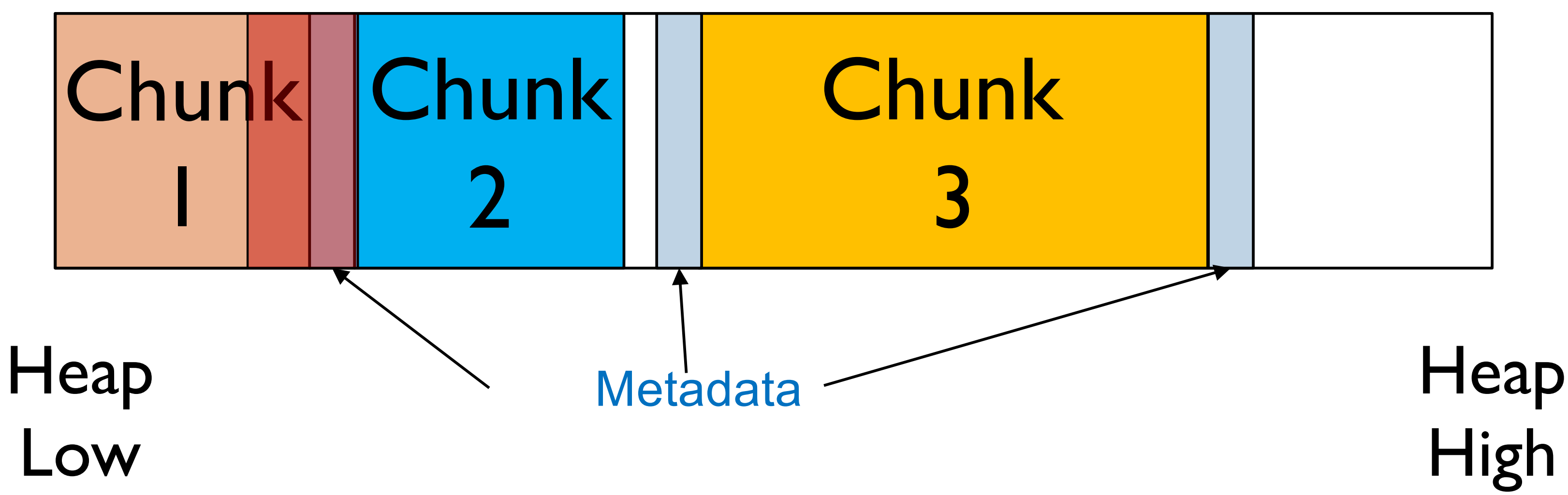
- Allocators maintain a doubly linked list of allocated and free “chunks”
  - Free a chunk by resetting the forward pointer of the back chunk and the back pointer of the forward chunk



- (Chunk1’s fd) Chunk2→bk→fd = Chunk2→fd; (Chunk3)
- (Chunk3’s bk) Chunk2→fd→bk = Chunk2→bk; (Chunk1)

Attacks on Heap Metadata

- How can you use a buffer overflow…
  - Say in Chunk1
- To exploit a "free" operation of Chunk2?



Attacks on Heap Metadata

- Modify the "fd" and "bk" pointer values of Chunk2
- Such that
  - Chunk2→bk is the location you want to write
    - Offset by distance to "fd" field
  - Chunk2→fd is the value you want to write
  - In Chunk2→bk→fd (location + fd) = Chunk2→fd (value)
- Result: A "write-what-where" vulnerability!
  - Or "arbitrary write primitive"

What is a Defense?

- How would you prevent this vulnerability?
  - Hint: What invariant would you expect for the forward and back pointers of Chunk2 prior to freeing it?

What is a Defense?

- How would you prevent this vulnerability?
  - Chunk2→bk→fd = ???
  - Chunk2→fd→bk = ???

What is a Defense?

