CS260 – Advanced Systems Security

Memory Errors

April 2, 2025

Memory Errors

- Bugs in C/C++ programs can cause memory errors
  - C/C++ does not ensure memory safety
- Memory errors and the ability to exploit them have been known for over 50 years
  - And exploited in practice since the Morris worm (1988)
- Microsoft and Google report that over 70% of vulnerabilities are still from memory errors

Memory Errors

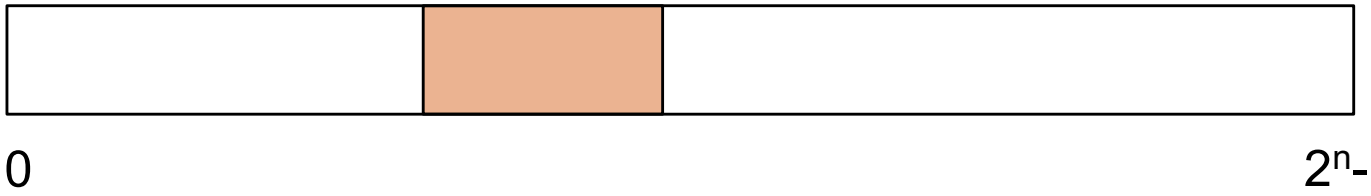
- What are the causes of memory errors?

Cause of Memory Errors

- In C/C++, objects and their memory references are separate things
  - Memory references: Pointers
  - Objects: Dynamically allocated on stack and heap
- Memory references and object allocations do not always correspond to each other
  - C/C++ (try to) use pointers to reference the memory locations of memory objects
  - The values (memory locations) of pointers may be assigned independently from object allocations

C/C++ Memory Model

- C allows programmers to access memory flexibly
  - Like a giant array of virtual memory



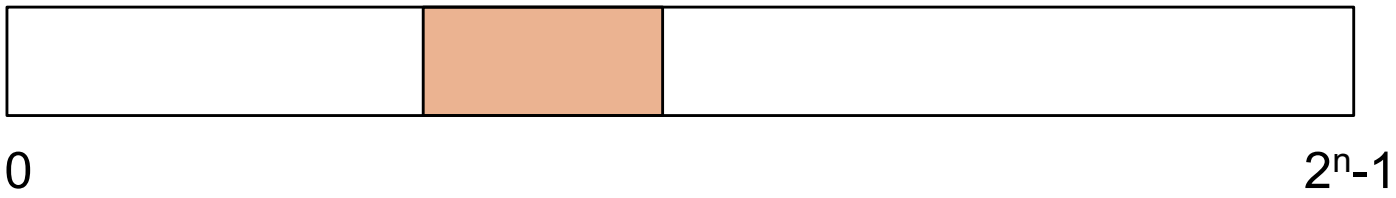
  - An object (in brown) can be allocated anywhere in the array
    - `char *x = (char *)malloc(size);`
  - Your program gets a reference (pointer) to the location of your object - the “array” - that is in virtual memory
    - It is up to the programmer to set and use the pointer correctly to access the object
    - I.e., the programmer must keep them ”in sync”

Memory and Type Safety

- Bugs in C/C++ programs can cause memory errors
  - C/C++ does not ensure memory safety
    - A pointer (reference) assigned to an object is not restricted to that object's memory region or lifetime
  - C/C++ does not ensure type safety
    - A pointer (reference) assigned to an object is not restricted to that object's data type
- We will look at the causes of memory errors
  - And a little bit about how to avoid them

C/C++ and Memory Safety

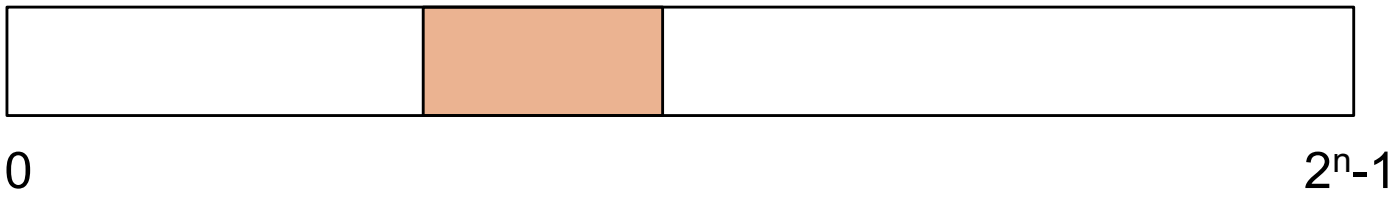
- An object (in brown) can be allocated anywhere in the array
  - `char *x = (char *)malloc(size);`



- Pointer arithmetic
  - `x = x+n;`
- What happens?

C/C++ and Memory Safety

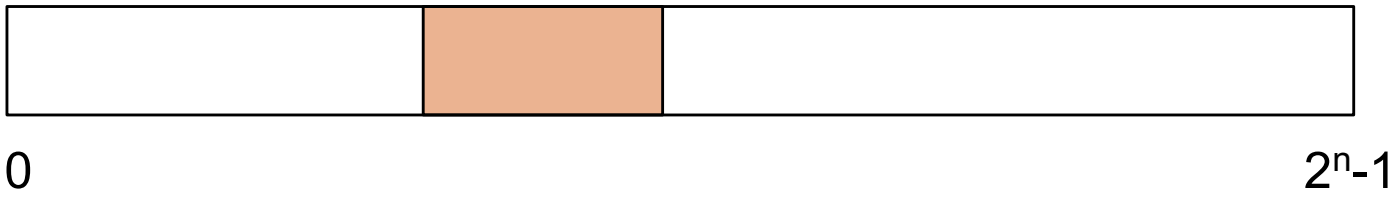
- An object (in brown) can be allocated anywhere in the array
  - `char *x = (char *)malloc(size);`



