CISC 3325 - INFORMATION SECURITY

Software Security

Adapted from *Security in Computing, Fifth Edition*, by Charles P. Pfleeger, et al. (ISBN: 9780134085043). Copyright 2015 by Pearson Education, Inc. All rights reserved

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

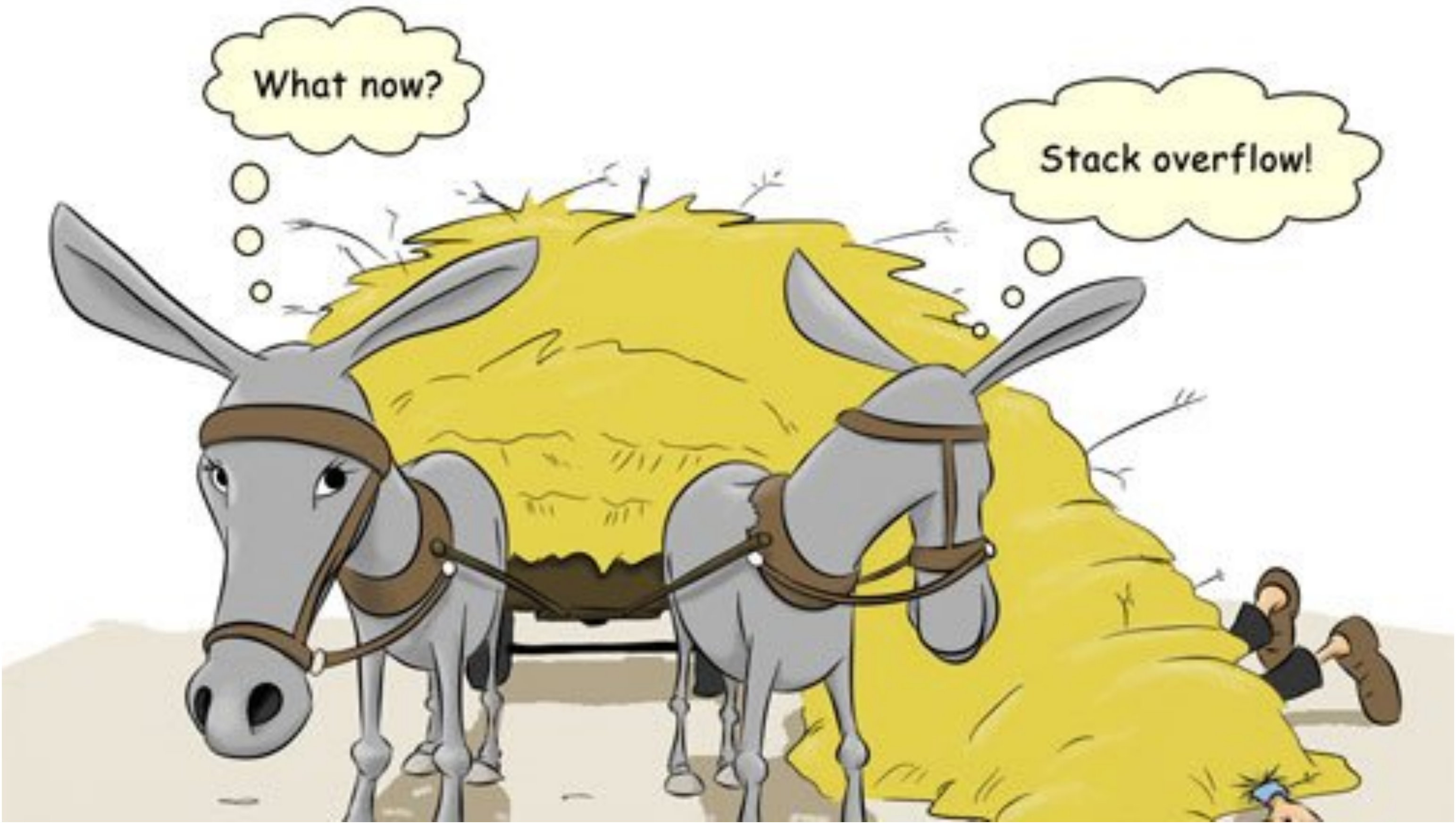
Software Security

- 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

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!

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 Overflows

- Occur when data is written beyond the space allocated for it
  - such as a 10^{th} byte in a 9-byte array
- In 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 Overflow

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

- 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	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.)