- How would you prevent this vulnerability?
  - Chunk2→bk→fd = Chunk2
  - Chunk2→fd→bk = Chunk2
- Thus, we check in every free(chunk)

```
assert(chunk→fd→bk == chunk)
assert(chunk→fd→bk == chunk)
```

- To detect any tampering prior to free-ing

Heap Overflows

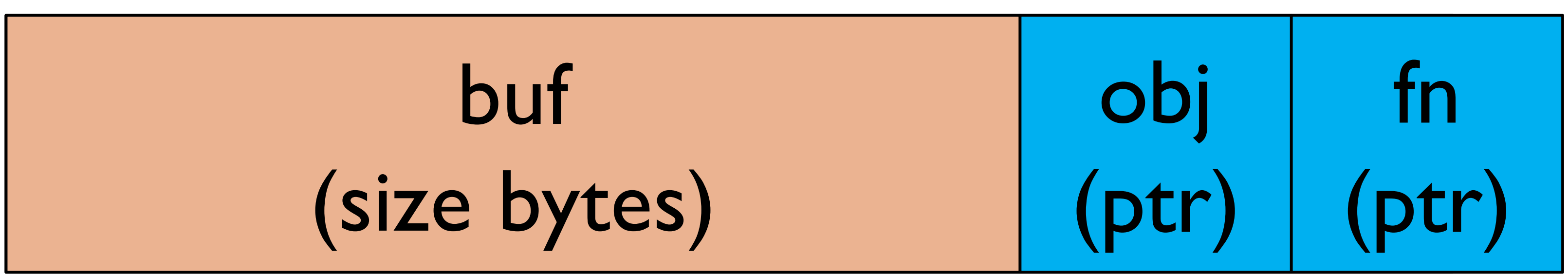
- Can be useful and hard to prevent

```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```

Heap Overflows

- Can be useful and hard to prevent

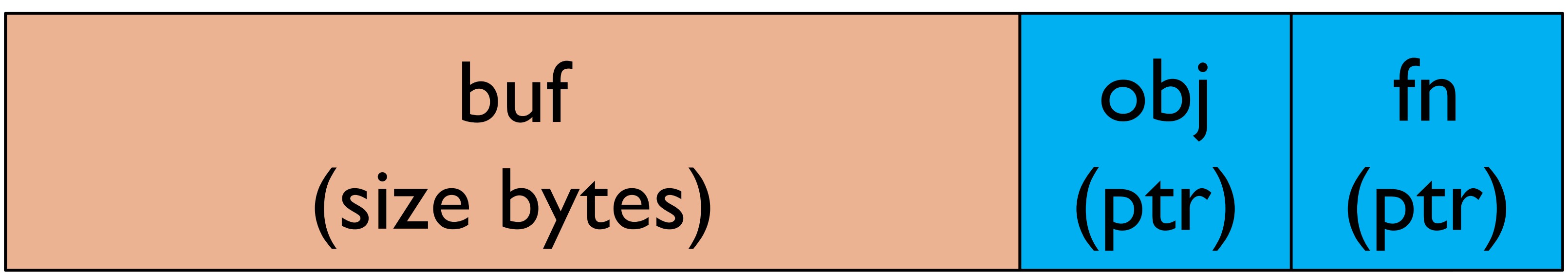
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Heap Overflows

- What can an overflow of "buf" cause?

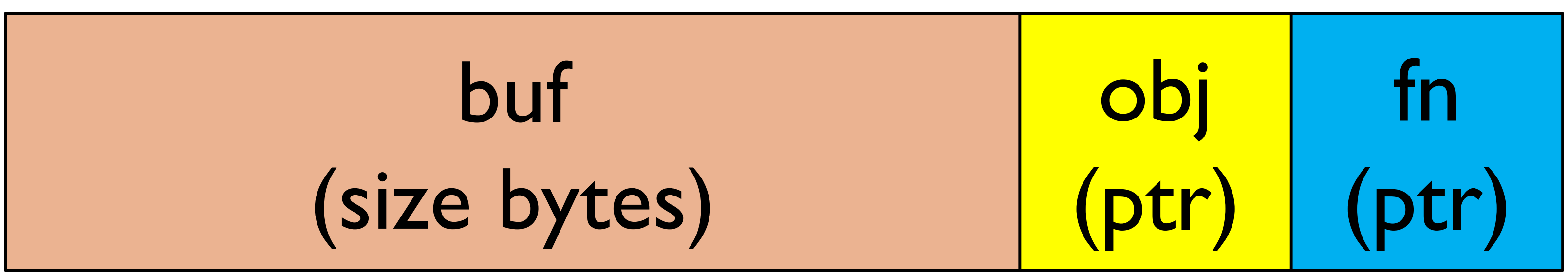
