CS165 – Computer Security

Memory Errors

January 23, 2024

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

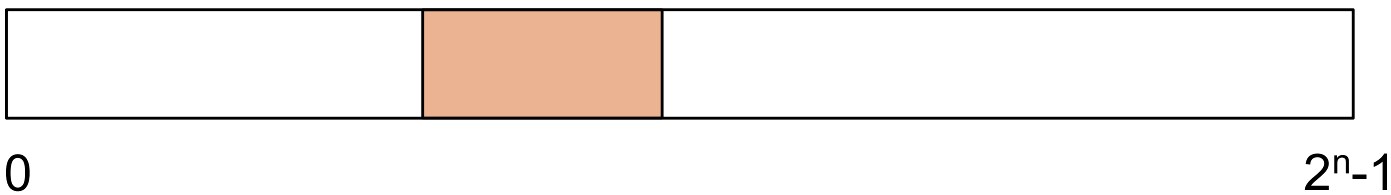
Cause of Memory Errors

- 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

Impact of Memory Errors

C/C++ Memory Model

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



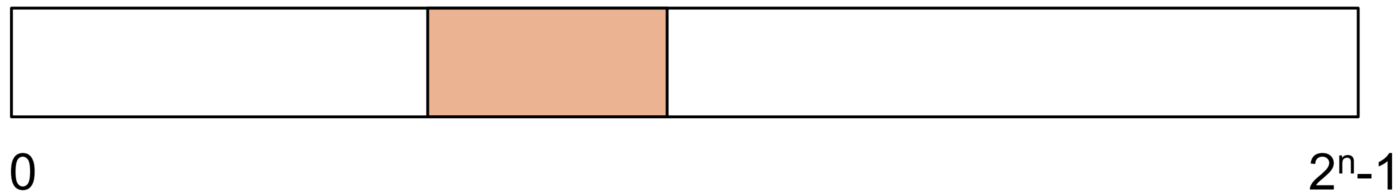
  - An object (in brown) can be allocated anywhere in the array
    - `char *x = (char *)malloc(size);`
  - Your program gets a reference (pointer) to the location of your object in the "array" that is virtual memory
    - It is up to the programmer to set and use the pointer correctly to access the object
    - 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
  - 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

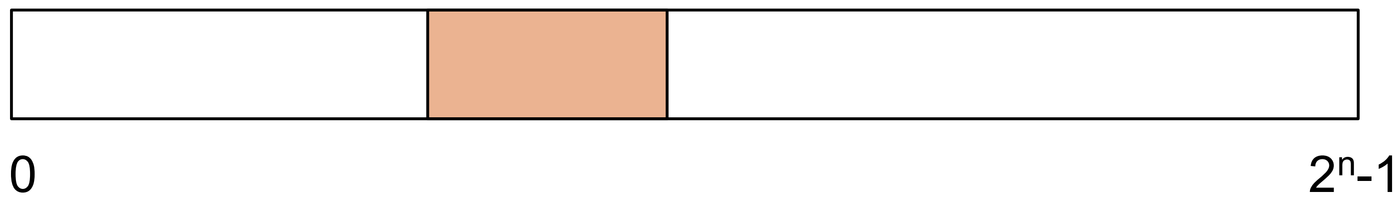
- An object (in brown) can be allocated anywhere in the array
  - `char *x = (char *)malloc(size);`



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

C/C++ and Memory Safety

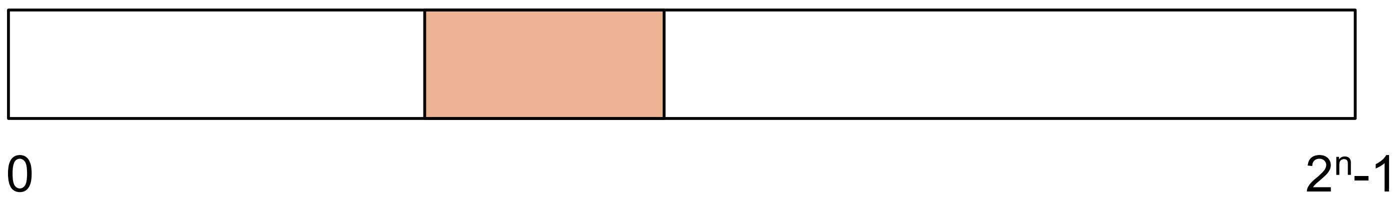
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



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

C/C++ and Memory Safety

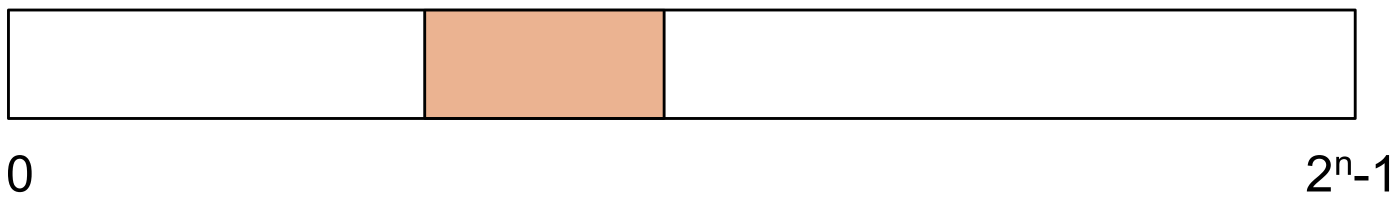
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- 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