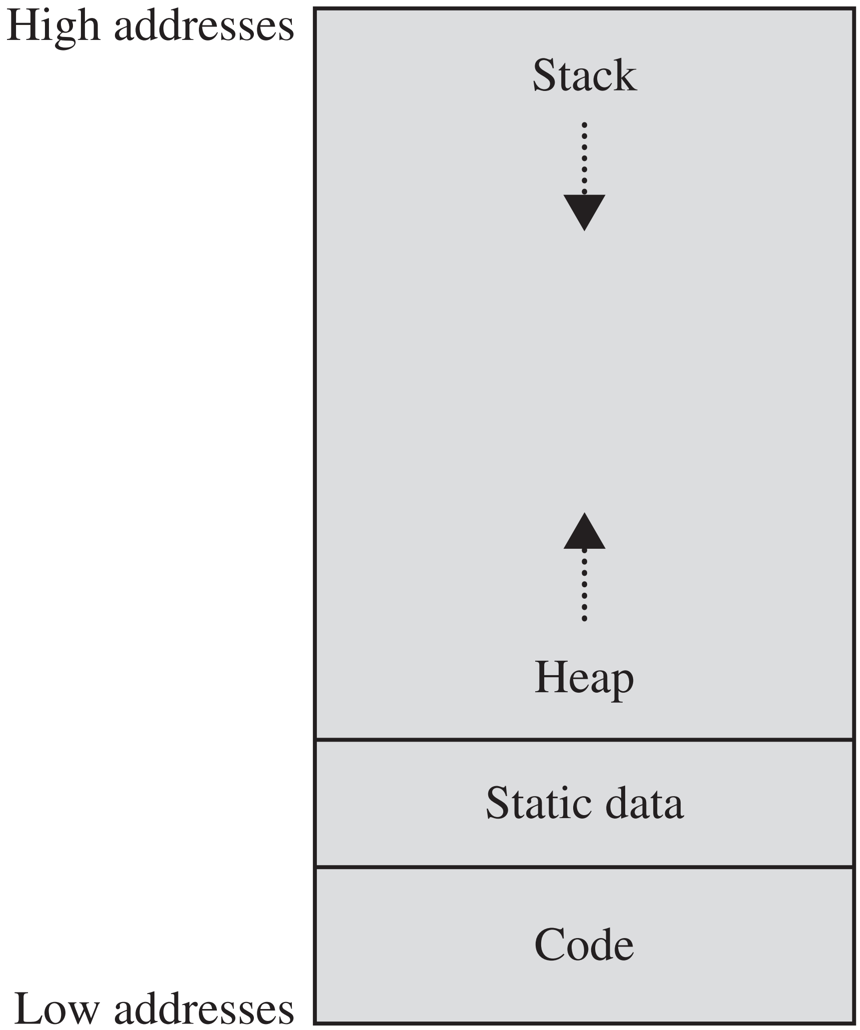
- How to prevent this?
- Replace call to [strcpy]() with [strncpy]()
  - strncpy takes the maximum capacity of A as an additional parameter
    - ensures that no more than this amount of data is written to A:

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

Buffer Overflow

- [Buffer Overflow Basics]