- Pointer arithmetic
  - `x = x+n;`
- What happens?
  - Change the value of the pointer x by n, regardless of how much x is
  - Positive or negative

C/C++ and Memory Safety

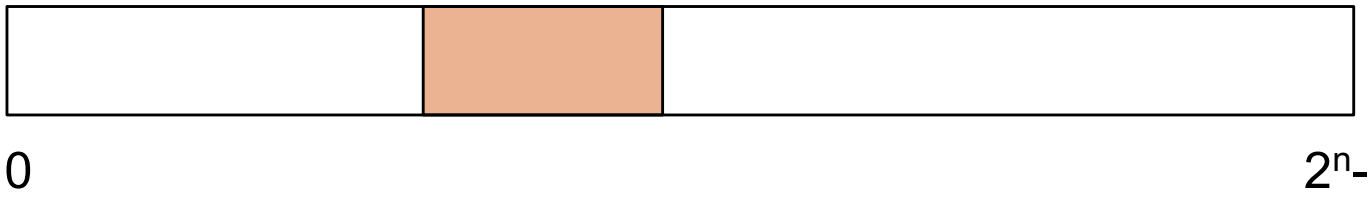
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Pointer arithmetic
  - `x = x+n;`
- What happens when the following is run after?
  - `strcpy(x, "string");`

C/C++ and Memory Safety

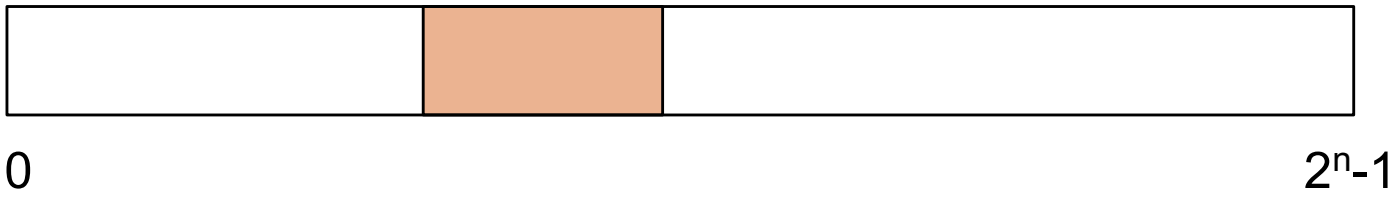
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Deallocate memory associated with the pointer `x`
  - `free(x);`
- What happens when the follow is run after the “free”?
  - `strcpy(x, “string”);`
- ”string” is written at location `x`, even if x is outside of the memory region

C/C++ and Memory Safety

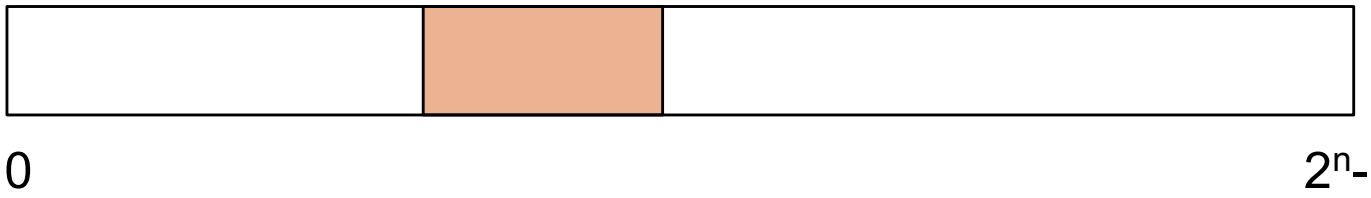
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Deallocate memory associated with the pointer `x`
  - `free(x);`
- What does the “free” command do?

C/C++ and Memory Safety

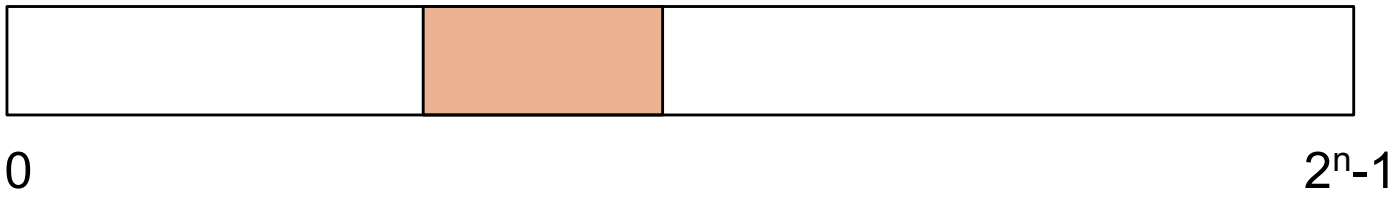
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Deallocate memory associated with the pointer `x`
  - `free(x);`
- What does the “free” command do?
  - Allow the memory region at `x` to be reused by another allocation

C/C++ and Memory Safety

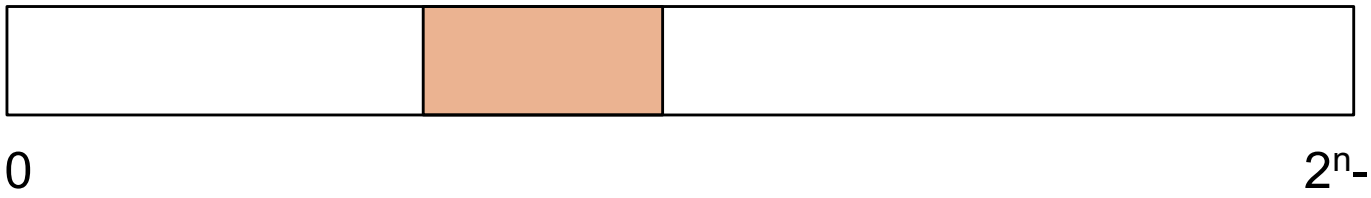
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Deallocate memory associated with the pointer `x`
  - `free(x);`
- What happens when the following is run after the “free”?
  - `strcpy(x, “string”);`

