# COMPUTER SECURITY

## Programs and Programming

## PROGRAMS AND PROGRAMMING

## Objectives for today

- Learn about memory organization, buffer overflows, and relevant countermeasures
- Common programming bugs, such as off-by-one errors, race conditions, and incomplete mediation
- Survey of past malware and malware capabilities
- Virus detection
- Tips for programmers on writing code for security

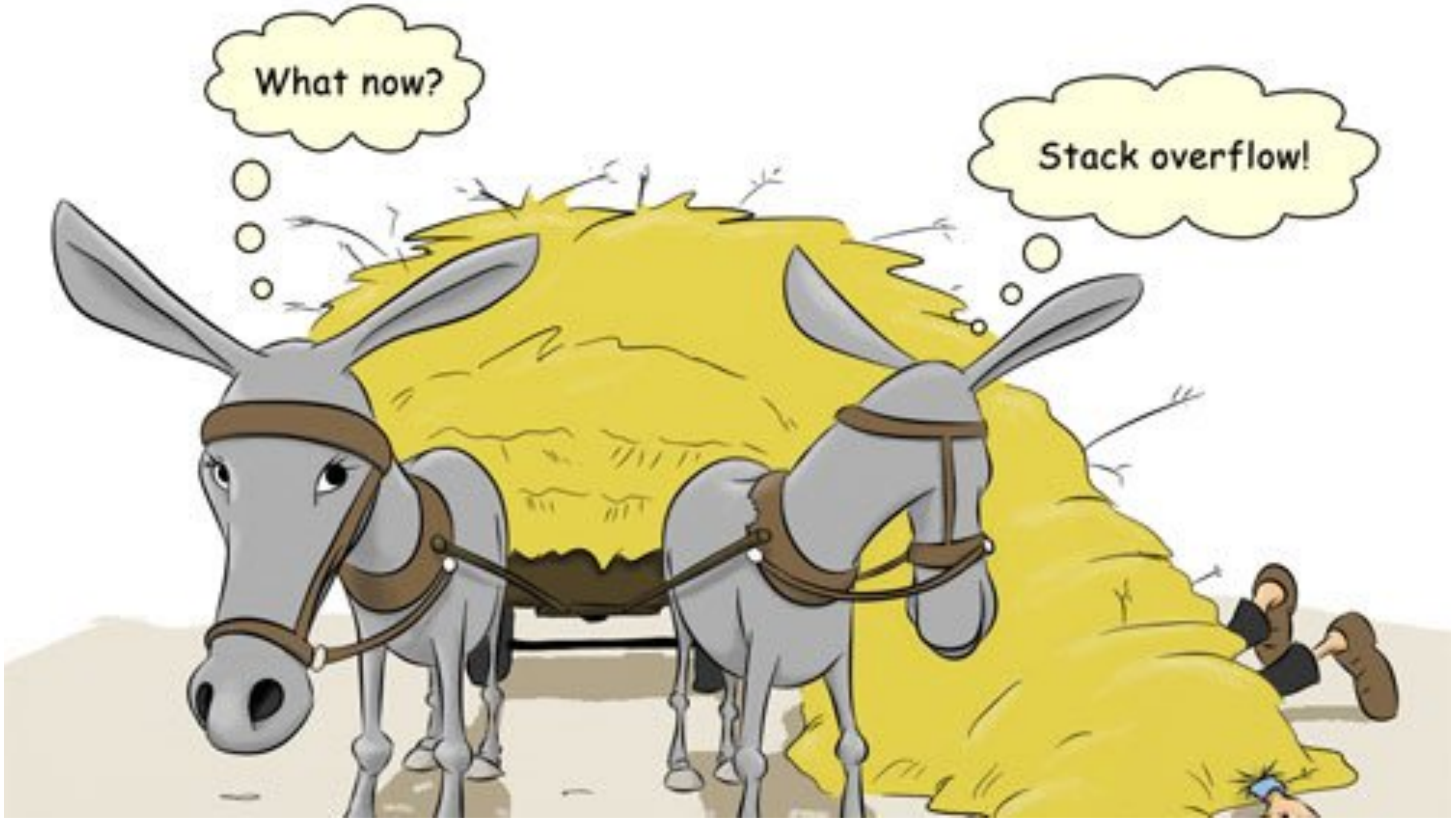
## Software Security

- Focuses on secure design and implementation of software
  - Using different languages, tools, methods, etc.
- Tends to focus on code
- In contrast, anti-viruses and firewalls treat code as blackbox:
  - build “walls” around software
  - Attackers can bypass these defenses

## Software Vulnerabilities

- A weakness in the program
  - May be explored by an attacker
    - Causing system to behave differently then expected
- Attacks may be based on the operating system and programming language
- Goals include manipulating the computer’s memory or control program’s execution

## Stack Buffer Overflow



https://www.linkedin.com/pulse/buffer-overflow-exploits-protection-mechanisms-roman-postanciuc/

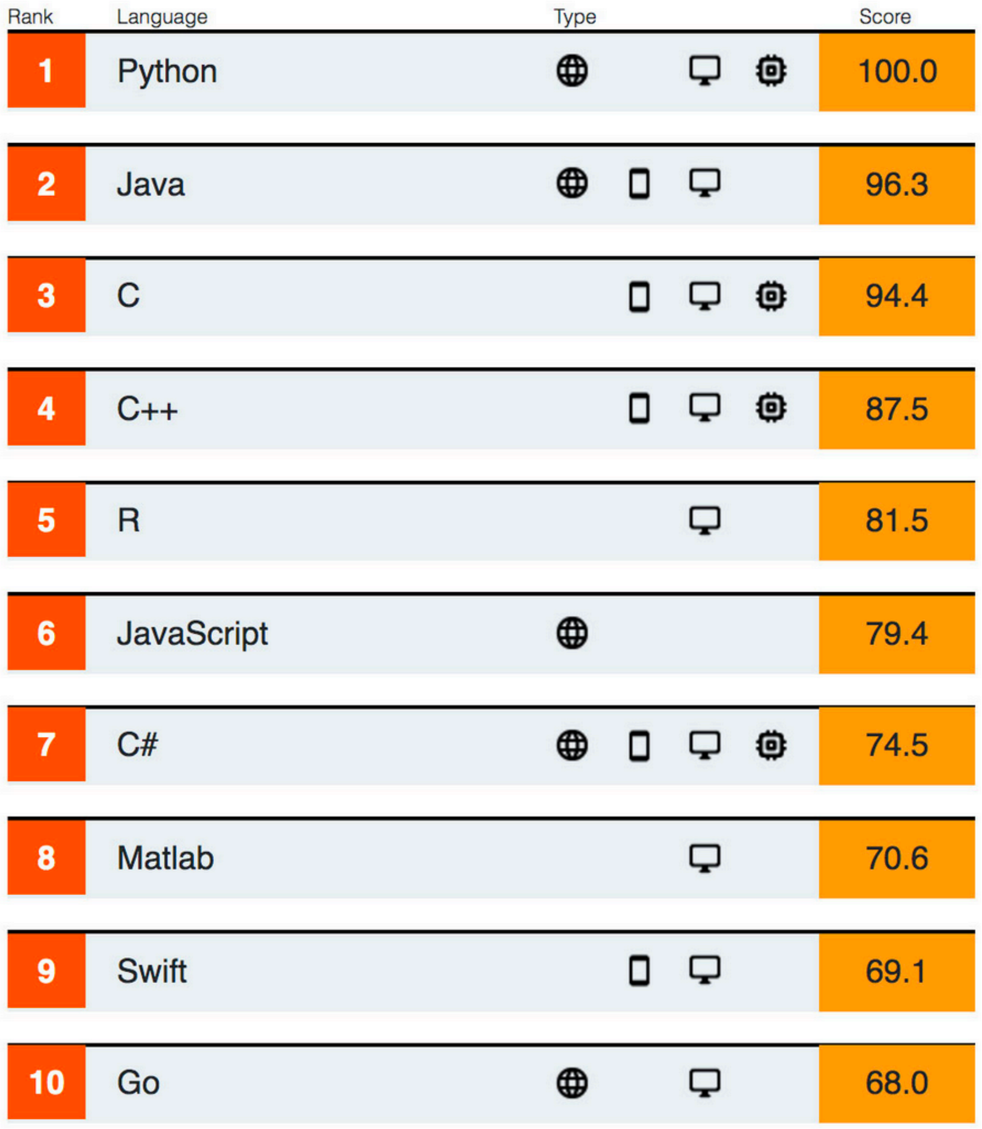
## Buffer Overflow

- A bug that affects low-level code
  - Typically in C and C++
- Has significant security implications
- Normally an attack with this bug will simply crash
- An attacker can cause much worse results:
  - Steal private information
  - Corrupt variables information
  - Run code of an attacker’s choice