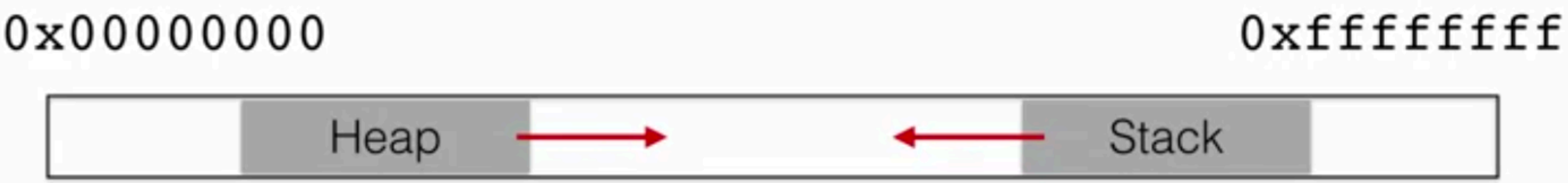
Memory Allocation



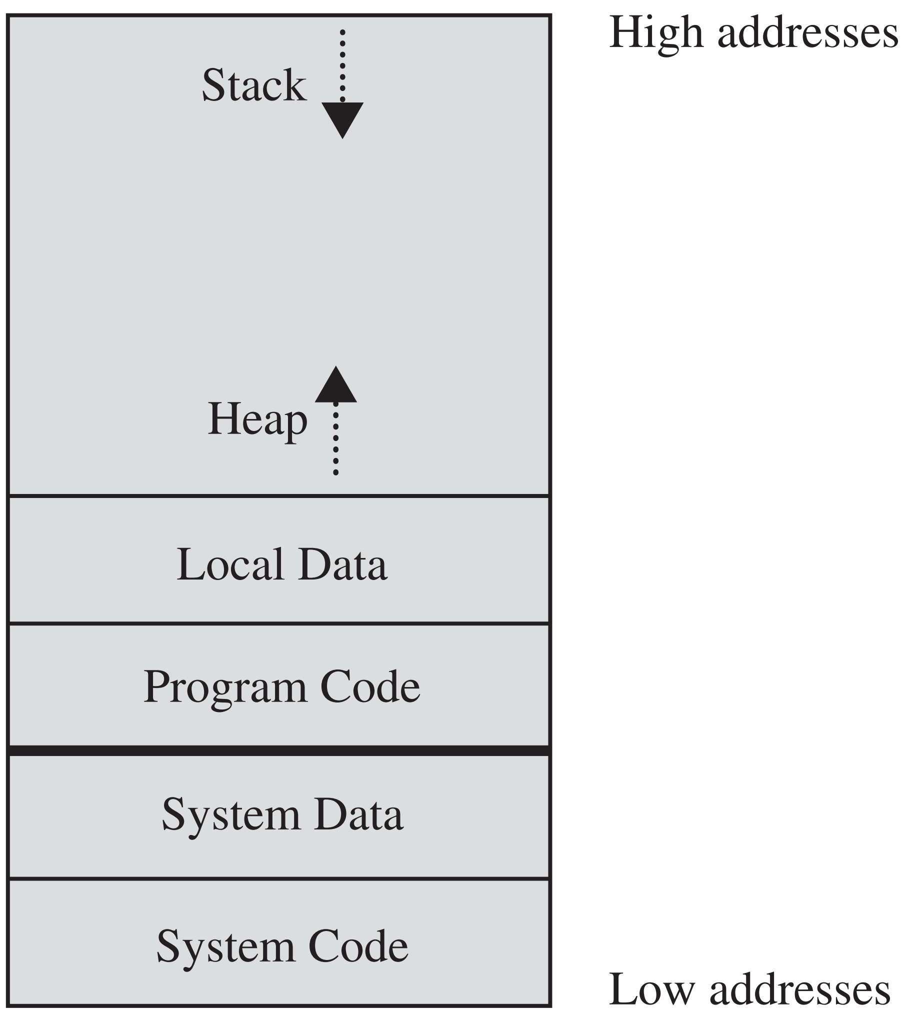
Memory Allocation

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

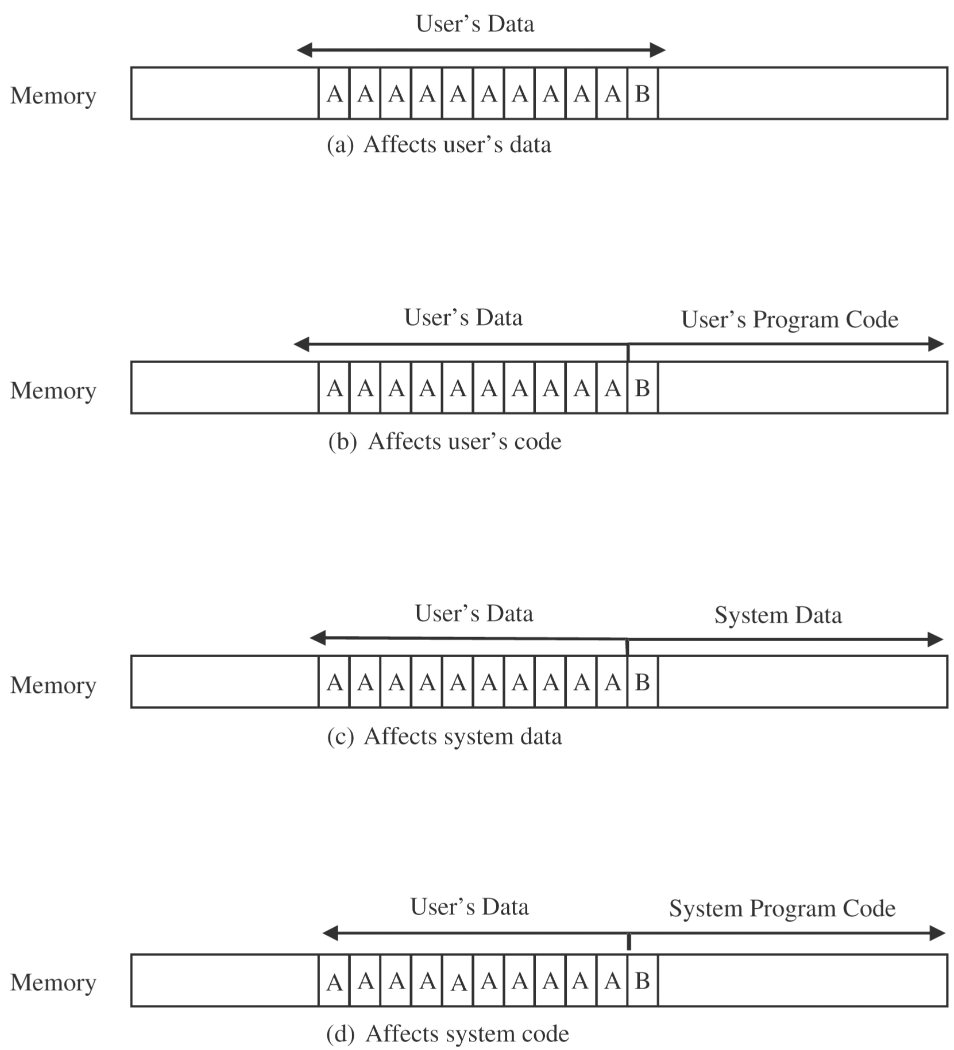
Memory Allocation



Memory Organization



