Chapter 7. Heap Vulnerability

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Topics

- **Brief background of dynamic memory allocation***
  - Linked list of free blocks
  - Heap metadata
- **Buffer overflow in heap**
  - Common patterns of heap buffer overflow
  - Exploiting heap buffer overflow
- **Use-after-free**
  - Principle and examples of use-after-free
  - Exploitation of use-after-free vulnerability

***** Take ***System Programming*** (CSE4100) for more details

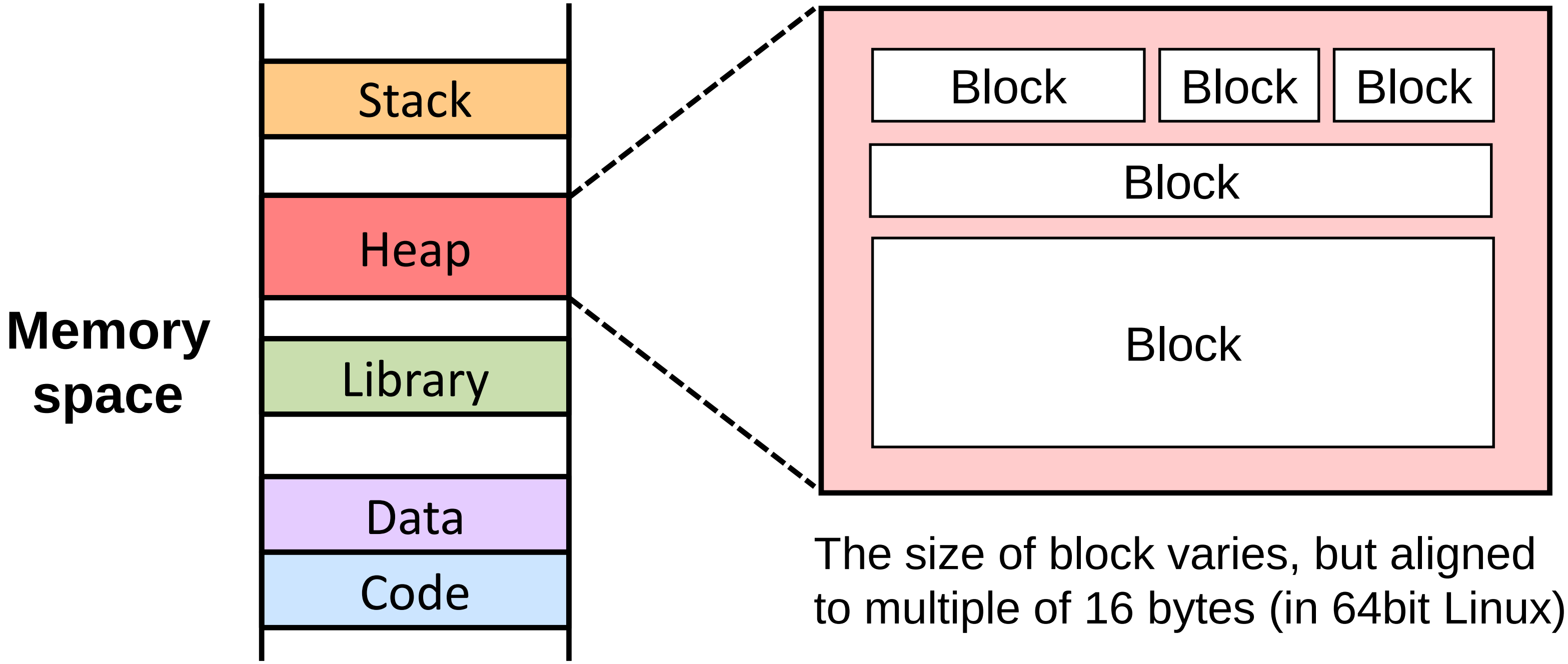
Dynamic Memory Allocation

- **Often, the size of data to process is decided at runtime**
  - Ex) Length of input string is provided by user
  - Ex) Must add a node to the linked list whenever a user requests
- **We can use dynamic memory allocation**
  - **`void *malloc(size_t size);`**
  - **`void free(void *ptr);`**

```
// Size of buffer is decided by user input
size_t input_size = read_size_t();
char *p = (char*) malloc(input_size);
if (p == NULL)
    return;
fgets(p, input_size, stdin);
```