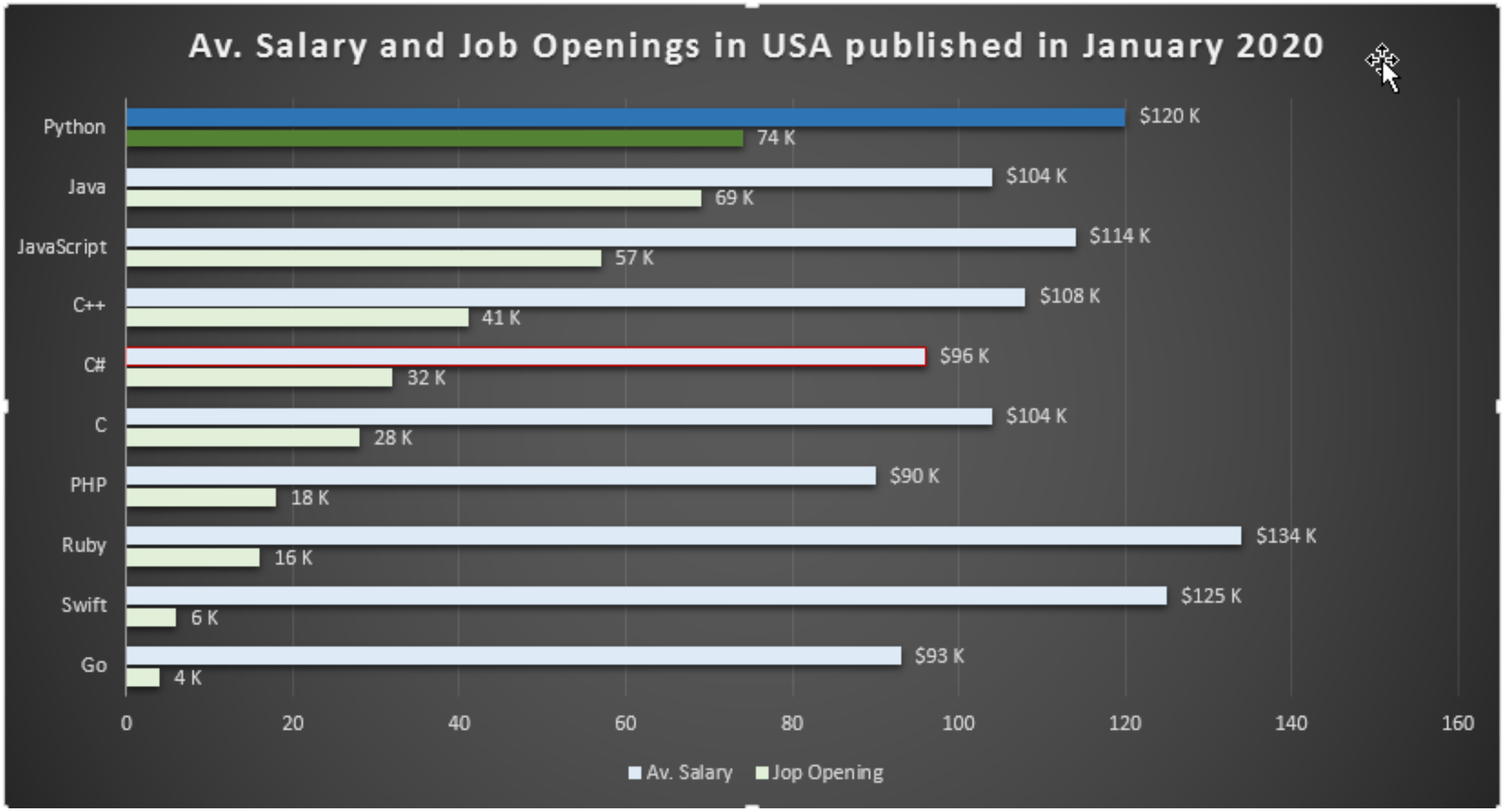
## Why examine buffer overflows?

- Still occur today
- Many bugs developed to take advantage of them
- Share common features with other bugs
  - With many defenses developed against them

# The Top Programming Languages 2019 (from IEEE Spectrum)



| Rank | Language | Type | Score |
|---|---|---|---|
| 1 | Python | 🌐 🖥 ⚙ | 100.0 |
| 2 | Java | 🌐 📱 🖥 | 96.3 |
| 3 | C | 📱 🖥 ⚙ | 94.4 |
| 4 | C++ | 📱 🖥 ⚙ | 87.5 |
| 5 | R | 🖥 | 81.5 |
| 6 | JavaScript | 🌐 | 79.4 |
| 7 | C# | 🌐 📱 🖥 ⚙ | 74.5 |
| 8 | Matlab | 🖥 | 70.6 |
| 9 | Swift | 📱 🖥 | 69.1 |
| 10 | Go | 🌐 🖥 | 68.0 |

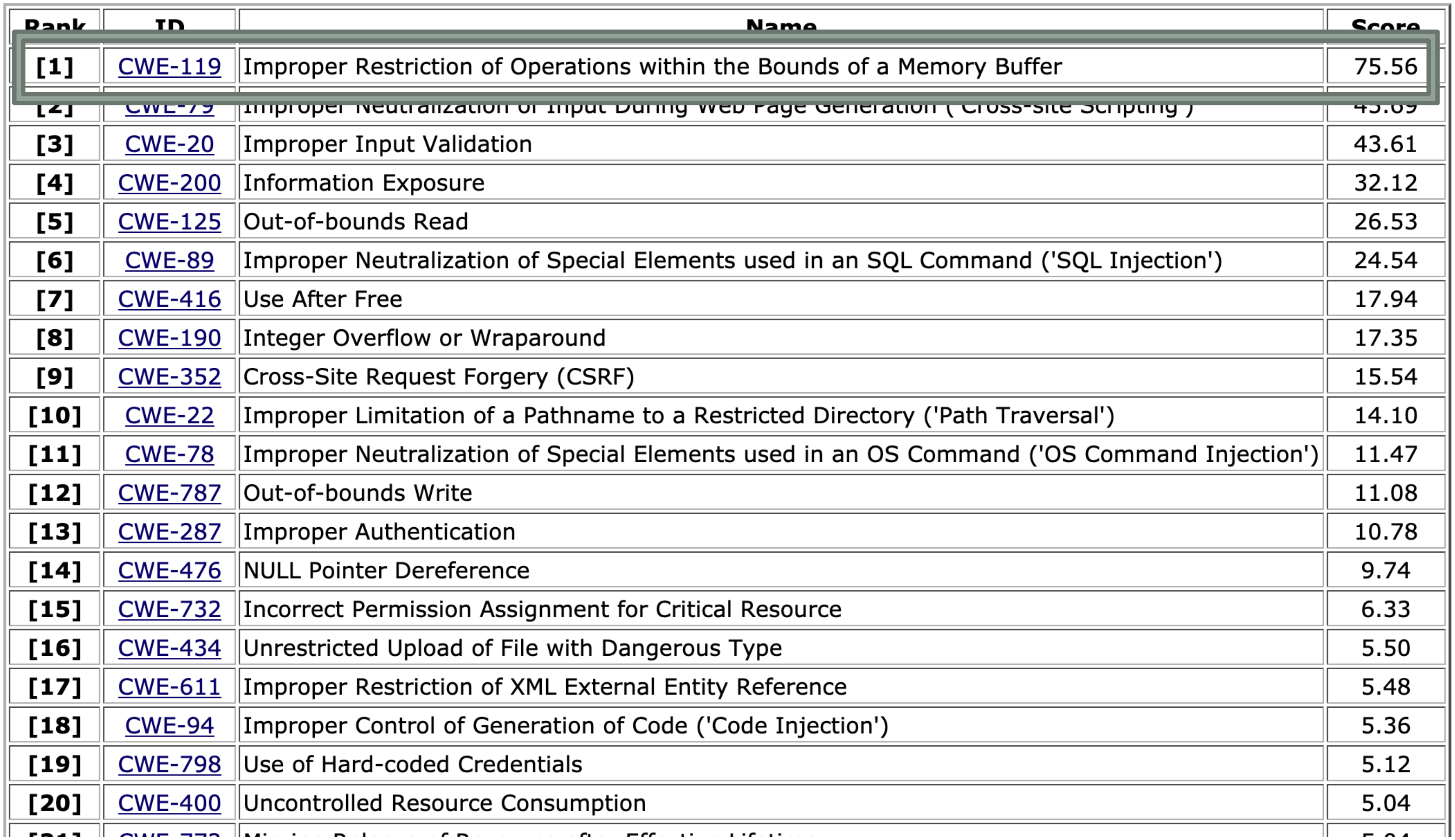


- https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020-4462eb7d8d3e

## Critical Systems Written in C/C++

- Most OS kernals and utilities
  - X windows server, shell, etc.
- Many high-performance servers
  - Microsoft SQL, Mysql, Apache httpd, etc.
- Embedded systems
  - Cars, Mars rover, etc.
- A successful attack on such systems has tremendous consequences!