Where a Buffer Can Overflow



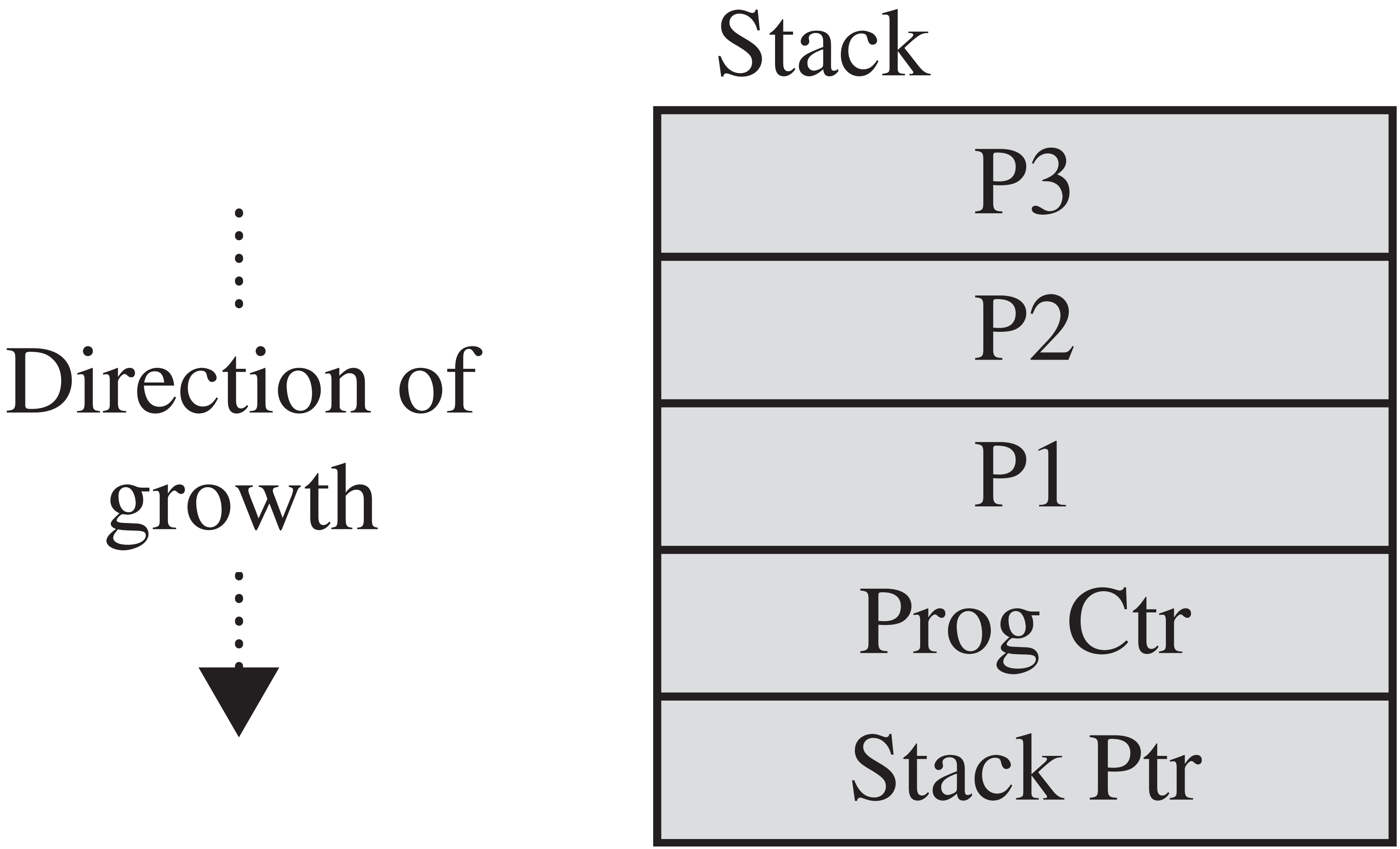
Stack and Functions

- 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

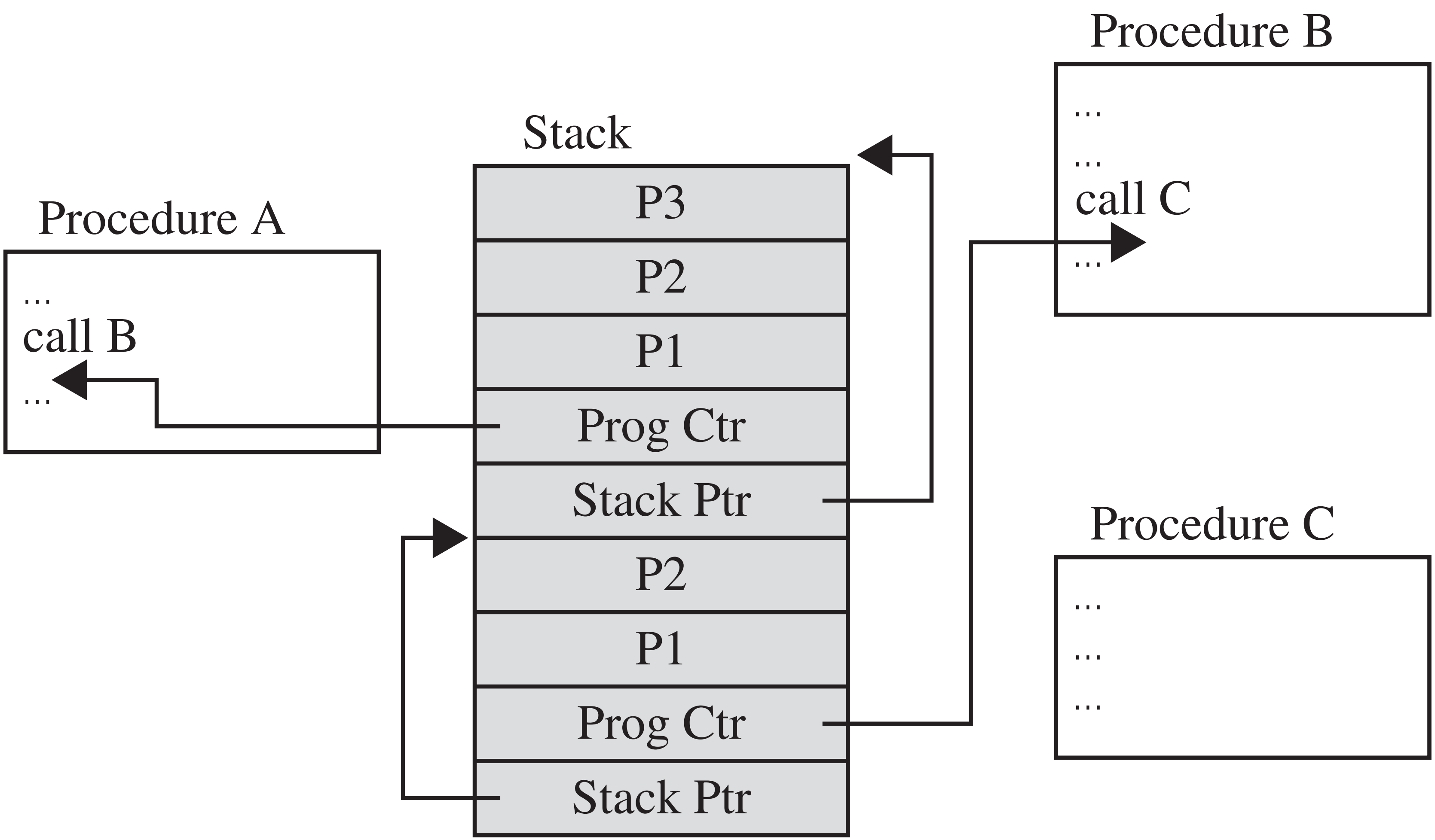
Stack and Functions

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

The Stack