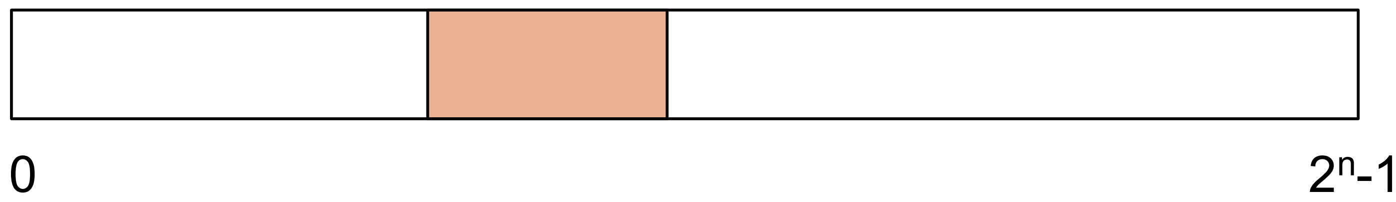
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



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

C/C++ and Memory Safety

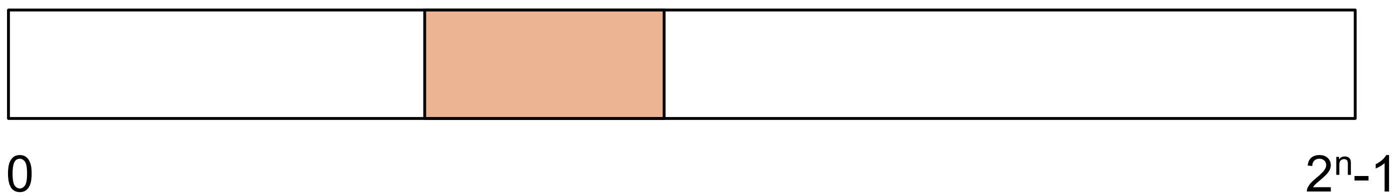
- An object (in brown) can be deallocated at any time
  - `char *x = (char *)malloc(size);`



- 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

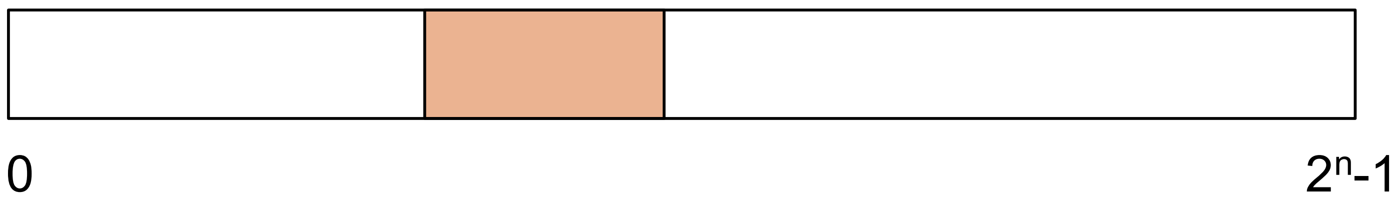
- An object (in brown) can be assigned a type
  - `char *x = (char *)malloc(size);`



- 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

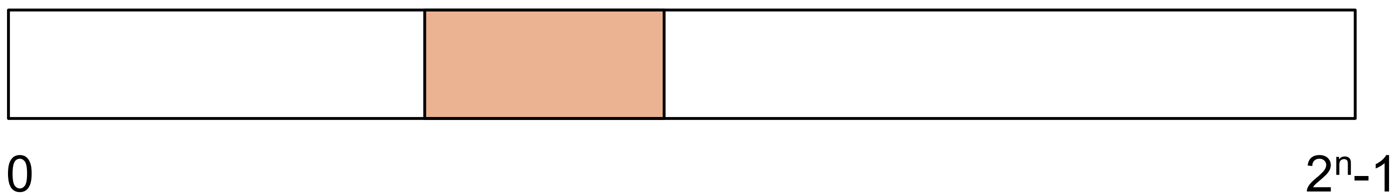
- An object (in brown) can be assigned a type
  - `char *x = (char *)malloc(size);`



- 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

```
#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”

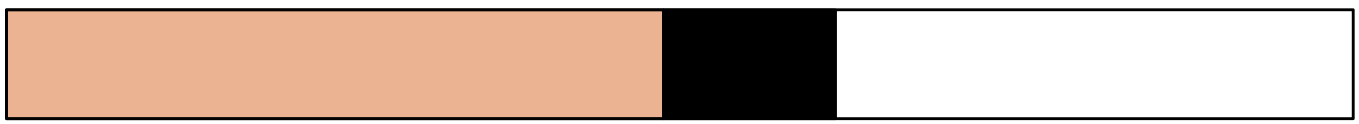
```
#include <stdio.h>

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?