C/C++ and Memory Safety

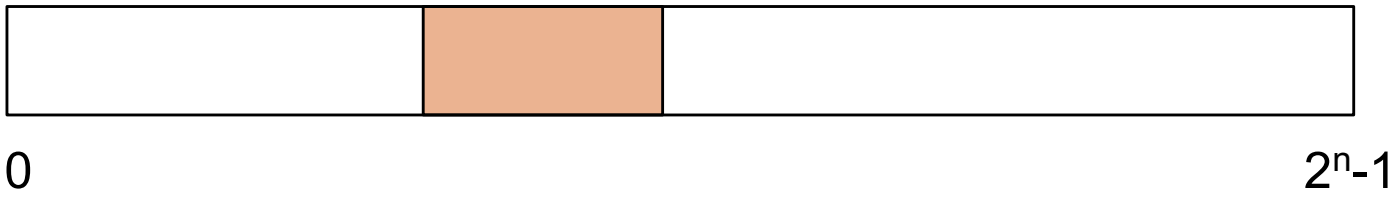
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- Deallocate memory associated with the pointer `x`
  - `free(x);`
- What happens when the follow is run after the “free”?
  - `strcpy(x, “string”);`
- ”string” is written at location `x`, even if something else has been allocated there

C/C++ and Type Safety

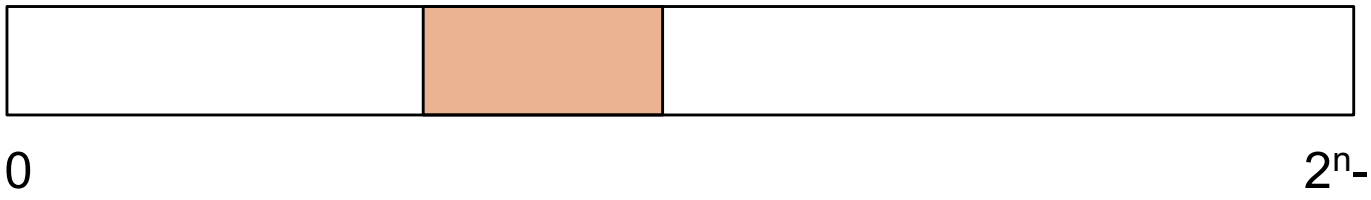
- An object (in brown) can be assigned a type
  - `char *x = (char *)malloc(size);`



- More specifically, the pointer is assigned a type
  - In this case, an array of 1-byte objects
- Used to interpret the values in the memory region
  - E.g., as a string

C/C++ and Type Safety

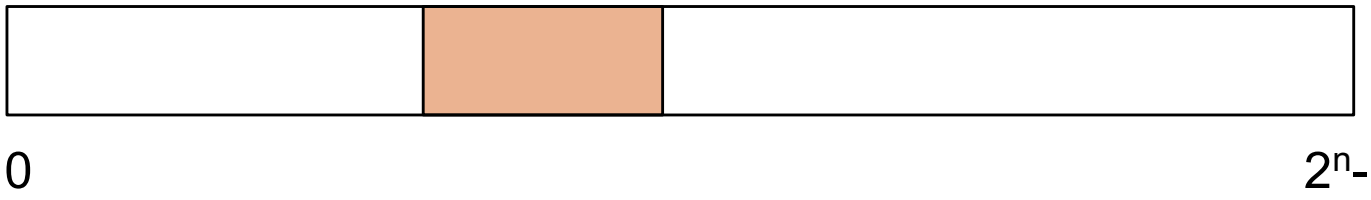
- An object (in brown) can be assigned a type
  - `char *x = (char *)malloc(size);`



- But, we can assign another pointer to reference the same memory using a different type (type cast)
  - `int *y = (int *)x;`
- Say an integer is 4 bytes, so the value is the first 4 characters assigned to the “string”
  - Nothing limits you in C
  - Other languages do prevent this kind of type cast

C/C++ and Type Safety

- An object (in brown) can be assigned a type
  - `char *x = (char *)malloc(size);`



- But, we can assign another pointer to reference the same memory using a different type (type cast)
  - `int *y = (int *)x;`
- Say an integer is 4 bytes, so the value is the first 4 characters assigned to the "string"
  - So, you cannot trust that a memory region's type (i.e., of the values assigned there) corresponds to the type of the pointer used to access the region – not type safe

Memory Error Vulnerability

- This code has a flaw

```c
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

Memory Error Vulnerability

- Suppose an adversary can provide “source”
  - May be larger than the memory space of “buffer”

```
#include <stdio.h>

int function( char *source )
{
  char buffer[10];

  sscanf( source, "%s", buffer );
  printf( "buffer address: %p\n\n", buffer );
  return 0;
}

int main( int argc, char *argv[] )
{
  function( argv[1] );
}
```

What Is Happening?

- Fill buffer to length of allocated buffer (10)
  - Scanf – Has no termination

What is happening?

- Fill buffer to length of allocated buffer (10)
  - Scanf – input a string (source) of length 5

  - Null termination of string (optional, in black)

What is happening?

- But, the string source may be >=10 bytes
  - 10 bytes – no room for the terminator byte

  - Write beyond the end of the allocated memory for buffer

  - Nothing stops that
    - What is beyond the end of one allocated region?

What is happening?

- But, the string source may be >=10 bytes
  - 10 bytes – no room for the terminator byte

  - Write beyond the end of the allocated memory for buffer

  - Nothing stops that
    - What is beyond the end of one allocated region?
      - Other objects that should not be accessed
      - Called a spatial memory error

More Complex Vulnerability

- Another flaw

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

More Complex Vulnerability

- Another flaw

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct test {
  char buffer[10];
  int (*fnptr)( char *, int );
};

int function( char *source )
{
  int res = 0, flags = 0;
  struct test *a = (struct test*)malloc(sizeof(struct test));
  printf( "buffer address: %p\n\n", a->buffer );
  a->fnptr = open;
  strcpy( a->buffer, source );
  res = a->fnptr(a->buffer, flags);
  printf( "fd: %d\n\n", res );
  return 0;
}

int main( int argc, char *argv[] )
{
  int fd = open("stack.c", O_CREAT);

  function( argv[1] );

  exit(0);
}
```