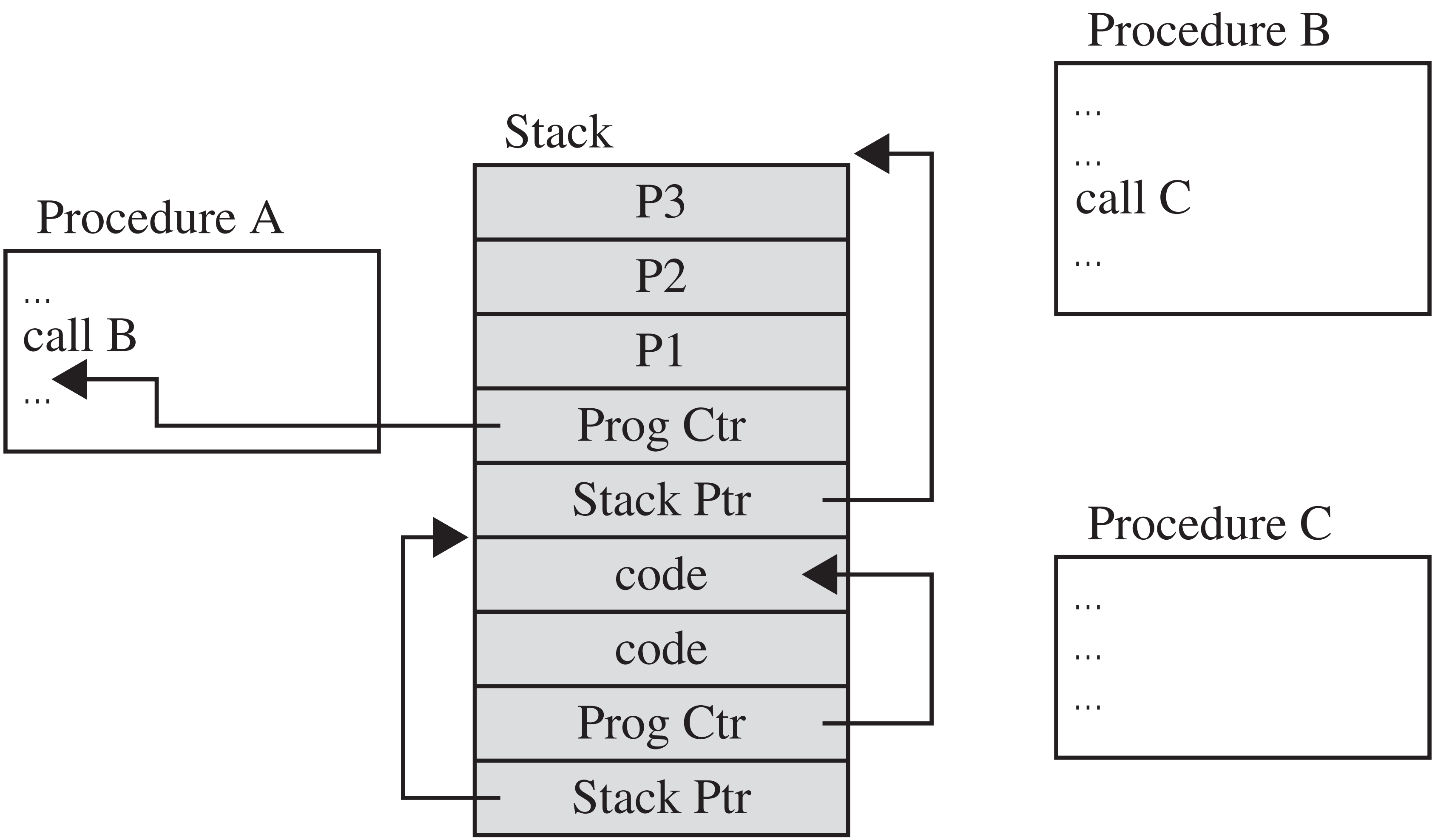
The Stack after Procedure Calls



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

- Instead of pointing at procedure B, the program counter is pointing 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 Attack

