RECITATION 1

Ayşe Sayın

Istanbul Technical University

sayinays@itu.edu.tr

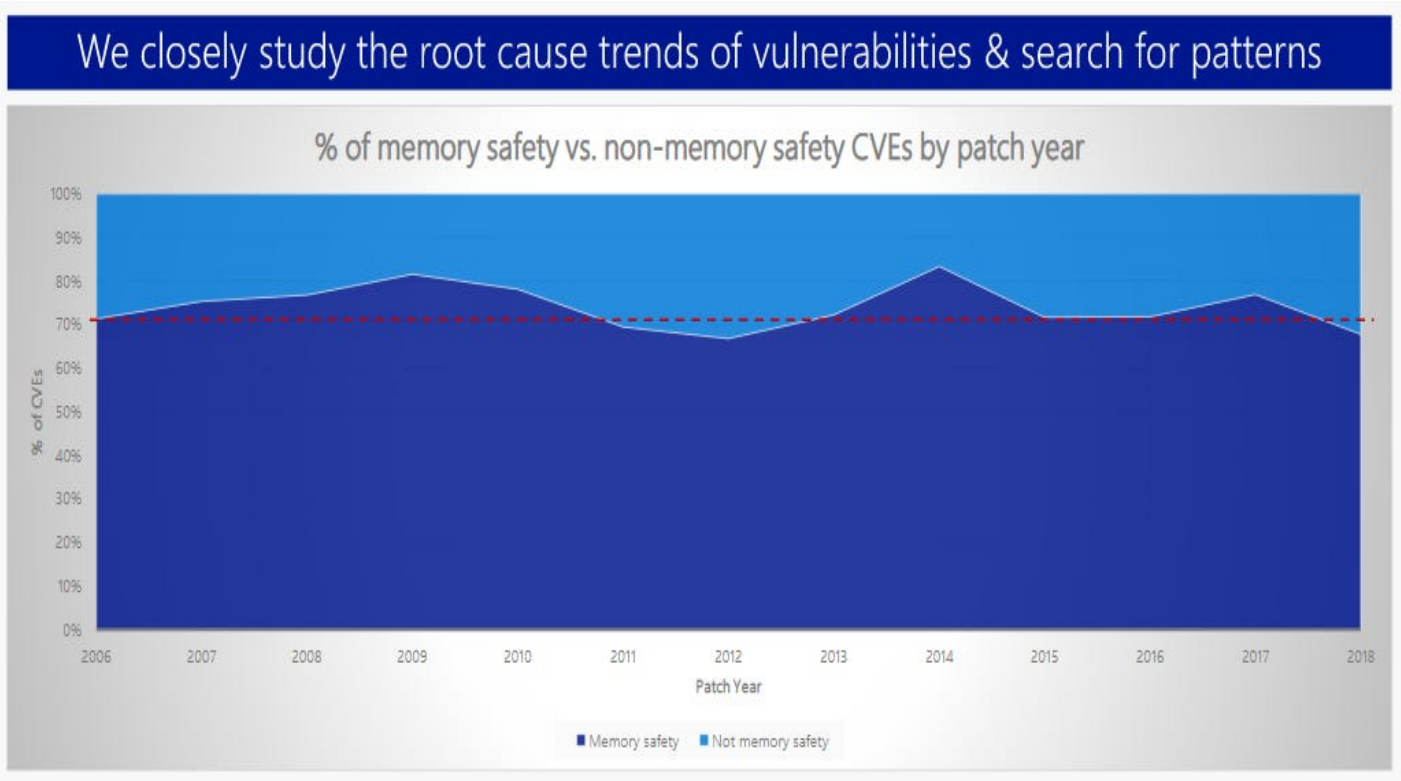
March 22, 2022

Overview

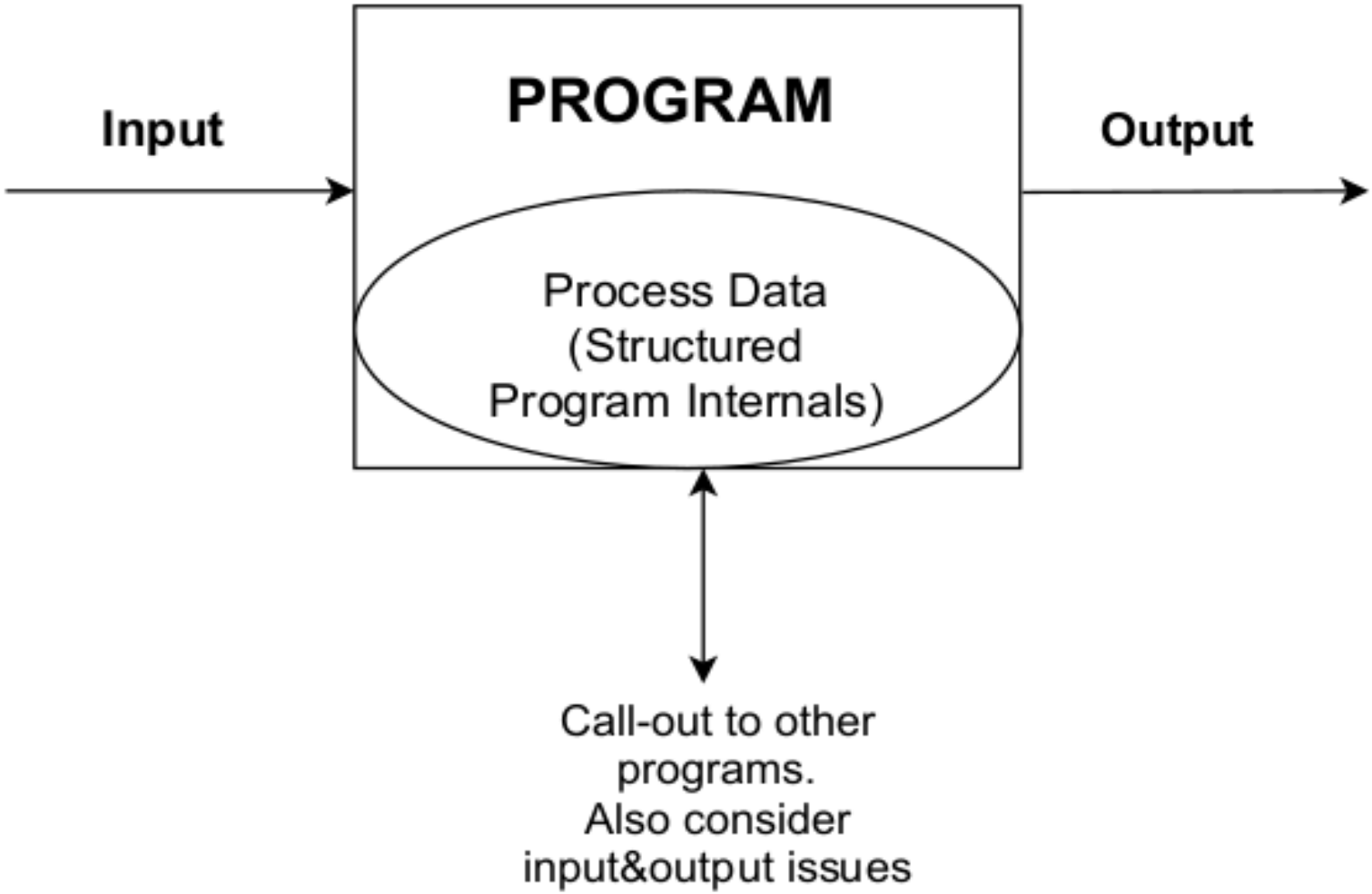
1. Buffer Overflow
   - Memory safety exists
   - Abstract view of a program
   - What is a buffer overflow?
   - A buffer overflow example
   - Address space of a process
   - Abstract data type: Stack
   - Calling a procedure
   - Function prologue
   - Stack: after prologue
   - What happens if we overwrite?
   - Inserting code in the buffer overflow attack
   - One possible result after attack

Memory safety exists

- %70 of Microsoft vulnerabilities 2006-2018 memory safety issues
  - Accesses system memory in a way that exceeds its allocated size & memory addresses (for example, a buffer overflow)
  - Problem occurs if programming language/mode isn't memory-safe



Abstract view of a program



What is a buffer overflow?

Buffer overflow occurs when:

- Fixed-length data buffer
- At least one value is written outside the buffer's boundaries
- Can occur when reading input or later processing the data