Layout of Heap Memory

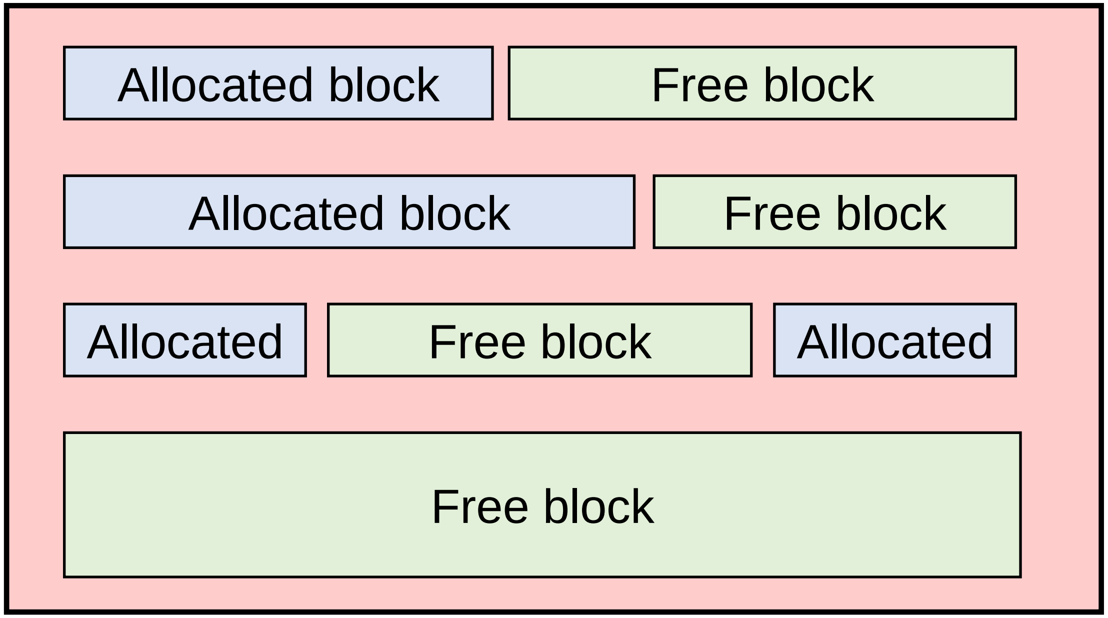
- **Memory allocator can increase the size of heap area (the red section) by invoking system calls like `sbrk()`**
- **Inside the heap memory area, small memory blocks (sometimes called chunks) are created and managed**



The size of block varies, but aligned to multiple of 16 bytes (in 64bit Linux)

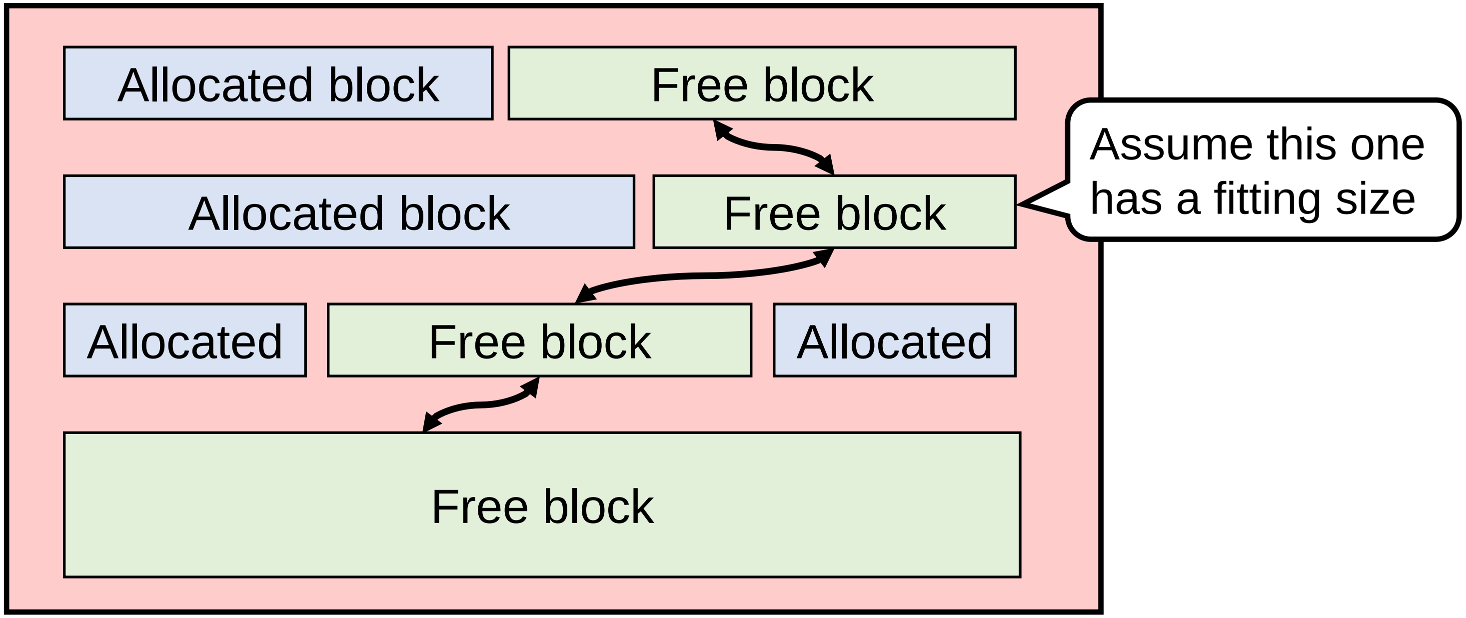
Heap Memory Block

- **When the program calls `malloc()`, the memory allocator will return one of the available blocks**
  - Block that is returned by **`malloc()`** is called ***allocated*** block
  - Other blocks are called ***free (not allocated)*** blocks