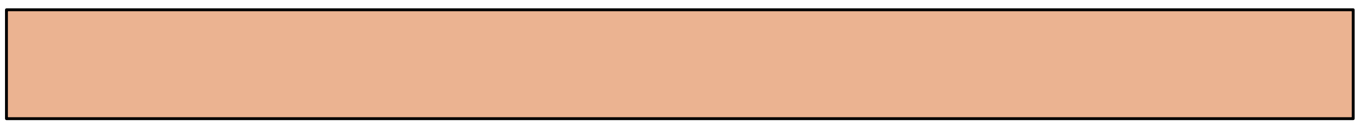
- Fill buffer to length of allocated buffer (10)
  - Scanf – input a string (source) of length 5



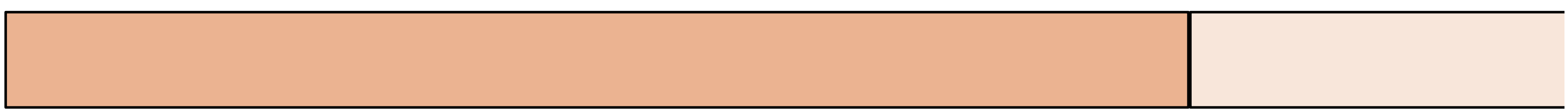
  - Null termination of string (optional)

What is happening?

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



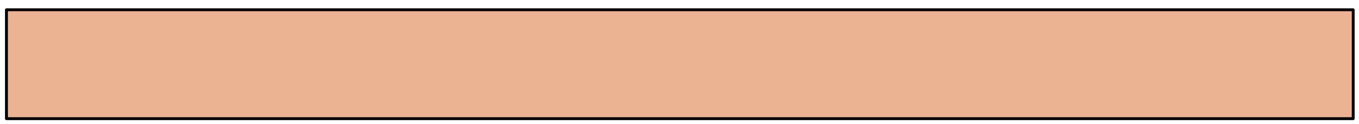
  - Write beyond the end of the allocated memory for buffer



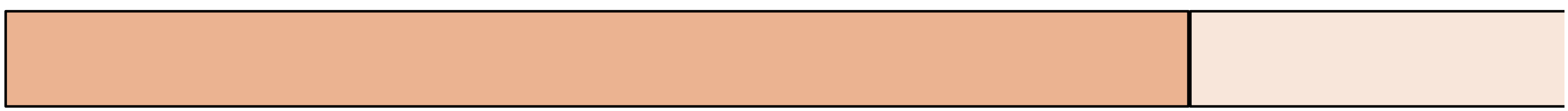
  - Nothing stops that
    - What is beyond the end of one allocated region?

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

```
#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);
}
```