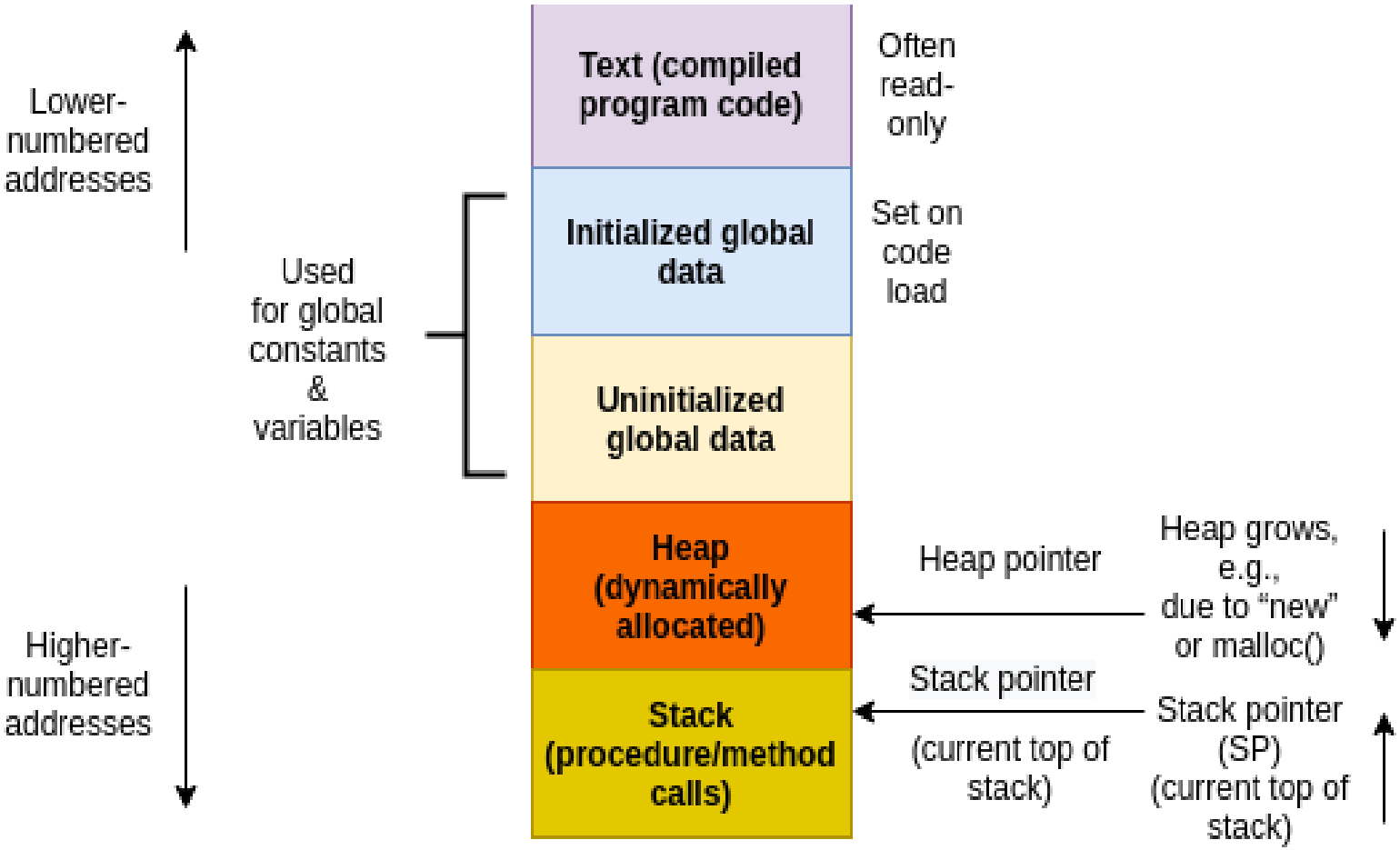
A buffer overflow example

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    // Only 10 bytes for command including termination char
    char command[10];

    printf("Your command?\n");