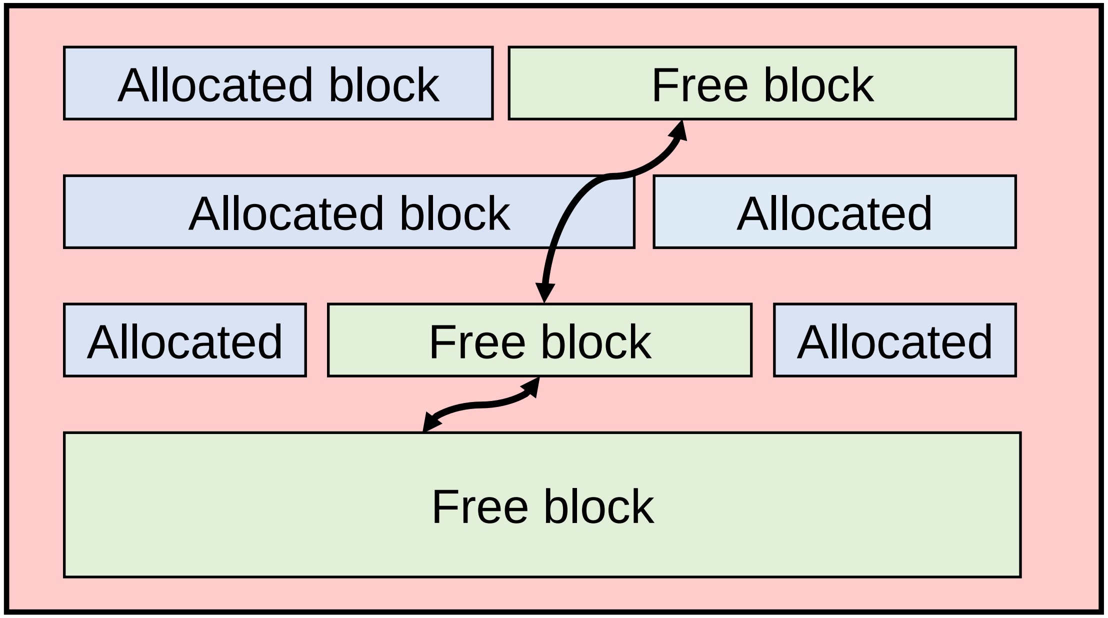
List of Free Blocks

- **The memory allocator manages linked list of free blocks**
  - When the program calls `p = malloc()`:
    - A block of fitting size is found from the list and returned
    - Or a free block with larger size can be split and returned



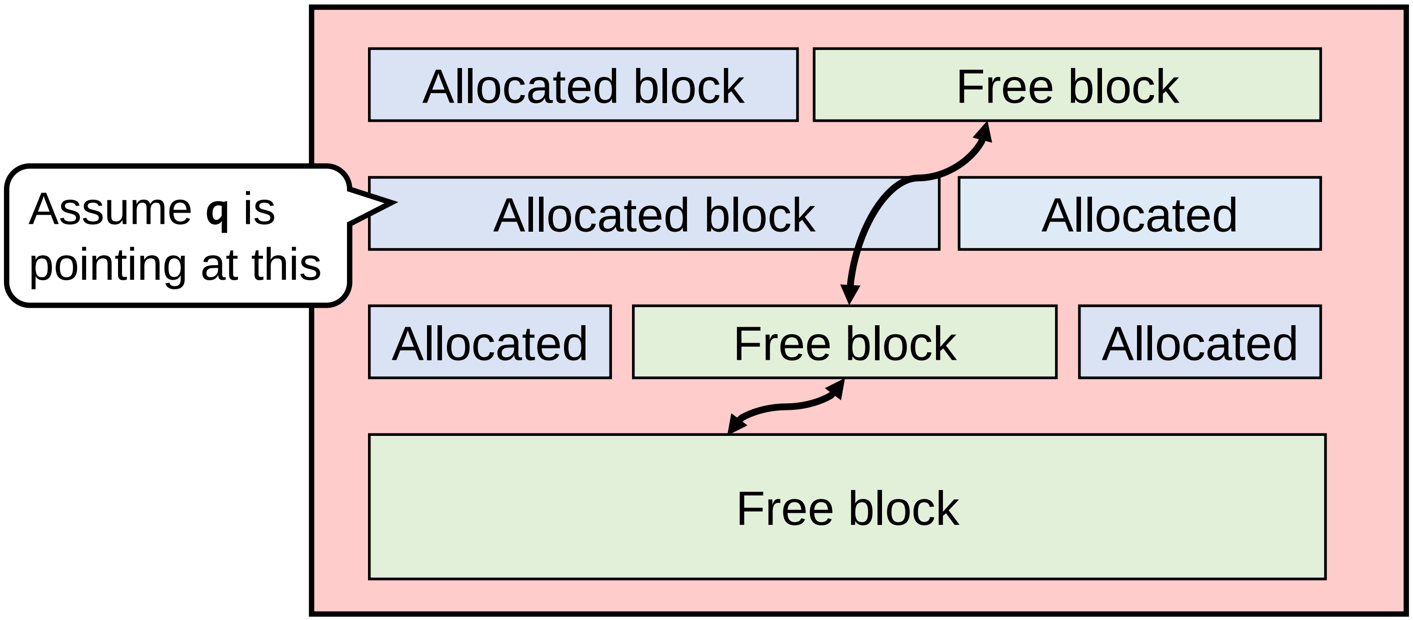
List of Free Blocks

- **The memory allocator manages linked list of free blocks**
  - When the program calls `p = malloc()`:
    - A block of fitting size is found from the list and returned
    - Or a free block with larger size can be split and returned



