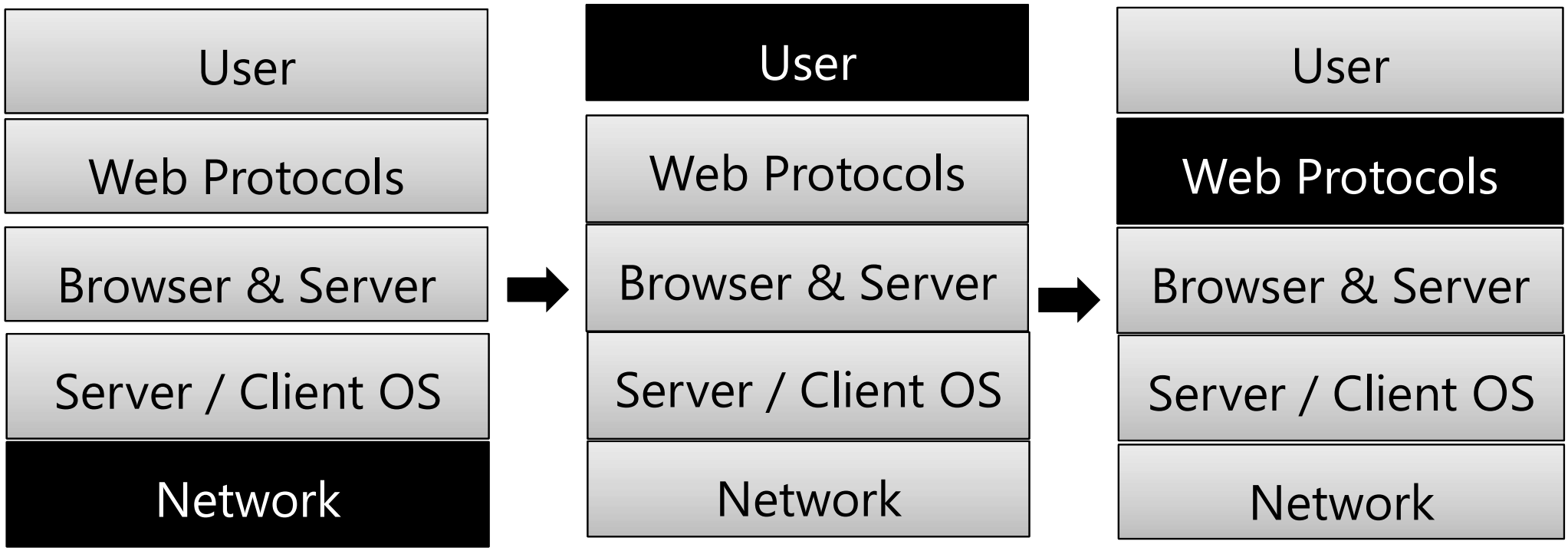
Memory Safety Vulnerabilities

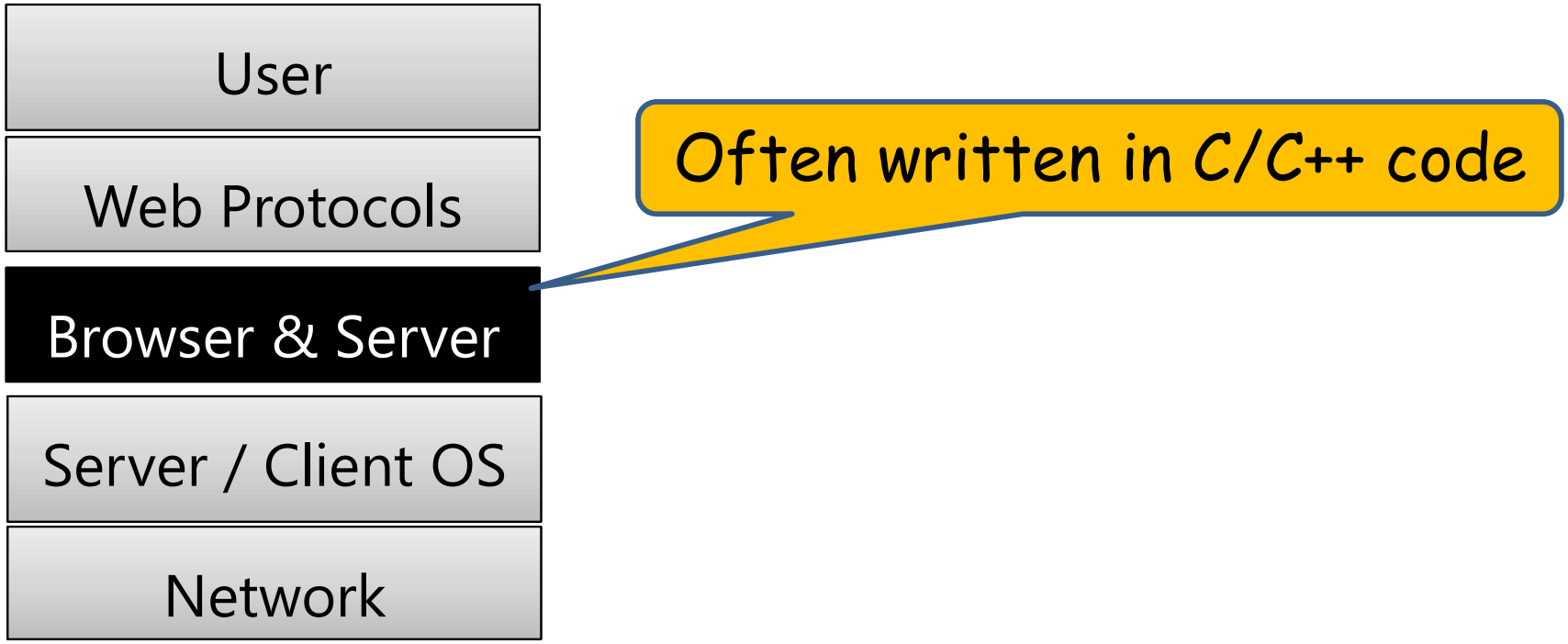
Prateek Saxena

Recap: Threat Models



So far…

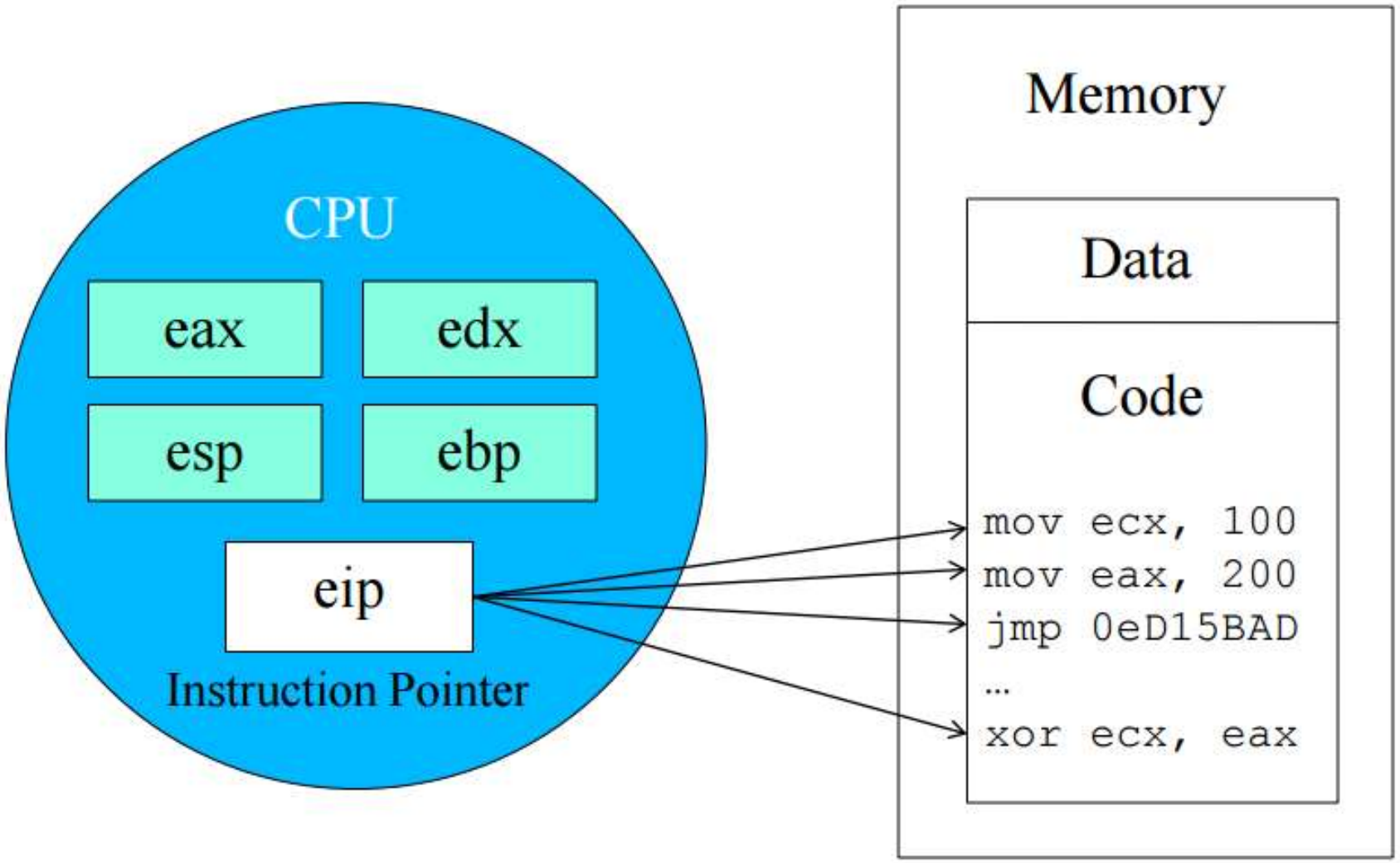