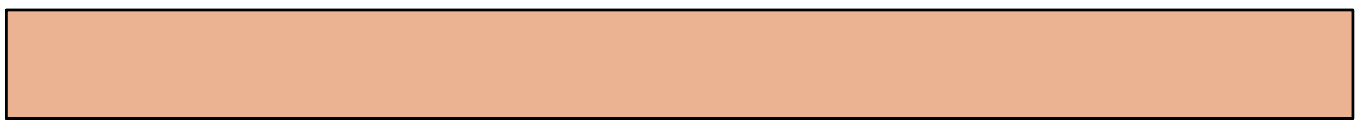
More Complex Vulnerability

- Another flaw

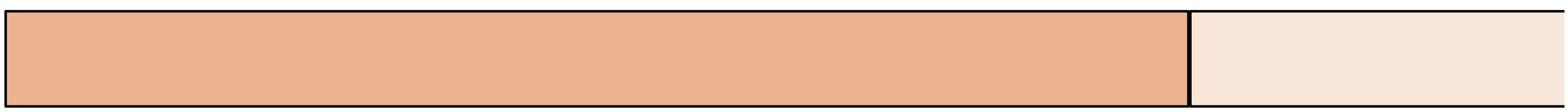
```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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?

What Is Going Wrong?

- Both of these functions process "strings"?
  - What is a string?

What Is Going Wrong?

- Both of these functions process “strings”?
- What is a string?
  - Sequence of bytes terminating with a null byte
- But, C/C++ do not differentiate strings from arrays of bytes (char *)
  - Which need not be null-terminated
  - What happens then?

What Is Going Wrong?

- Both of these functions process "strings"?
- What is a string?
  - Sequence of bytes terminating with a null byte
- But, C/C++ do not differentiate strings from arrays of bytes (char *)
  - Need not be null-terminated
  - What happens when you read a string w/o a null-terminating byte?
- Keep reading the value until you hit a null byte

String 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

Obvious Solution in C

- “Obvious” solution is to always enforce bounds

Function w/o Bounds Checks

- gets(3) – reads input without checking. Don't use it!
- strcpy(3) – *strcpy(dest, src)* – copies from *src* to *dest*
  - If *src* longer than *dest* buffer, keeps writing!
- strcat(3) – *strcat(dest, src)* – appends *src* to *dest*
  - If src+data-in-dest longer than dest buffer, keeps writing!
- Many other dangerous functions, e.g.:
  - realpath(3), getopt(3), getpass(3)
  - streadd(3), strecpy(3), and strtrns(3)

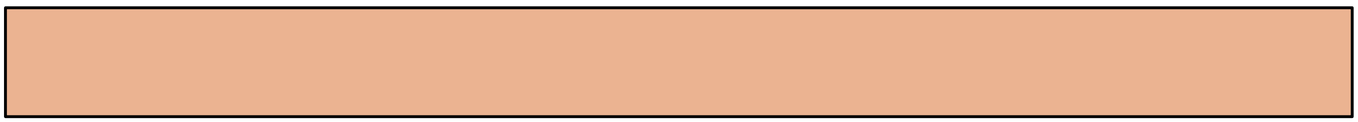
- Don't use these!

Traditional Solutions

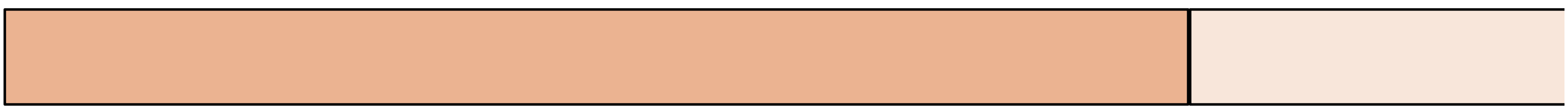
- Depend mostly on strncpy(3), strncat(3), sprintf(3)
  - Can be hard to use correctly
  - char **strncpy*(char *DST, const char *SRC, size_t LENGTH)
    - Copy bytes from SRC to DST
    - Up to LENGTH bytes; if less, NULL-fills
- If LENGTH is the size of the DST memory region
  - Can fill memory region without null-terminator
    - Thus, does not guarantee creating a C string
  - Can truncate "in the middle," leaving malformed data
    - Yet difficult to detect when it happens
- Not a correct solution

strncpy(buffer, “0123456789”, 10)

- Strncpy stops the copy after 10 bytes
  - Since buffer is 10 bytes – no room for the terminator byte



  - Prevents any write beyond the end of the allocated memory for buffer if the “size” argument is correct



  - But, nothing guarantees that

Traditional Solution – That Works!

- Available now: snprintf(3), vsnprintf(3)
  - Essentially the same functions, although arg format differs
- int *snprintf*(char *S, size_t N, const char *FORMAT, ...);
  - So, you should use this for safe programming today