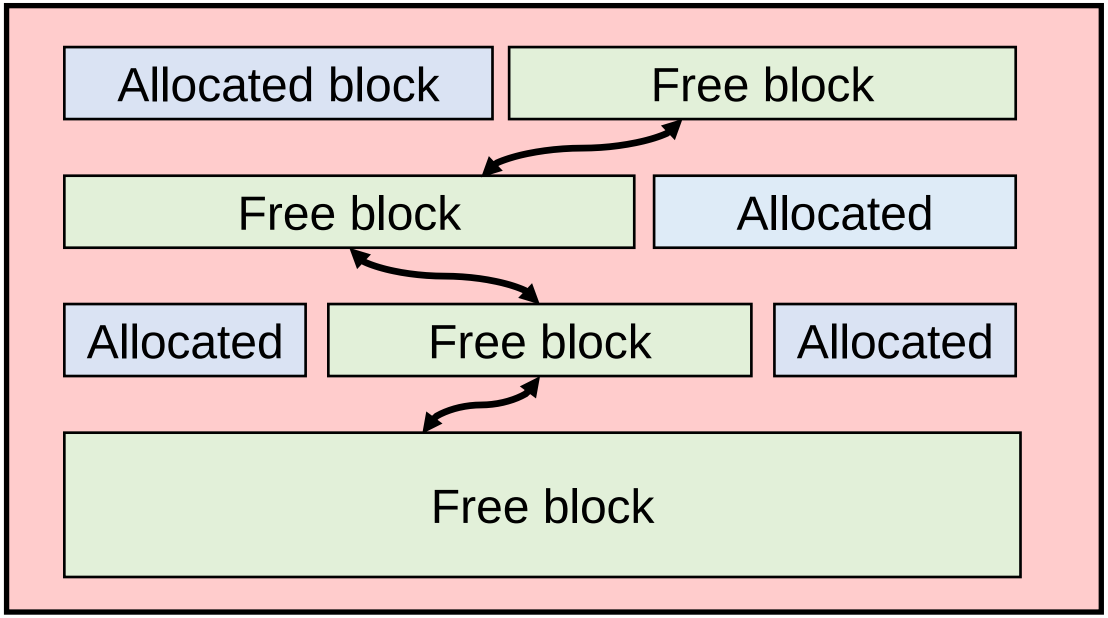
List of Free Blocks

- **The memory allocator manages linked list of free blocks**
  - When the program calls `free(q)`:
    - The block pointed by **q** is inserted to the list of free blocks
    - Now the block space is used to store a pointer to next node



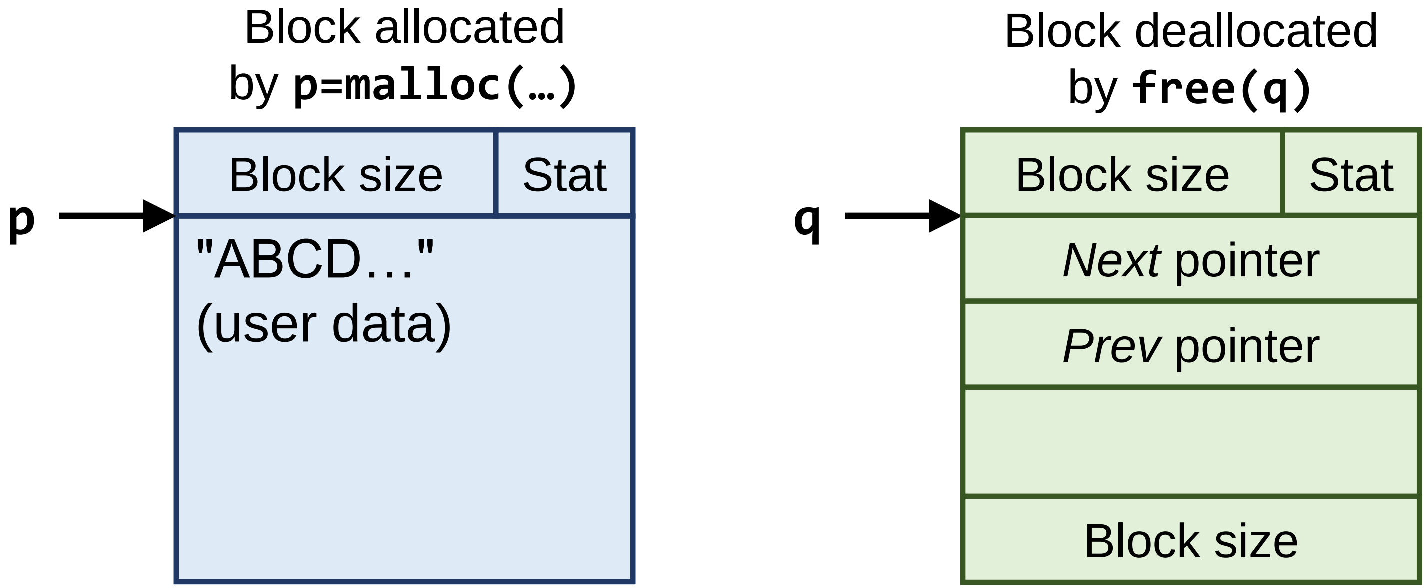
List of Free Blocks

- **The memory allocator manages linked list of free blocks**
  - When the program calls `free(q)`:
    - The block pointed by `q` is inserted to the list of free blocks
    - Now the block space is used to store a pointer to next node



Metadata in Heap Memory Block

- **To manage blocks in this way, the memory allocator must store various information other than user data**
  - Ex) Size of block, status of current/adjacent block, pointer to next/previous node of a doubly linked list …
  - Such additional information is often called ***metadata***



Real-world memory allocators are much more complex than this

But this will be enough for our course

Topics

- **Brief background of dynamic memory allocation**
  - Linked list of free blocks
  - Heap metadata
- **Buffer overflow in heap**
  - Common patterns of heap buffer overflow
  - Exploiting heap buffer overflow
- **Use-after-free**
  - Principle and examples of use-after-free
  - Exploitation of use-after-free vulnerability