Today: Threat Model



Benign-but-buggy Application Code:
Application stack is well-intentioned but has a coding bug.

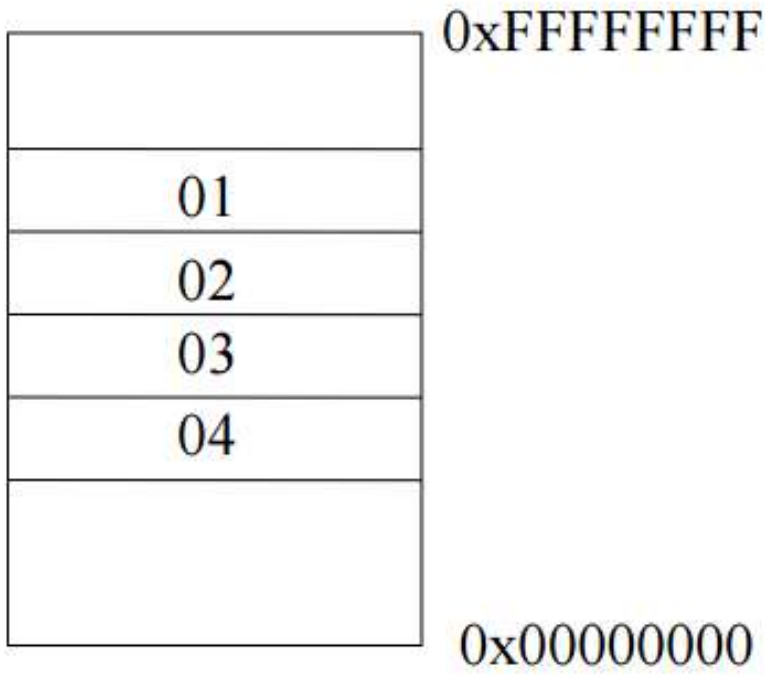
Background:
Primer on Systems Programming