Strcpy

- Essentially, the same problem as for scanf
  - 10 bytes – no room for the terminator byte

  - Write beyond the end of the allocated memory for buffer

  - Nothing stops that
    - What is beyond the end of one allocated region?
- NOTE: Strncpy does not fully fix this problem

Exploiting Spatial Errors

- What can we exploit
  - When we write beyond the end of the allocated memory for buffer

- What is beyond? Other data
  - In C, data may include pointers
  - In C, pointers may be function pointers
  - The C runtime stores runtime metadata on the stack
    - Return addresses (function pointers)
- Changing of any of these may enable hijacking

String Copy Issues

- Issues with C/C++ arrays of bytes
  - May be longer than memory region (bounds)
  - May not be terminated by a null byte (bounds)
  - May be terminated before expected (truncate)
- Each of these issues may lead to problems
  - If undetected

Type Errors

- Errors that permit access to memory according to a multiple, incompatible formats
  - These are called type errors
  - Access using a different “type” than used to format the memory
- Most of these errors are permitted by simple programming flaws
  - Of the sort that you are not taught to avoid
  - Let’s see how such errors can be avoided
- Some of the changes are rather simple

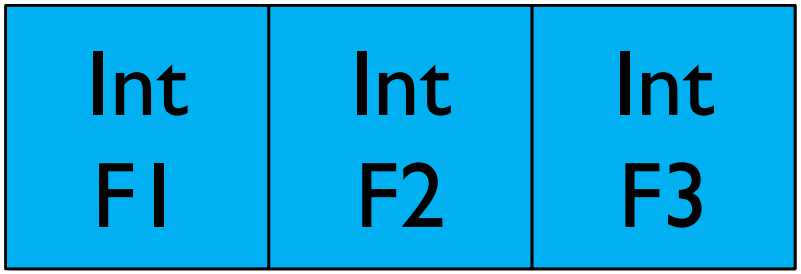
Other Error Prone Type Casts

- Downcasts – Cast to a larger type; allows overflow
  - t1 *p, t2 *q; // declare pointers
  - p = (t1 *) malloc(sizeof (t1)); // allocate t1 object, define p
  - p→field = value; // suppose this is an int field
  - q = (t2 *)p; // downcast, t2 is a larger type
  - q→extra= value2; // overflow memory of object
- E.g., t2 is a child type of t1
  - So, the size of type t2 is greater than the size of type t1
  - "extra" field is added to the type t1 to create type t2

Exploiting Type Errors

- "p" is assigned to an object of type t1

"P"

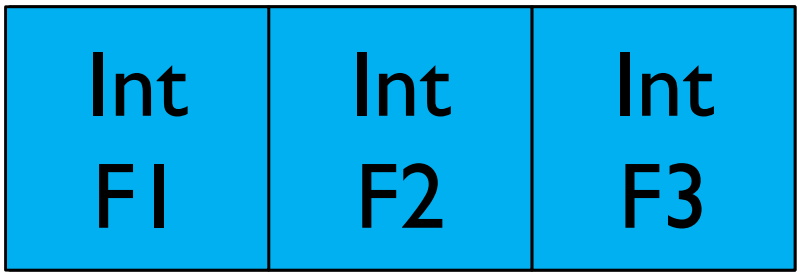


- Only memory large enough for t1 is allocated

Exploiting Type Errors

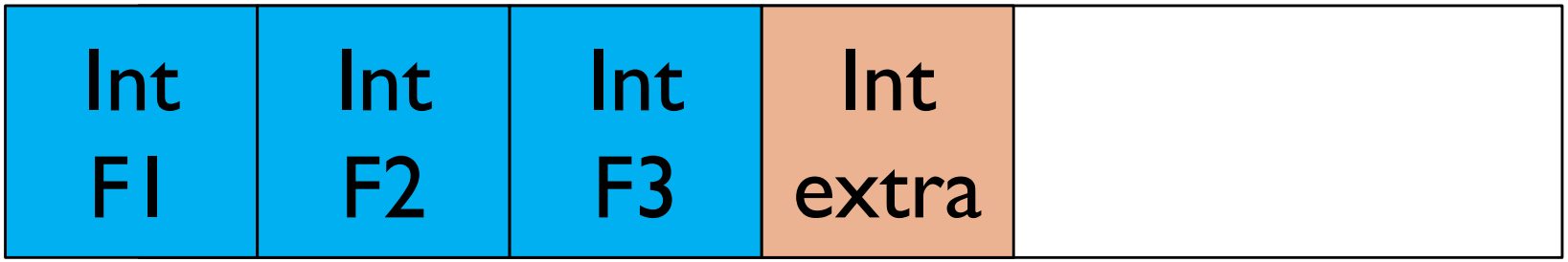
- "p" is assigned to an object of type t1

"p"



- But, if we assign a pointer of type t2 to the object

"q"