  - Replaces strcpy and others directly

Traditional Solution – That Works!

- int *snprintf*(char *S, size_t N, const char *FORMAT, ...);
  - Writes output to buffer S up to N chars (bounds check)
  - Always writes '\0' at end if N>=1 (terminate)
  - Returns "length that would have been written" or negative if error (reports truncation or error)
- Thus, achieves goals of correct bounds checking
  - Enforces bounds, ensures correct C string, and reports truncation or error
    - len = snprintf(buf, buflen, "%s", original_value);
    - if (len < 0 || len >= buflen) … // handle error/truncation

Scanf and Friends

- What about other functions like scanf?
  - scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf
    - all unsafe by default

Scanf and Friends

- What about other functions like scanf?
  - scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf
    - all unsafe by default
  - Fortunately, these can be made safe quite easily
    - By leveraging auto-resizing option

Scanf and Friends

- What about other functions like scanf?
  - scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf
    - all unsafe by default
  - Instead, use “%ms” to auto-resize
    - char *buffer = NULL;   // Must be set to NULL
    - scanf(buffer, “%ms”);
  - Allocates memory for the buffer dynamically to hold input safely – null-terminated, no truncation required
- Note: also, can use for other functions that process input like getline
- Note: You need to deallocate when completely done

Type Errors

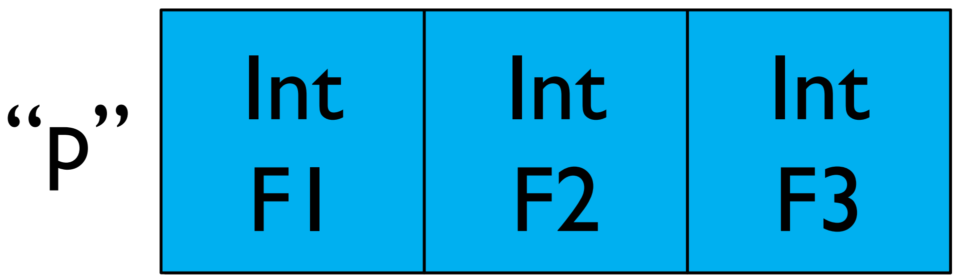
- Errors that permit access to memory according to a multiple, incompatible formats
  - These are called type errors
  - Access using a different "type" than used to format the memory
- Most of these errors are permitted by simple programming flaws
  - Of the sort that you are not taught to avoid
  - 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

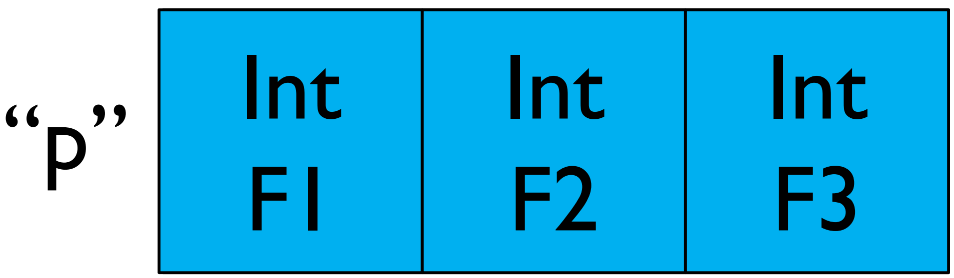
- "p" is assigned to an object of type t1



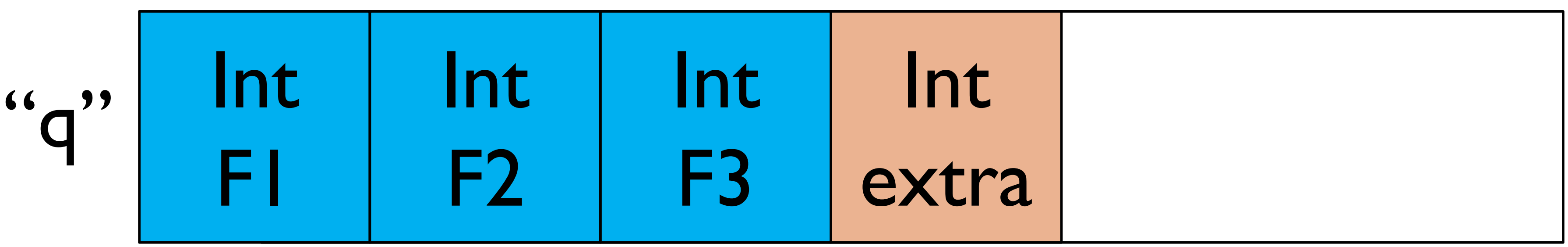
- Only memory large enough for t1 is allocated

Exploiting Type Errors

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



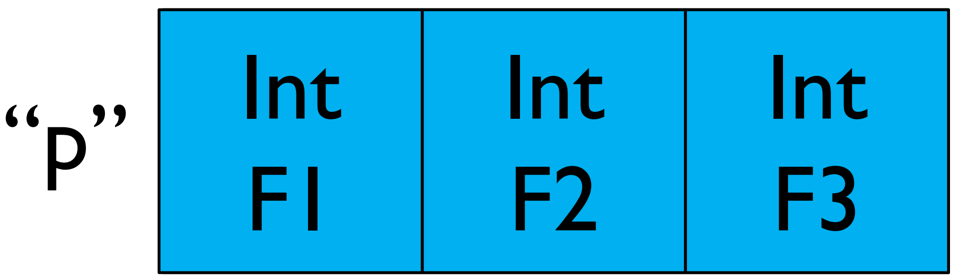
- But, if we assign a pointer of type t2 to the object



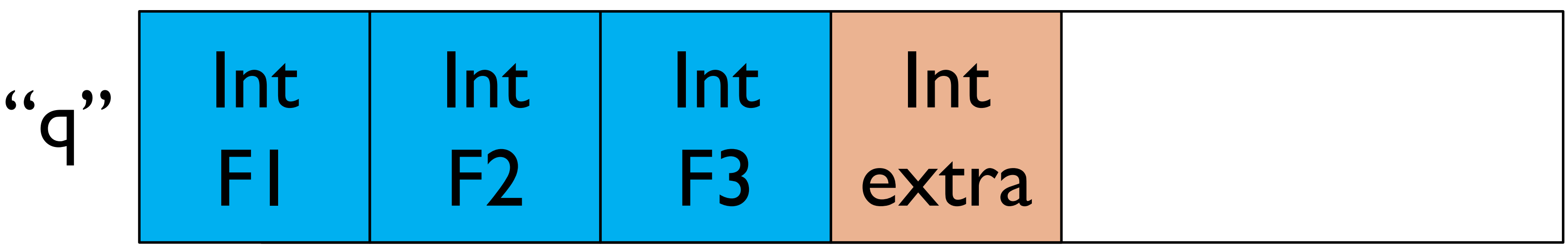
- This is what can be referenced by "q"
  - "q" of type t2 thinks it is referencing a larger region

Exploiting Type Errors

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



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



- 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

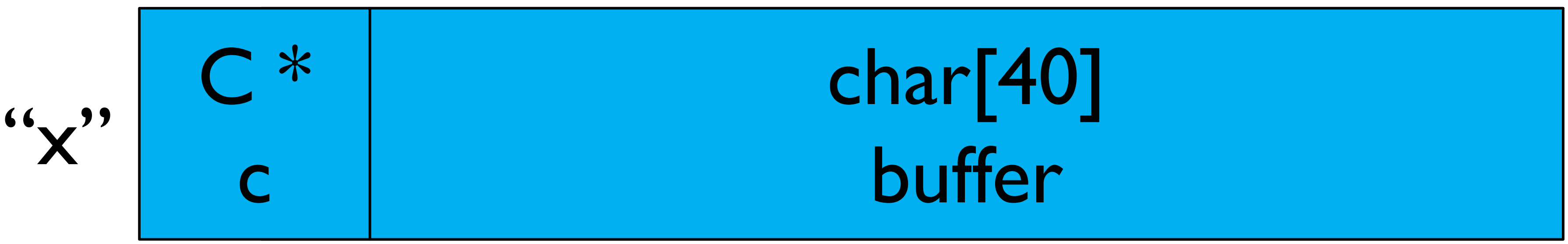
```
struct A {
struct C *c;
char buffer[40];
};