# 2019 CWE Top 25 Software Errors



| Rank | ID | Name | Score |
|---|---|---|---|
| [1] | CWE-119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 75.56 |
| [2] | CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') | 45.69 |
| [3] | CWE-20 | Improper Input Validation | 43.61 |
| [4] | CWE-200 | Information Exposure | 32.12 |
| [5] | CWE-125 | Out-of-bounds Read | 26.53 |
| [6] | CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | 24.54 |
| [7] | CWE-416 | Use After Free | 17.94 |
| [8] | CWE-190 | Integer Overflow or Wraparound | 17.35 |
| [9] | CWE-352 | Cross-Site Request Forgery (CSRF) | 15.54 |
| [10] | CWE-22 | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') | 14.10 |
| [11] | CWE-78 | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') | 11.47 |
| [12] | CWE-787 | Out-of-bounds Write | 11.08 |
| [13] | CWE-287 | Improper Authentication | 10.78 |
| [14] | CWE-476 | NULL Pointer Dereference | 9.74 |
| [15] | CWE-732 | Incorrect Permission Assignment for Critical Resource | 6.33 |
| [16] | CWE-434 | Unrestricted Upload of File with Dangerous Type | 5.50 |
| [17] | CWE-611 | Improper Restriction of XML External Entity Reference | 5.48 |
| [18] | CWE-94 | Improper Control of Generation of Code ('Code Injection') | 5.36 |
| [19] | CWE-798 | Use of Hard-coded Credentials | 5.12 |
| [20] | CWE-400 | Uncontrolled Resource Consumption | 5.04 |
| [21] | CWE-772 | Missing Release of Resource after Effective Lifetime | 5.04 |

https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

## Attacks that took advantage of buffer overflow

- Morris attack
    - Used buffer overflow vulnerability in fingerd and VAXes
- CodeRed attack
    - Exploited overflow in MS-ISS server
        - 300,000 machines infected in 14 hours
- SQL slammer
    - Exploited overflow in MS-SQL server
    - 75,000 machines infected in 10 minutes

## What we can do

- Understand how these attacks work
  - And how to defend against them
- Different aspects involved:
  - Compiler
  - OS
  - Computer architecture

## Buffer Overflow

- ***Buffer***: contiguous memory associated with a variable or field
  - Common in C/C++:
    - All strings are NULL-terminated arrays of characters

## Buffer Overflow

- ***Overflow***: put more characters into the buffer than it can hold
  - Characters “spill” beyond the buffer
    - Into other parts of the computer memory space
  - Most compilers assume program does not overflow
    - Do not check for it, will process such memory access beyond bounds

## Buffer Overflows

- Occur when data is written beyond the space allocated for it
  - such as a 10th byte in a 9-byte array
- A typical exploitable buffer overflow:
  - An attacker’s inputs are expected to go into regions of memory allocated for data
    - those inputs are instead allowed to overwrite memory holding executable code

## Buffer Overflows

- The trick for an attacker is:
  - finding buffer overflow opportunities
    - lead to overwritten memory being executed
  - finding the right code to input

## How Buffer Overflows Happen

```
char sample[10];

int i;

for (i=0; i<=9; i++)
   sample[i] = ‘A’;

sample[10] = ‘B’;
```

## Programming Errors

- **Buffer overflow** attack can cause crash
  - Input is longer than variable buffer, overwrites other data
  - Example: program in C:

```
char A[8] = "";
unsigned short B = 1979;
```

# Buffer Overflow (example)

Initially, A contains nothing but zero bytes, and B contains the number 1979

| variable name | A | | | | | | | | B | |
|---|---|---|---|---|---|---|---|---|---|---|
| value | [null string] | | | | | | | | 1979 | |
| hex value | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 07 | BB |

## Buffer Overflow (example)

- The program attempts to store the null-terminated string "excessive“

```
strcpy(A, "excessive");
```

- "excessive" is 9 characters long and encodes to 10 bytes including the null terminator
  - but A can take only 8 bytes
  - By failing to check the length of the string, it also overwrites the value of B:

## Buffer Overflow (example)

| variable name | A | | | | | | | | B | |
|---|---|---|---|---|---|---|---|---|---|---|
| value | 'e' | 'x' | 'c' | 'e' | 's' | 's' | 'i' | 'v' | 25856 | |
| hex | 65 | 78 | 63 | 65 | 73 | 73 | 69 | 76 | 65 | 00 |

## Buffer Overflow (cont.)

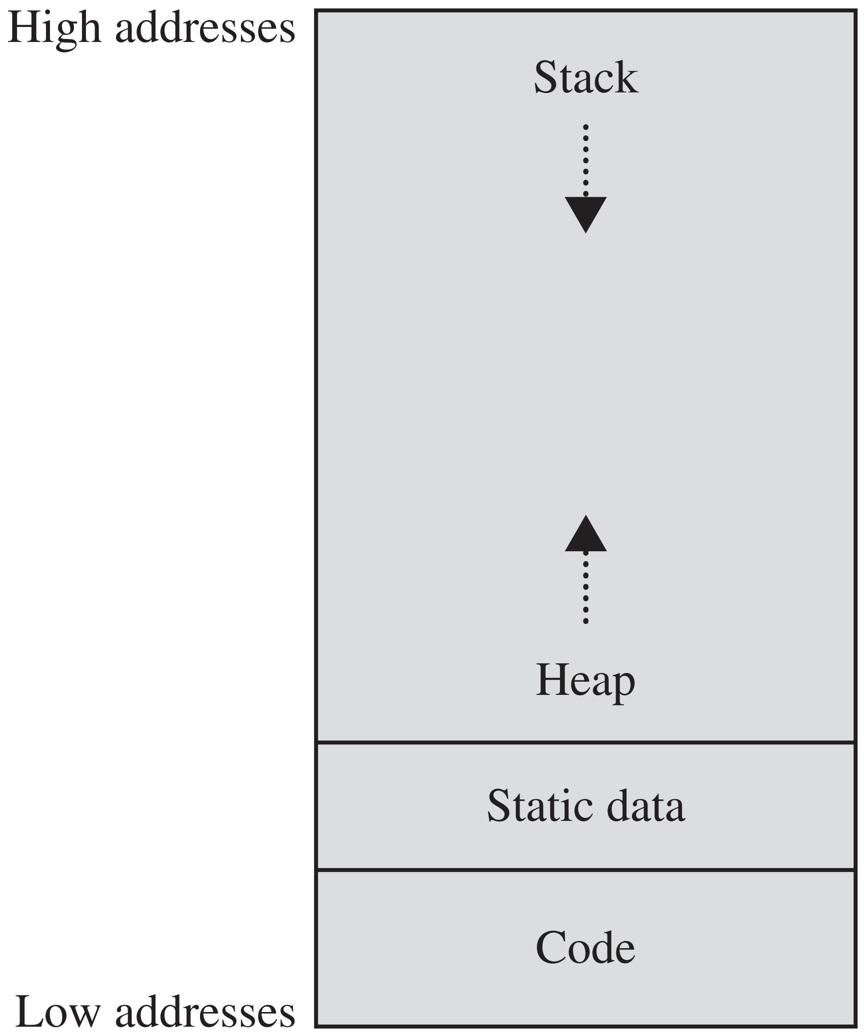
- How to prevent this?
- Replace call to strcpy with strncpy
  - strncpy takes the maximum capacity of A as an additional parameter
    - ensures that no more than this amount of data is written to A:

```
strncpy(A, "excessive", sizeof(A));
```

## Buffer Overflow

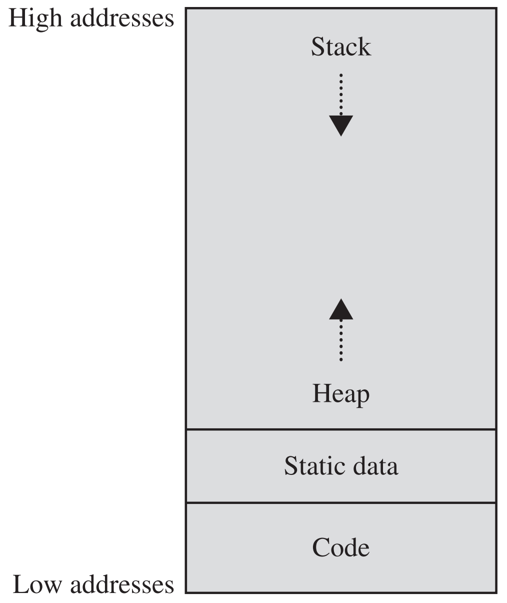
- Buffer Overflow Basics

# Memory Allocation

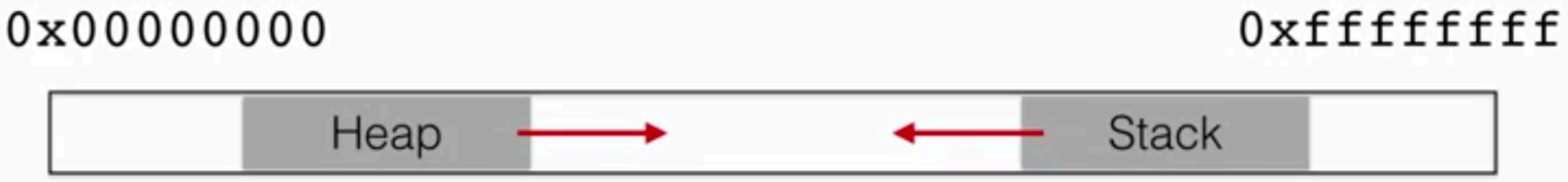


## Memory Allocation

- Memory organization:
  - Code and data separated
  - The heap grows up toward high addresses
  - The stack grows down from the high addresses.