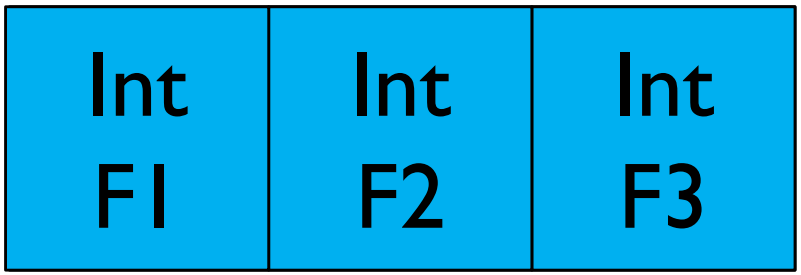


- This is what can be referenced by "q"
  - "q" of type t2 thinks it is referencing a larger region

Exploiting Type Errors

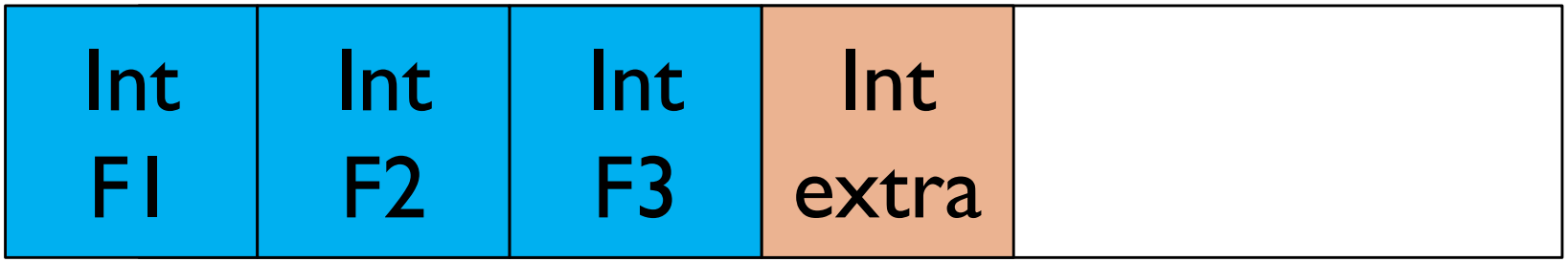
- "p" is assigned to an object of type t1

"p"



- But, if we assign a pointer of type t2 to the object

"q"



- What will happen when the program accesses "q→extra"?

What Can Go Wrong?

- Downcasts – Cast to a larger type; causes overflow
  - t1 *p, t2 *q; // declare pointers
  - p = (t1 *) malloc(sizeof (t1)); // allocate t1 object, define p
  - p→field = value; // suppose this is an int field
  - q = (t2 *)p; // down cast, t2 is a larger type
  - q→extra = value2; // overflow memory of object
- By downcasting to the larger type t2 with the "extra" field, gives the adversary the ability to read/write beyond the memory region allocated
  - Memory region is "sizeof(t1)" in size