Basics:
The x86 Machine Model



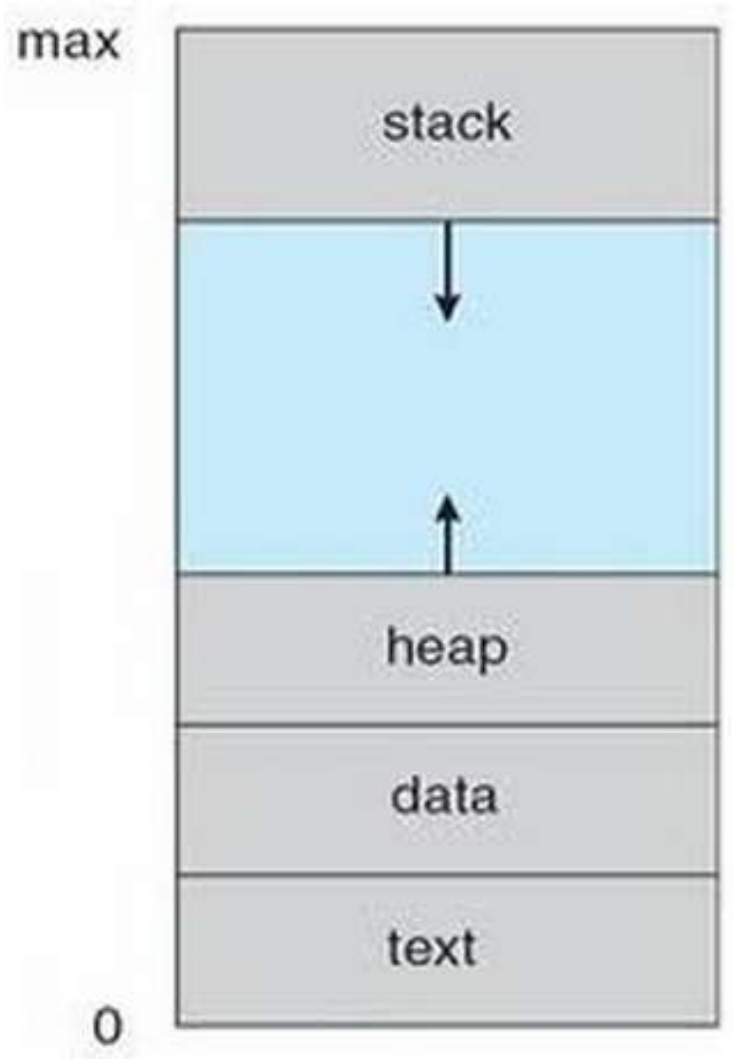
Basics:
The x86 Machine Model

- Both code and data are represented as numbers
- Code
  - `lea ecx, [esp+4]` represented as `0x8d 0x4c 0x24 0x04`
- Data
  - On Intel CPUs, least significant bytes is put at lower addresses
  - It is called little endian
  - For example, `0x01020304`



Basics:
The x86 Machine Model

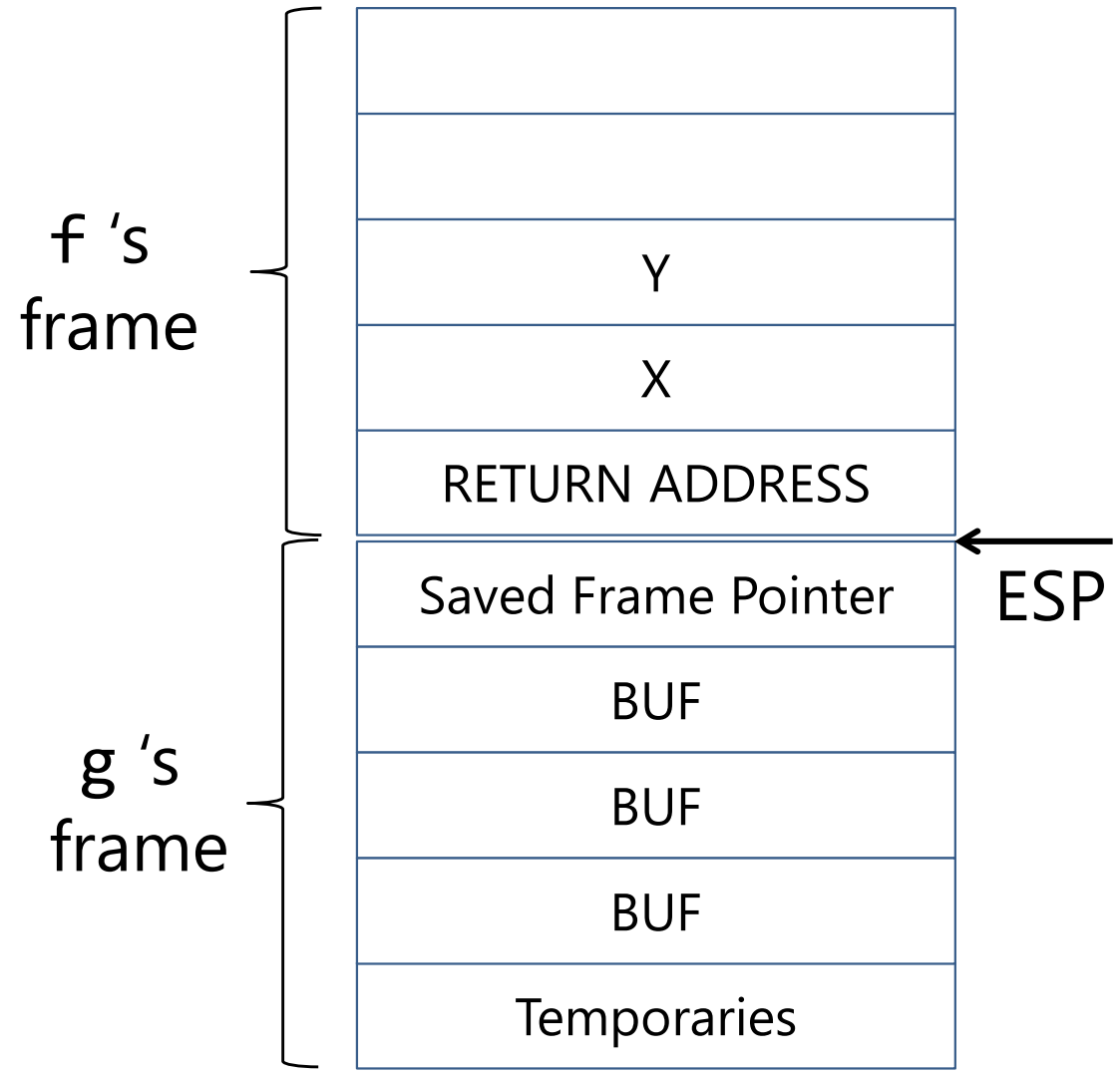
- Registers, Instructions, Stack, EIP
- Addressing modes, offset addresses
  - `mov 0x12[ebp], ecx`
- Stack grows down, other memory accesses move up.