Buffer Overflow in Heap

- **Heap BOF can occur for similar reasons to stack BOF**
  - Ex) Calling unsafe library functions like **`gets()`** or **`scanf("%s")`**
  - Ex) Allowing array to be accessed with arbitrary index
- **Moreover, it is easier to make mistakes in heap because the allocation size is usually affected by user input**
  - Consider the example below: what can go wrong here?

```
uint item_count = read_uint();
int *arr = (int*) malloc(item_count * 4); // int is 4-byte
if (arr == NULL)
    return;
uint idx = read_uint();
if (idx < item_count)
    arr[idx] = 1;
```

Buffer Overflow in Heap

- **Heap BOF can occur for similar reasons to stack BOF**
  - Ex) Calling unsafe library functions like **`gets()`** or **`scanf("%s")`**
  - Ex) Allowing array to be accessed with arbitrary index
- **Moreover, it is easier to make mistakes in heap because the allocation size is usually affected by user input**
  - Consider the example below: what can go wrong here?

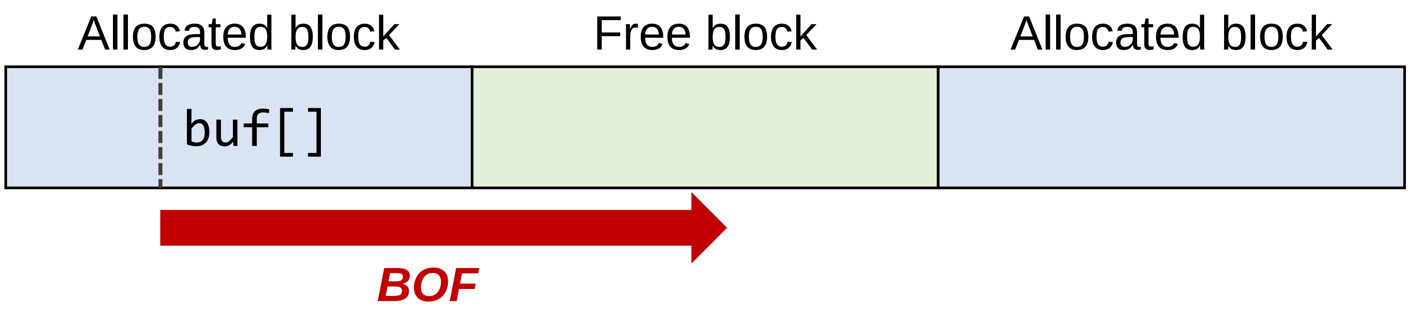
```
uint item_count = read_uint();
int *arr = (int*) malloc(item_count * 4); // int is 4-byte
if (arr == NULL)
    return;
uint idx = read_uint();
if (idx < item_count)
    arr[idx] = 1;
```

Input: 0x40000001

Size: 0x4 ***(integer overflow)***

Exploiting Heap BOF

- **How can an attacker exploit BOF that occurs in heap?**
  - Unlike stack, there is **no saved return address** in heap
- **First, corruption of a structure (or object in `C++`) may lead to security issues, such as control hijack**
- **Second, the corruption of metadata in heap blocks can cause problems as well**



Corruption of Structure/Object

- **Assume a structure (or object in C++) allocated in heap**
- **If there is a field (or property) whose type is function pointer, corrupting such field can lead to control hijack**
  - C++ object often contains a pointer to function pointer table (which is known as **virtual table**)

```
struct S {
  char buf[16];
  void (*handler)(char *s);
};

void f(void) {
  struct S *s = malloc(sizeof(struct S));
  gets(s->buf); // Buffer overflow (may corrupt "handler")
  s->handler("input msg");
}
```

Corruption of Metadata

- **Recall that memory allocator often maintains a (doubly) linked list of free blocks**
- **What if the `Next` or `Prev` pointer within a free block is corrupted by buffer overflow?**
  - When such block is removed from the linked list, dangerous memory operations can occur

Block size	Stat
Next (corrupted)	
Prev (corrupted)	
Block size	

```
remove_from_list(b) {
  Block *n = b->next;
  Block *p = b->prev;
  n->prev = p;
  p->next = n;
}
```

Modern Memory Allocators

- **Recent memory allocators are equipped with many protection mechanisms against metadata corruption**
  - The code below shows a patch introduced in 2004
- **As a result, nowadays it is rather hard to exploit a heap buffer overflow by corrupting block metadata**

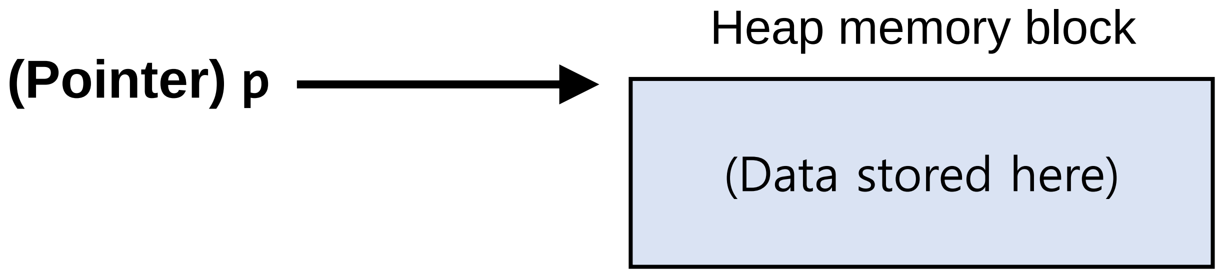
```
#define unlink(P, BK, FD) {
  FD = P->fd ;
  BK = P->bk ;
  if (FD->bk != P || BK->fd != P) error(); // Validation
  else {
    FD->bk = BK;
    BK->fd = FD;
  }
}
```