Type Confusion

- Many effective attacks exploit data of another type

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

Type Confusion

- Adversary can abuse ambiguity to control writes

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

Type Confusion

- Adversary can abuse ambiguity to control writes

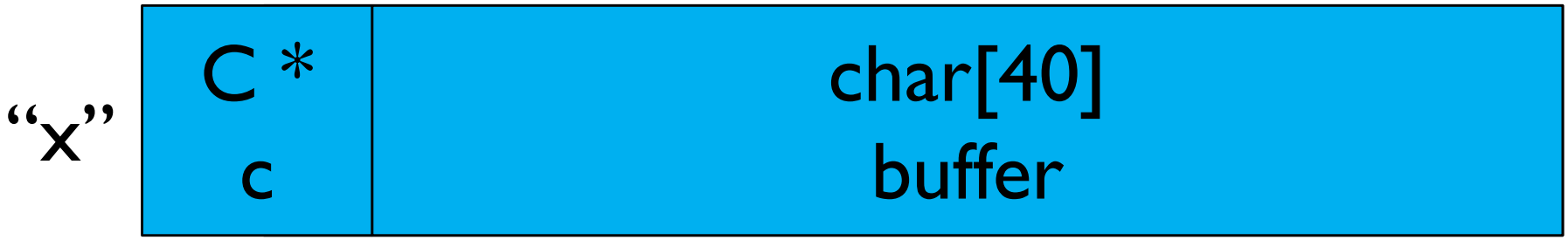
```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

- Arbitrary Write Primitive!
  - Adversary controls the value to write and the location of the write
  - Allow adversary to write an arbitrary value to an arbitrary location

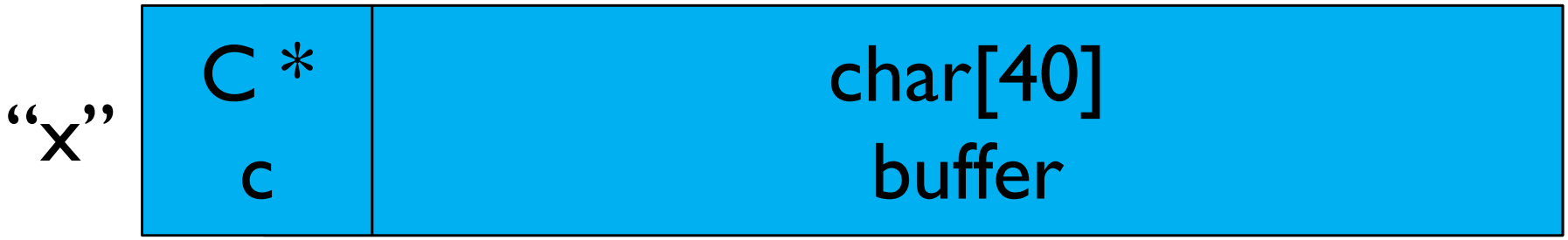
Exploiting Type Errors

- Type A is unrelated to type B

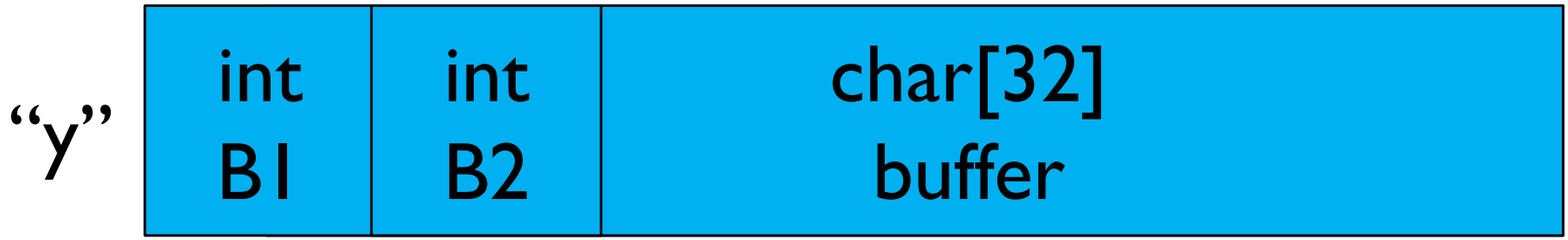


Exploiting Type Errors

- Type A is unrelated to type B



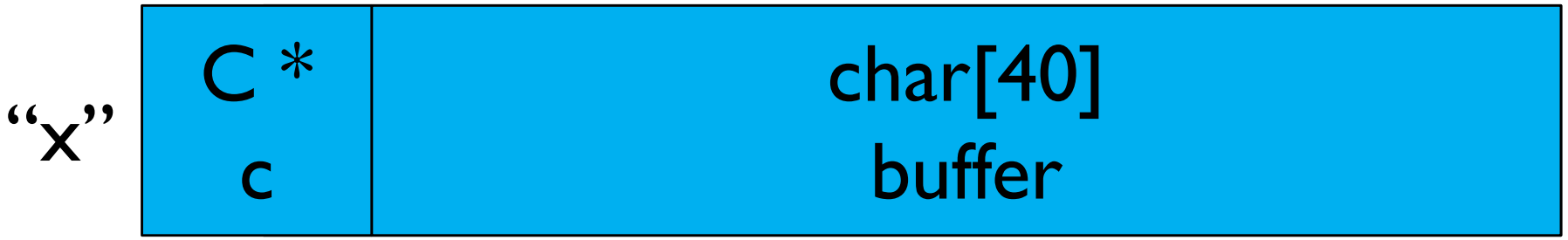
- Type casting "x" to be referenced by "y" of type B



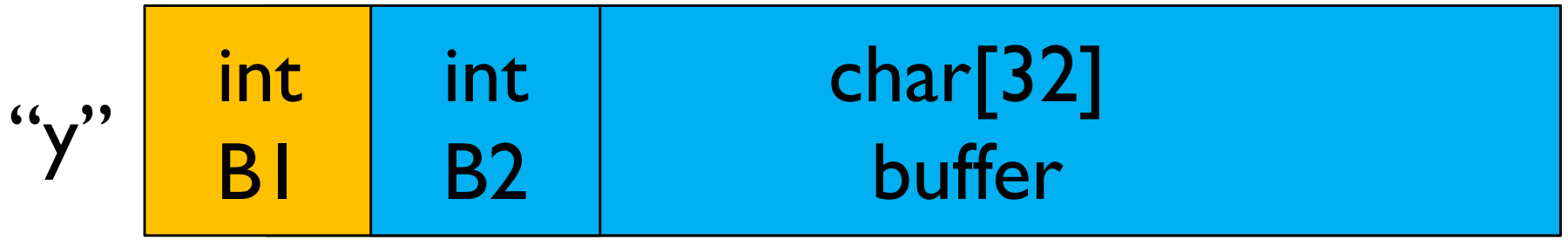
- Why could this become a problem?

Exploiting Type Errors

- Type A is unrelated to type B



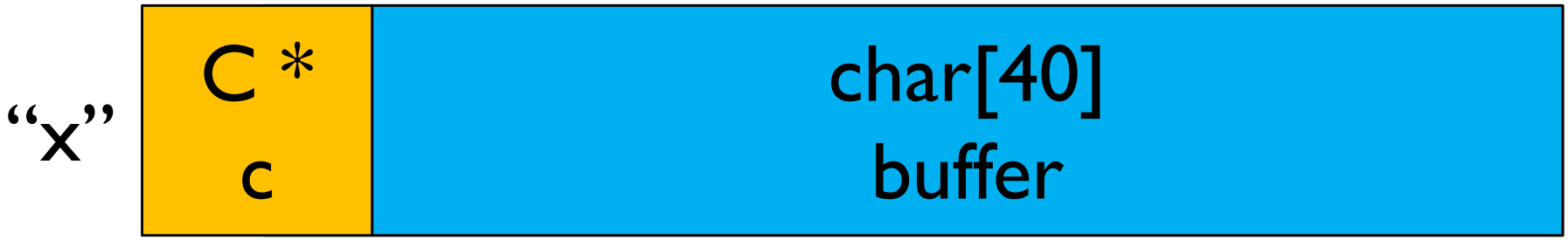
- Type casting "x" to be referenced by "y" of type B



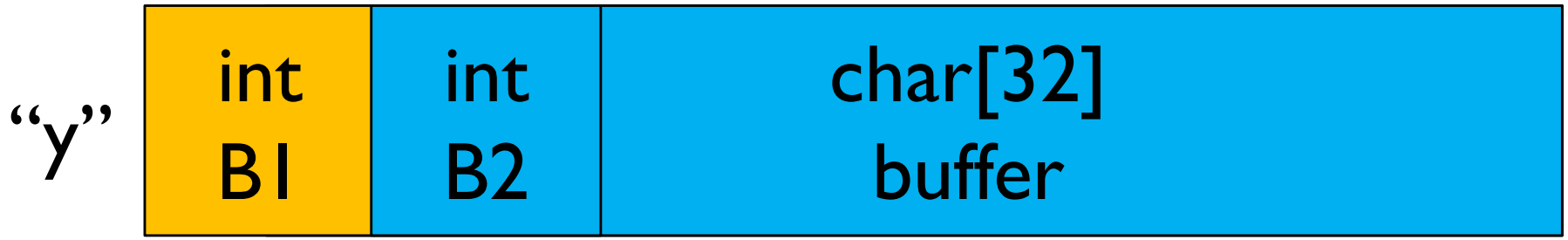
- The code allows assignment of field B1

Exploiting Type Errors

- Type A is unrelated to type B



- Type casting "x" to be referenced by "y" of type B



- The code allows assignment of field B1 of y, which corresponds to field c of x (a data pointer)

Type Confusion

- Adversary can abuse ambiguity to control writes