Stack Frames

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```



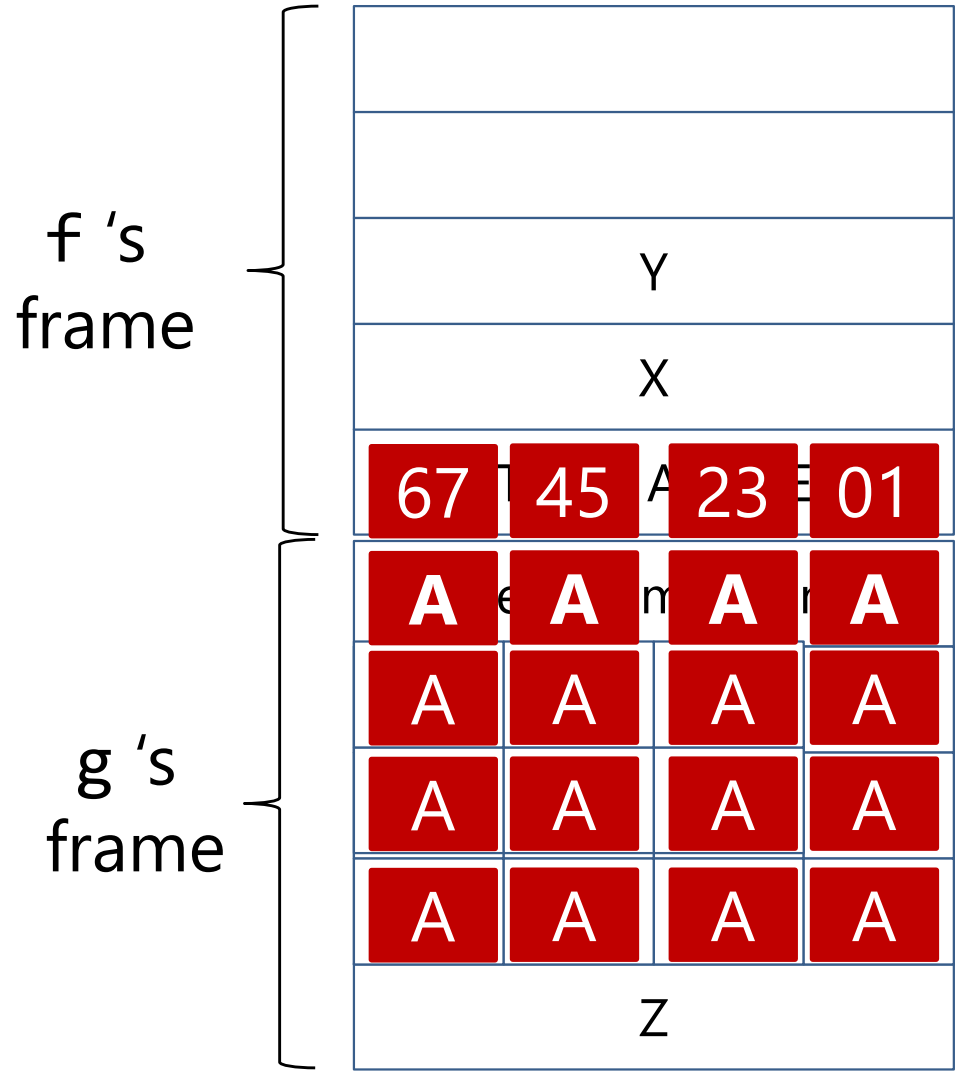
What address will it return to?

Spatial Memory Errors: Buffer Overflows

Buffer Overflows

```
int f() {
…
  g (x, y);
}

int g(int x, int y) {
  char buf[50];
  scanf(“%s”, buf);
}
```

```
.g

push ebp

…
call scanf
…
pop ebp
ret
```