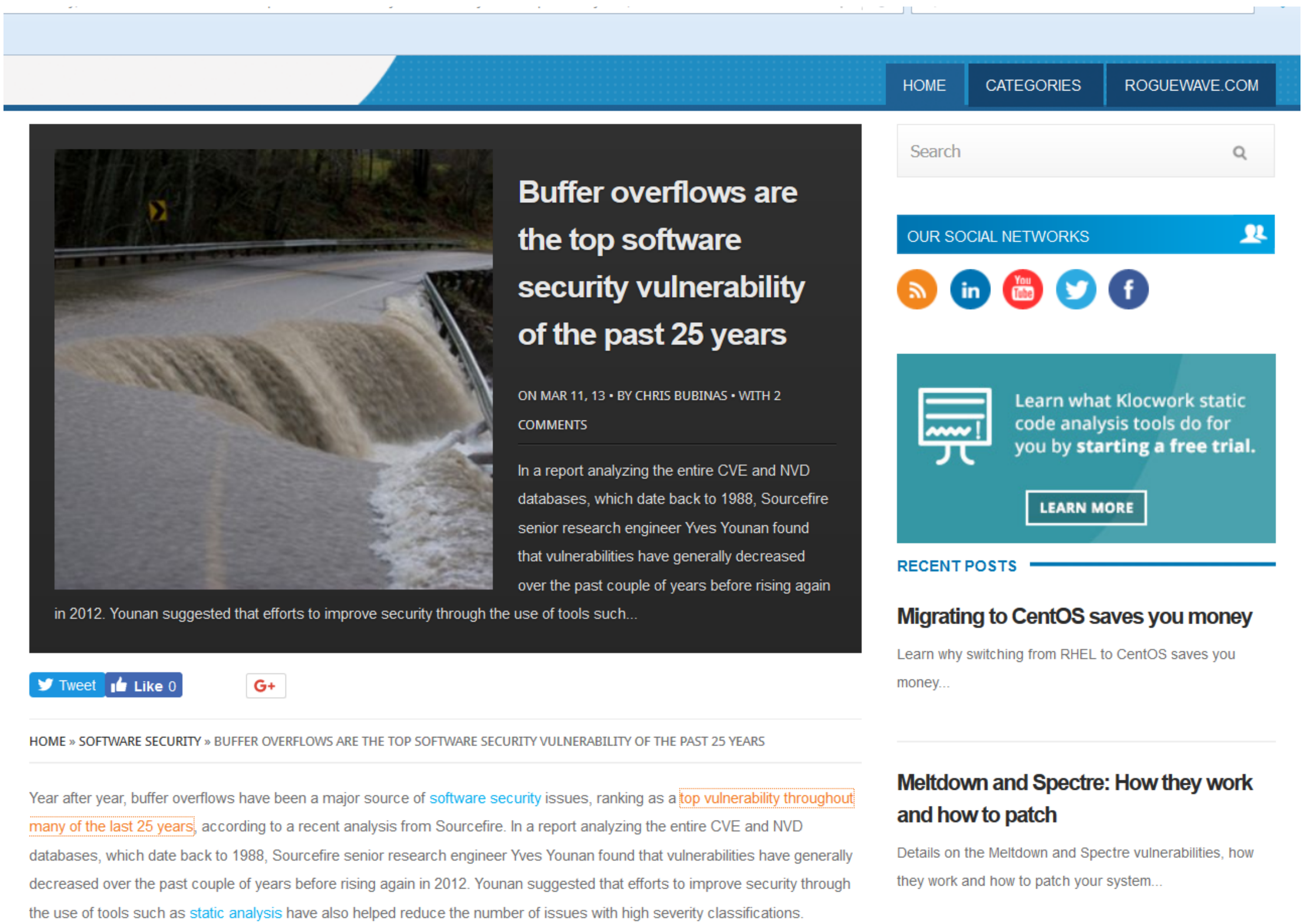
- Buffer overflow can happen by accident
  - Or a malicious attack
- Example of an attack on a web server:
  - The server accepts user input from a name field on a web page
    - Into an unchecked buffer variable
    - The attacker supplies a malicious code as input
  - The code read by the server overflows 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
- Overwriting a program's instructions gives attackers that program's execution privileges
- Overwriting operating system instructions gives attackers the operating system'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!



HOME CATEGORIES ROGUEWAVE.COM

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...

Tweet Like 0 G+

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.