```
struct A {
struct C *c;
char buffer[40];
};

struct B {
int B1;
int B2;
char info[32];
};
```

```
x = (struct A *)malloc(sizeof(struct A));
y = (struct B *)x;
y->B1 = adversary-controlled-value;
x->c->field = adversary-controlled-value-also;
```

- Arbitrary Write Primitive!
  - Adversary controls the value to write and the location of the write
  - Allow adversary to write an arbitrary value to an arbitrary location

Temporal Memory Errors

- Exploit inconsistencies in the assignment of pointers to memory regions
  - Use-before-initialization
    - Prior to a pointer being assigned to an object (memory region)
  - Use-after-free
    - Use a pointer in a statement after the memory region to which has been assigned has been deallocated
      - And something has been allocated there in its place
- The most common vector for exploits today

Memory Life Cycle

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

Memory Life Cycle

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

Memory Life Cycle - Violated

- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- Normal lifecycle between a pointer and object
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - free(p); // deallocate object

What Is Going Wrong?

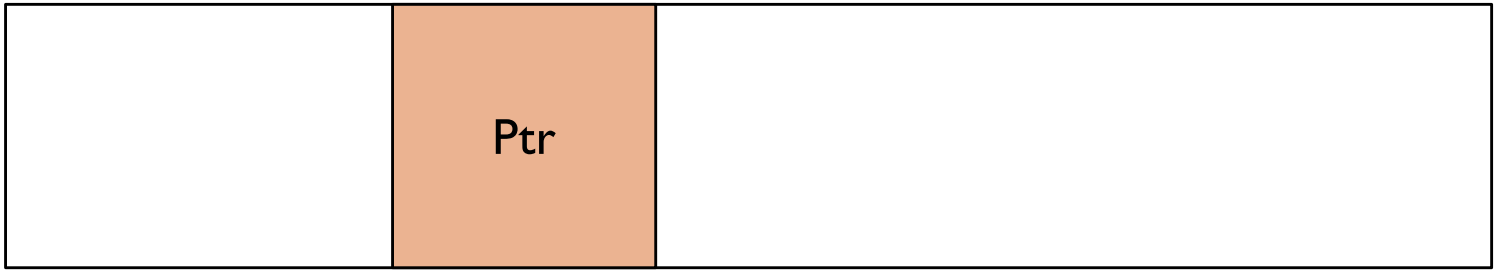
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object

Use-Before-Initialization (UBI)

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p, size, "%s", original_value); // use pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object
- Called “use before initialization” (UBI)
  - Allows an adversary to reference a value that happens to be at the location that “p” is declared (not an assignment)
  - Could be anywhere

Why UBI Is A Problem

- Use before initialization



- Questions to explore
  - Where is the pointer allocated in memory?
    - Can the adversary control what is written to that location
  - What is the pointer's value at initialization?
    - Can this reference a useful target object to attack?

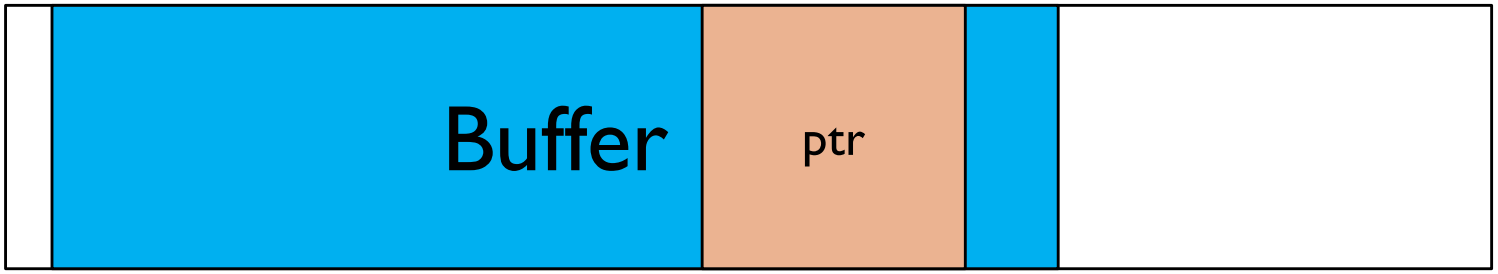
Why UBI Is A Problem

- Use before initialization

- Assume function “A” calls functions “B” and “C”
  - When function “B” is called, a new stack frame is created
  - Using memory in the stack region