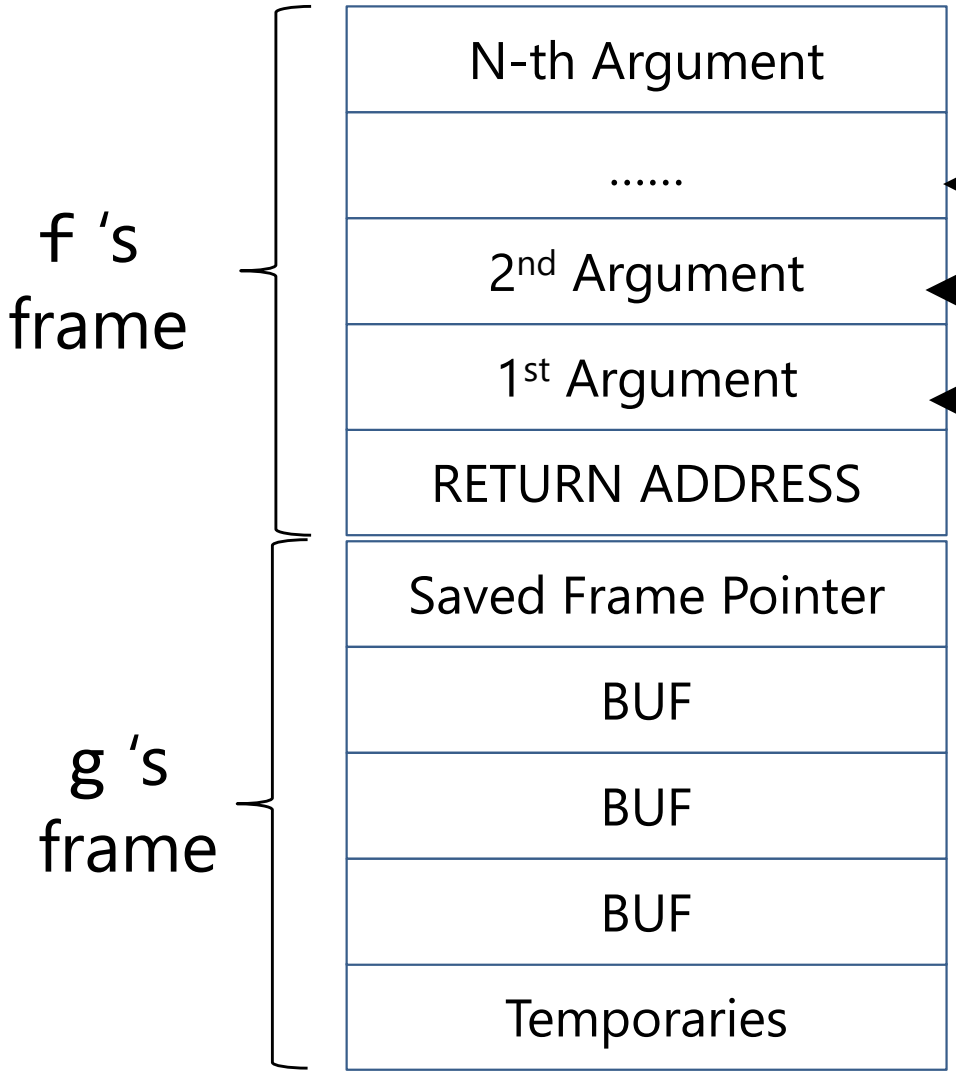


What address will it return to?

Refer to Tutorial 1
(LumiNUS Self-Study material)

Spatial Memory Errors: Format String Bugs

Variable Argument Functions (Example – Printf / Scanf)

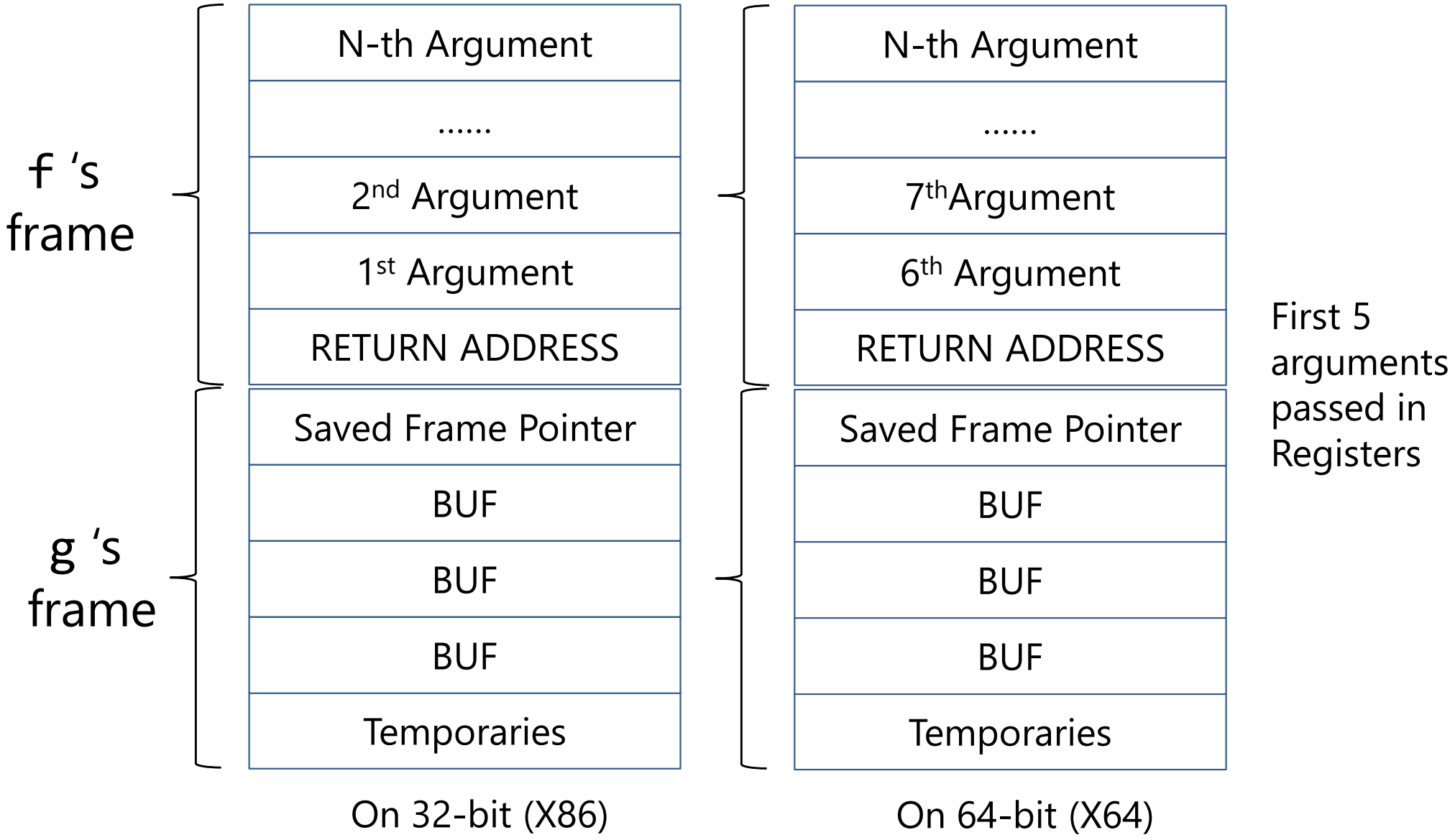


The format string is "interpreted" by the "Variable-argument" function (callee). In this example, there are expected to be 2 more arguments after the format string: the first one is a character string pointer, and the next one is integer

On 32-bit (X86)

Format String Vulns.