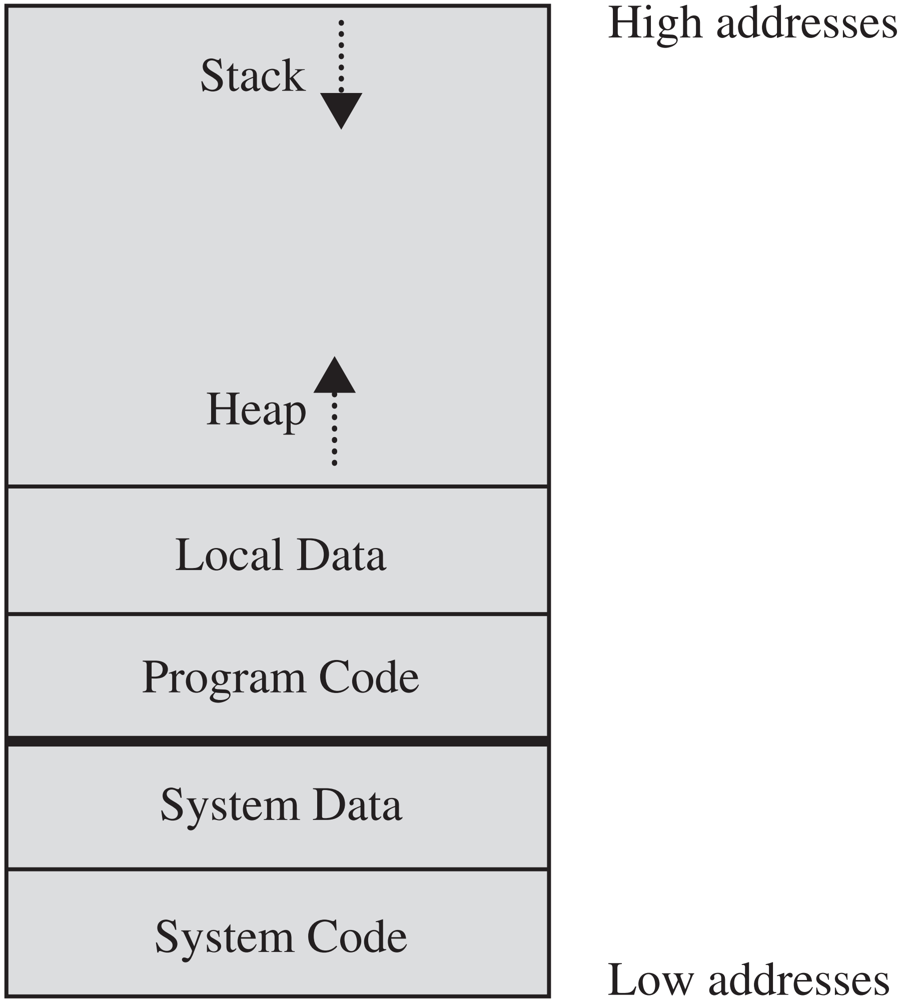
## Memory Allocation



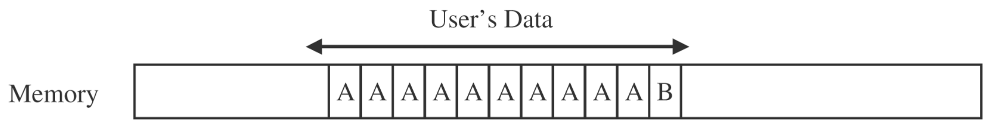
## Memory Organization

- An attack that takes place inside a given program can affect :
  - that program
  - the rest of the system
- Different memory parts hold different components:
  - system data/code
  - program code and its local data
- A malicious program may affect different parts of the system