  - Suppose there is a string “buffer” built from adversary input
  - Then, function ”B” returns

Why UBI Is A Problem

- Use before initialization



- Assume function “A” calls functions “B” and “C”
  - When function “C” is called, a new stack frame is created
  - Using memory in the stack region – used by function “B”
  - Suppose there is a local variable pointer “ptr” declared in function “C”
  - But, ”ptr” is not initialized – what is the value of “ptr”?

What Is Going Wrong?

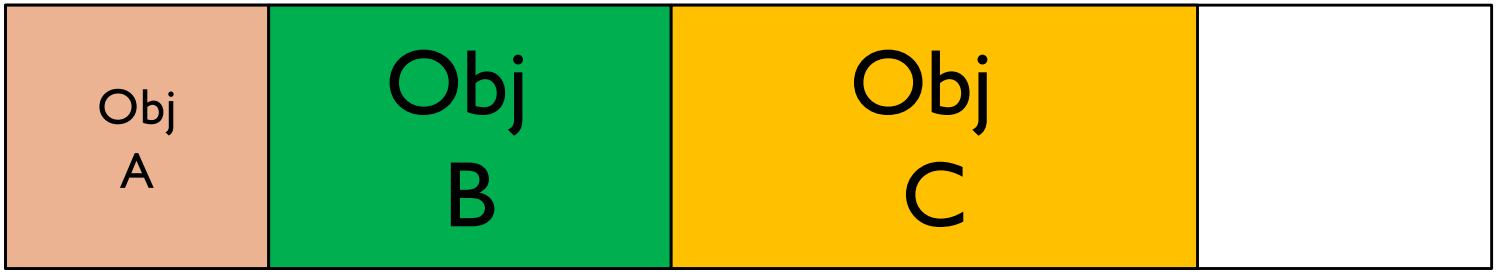
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer

Use-After-Free (UAF)

- A pointer may reference a memory region that does not hold a defined (assigned) object
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size); // define pointer to object
  - free(p); // deallocate object – release memory for reuse
  - len = snprintf(p, size, "%s", original_value); // use pointer
- Called “use after free” (UAF)
  - Allows an adversary to reference a memory region that may be allocated to a different object
  - I.e., imagine a malloc between the free and use

Why Is UAF a Problem

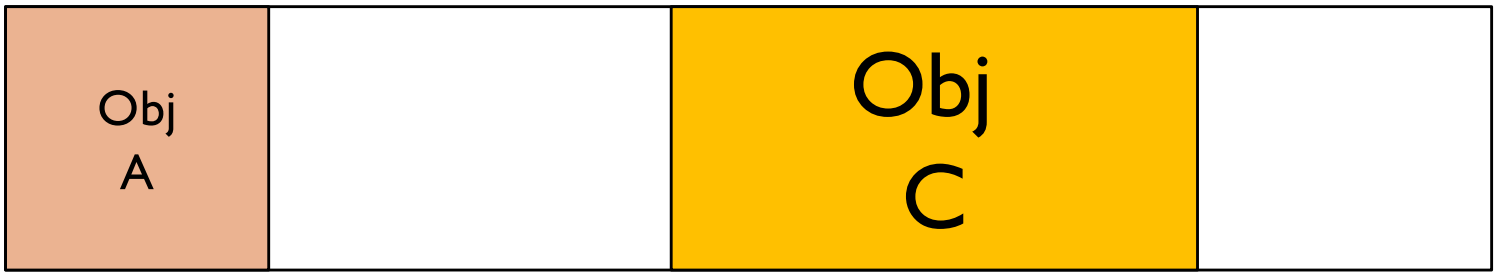
- Use after free



- Assume you have a heap as shown
  - Focus on object ”B”
  - You have a reference to “B” – say pointer “b”

Why Is UAF a Problem

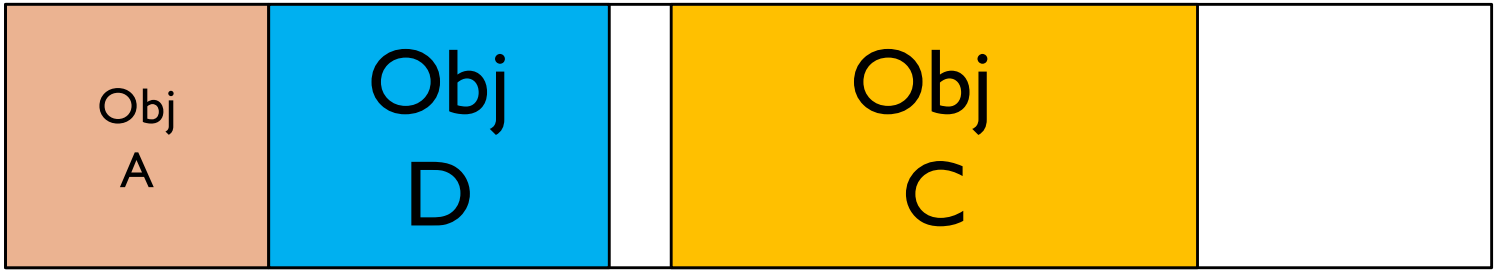
- Use after free



- Assume you have a heap as shown
  - Object ”B” is deallocated
  - And you still have a reference to “B” – e.g., pointer “b”
  - And, pointer “b” may have “uses” after the deallocation of object ”B”
  - But, the allocator is free to reuse the memory region

Why Is UAF a Problem

- Use after free



- Assume you have a heap as shown
  - The allocator chooses to use the memory region for object "D"
  - So, a "use" of pointer "b" will access the object "D" instead
  - What determines the values referenced by "b"?

Conclusions

- Memory errors are still the most common cause of vulnerabilities
- They are caused by C/C++ allows objects (memory regions) and pointers (references to memory locations) to be defined and managed separately
- Thus, C/C++ are neither memory safe nor type safe
- Which leads to spatial, type, and temporal errors

Questions