Architectural Differences (x86 vs x64)

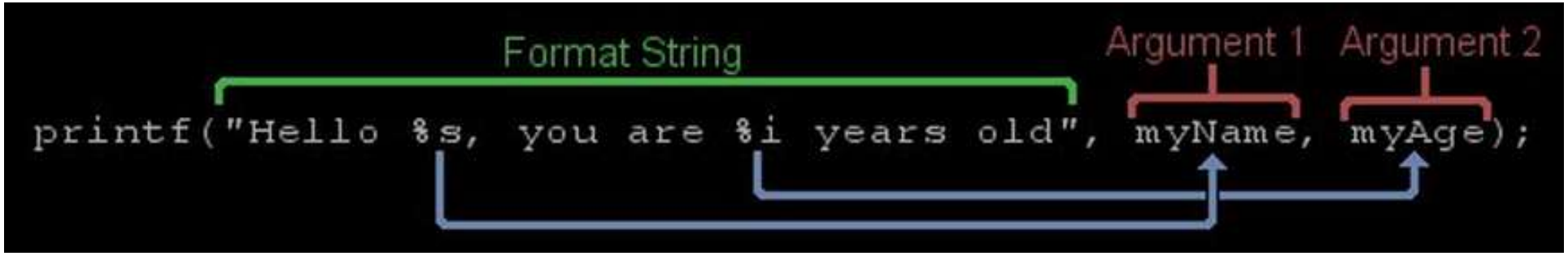


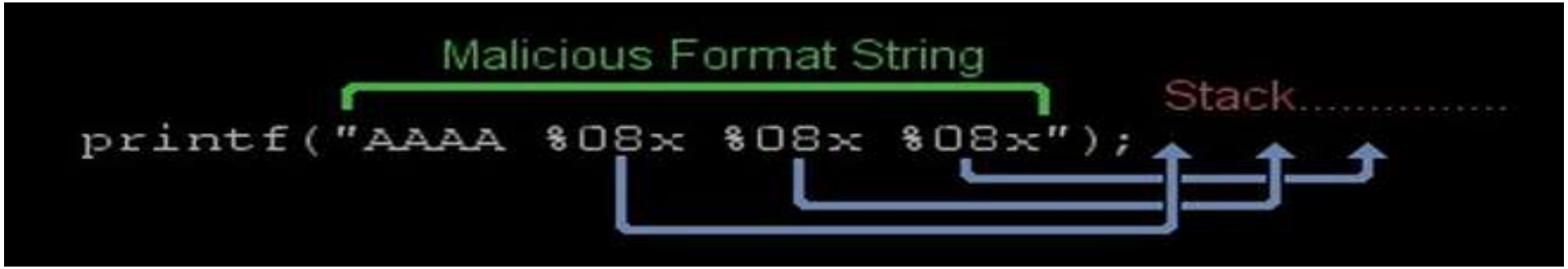
Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);
    printf("What is the access code? ");

     scanf(“%d”, &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
…}
```

Format String Vulns.

Format String Vulnerabilities



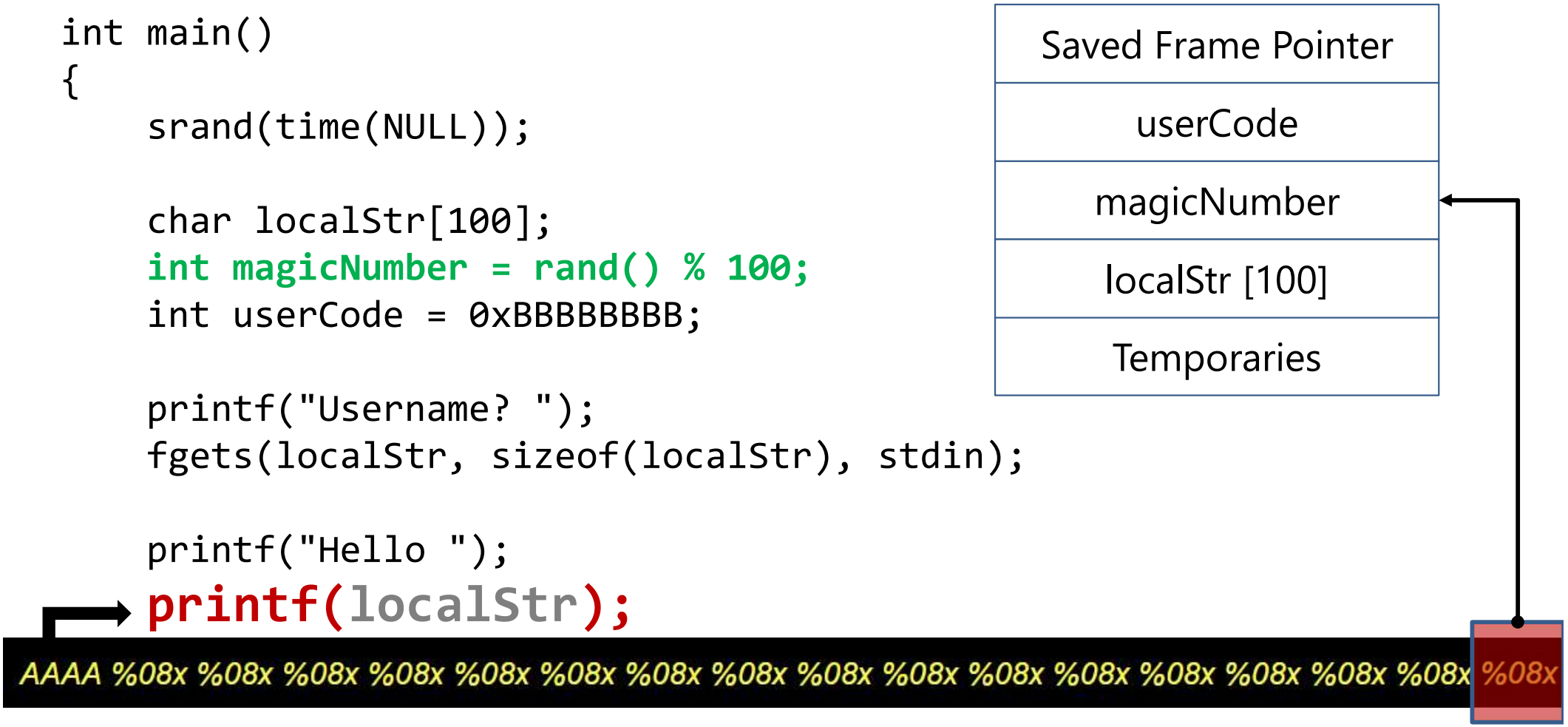


Format String Vulns.

Format String Vulnerabilities