    // gets provides no protection against buffer overflow
    gets(command);
    printf("Your command was: %s\n", command);
}
```

Address space of a process



Abstract data type: Stack

- Memory area set aside to implement calls to procedure/function/method/subroutine
- Stack is used to control flow
  1. When you call a procedure, where it *came from* is *pushed* on stack
  2. When a procedure returns, the *where the procedure came from* is *popped* from stack

Abstract data type: Stack

- Stack is used for other data in many cases
  1. Parameters passed to procedures
  2. Procedure local variables
  3. Return values from procedure

Calling a procedure

Given this C program:

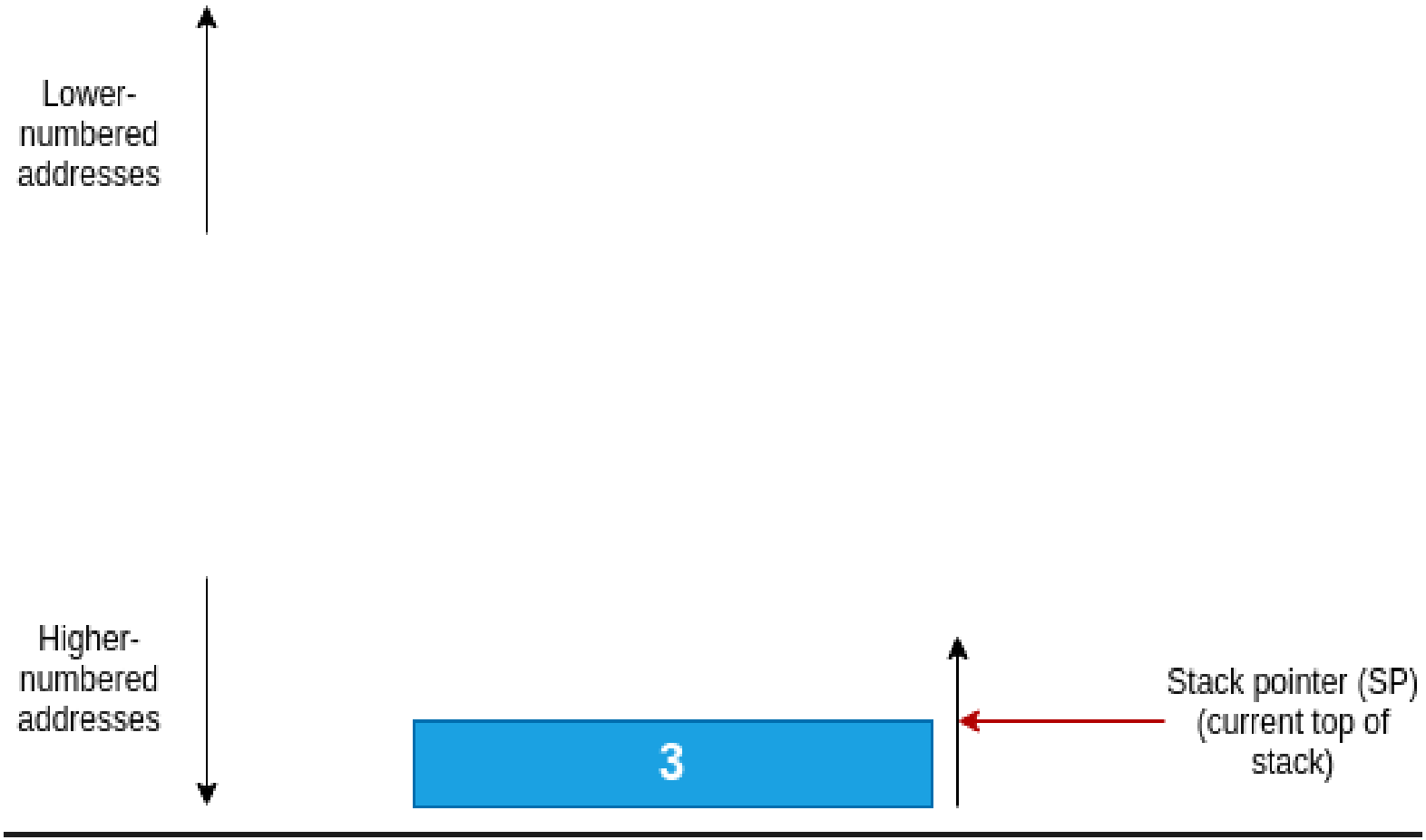
```
void main(){
    f(1, 2, 3);
}
```

Calling a procedure

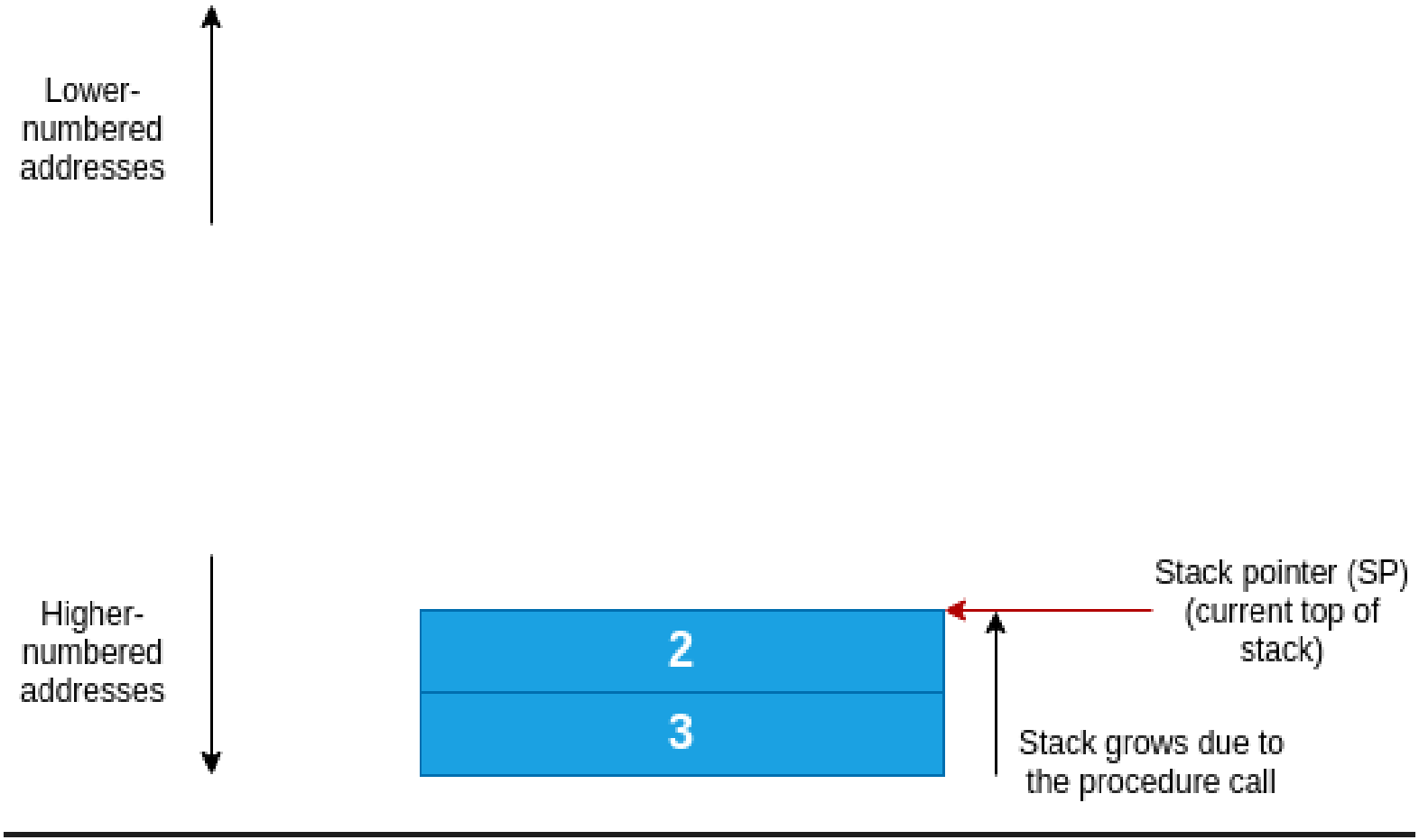
The invocation of f() might generate assembly:

```
pushl $3 ; constant 3
pushl $2 ; Most C compilers push in reverse order
pushl $1
call f
```

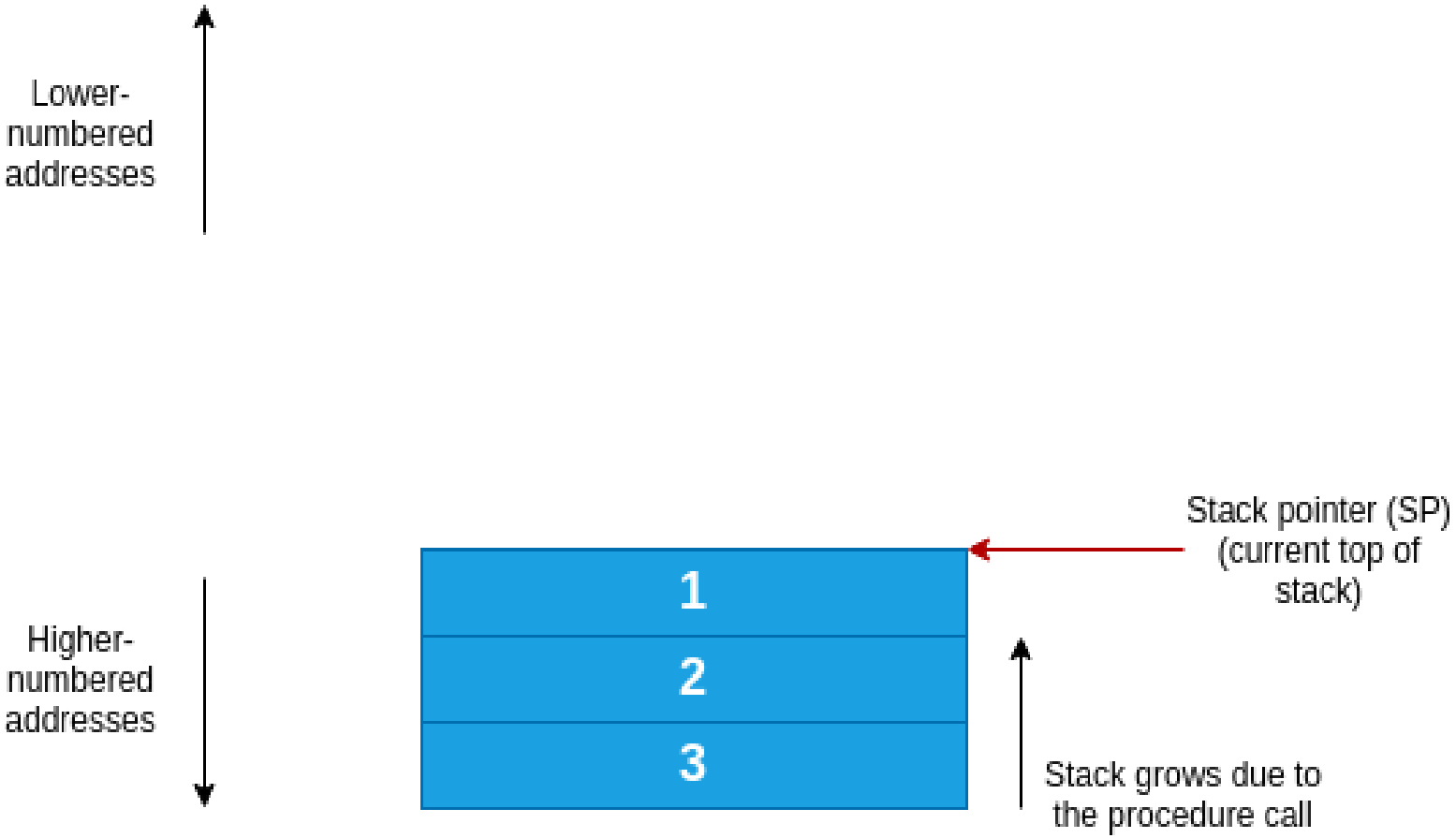
Stack: after calling a procedure



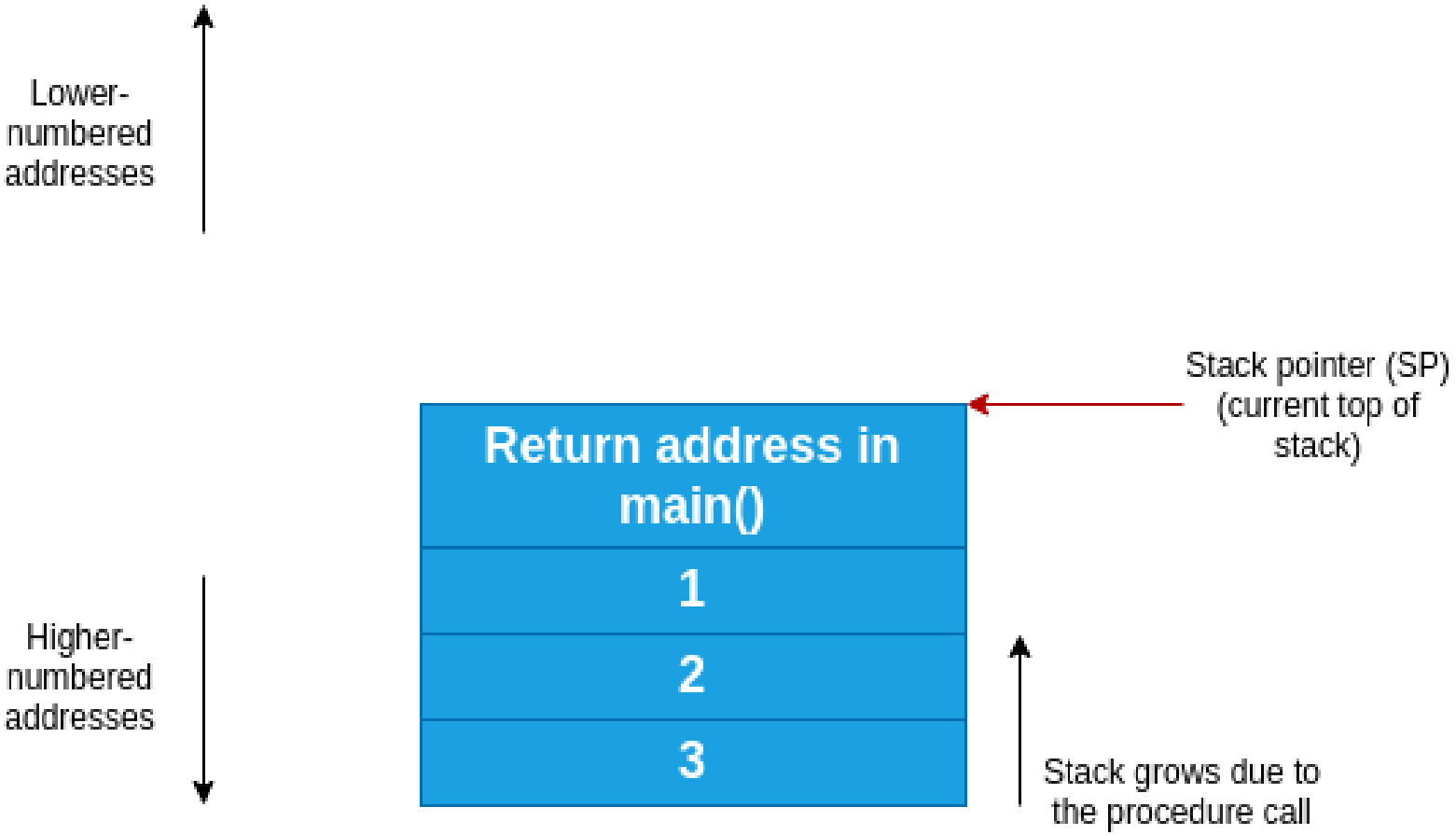