Topics

- **Brief background of dynamic memory allocation**
  - Linked list of free blocks
  - Heap metadata
- **Buffer overflow in heap**
  - Common patterns of heap buffer overflow
  - Exploiting heap buffer overflow
- **Use-after-free**
  - Principle and examples of use-after-free
  - Exploitation of use-after-free vulnerability

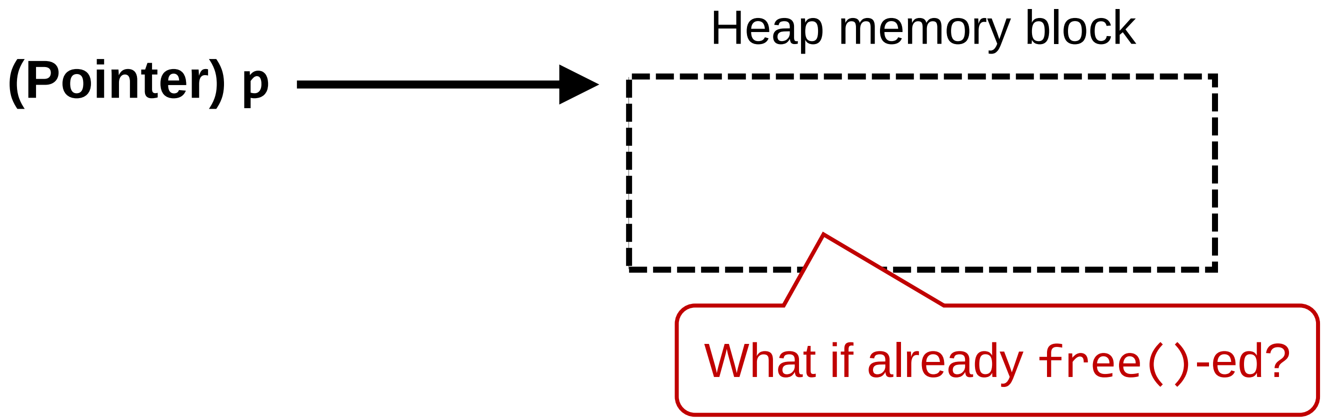
Use-After-Free (UAF)

- **Another popular source of heap vulnerability**
- **Accessing a heap memory location that has already been deallocated with `free()`**
  - Has been a well-known concept in computer science
  - Sometimes called *use of a dangling pointer*
  - Gained lots of interest from security researchers around 2010



Use-After-Free (UAF)

- **Another popular source of heap vulnerability**
- **Accessing a heap memory location that has already been deallocated with `free()`**
  - Has been a well-known concept in computer science
  - Sometimes called *use of a dangling pointer*
  - Gained lots of interest from security researchers around 2010



Use-After-Free Example

- **In the code below, the heap memory block pointed by `struct S *s` is deallocated**
- **But this memory location is accessed after `free()`**
  - After the deallocation, cannot guarantee what is stored there

```
struct S {
  int x;
  ...
};

struct S *s = (struct S*) malloc(sizeof(struct S));
...
free(s);
...
printf("x = %d\n", s->x); // Accessing deallocated memory
```

More Realistic Example (1)

- **Assume that there are multiple pointers that are pointing at the allocated memory block**
- **It can be hard to remember all the pointer variables that are related to the deallocated memory block**

```
struct S *s1 = (struct S*) malloc(sizeof(struct S));
struct S *s2 = s1;
px = &(s1->x);
...
free(s1); // OK, let’s not use 's1' anymore
...
// But forgot about 's2' or 'px'
s2->x = 100;
printf("x = %d\n", *px);
```

More Realistic Example (2)

- **Assume a global pointer variable pointing at heap block**
- **If the pointed heap block is deallocated, must invalidate the pointer (for example, by setting into `NULL`)**
  - Forgetting to do so may lead to ***use-after-free***

```
struct S* items[32]; // Each element is a pointer to struct

void delete(int idx) {
  if (idx < 0 || idx >= 32)
    return;
  if (items[idx] != NULL) {
    free(items[idx]);
  }
  // Didn't we forget something here?
}
```

Security Impact of Use-After-Free

- **Now let's think about the severity of this bug**
- **Consider the example code below**
  - `printf()` after `free()` will print out old data in the free block
  - Following `s->x=100;` will update unused space in the free block
  - In some case, these may disclose or corrupt the heap metadata
- **But is that all?**
  - Or is it possible to exploit UAF and **hijack the control-flow?**

```
struct S *s = (struct S*) malloc(sizeof(struct S));
...
free(s);
printf("x = %d\n", s->x); // Using 's'
s->x = 100;
```