```
#include <stdio.h>

int main()
{
    srand(time(NULL));

    char localStr[100];
    int magicNumber = rand() % 100;
    int userCode = 0xBBBBBBBB;

    printf("Username? ");
    fgets(localStr, sizeof(localStr), stdin);

    printf("Hello ");
    printf(localStr);

    scanf("%d", &userCode);
    if (userCode == magicNumber)
        printf("You win!\n");
… }
```

The "main" stack frame



[Format String Vulns.]

Format String Specifiers

Format String: Example "%s%d"

Format Specifiers

%d - print as number
%p - print as pointer
%c - print as character
%s - read from the address provided and print bytes until the NULL byte is reached
%n - write number of bytes already printed in the address provided
<n>$ - accesses the nth positional argument with respect to printf (ex: %5$p)

Format String Vulns.

Refer to Tutorial 2
(LumiNUS Self-Study material)

Temporal Memory Errors: Use-after-free & Double Free

Lifetime & Scope of Variables

- **Scope**:
  - Region of code where a variable can be accessed
  - E.g. Global, Function-local, Heap (dynamic)
- **Lifetime**:
  - Portion of program execution during which storage is guaranteed
  - E.g. Auto vs. static

Are Programming Language Abstractions

- Not instruction set / hardware abstractions

```
1. int z=0;
2. int g(int x, int y) {
3.       char* buf;
4.       buf = malloc (50);
5.       scanf(“%s”, buf);
6.       free (buf);
7. …
8. }
```

Variable	Scope	Lifetime
z	Global, Line 1-8	Throughout the program
x	Local, Line 3-7	Execution of g
y	Local, Line 3-7	Execution of g
buf	Local, Line 3-7	Execution of g
	Constant Literal, Line 5	Execution of Line 5 (undefined?)
“%s”	Heap, Line 4-7	Line 5-6
*buf		

Question

```
int foo() {
int *p;
            {
            int x = 5;
            p = &x;
            }
return *p;
}
```

What will this program return?

Answer:
Undefined behavior as per C11 standard

Why?

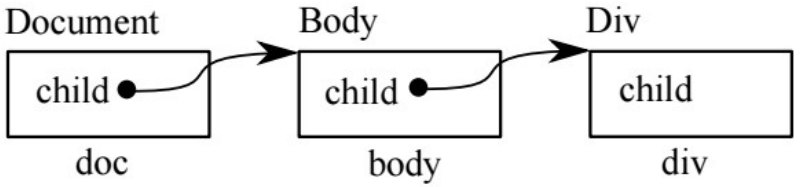
- The lifetime of x is within the inner {}
- The compiler can choose to remove the storage for “x”
- The pointer p is in scope at the last line
- The pointed-to object, however, is accessed out of scope!

Temporal Memory Errors

Example: Use-after-free

Temporal Mem Error: When program accesses mem. beyond its valid lifetime!

```
1  class Div: Element;
2  class Body: Element;
3  class Document {
4    Element* child;
5  };
6
7  // (a) memory allocations
8  Document *doc = new Document();
9  Body *body = new Body();
10 Div *div = new Div();
11
12 // (b) using memory: propagating pointers
13 doc->child = body;
14 body->child = div;
15
16 // (c) memory free: doc->child is now dangled
17 delete body;
18
```



Dangling Pointers [NDSS'15]

Type Errors:
Integer Overflow

A Puzzle:
Is this code free from buffer overflow?

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);

      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

OWASP – Integer Overflow Integer Overflow (Sec 1,2,3)

Integer Overflow