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

“x”	C * c	char[40] buffer

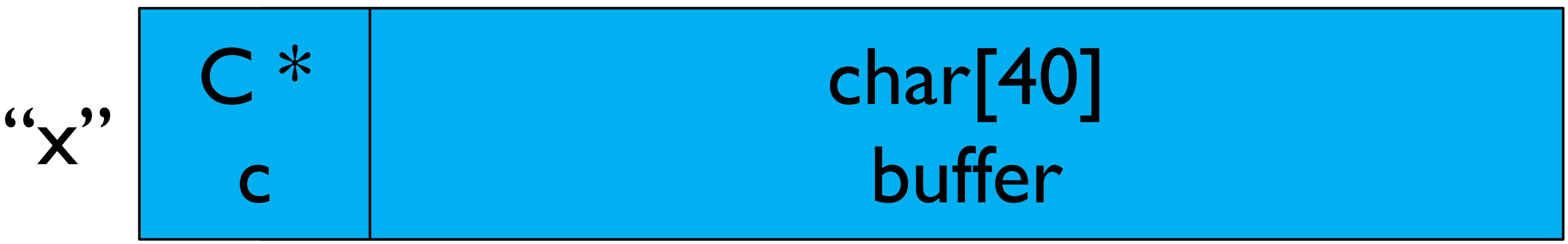
- Type casting “x” to be referenced by “y” of type B

“y”	int B1	int B2	char[32] buffer

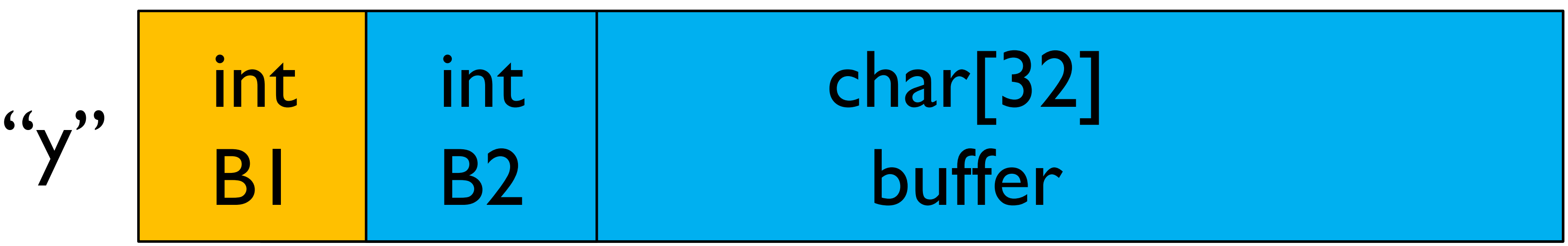
- Why could this become a problem?

Exploiting Type Errors

- Type A is unrelated to type B



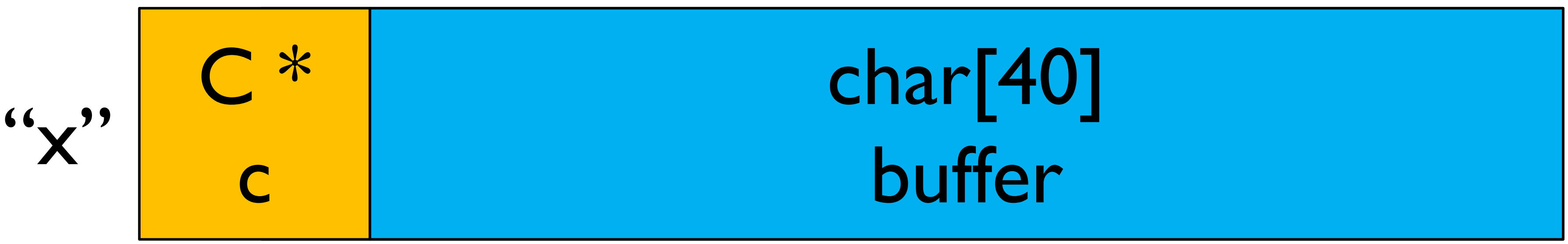
- Type casting "x" to be referenced by "y" of type B



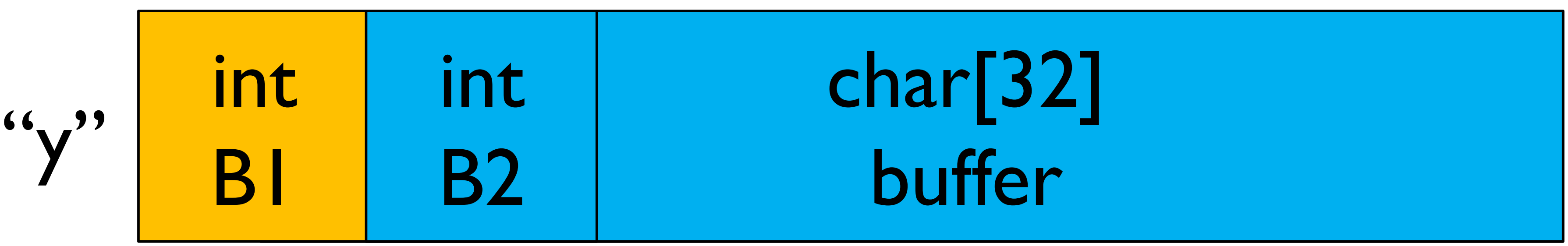
- The code allows assignment of field B1

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

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

Who Would Do That?!