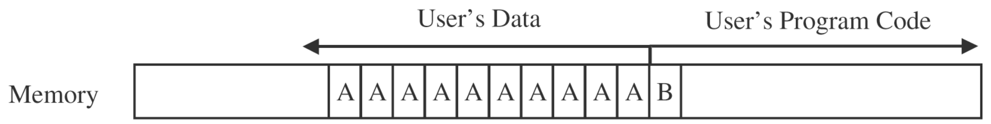
# Memory Organization



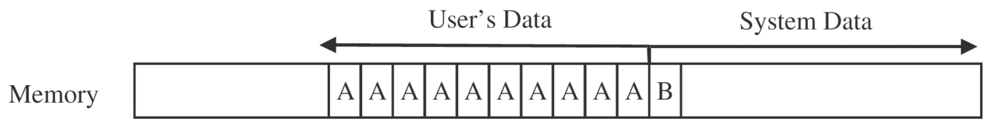
# Where a Buffer Can Overflow



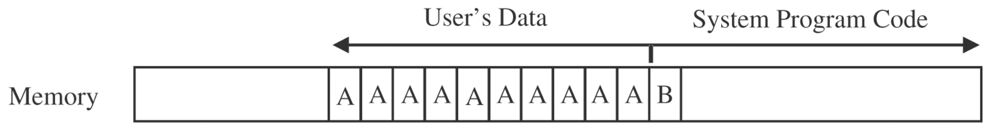
(a) Affects user's data



(b) Affects user's code

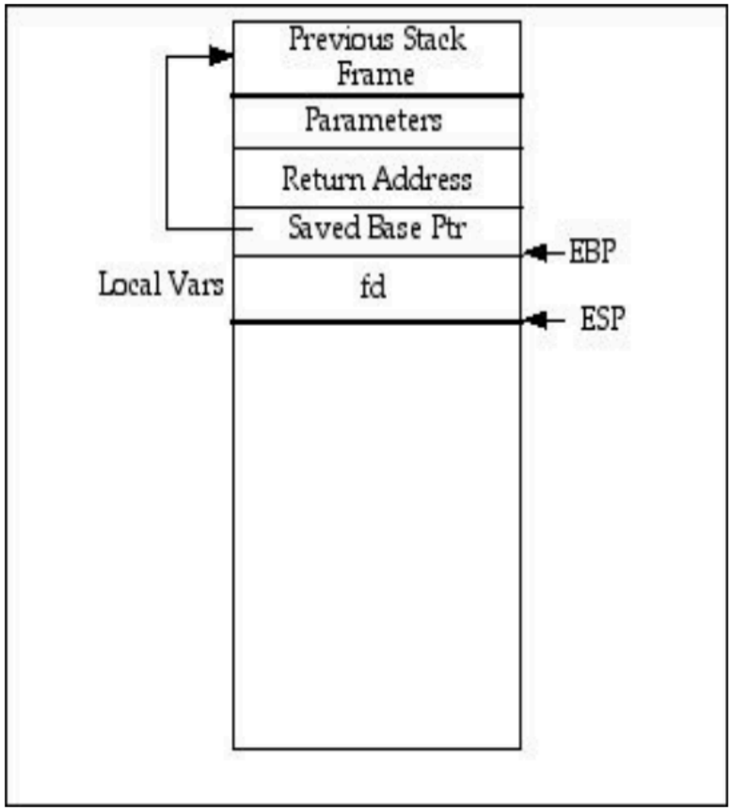


(c) Affects system data

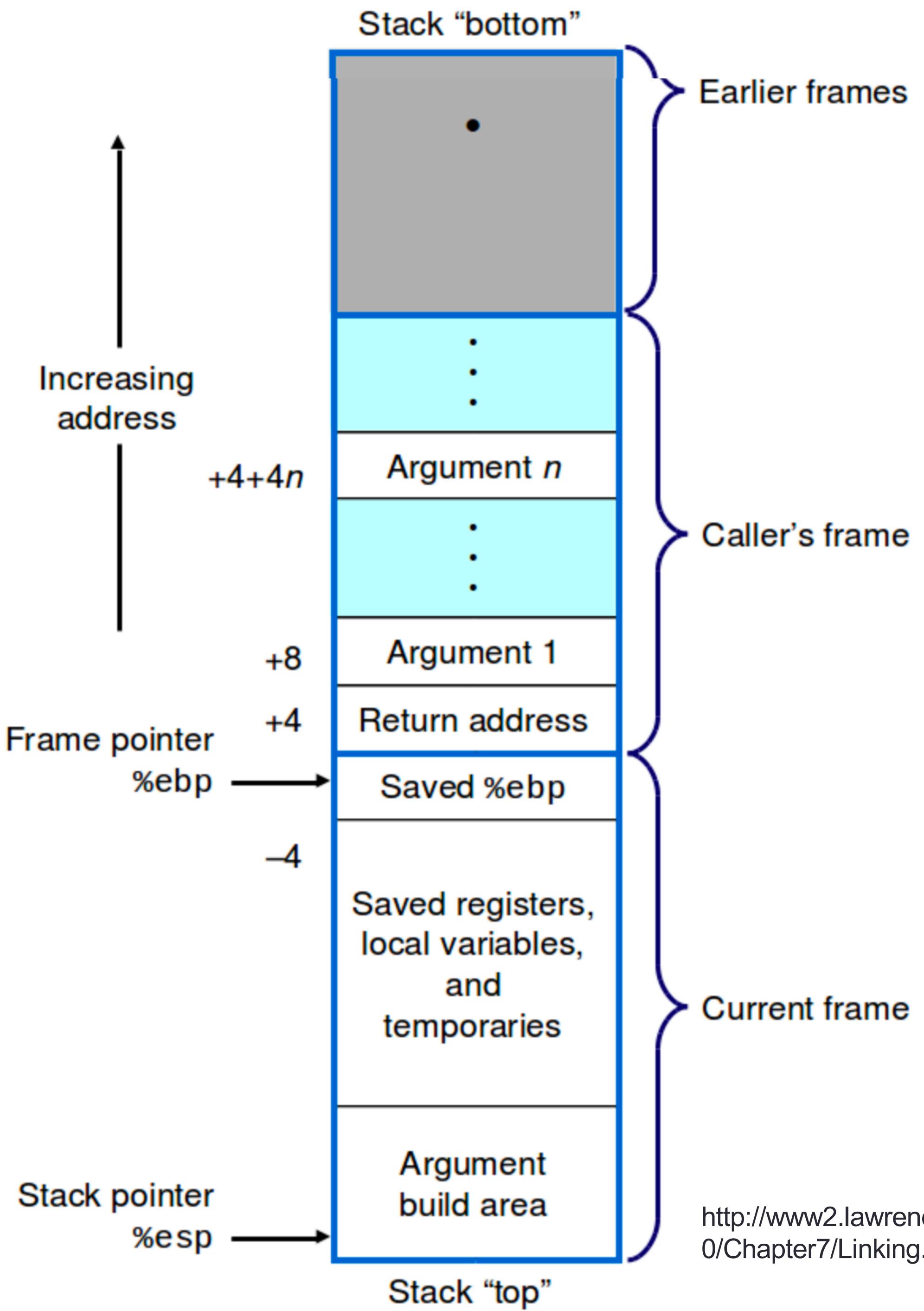


(d) Affects system code

# LINUX PROCESS MEMORY LAYOUT



http://post.queensu.ca/~trd/377/tut5/stack.html



http://www2.lawrence.edu/fast/GREGGJ/CMSC480/Chapter7/Linking.html

## LINUX PROCESS MEMORY LAYOUT

- Similar attacks also happen on other architectures
- EBP: “Frame pointer”: points to the start of the current call frame on the stack
- ESP: “Stack pointer”: points to the current stack
    - PUSH: Decrement the stack pointer and store something there
    - POP: Load something and increment the stack pointer

## Stack and Functions

- Calling functions:
  - Push arguments onto the stack
  - Push the return address
    - Where the control will return to at end of function
  - Jump to function’s address

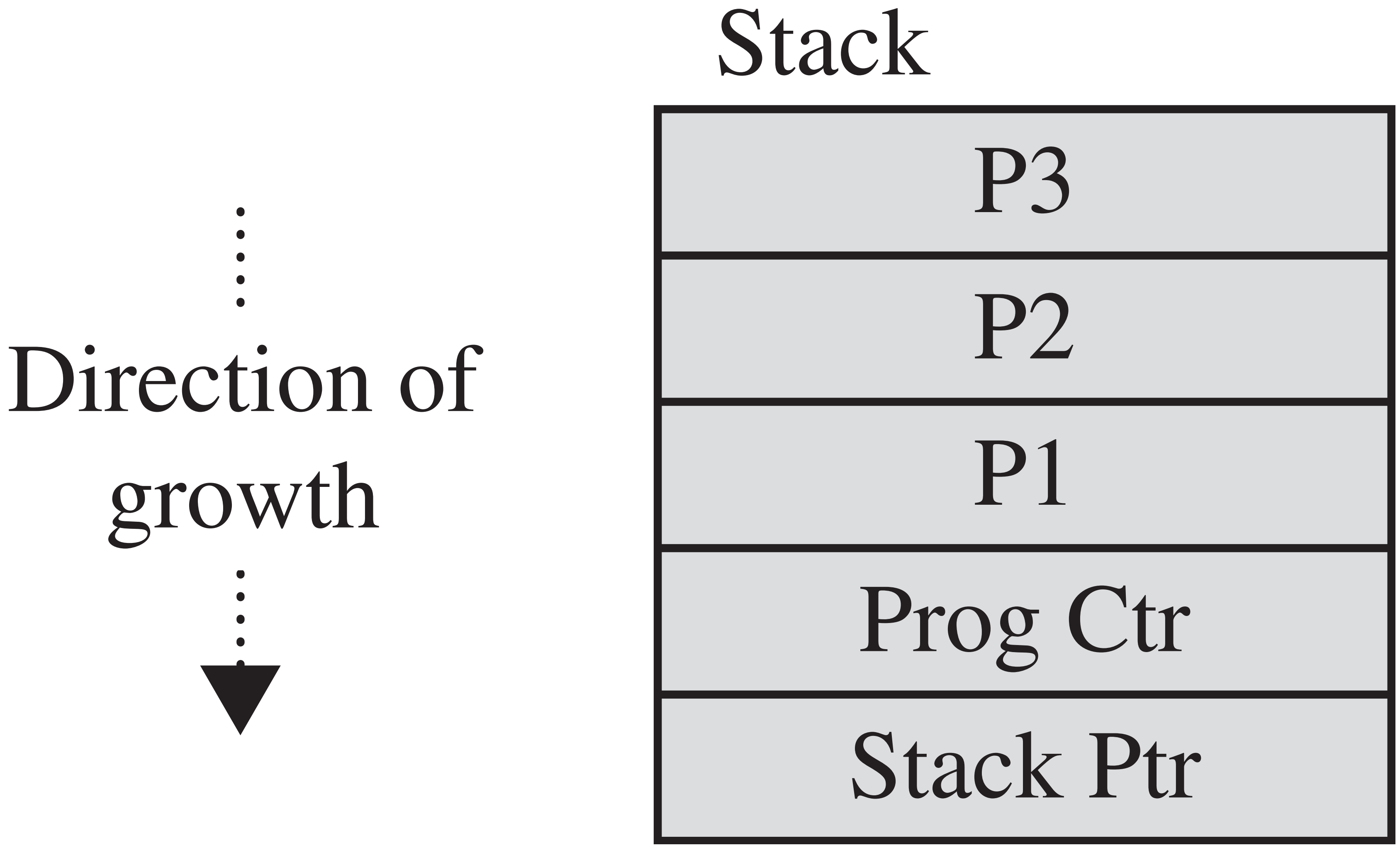
## Stack and Functions

- Called function:
  - Push the old frame pointer onto stack
  - Set frame pointer to end of stack
  - Push local variables onto stack
    - Overwriting buffers may overwrite the old frame pointer
  - Performs actions
  - Restores all variables

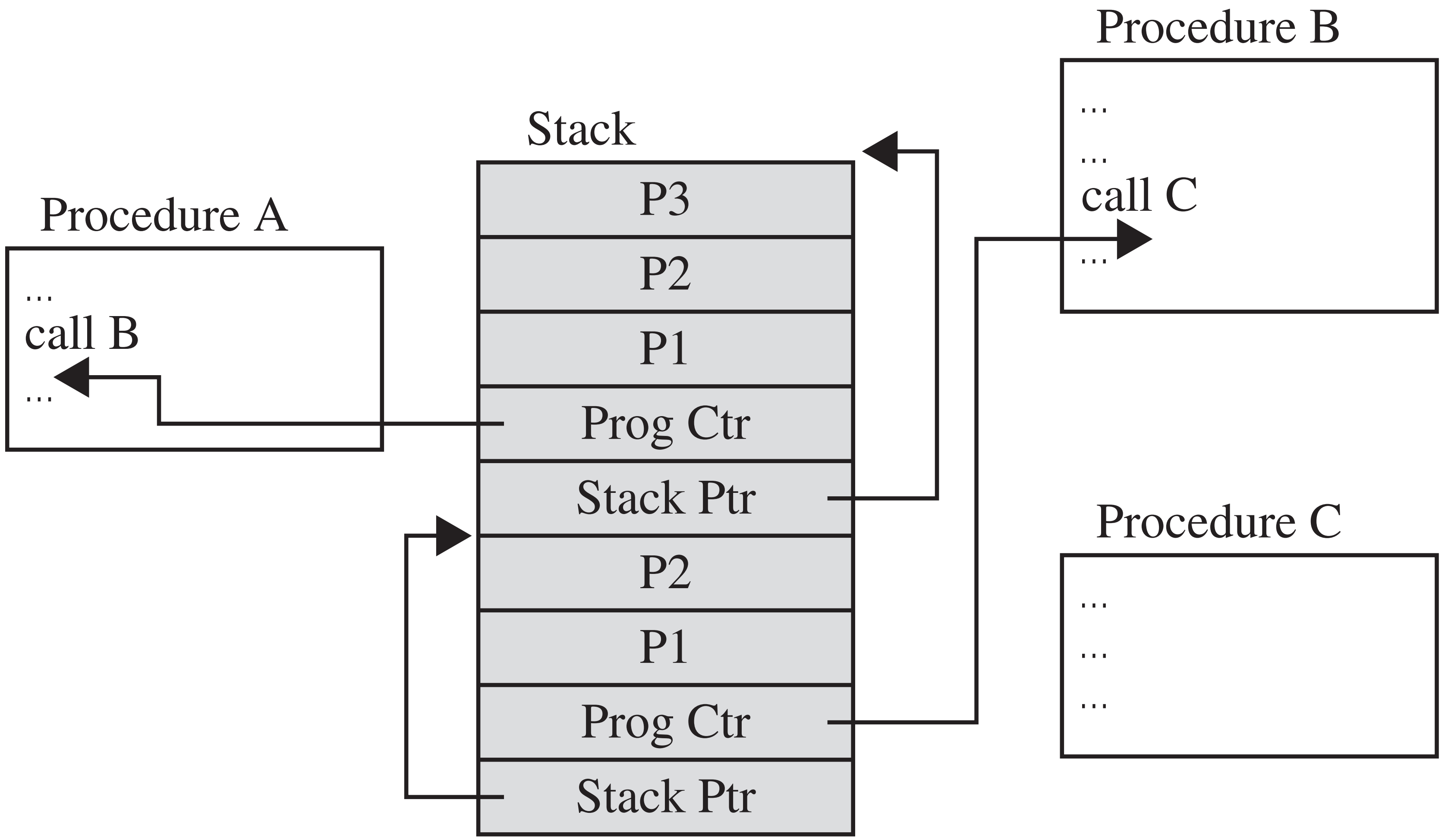
## Stack and Functions

- Returning function:
  - Reset the previous stack frame
  - Jump back to return address
    - If return address was overwritten in stack, program will go to an unknown location