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Heap Overflows

- What can an overflow of "buf" cause? Change "obj"
  - What attacks are possible?

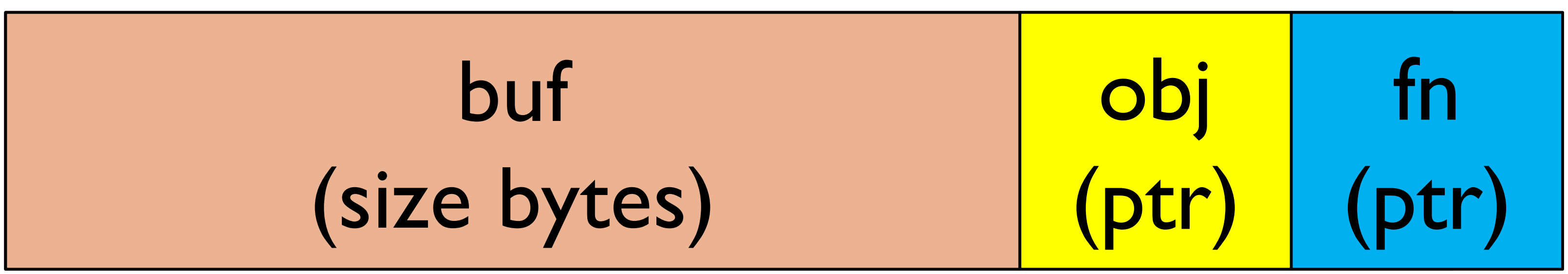
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Heap Overflows

- What can an overflow of “buf” cause? Change “obj”
  - Read/write arbitrary locations defined by adversary

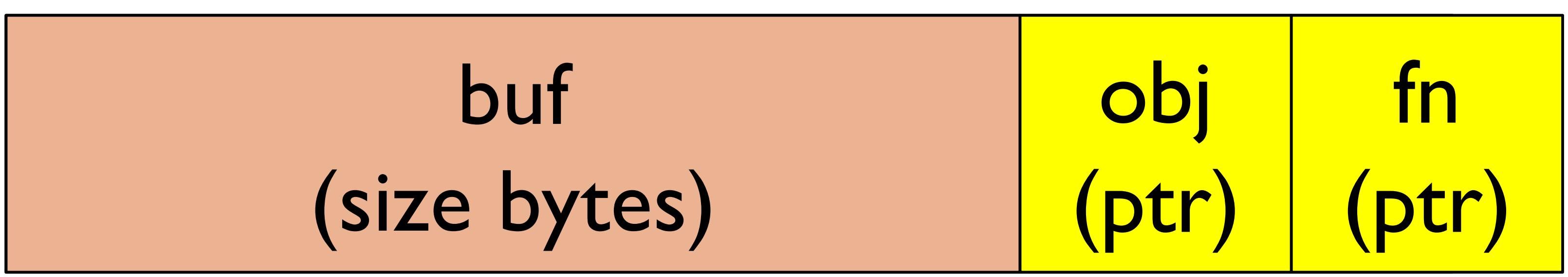
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Heap Overflows

- What can an overflow of "buf" cause? Change "fn"
  - What attacks are possible?

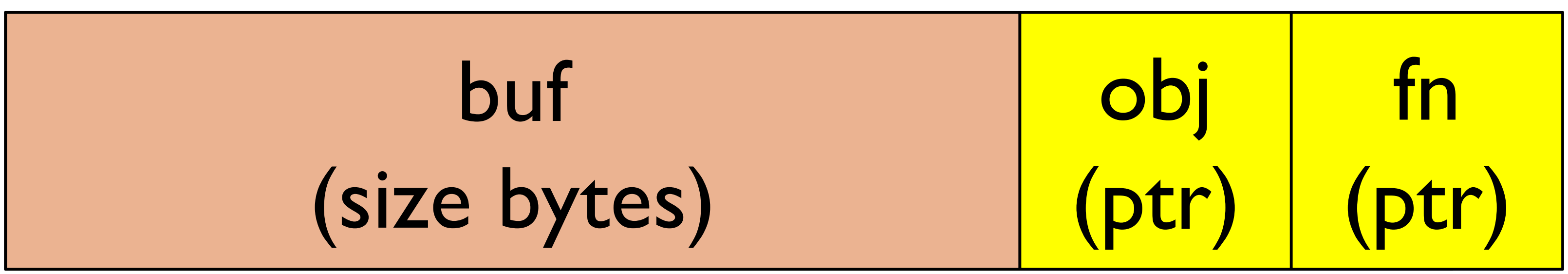
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Heap Overflows