Stack: after calling a procedure



Stack: after calling a procedure



Stack: after calling a procedure



Function prologue

Imagine f() has local variables, e.g. in C:

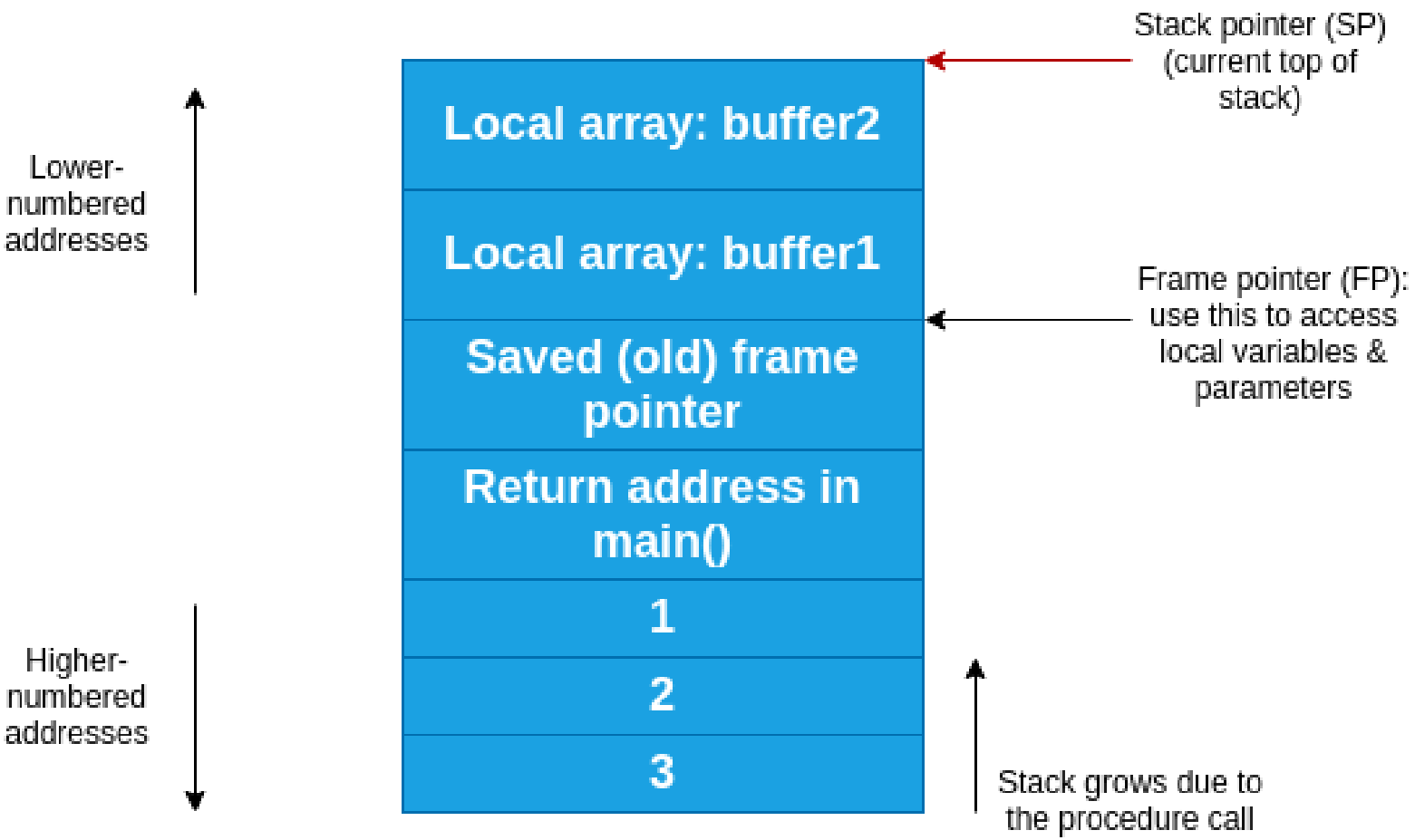
```