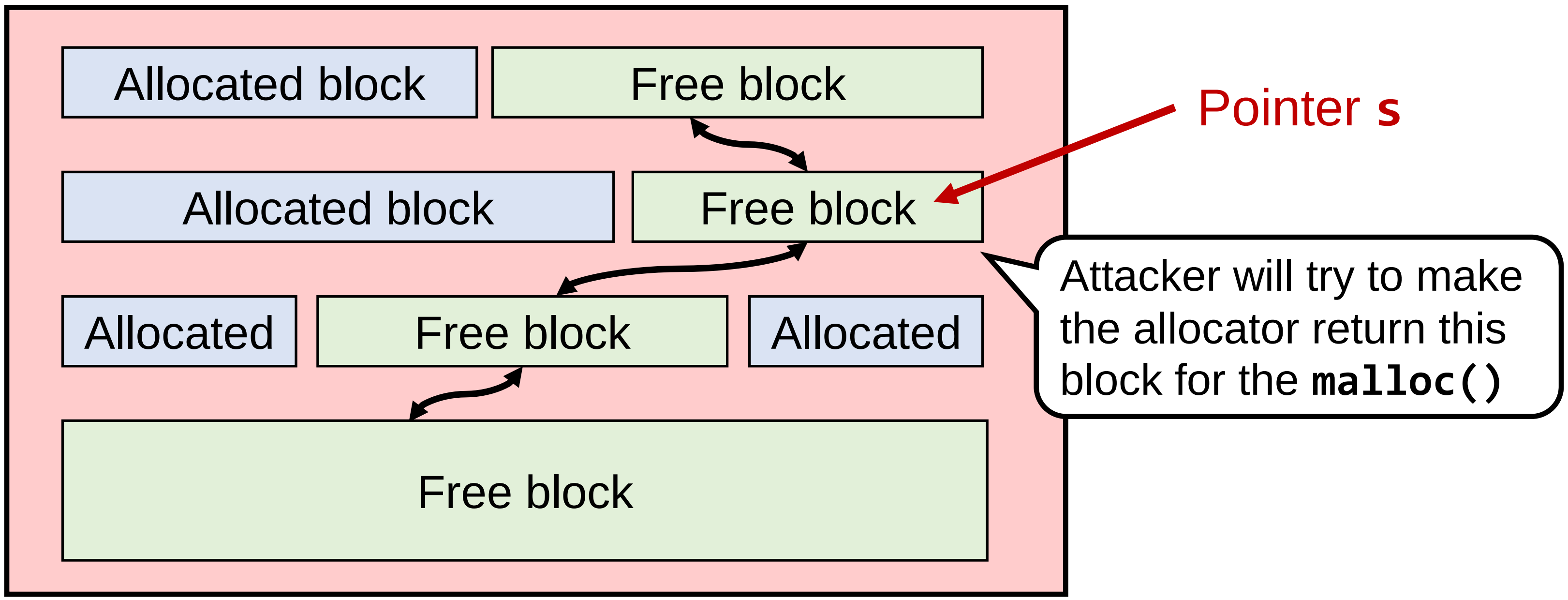
Free blocks can be re-allocated

- **In practice, often there is an interval between the `free()` and use-after-free**
- **What if there are `malloc()` calls within that interval?**
- **Memory block pointed by 's' may be allocated again**

```
struct S *s = (struct S*) malloc(sizeof(struct S));
...
free(s);
...
malloc(...);
malloc(...);
...
printf("x = %d\n", s->x); // Using 's'
s->x = 100;
```

Controlling the Heap Memory

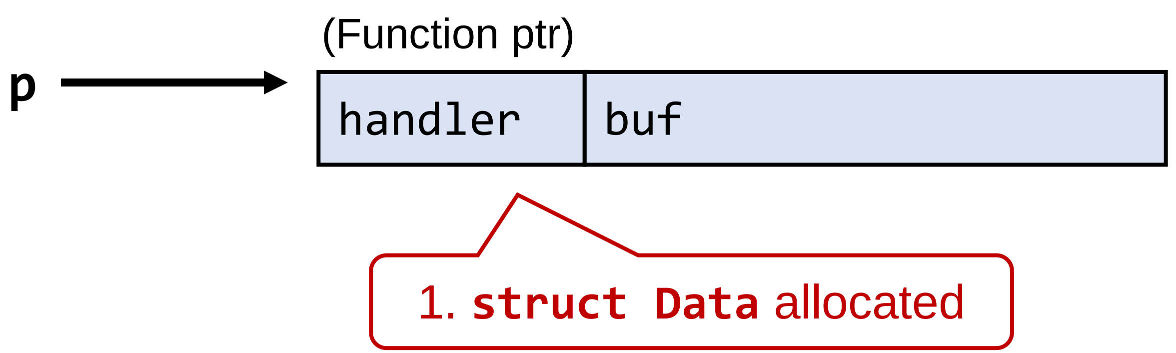
- **Recall that free blocks are managed as a linked list**
  - The allocator will find a block with ***fitting size*** from this list
- **By carefully deciding the size of `malloc()`, the attacker can enforce the wanted block to be returned**



Exploiting Use-After-Free

- Assume that struct **`Data`** has a character array field (**`buf`**) and a function pointer field (**`handler`**)