- What can an overflow of "buf" cause? Change "fn"
  - Execute adversary-chosen code

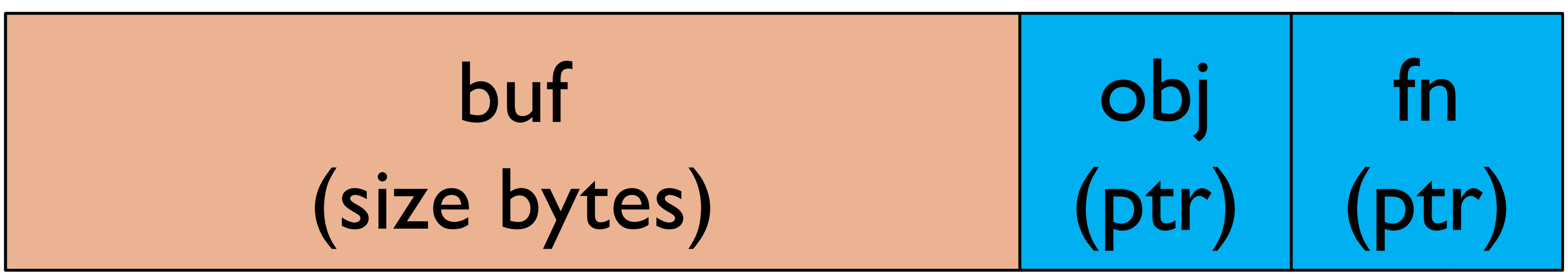
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Defenses for Heap Overflows

- None really – e.g., canaries are expensive

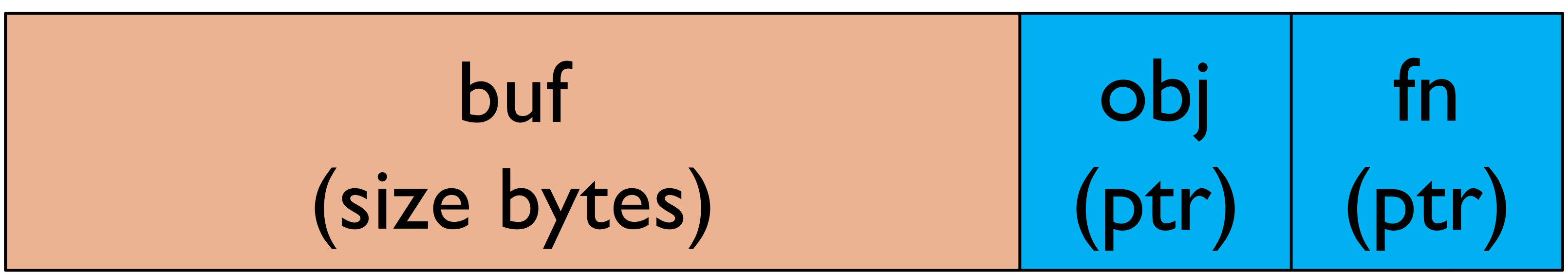
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Defenses for Heap Overflows

- None really – e.g., ASLR doesn't help – why not?

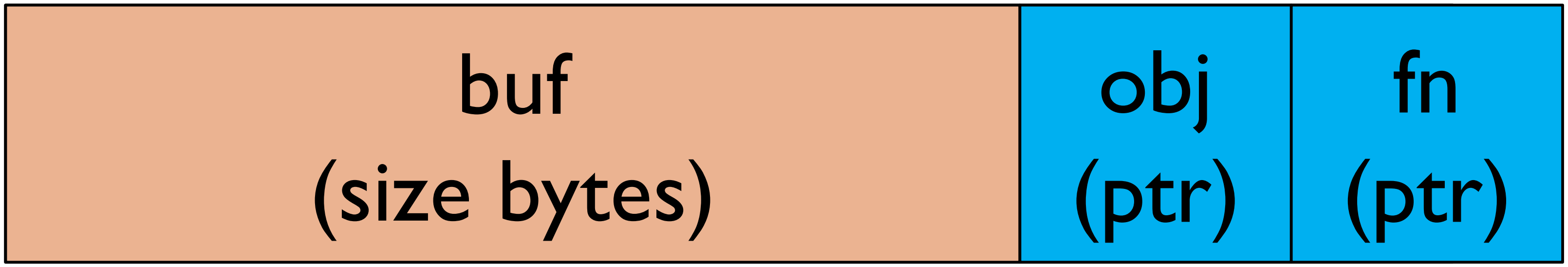
```
struct x  {
      char buf[size];
      data *obj;
      void (*fn)();
};
```



Defenses for Heap Overflows

- None really – e.g., NX does not help – like ROP