- How could such an error happen?

Who Would Do That?!

- How could such an error happen?
- Several ways
  - Type casts
  - Unions – use the same memory with multiple formats
  - Use-before-initialization (UBI)
  - Use-after-free (UAF)
- The last two are due to bugs created because C/C++ requires the programmer manage memory
  - Temporal errors

Unions

- Example of a union data structure

Defining a `union` typed variable:

- Just like a `struct` data type, you can **define *variables*** of a `union` data type after you have **defined** the ***structure* of a `union`** data type

  Example:

```
union myExample     // Union definition
{
   int     a;
   double  b;
   short   c;
   char    d;
};

union myExample  x;  // Define a variable of the type union myExample
```

Observe that:

- ***Every* member variable** in a `union` **typed variable** start at the ***same* memory address**
- The **number of bytes** used to store a **member variable** depends on the ***size* (= data type)** of the **member variable**,
  - `a` uses **4** because it is an `int` type variable
  - `b` uses **8** because it is an `double` type variable
  - And so on.
- The **size** of a `union` **typed variable** is equal to the **size** of the ***largest* component variable**

http://www.cs.emory.edu/~cheung/Courses/255/Syllabus/2-C-adv-data/union.html#:~:text=A%20union%20data%20structure%20is,variables%20at%20any%20one%20time

Unions

- Example of a union data structure
  - We can **easily** show the above **facts** with the following **C program**:

```
union myUnion    // Union structure
{
   int    a;
   double b;
   short  c;
   char   d;
};

struct myStruct     // Struct with the same member variables
{
   int    a;
   double b;
   short  c;
   char   d;
};

int main(int argc, char *argv[])
{
   struct myStruct s;      // Define a struct
   union myUnion   u;      // and a union variable

   // Print the size and the address of each component

   printf("Structure variable:\n");
   printf("sizeof(s) = %d\n", sizeof(s) );
   printf("Address of s.a = %u\n", &(s.a) );
   printf("Address of s.b = %u\n", &(s.b) );
   printf("Address of s.c = %u\n", &(s.c) );
   printf("Address of s.d = %u\n", &(s.d) );

   putchar('\n');

   printf("Union variable:\n");
   printf("sizeof(u) = %d\n", sizeof(u) );
   printf("Address of u.a = %u\n", &(u.a) );
   printf("Address of u.b = %u\n", &(u.b) );
   printf("Address of u.c = %u\n", &(u.c) );
   printf("Address of u.d = %u\n", &(u.d) );
}
```

Output:

```
Structure variable:
sizeof(s) = 24
Address of s.a = 4290768696
Address of s.b = 4290768704
Address of s.c = 4290768712
Address of s.d = 4290768714

Union variable:
sizeof(u) = 8
Address of u.a = 4290768688    (Same location !!!)
Address of u.b = 4290768688
Address of u.c = 4290768688
Address of u.d = 4290768688
```

http://www.cs.emory.edu/~cheung/Courses/255/Syllabus/2-C-adv-data/union.html#:~:text=A%20union%20data%20structure%20is,variables%20at%20any%20one%20time

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

- 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?

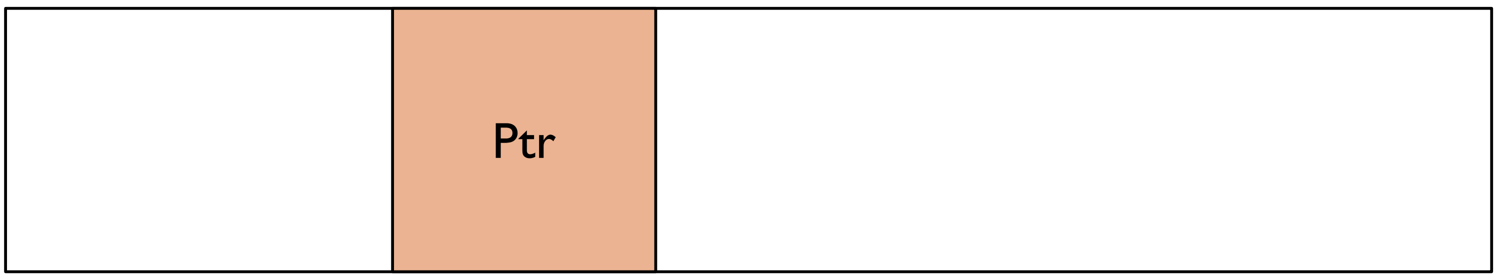
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - len = snprintf(p,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