## The Stack



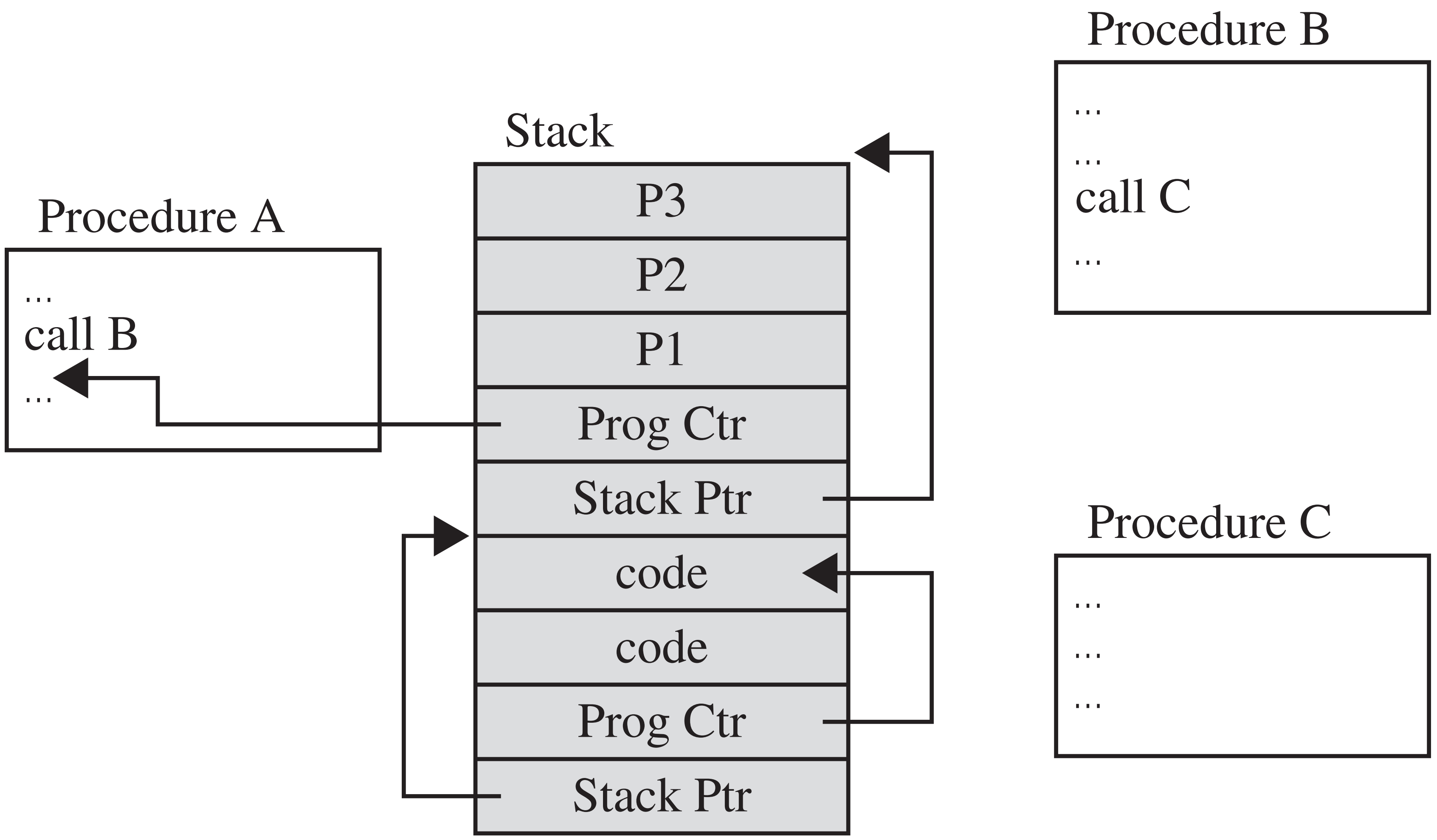
## The Stack after Procedure Calls



## The Stack after Procedure Calls

- When procedure A calls procedure B, procedure B gets added to the stack
  - along with a pointer back to procedure A
- When procedure B is finished running, it can get popped off the stack
  - procedure A will just continue executing where it left off.

# Compromised Stack



## Compromised Stack

- Program counter pointed to procedure B
- Changed to point at code that’s been placed on the stack
  - as a result of an overflow.

## Overwriting Memory for Execution

- Overwrite the program counter stored in the stack
- Overwrite part of the code in low memory, substituting new instructions
- Overwrite the program counter and data in the stack
  - so that the program counter points to the stack

## Buffer Overflow

- Buffer Overflow Example

## Buffer Overflow Attack

- Buffer overflow can happen by accident
  - Or a malicious attack
- Example of an attack on a web server:
  - The server accepts user input from a name field on a web page
    - Into an unchecked buffer variable
    - The attacker supplies a malicious code as input
  - The code read by the server overflows part of the application code
  - The web server now runs the malicious code

## Harm from Buffer Overflows

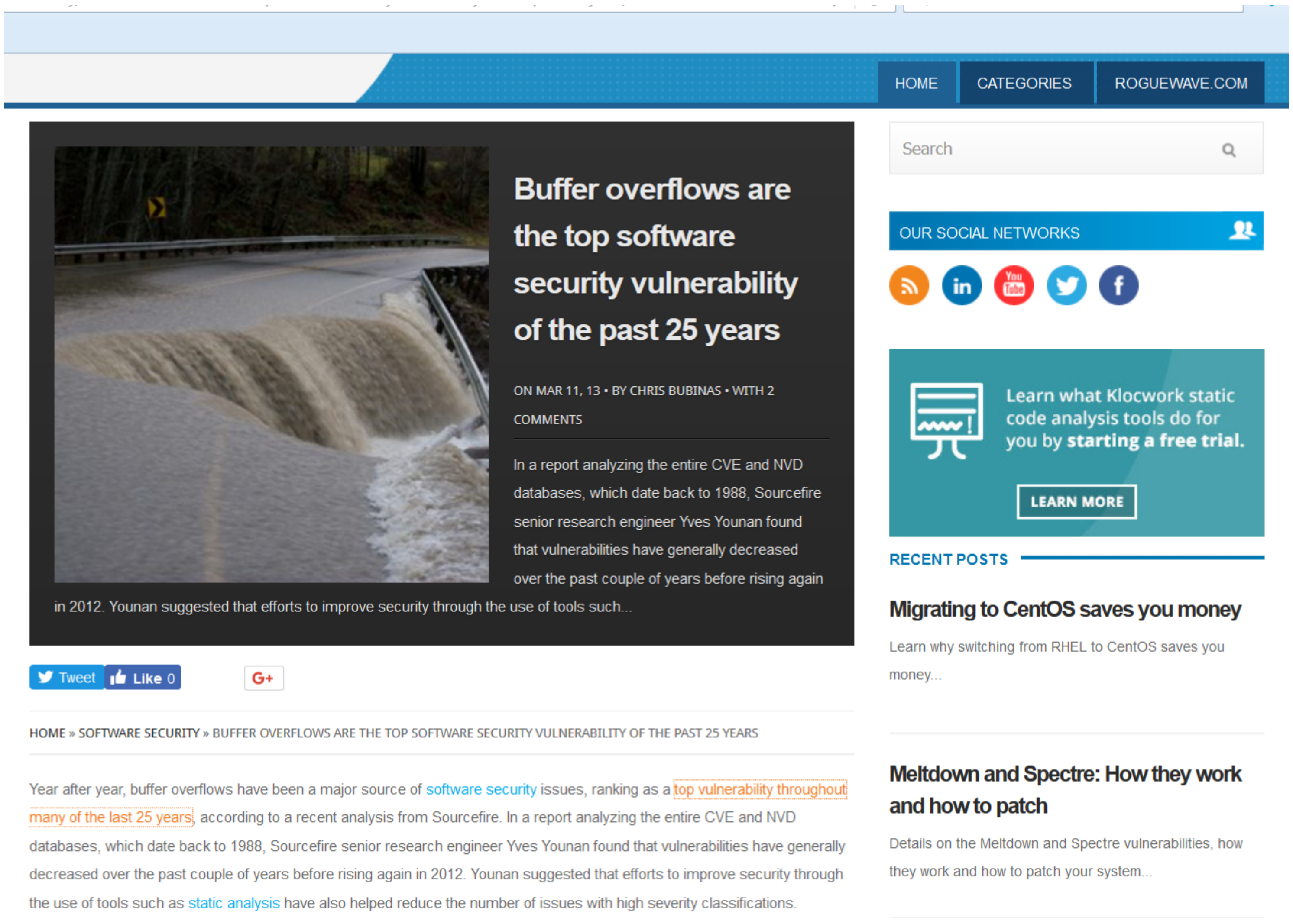
- Overwrite:
  - Another piece of your program’s data
  - An instruction in your program
  - Data or code belonging to another program
  - Data or code belonging to the operating system

## Harm from Buffer Overflows

- Overwriting a program’s instructions gives attackers that program’s execution privileges
- Overwriting operating system instructions gives attackers the OS’s execution privileges

## Buffer Overflow Attack

- Buffer overflows are a major source of security issues
- Software tools help decrease server issues
  - But vulnerability still exists and exploited
- Buffer overflows account for 14% of all vulnerabilities in the last 25 years
  - But 23% of the high severity vulnerabilities and 35% of critical vulnerabilities!