```
struct x  {
     char buf[size];
     data *obj;
     void (*fn)();
};
```

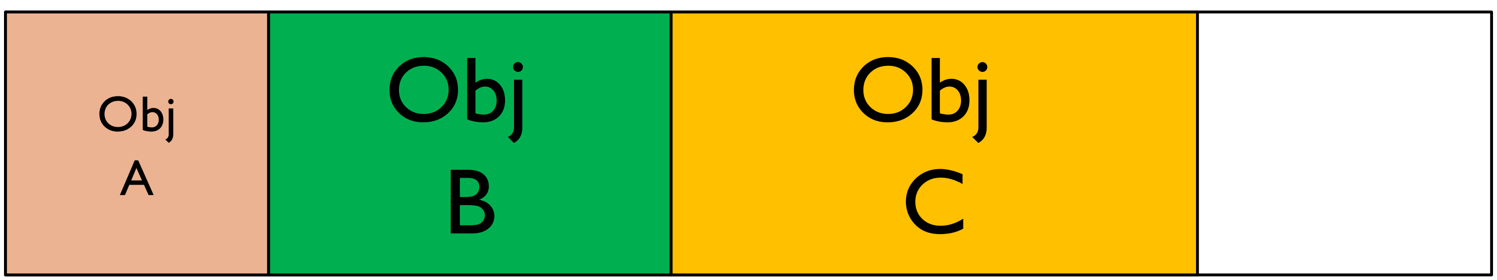


Attacks on Memory Reuse

- Attacks also exploit the inconsistencies caused in the reuse of memory on the heap
- Inconsistencies
  - Your program may reclaim memory
    - And reuse that memory region for another object
  - But, the pointers to the original object (i.e., memory location prior to reclamation) may remain
    - And be used after the reuse
- Example
  - Use-after-free

Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say “b”)
- Accessible: Adversary can control data written using the freed pointer
  - memcpy(b, adv-data, size);
- Exploit: Obtain a “write primitive”



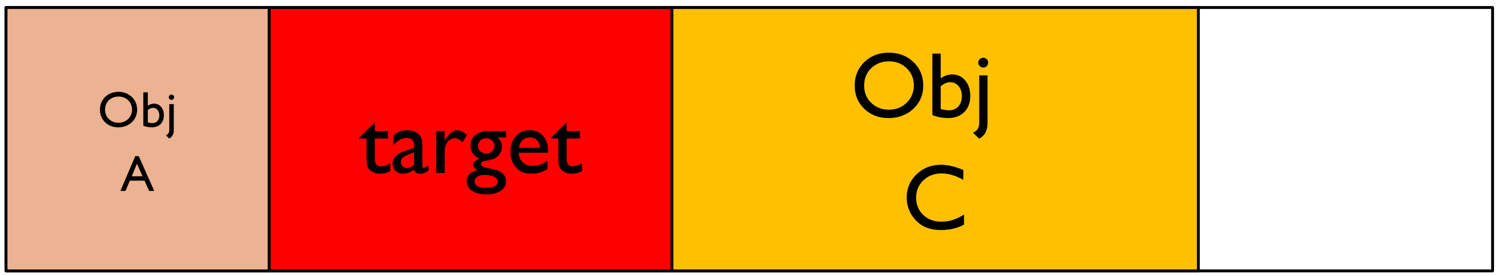
Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid
- Accessible: Adversary can control data written using the freed pointer
- Exploit: Obtain a “write primitive”

- Hold on: just using a reference to freed memory isn’t really a problem, is it?
  - What is missing from above?

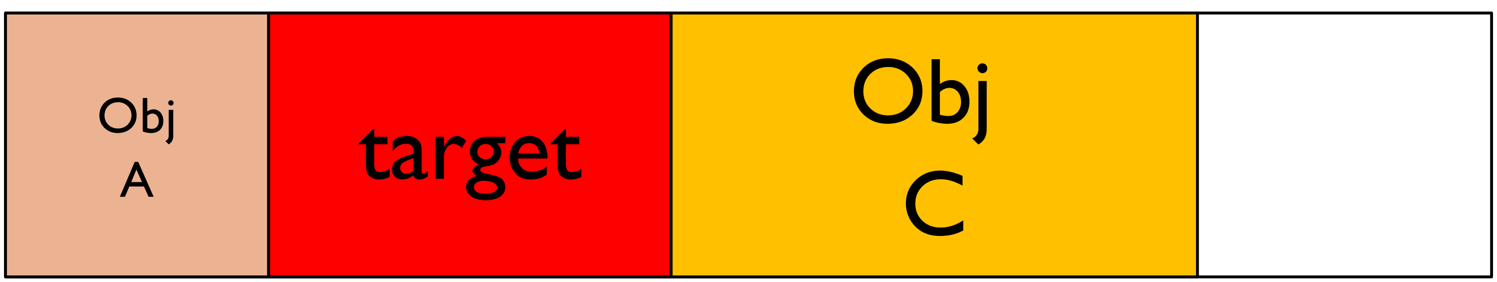
Use After Free

- Flaw: Program frees data on the heap, but then references that memory as if it were still valid
  - E.g., pointer to Obj B (say "b")