void f(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
    strcpy(buffer2, "This is a very long string!!!!!!!");
}
```

Function prologue

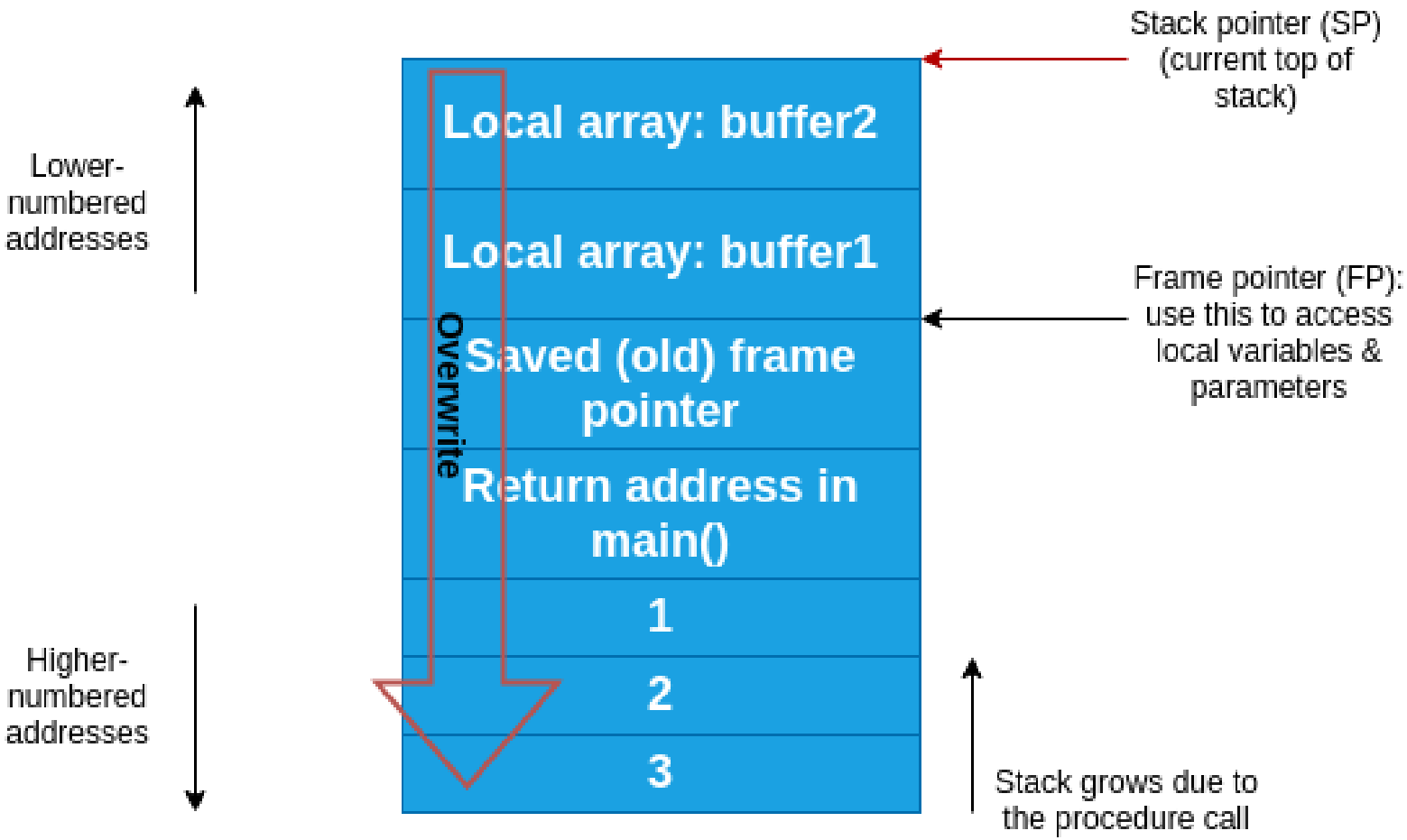
Typical x86-32 assembly on entry of f():