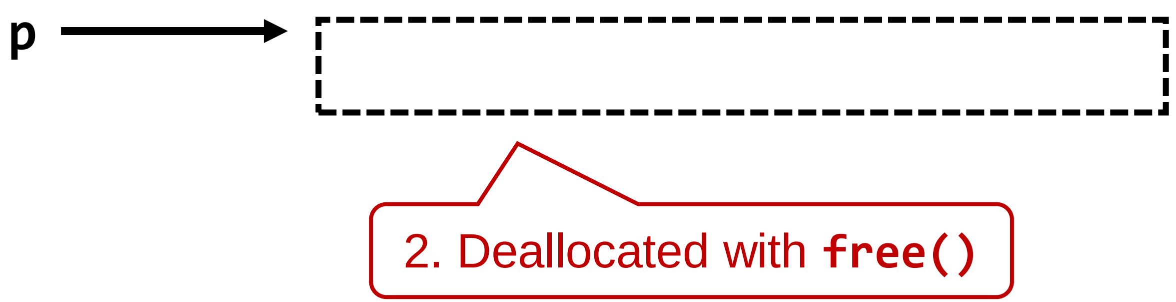
```
struct Data *p = (struct Data*) malloc(sizeof(struct Data));
...
free(p);
...
len = read_size_t();
char *str = (char *) malloc(len); // Will read in string
...
p->handler(p->buf); // Use-after-free
```



Exploiting Use-After-Free

- Assume that struct **`Data`** has a character array field (**`buf`**) and a function pointer field (**`handler`**)

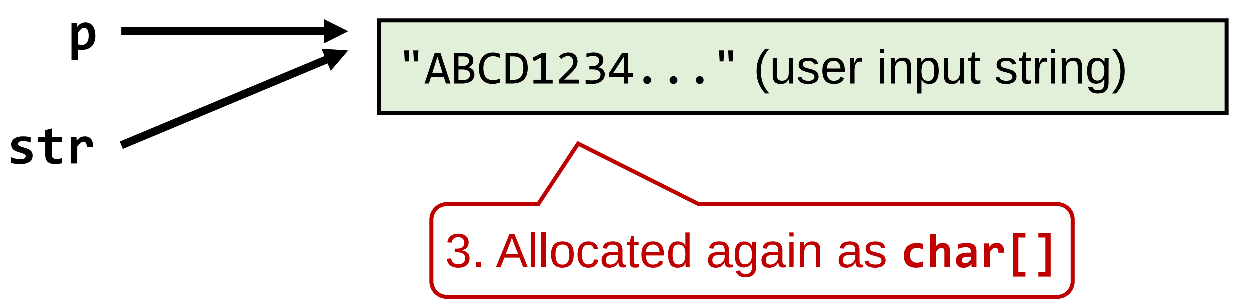
```
struct Data *p = (struct Data*) malloc(sizeof(struct Data));
...
free(p);
...
len = read_size_t();
char *str = (char *) malloc(len); // Will read in string
...
p->handler(p->buf); // Use-after-free
```



Exploiting Use-After-Free

- Assume that struct **`Data`** has a character array field (**`buf`**) and a function pointer field (**`handler`**)

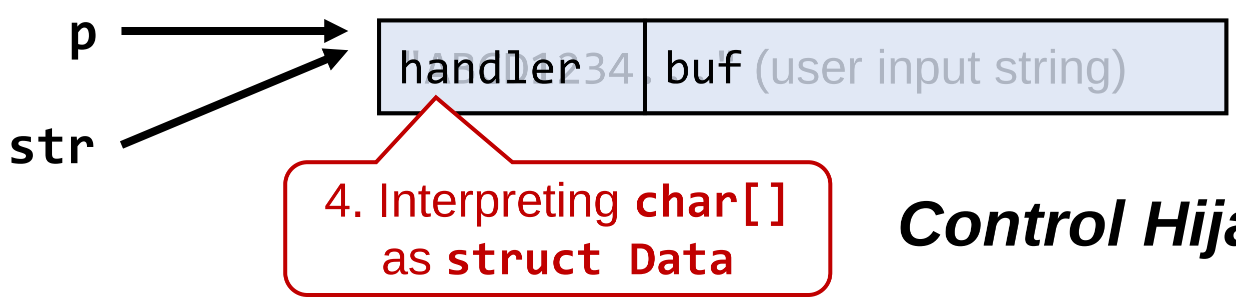
```
struct Data *p = (struct Data*) malloc(sizeof(struct Data));
...
free(p);
...
len = read_size_t();
char *str = (char *) malloc(len); // Will read in string
...
p->handler(p->buf); // Use-after-free
```



Exploiting Use-After-Free

- Assume that struct **Data** has a character array field (**buf**) and a function pointer field (**handler**)

```
struct Data *p = (struct Data*) malloc(sizeof(struct Data));
...
free(p);
...
len = read_size_t();
char *str = (char *) malloc(len); // Will read in string
...
p->handler(p->buf); // Use-after-free
```



Control Hijack!

Use-After-Free in Stack?

- **Can use-after-free occur in stack as well?**
- **There is no `malloc()` in stack, but the allocation and deallocation of stack frame is conceptually similar**
- **If a function returns a pointer to its local variable, use of that pointer will access invalid memory location**
  - But this type of bug occurs rarely (easy to avoid)

```
char *f(void) {
  char buf[32];
  char *p = buf;
  return p; // But buf does not exist after f() returns
}
```

Side-note: Uninitialized Data Use

- **UAF can be thought as *accessing memory too late***
- **In contrast, program may also access *memory too early***
  - If a variable is used without being initialized, garbage values stored there will be used (use-before-initialize)
  - But the term ***uninitialized data/variable use*** is more popular
  - Can occur in any memory region (*stack*, *heap*, or *data*)
- **Memory disclosure or even code execution can be possible in some cases**

```
void f(void) {
  struct S s;
  read(0, s.buf, 32); // Using uninitialized pointer value
  ...
}
```