- Accessible: Adversary can control data written using the freed pointer
  - memcpy(b, adv-data, size);
- Exploit: Obtain a "write primitive" to a target object



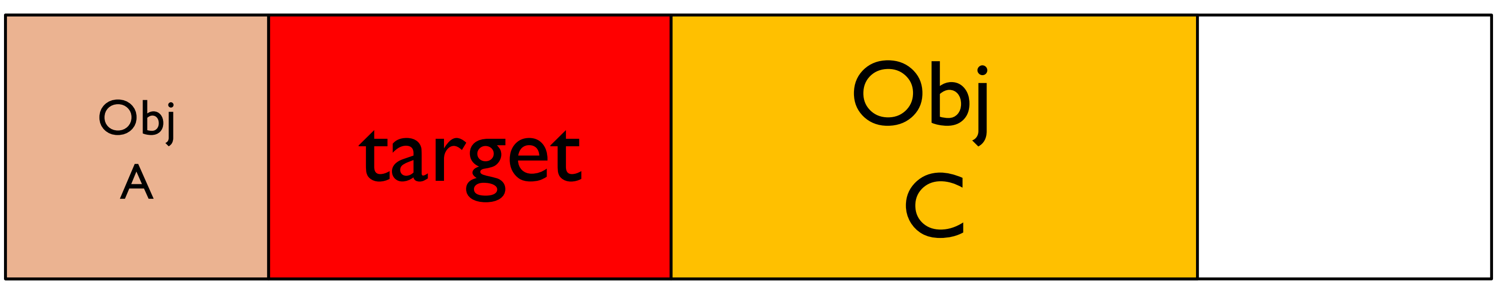
Use After Free

- Challenge: Get the program to allocate the adversary-chosen target object in the same location as the freed object
  - Need to cause the program to “malloc” a target
  - The location of allocation depends on the allocator
  - What can you do?



Use After Free

- Challenge: Get the program to allocate the adversary-chosen target data in the same location as the freed object
  - What can you do?
    - Heap spraying: cause the allocation of lots of objects in hope one lands where you (the adversary) wants
    - E.g., Get the program to run “malloc” for your object of choice many times until target is likely allocated at B



Conclusions

- Heap errors are now the most commonly exploited vulnerabilities
- Attacks on the heap may exploit the heap metadata and/or data (spatially or temporally)
- While these are similar in spirit to stack exploits, heap attacks can be more varied
  - Due to the more complex allocation/deallocation
- Major focus is to figure out how to prevent heap attacks in a manner that is reliable, but not too expensive

Questions