```
void bad_function(char *input)
{
      char dest_buffer[32];
      char input_len = strlen(input);   // Range? [0, 255]
                                              Or [-128, 127]
            -100
      if (input_len < 32)
      {
            strcpy(dest_buffer, input);
            printf("The first command line argument is %s.\n", dest_buffer);
      }
      else
      {
            printf("Error - input is too long for buffer.\n");
      }
```

OWASP – Integer Overflow Integer Overflow (Sec1,2,3)

Implicit Type Conversions

Operation	Operand Values	Overflow / Underflow?
SUB – 32 Bit signed	-2^31 – 2^31	Underflow
ADD – 32 Bit Unsigned	2^32 + 2^32	Overflow
MUL – 32 Bit Unsigned	2^20 * 2^20	Overflow

The actual range of values for each type is compiler / machine –dependent. When operands of different types widths are used, C99 standard **automatically (implicitly) converts types**.

Type Promotions:
bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

Signed / Unsigned Coercions:
Defined by the C standard

https://stackoverflow.com/questions/50605/signed-to-unsigned-conversion-in-c-is-it-always-safe
https://www.geeksforgeeks.org/type-conversion-c/

Why doesn't hardware complain?

- Hardware: Arithmetic doesn't distinguish signed and unsigned integers --- works for both

Unsigned $\sum_{i=0}^{n-1} x_i 2^i$

$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$ Signed

$i=0$

An advantage of two's complement is that signed and unsigned addition can be performed using the same operation. The same is true for subtraction and multiplication. Historically, this was advantageous because fewer instructions needed to be implemented. Also, unlike the *ones' complement* and *sign-magnitude* representations, two's complement has only one representation for zero. A drawback of two's complement is that its range, $-2^{n-1} \ldots 2^{n-1} - 1$, is asymmetric. Thus, there is a representable value, -2^{n-1}, that does not have a representable additive inverse—a fact that programmers can and do forget.

When an n-bit addition or subtraction operation on unsigned or two's complement integers overflows, the result "wraps around," effectively subtracting 2^n from, or adding 2^n to, the true mathematical result. Equivalently, the result can be considered to occupy $n+1$ bits; the lower n bits are placed into the result register and the highest-order bit is placed into the processor's carry flag.

TOSEM'15 paper – section 1,2,3 are relevant

Integer Overflow (TOSEM'15)

Why doesn't hardware complain?

- In C99 on x64, conversions reinterpret the bit representation!

3.2. The Usual Arithmetic Conversions

Most integer operators in C/C++ require that both operands have the same type and, moreover, that this type is not narrower than an `int`. The collection of rules that accomplishes this is called *the usual arithmetic conversions*. The full set of rules encompasses both floating point and integer values; here we will discuss only the integer rules. First, both operands are *promoted*:

> If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

If the promoted operands have the same type, the usual arithmetic conversions are finished. If the operands have different types, but either both are signed or both are unsigned, the narrower operand is converted to the type of the wider one.

If the operands have different types and one is signed and the other is unsigned, then the situation becomes slightly more involved. If the unsigned operand is narrower than the signed operand, and if the type of the signed operand can represent all values of the type of the unsigned operand, then the unsigned operand is converted to signed. Otherwise, the signed operand is converted to unsigned.

These rules can interact to produce counterintuitive results. Consider this function:

```
int compare (void) {
  long a = -1;
  unsigned b = 1;
  return a > b;
}
```

For a C/C++ implementation that defines `long` to be wider than `unsigned`, such as GCC for x86-64, this function returns zero. However, for an implementation that defines `long` and `unsigned` to have the same width, such as GCC for x86, this function returns one. The issue is that on x86-64, the comparison is between two signed integers, whereas on x86, the comparison is between two unsigned integers, one of which is very large. Some compilers are capable of warning about code like this.

C/C++ don't define exactly for all cases!

Where they do define, the behavior is implementation (size) specific to compilers, and varies by architecture!

Int Overflow (TOSEM)

Real-life Incidents Due to Integer Bugs

The New York Times Magazine

Little Bug, Big Bang

By James Gleick

Dec. 1, 1996

IT TOOK THE European Space Agency 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space.

CVE-2018-10299 Detail

Memory Safe ≠ Type Safe

Current Description

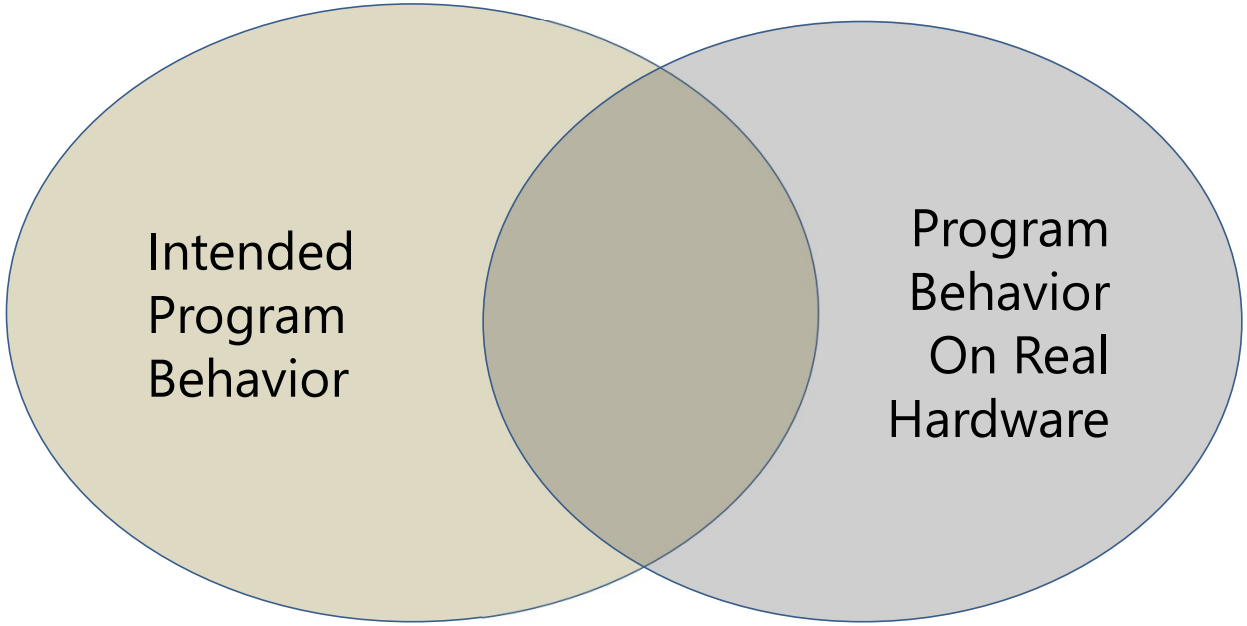
An integer overflow in the batchTransfer function of a smart contract implementation for Beauty Ecosystem Coin (BEC), the Ethereum ERC20 token used in the Beauty Chain economic system, allows attackers to accomplish an unauthorized increase of digital assets by providing two _receivers arguments in conjunction with a large _value argument, as exploited in the wild in April 2018, aka the "batchOverflow" issue.

Source: MITRE

− Hide Analysis Description

Summary & Key Takeaways

- Memory Errors / Vulnerabilities
  - Spatial & Temporal
- Worst case: Can give attackers capability to read / write any value anywhere in memory
- Hardware does not give memory safety



End of Segment!