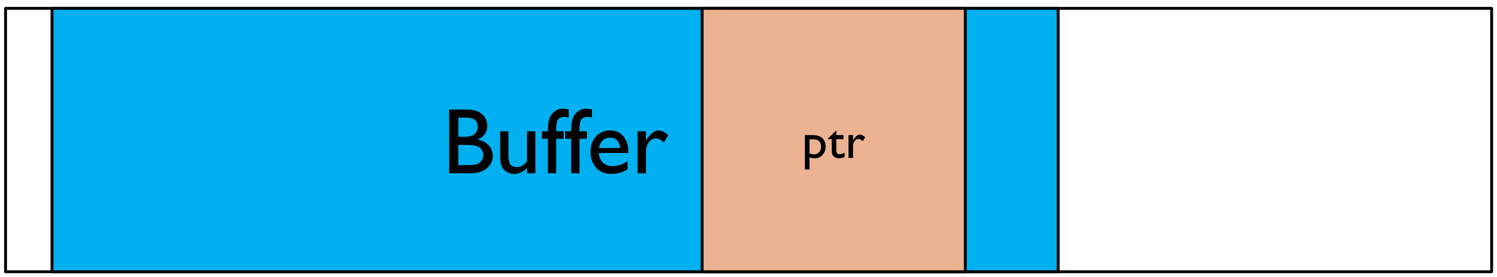
Why UBI Is A Problem

- 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?

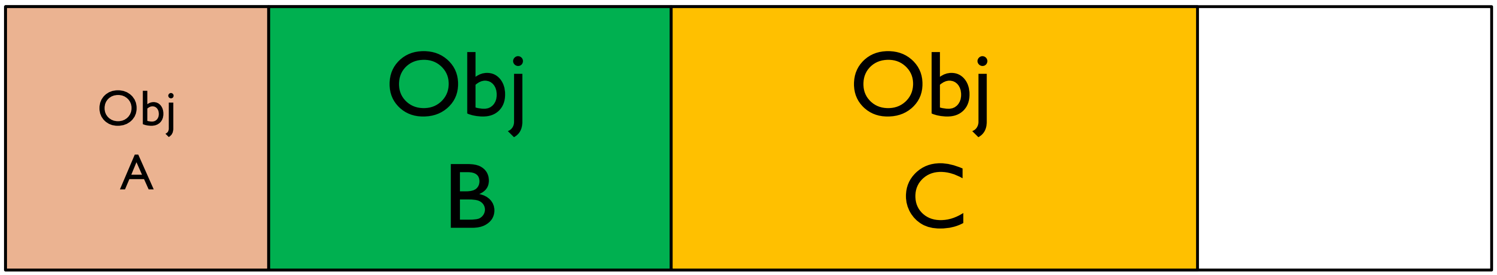
- We have objects (memory regions) and references (pointers)
  - What goes wrong in temporal errors?
- A pointer may reference (use) a memory region that does not hold the object to which the pointer was assigned
- What does ”p” reference upon use?
  - char *p; // declare pointer
  - p = (char *) malloc(size);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

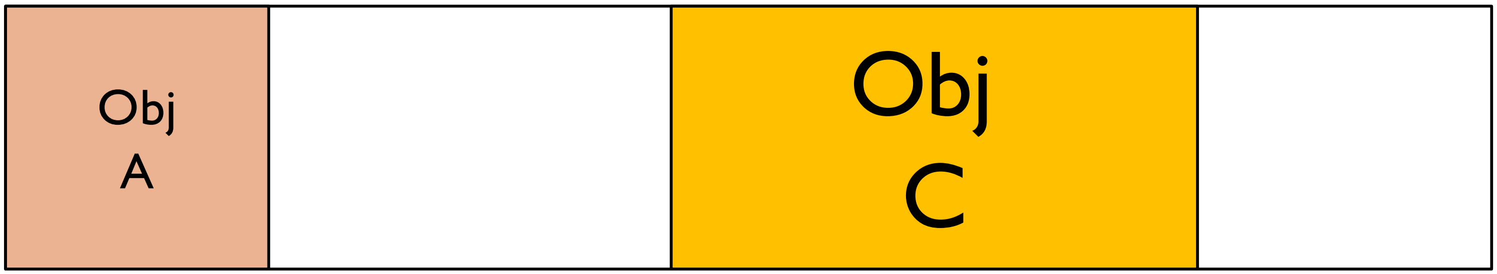
- Use after free



- 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

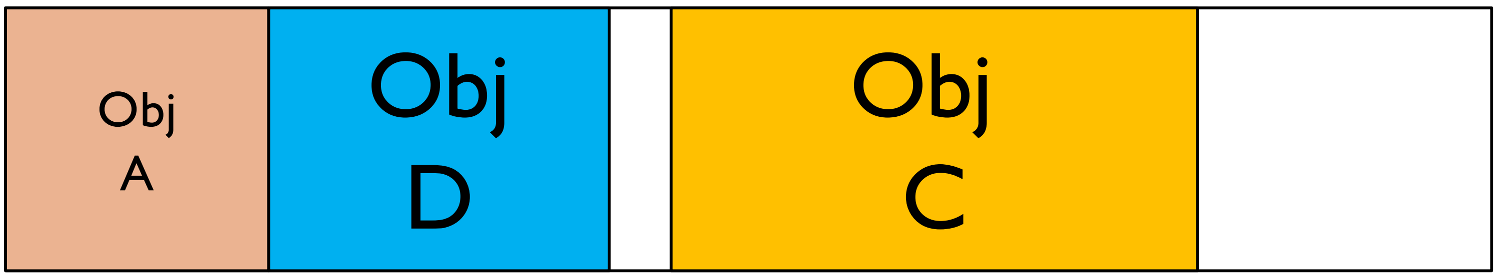
- Use after free



- 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