```
pushl %ebp ; Push old frame pointer (FP)
movl %esp,%ebp ; New FP is old SP
subl $20,%esp ; New SP is after local vars
; "$20" is calculated to be >= local var space
```

Stack: after prologue



Stack: overflowing



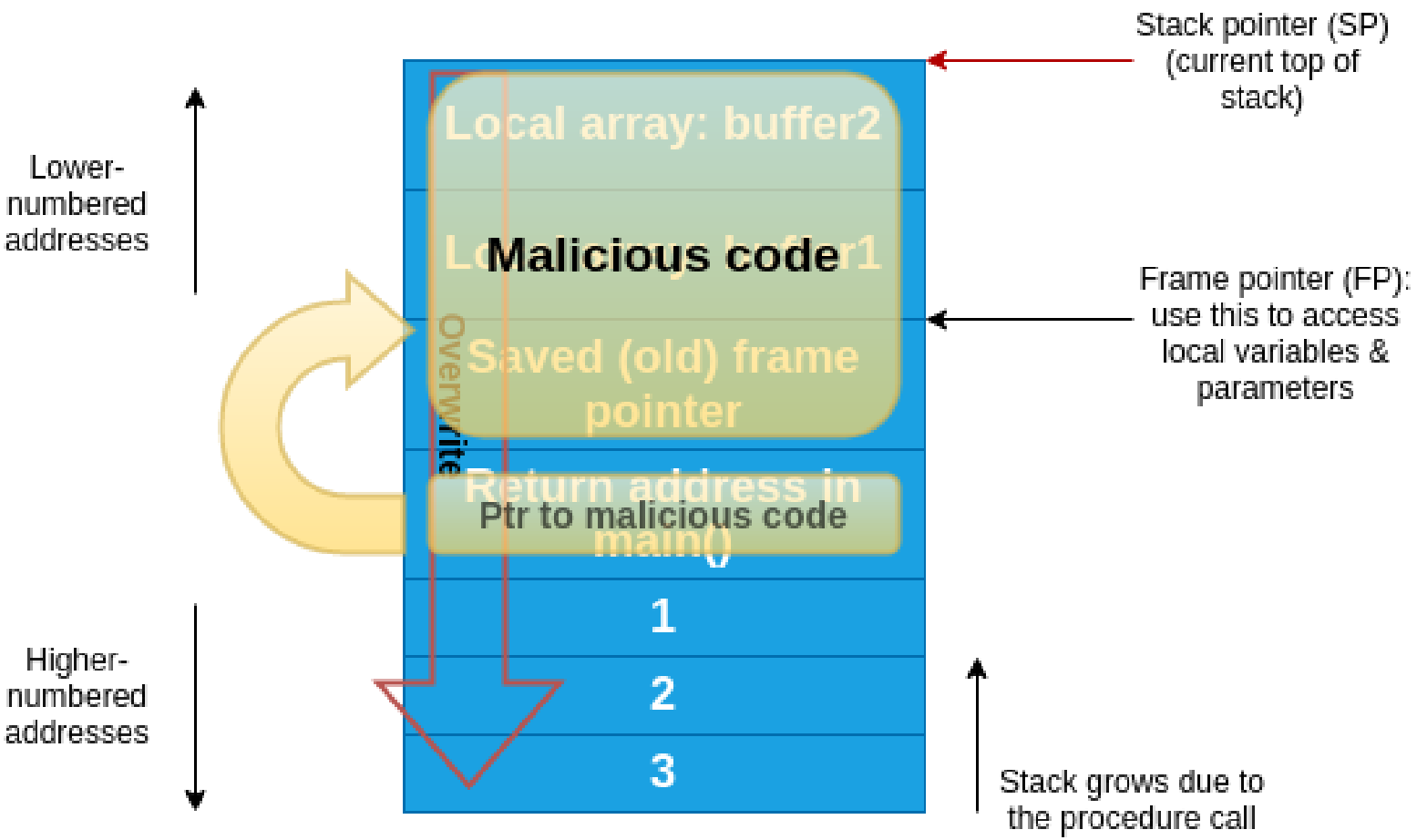
What happens if we overwrite?

- Overwrite higher addresses
- In our example, by using *buffer2* we can overwrite:
  1. Local values (buffer1)
  2. Saved frame pointer
  3. Return value (changing what we return to)
  4. Parameters to function
  5. Previous frames

Inserting code in the buffer overflow attack

- Send data that is too large, or will create overlarge data
- Attacker can modify return value to point to something that the attacker wants us to run (maybe with different parameters, too)

One possible result after attack



Smashing elsewhere

- *Heap* contains dynamically-allocated data
- *Data* contains global data
- If attacker can overwrite beyond buffer, can control other values (e.g., stored afterwards)

References

Dr. David A. Wheeler

Secure Software Design and Programming: Lecture 3