Search

OUR SOCIAL NETWORKS

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

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code Example 4 – Security Implications

```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, arg1);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

A	u	t	h	4d	65	21	00	%ebp	%eip	&arg1	

https://www.coursera.org/learn/software-security/lecture/r9BIO/buffer-overflow

Code Example 4 – Security Implications

- After running the code, the ‘authenticated’ variable is set to a non-zero value
  - So even if the user is not authenticated, the if statement will return ‘1’ instead of ‘0’

Other possible outcomes

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

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
- 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

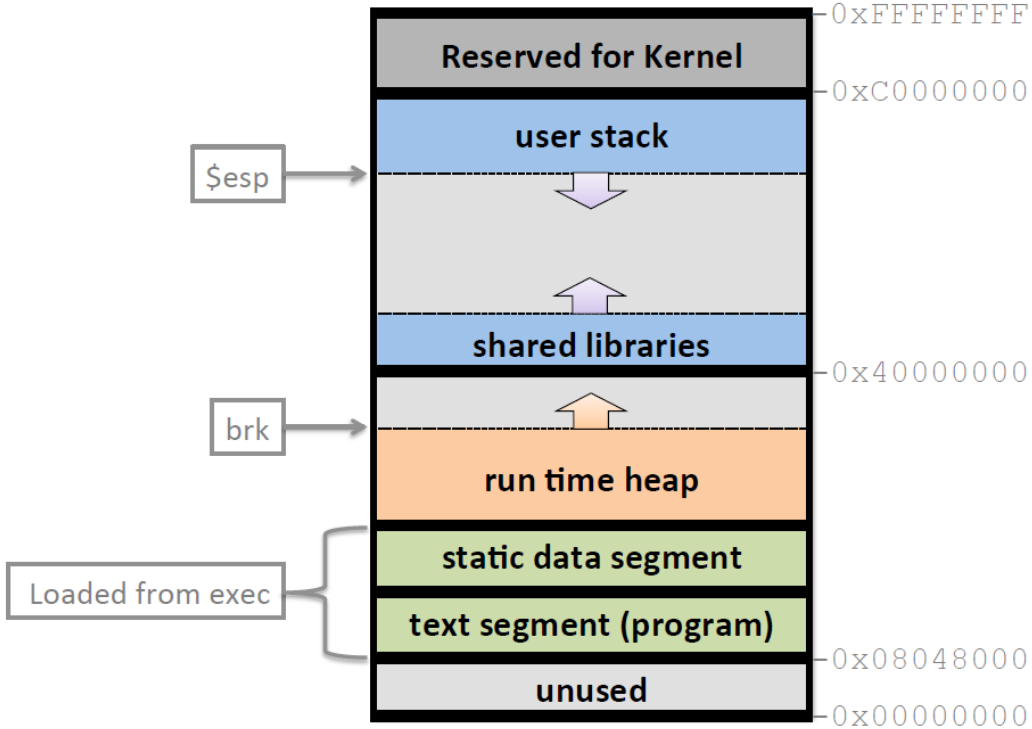
- Calling functions:
  - Push arguments onto the stack
  - Push the return address
    - Where the control will return to at end of function
  - Jump to function's address
- Called function:
  - Push the old frame pointer onto stack
  - Set frame pointer to end of stack
  - Push local variables onto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

Stack and Functions

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

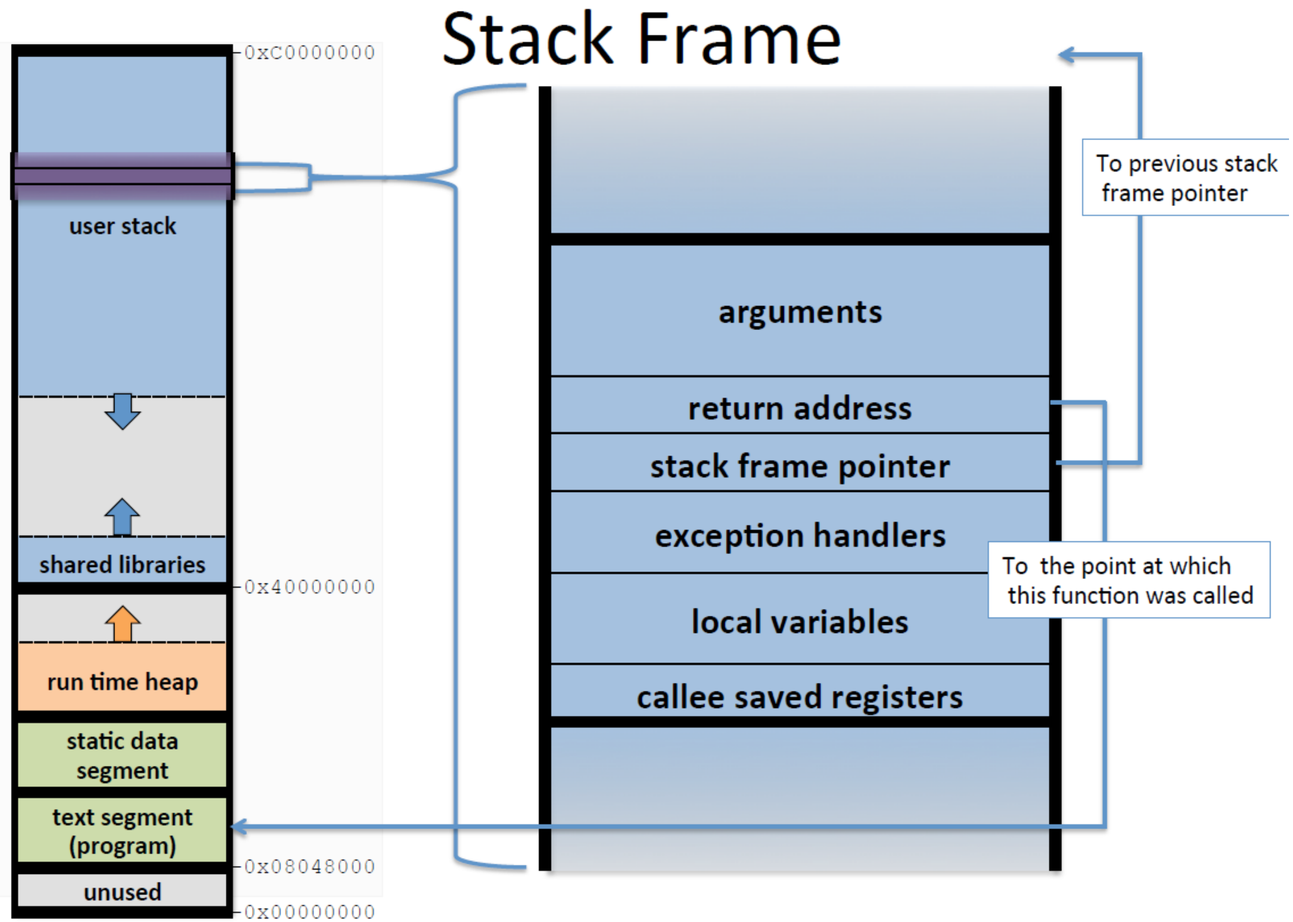
Code Injection Attack Example

Linux (32-bit) process memory layout

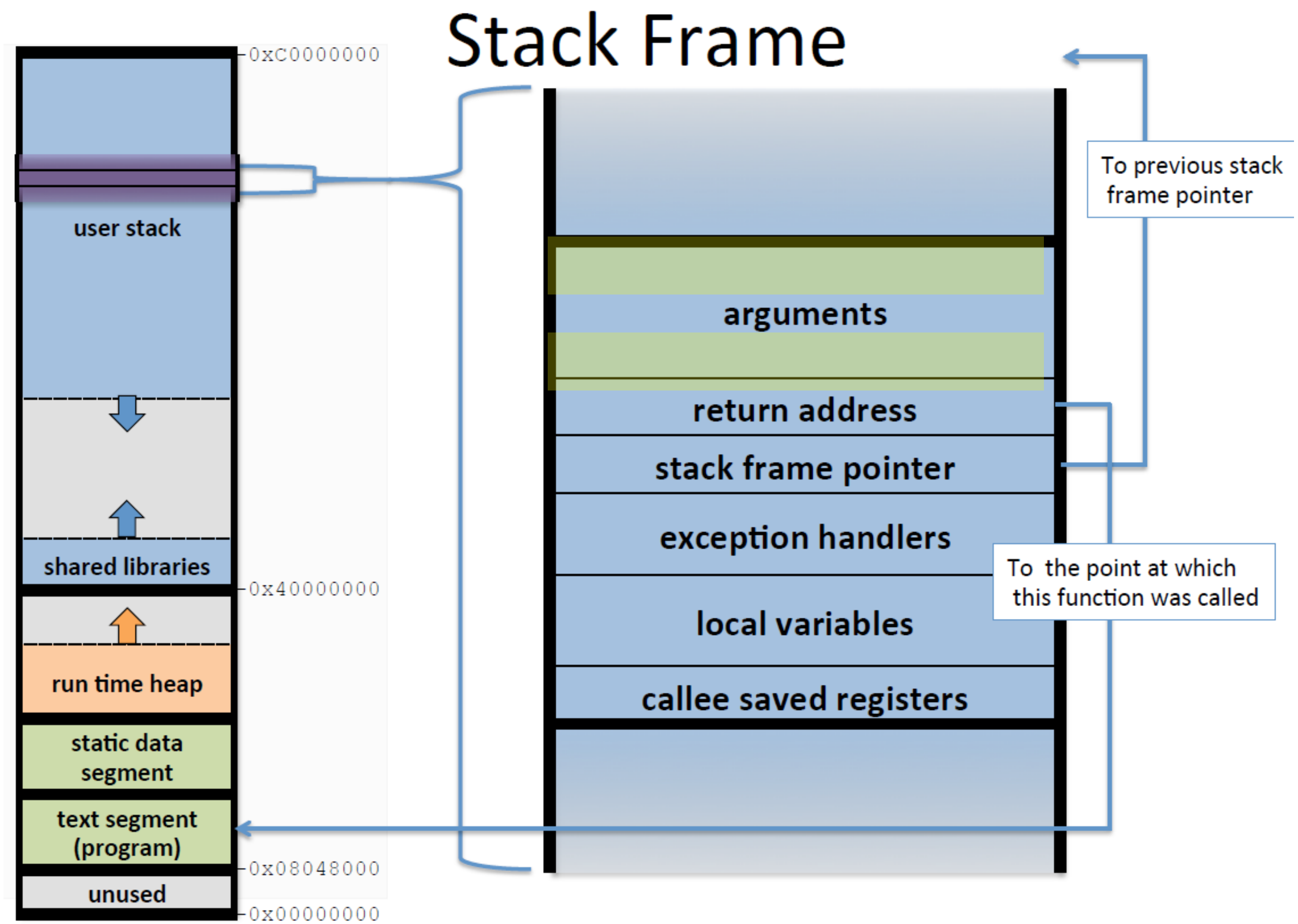


Code Injection Attack Example

- 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]

DEFENSES

Causes of Software Vulnerabilities

- Human-errors: human error is a significant source of software vulnerabilities and security vulnerabilities
  - Solution? Use automated tools
- Awareness: programmers not be focused on security
  - May be unaware of consequences
  - Solution: Learn about common types of security flaws
- Design flaws: software and hardware have design flaws and bugs
  - May not be designed well for security
    - Use better languages (Java, Python…)

Causes of Software Vulnerabilities

- Complexity: software vulnerabilities rise proportionally with complexity
  - Solution: design simple and well-documented software
- User input: accepting user input by internet can introduce software vulnerabilities
  - Data may be incorrect or fraudulent
    - Data can be designed to attack the receiving system
  - Solution: sanitize user input

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - If programmer doesn't make mistake, program will run correctly
  - We need to test for the absence of something
    - Security is a negative property!
      - "nothing bad happens, even in really unusual circumstances"
  - Normal inputs rarely stress security-vulnerable code

Testing for Software Security Issues

- How can we test more thoroughly?
  - Use random inputs
  - Create a testing plan
    - Allows defining a range of inputs

https://www.avyaancom/blog/why-hire-a-software-security-testing-company/

http://www.testnbug.com/2015/02/software-testing-types/

Testing for Software Security Issues