# Buffer overflows are the top software security vulnerability of the past 25 years

ON MAR 11, 13 • BY CHRIS BUBINAS • WITH 2 COMMENTS

In a report analyzing the entire CVE and NVD databases, which date back to 1988, Sourcefire senior research engineer Yves Younan found that vulnerabilities have generally decreased over the past couple of years before rising again in 2012. Younan suggested that efforts to improve security through the use of tools such...

HOME » SOFTWARE SECURITY » BUFFER OVERFLOWS ARE THE TOP SOFTWARE SECURITY VULNERABILITY OF THE PAST 25 YEARS

Year after year, buffer overflows have been a major source of software security issues, ranking as a top vulnerability throughout many of the last 25 years, according to a recent analysis from Sourcefire. In a report analyzing the entire CVE and NVD databases, which date back to 1988, Sourcefire senior research engineer Yves Younan found that vulnerabilities have generally decreased over the past couple of years before rising again in 2012. Younan suggested that efforts to improve security through the use of tools such as static analysis have also helped reduce the number of issues with high severity classifications.

Learn what Klocwork static code analysis tools do for you by **starting a free trial.**

LEARN MORE

RECENT POSTS

**Migrating to CentOS saves you money**

Learn why switching from RHEL to CentOS saves you money...

**Meltdown and Spectre: How they work and how to patch**

Details on the Meltdown and Spectre vulnerabilities, how they work and how to patch your system...

## Notes

- Buffer overflow may also be over-reading
    - Reading beyond allocated memory

## Code Example 1

- Is this function safe?

```
void vulnerable() {
        char buf[64];
        …
        gets(buf);
        …
        }
```

- How can you make it safer?

## Code Example 1 (cont.)

- Is this function safe?

```
void safe() {
        char buf[64];
        …
        fgets(buf, 64, stdin);
        …
        }
```

- Can we make it safer?

## Code Example 1 (cont.)

- Is this function safe?

```
void safe() {
        char buf[64];
        …
        fgets(buf, 64, stdin);
        …
        }
```

- What happens when the function changes over time?

## Code Example 1 (cont.)

- Function grows after a while...

```
void safe() {
        char buf[64];
        …
        …
        …
        …
        …
        fgets(buf, 64, stdin);
        …
        }
```

## Code Example 1 (cont.)

- A bigger buffer may be needed, a change may (eventually) occur:

```
void safe() {
        Char buf[64];
        …
        …



        …
        …
        fgets(buf, 128,stdin)
```

- How can we make it safer?

## Code Example 1 (cont.)

```
void safer() {
        char buf[64];
        …
        fgets(buf, sizeof buf, stdin);
        …
        }
```

## Code Example 2

- Is this function safe?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
memcpy(void *s1, const void *s2, size_t n);
```

## Code Example 2

- $Size\_t$ is an unsigned integer type
- Signed integer negative values change to a high positive number
  - when converted to an unsigned type
    - In libc routines, such as memcpy
- How can an attacker exploit this?

## Code Example 2

- Malicious users can often specify negative integers through various program interfaces
  - undermine an application's logic
- This happens commonly when a maximum length check is performed on a user-supplied integer
  - but no check is made to see whether the integer is negative

## Signed and unsigned integers

- If you have a function:

```
void f(unsigned int count) { }
```

- and call it with

```
f(-1);
```

the compiler will just pass some gigantic number into the function without even a warning

## Code Example 3

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL) return;
  memcpy(buf, data, len);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

- Is it safe?
  - No, vulnerable!
  - If len = 0xffffffff, then program allocates only 1  byte
    - Overflows

## Code Example 3

- Len chosen to be a large negative number
  - But program translates it into a positive number
    - If it was signed, the program would not allow to set it that large
  - Len+2 = 1 byte

- Is it safe?
  - No, vulnerable!
  - If len = 0xffffffff, then program allocates only 1  byte
    - Overflows

## Code Example 4 – Security Implications

- char buf[40];
- int authenticated=0;
- void vulnerable(){
    - gets(buf)
    - }

## Code Example 4 – Security Implications

- A login routine sets authenticated flag only if user proves knowledge of password
- What’s the risk?
  - Authenticated var stored immediately after buf
  - Attacker overwrites data after end of buf
  - Attacker supplies 41 bytes ($41^{st}$ set non-zero)
    - Makes authenticated flag true!
    - Attacker gains access: security breach!

## Other possible outcomes

- Attacker could insert his own code
  - Set the return pointer from the function to run its code

## Moreover

- These examples provide their own strings
- However, many times strings come from users
  - Users may be malicious
  - May be in form of text inputs, file inputs, etc.
- Validating user input is essential to software security!

## CODE INJECTION

## Code Injection

- Exploitation of a computer bug that is caused by processing invalid data
- Code injected into a vulnerable computer program and changes the course of execution
  - Using ***buffer overflow***
- Results can be disastrous
  - May allow computer worms to propagate

## Code Injection

- Must be machine code
  - Compiled and ready to run
- Can’t contain all-zero bytes
  - String copying functions will stop copying
- Code has to be completely self-contained
  - Can’t resolve memory addresses at run-time through loader

## Code Injection

- One possibility: run ***shellcode***
  - Provide general access to the system
    - through the command line
- Need to get the injected code to run
  - Store the address of the injected code on stack in the return address location

# Code Injection

- Finding the return address location is challenging
  - Without address randomization, stack always starts from same fixed address
    - Does not grow very deeply
    - Possible to find/guess the return address location

## Stack and Functions

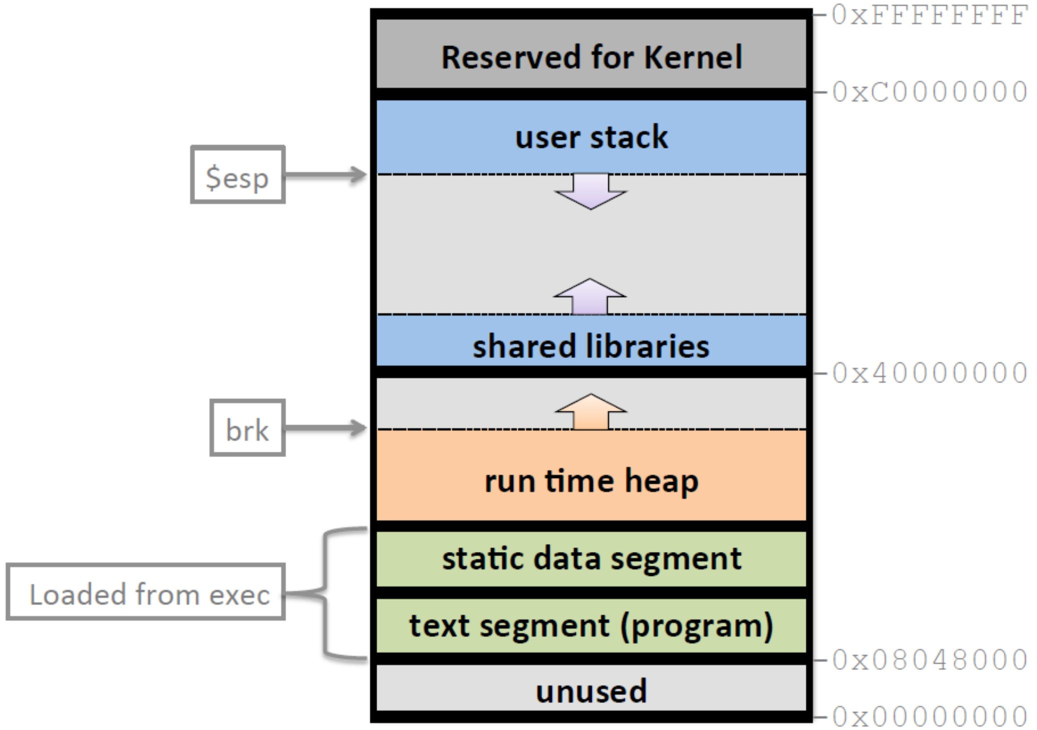
- Calling functions:
  - Push arguments onto the stack
  - Push the return address
    - Where the control will return to at end of function
  - Jump to function’s address
- Called function:
  - Push the old frame pointer onto stack
  - Set frame pointer to end of stack
  - Push local variables onto stack

## Stack and Functions

- Returning function:
  - Reset the previous stack frame
  - ***Jump back to return address***

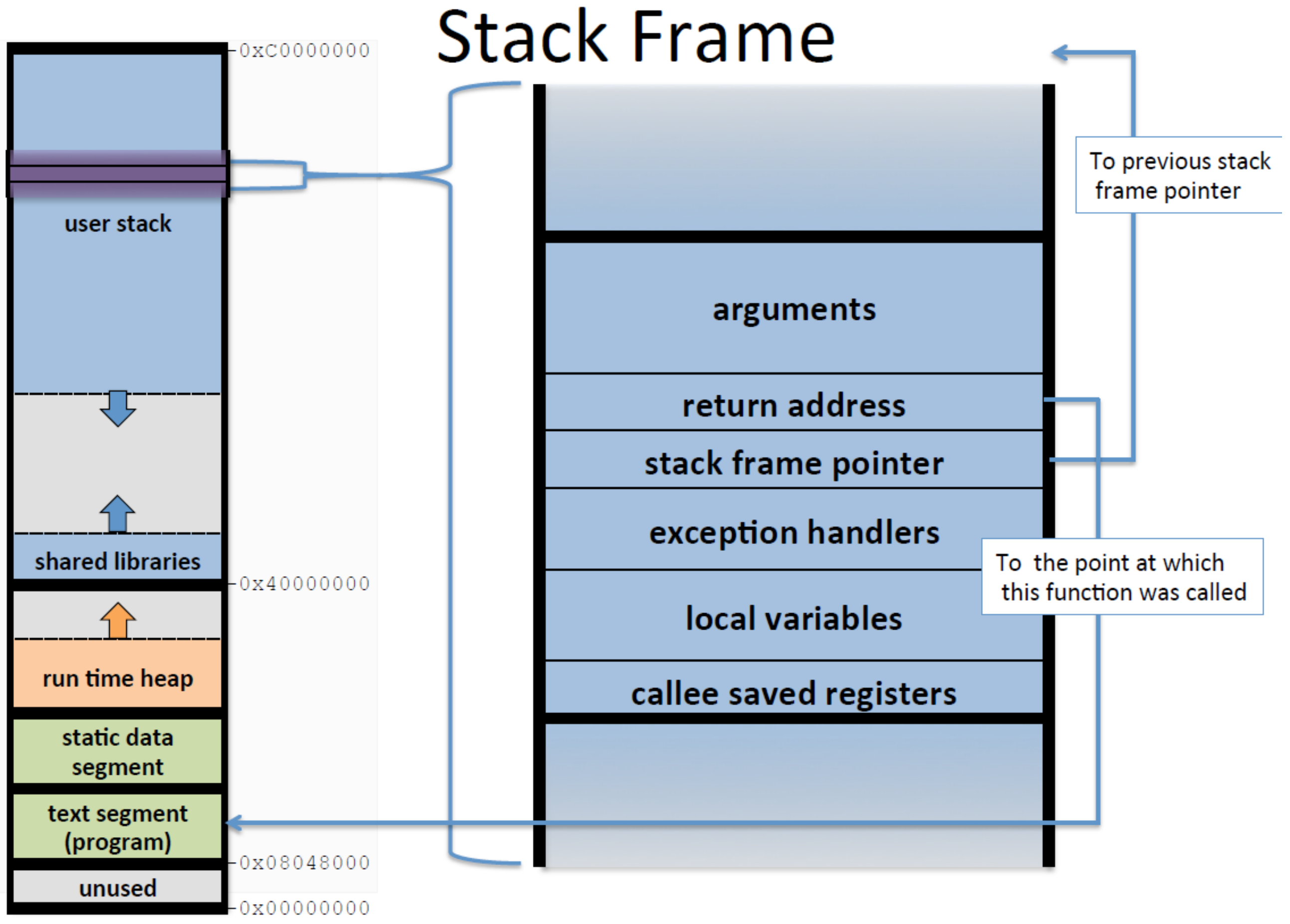
## Code Injection Attack Example

Linux (32-bit) process memory layout

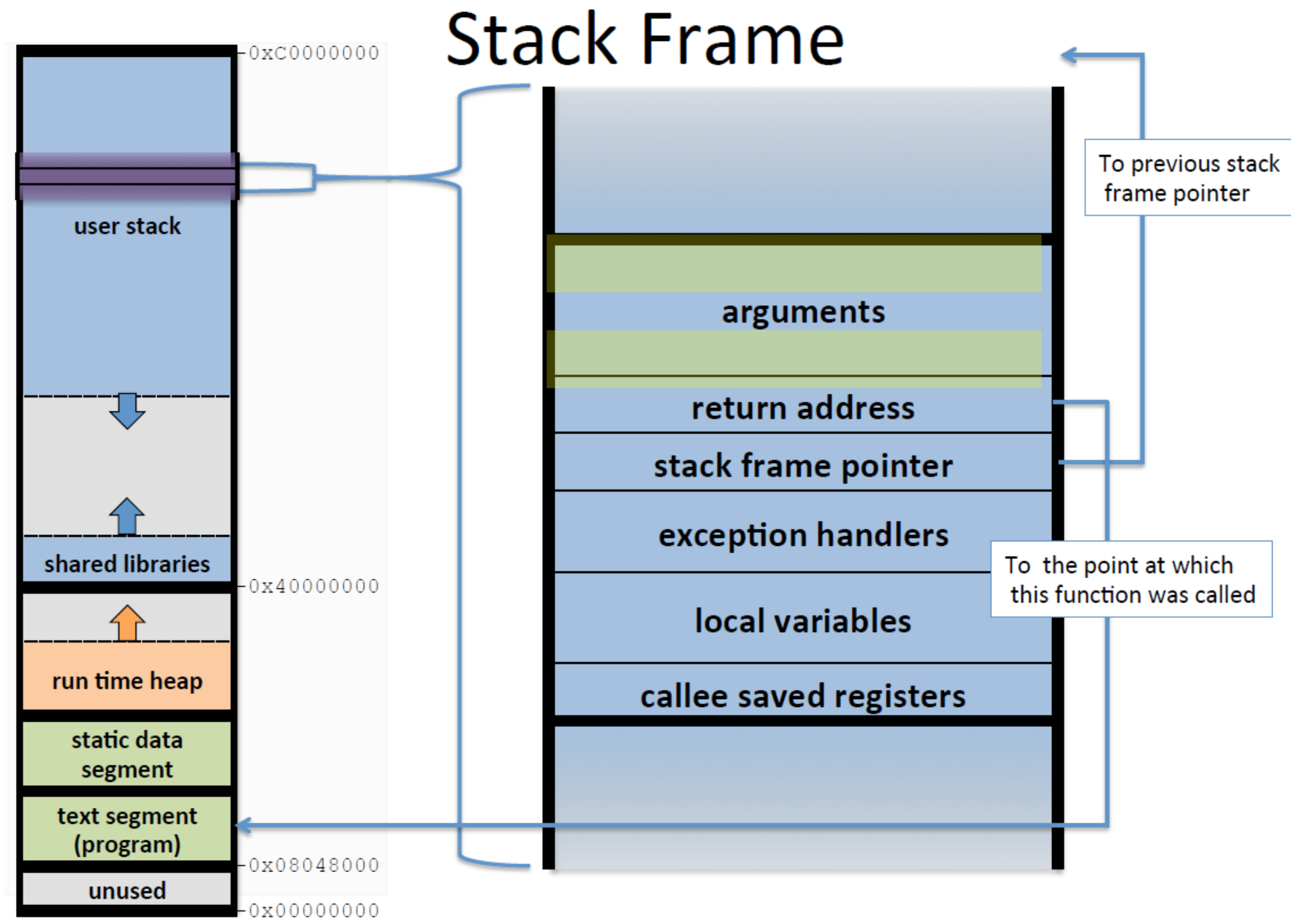


## Code Injection Attack Example

- What makes frame vulnerable to attacks?



## Code Injection Attack Example



## Code Injection

- Basic Stack Exploit:
  - Overwriting the return address allows an attacker to redirect the flow of program control.
  - Instead of crashing, this can allow arbitrary code to be executed.

## Code Injection

- Basic Stack Exploit example:
  - attacker chooses malicious code he wants executed (“shellcode”), compiles to bytes
  - includes this in the input to the program so it will get stored in memory somewhere
  - overwrites return address to point to it.

## Smashing the Stack

- Smashing the Stack

## MIDTERM REVIEW

# Topics we covered

- Security concepts
  - CIA Triad
  - Threat model
  - Risk management
- Authentication and access control
- Security principles
  - Developing robust approaches for current technology

# Topics we covered

- Cryptography
  - Symmetric cryptography
  - Key Exchange
  - Asymmetric cryptography
  - Cryptographic tools
    - Providing confidentiality and integrity

## Topics we covered

- Everything we covered in class is for the exam
- How to prepare:
  - Go over slides, homework
  - Book:
    - Chapters 1-3
    - Chapter 12

Questions?