- How can we test more thoroughly?
  - Use random inputs
  - Create a testing plan
    - Allows defining a range of inputs
- How do we tell if a problem was found?
  - Crash or other deviant behavior;
    - now more expensive checks justified

https://www.avyaan.com/blog/why-hire-a-software-security-testing-company/

http://www.testnbug.com/2015/02/software-testing-types/

WORKING TOWARDS SECURE SYSTEMS

Common Software Vulnerabilities

- Memory safety vulnerabilities
- Input validation vulnerabilities
- Race conditions
- Time-of-Check to Time-of-Use (TOCTTOU) vulnerability

Memory safety vulnerabilities

- Software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers
- Java is said to be memory-safe
  - its runtime error detection checks array bounds and pointer dereferences (*p).
- In contrast, C and C++ support arbitrary pointers with no provision for bounds checking
  - thus are termed **memory-unsafe**

Input validation vulnerabilities

- Program requires certain assumptions on inputs to run properly
- When program does not validate input correctly, it may get exploited
- Buffer overflow and SQL injection are just a few of the **attacks** that can result from improper data **validation**

Input validation vulnerabilities

- Input Validation Example:
  - Bank money transfer:
    - Check that amount to be transferred is non-negative and no larger than payer's current balance

Input validation vulnerabilities

- **SQL injection** is a code **injection** technique
  - used to attack data-driven applications
  - nefarious **SQL** statements are inserted into an entry field for execution
    - e.g. to dump the database contents to the attacker
- **Directory traversal** or **Path Traversal** is an HTTP attack
  - allows attackers to access restricted **directories**
  - execute commands outside of the web server's root **directory**

Input validation vulnerabilities

- **Cross-site scripting** is a computer security vulnerability
  - typically found in web applications
  - XSS enables attackers to inject client-side scripts into web pages viewed by other users

Overflow Countermeasures

- Staying within bounds
  - Check lengths before writing
  - Confirm that array subscripts are within limits
  - Double-check boundary condition code for off-by-one errors
  - Limit input to the number of acceptable characters
  - Limit programs’ privileges to reduce potential harm

Overflow Countermeasures

- Many languages have overflow protections
- Code analyzers can identify many overflow vulnerabilities
- Canary values in stack to signal modification

Incomplete Mediation

- Mediation: Verifying that the subject is authorized to perform the operation on an object
- Preventing incomplete mediation:
  - Validate all input
  - Limit users' access to sensitive data and functions
  - Complete mediation using a reference monitor

Race Condition

- Output is dependent on the sequence or timing of other uncontrollable events
- Becomes a vulnerability when events happen in a different order
  - than the programmer intended

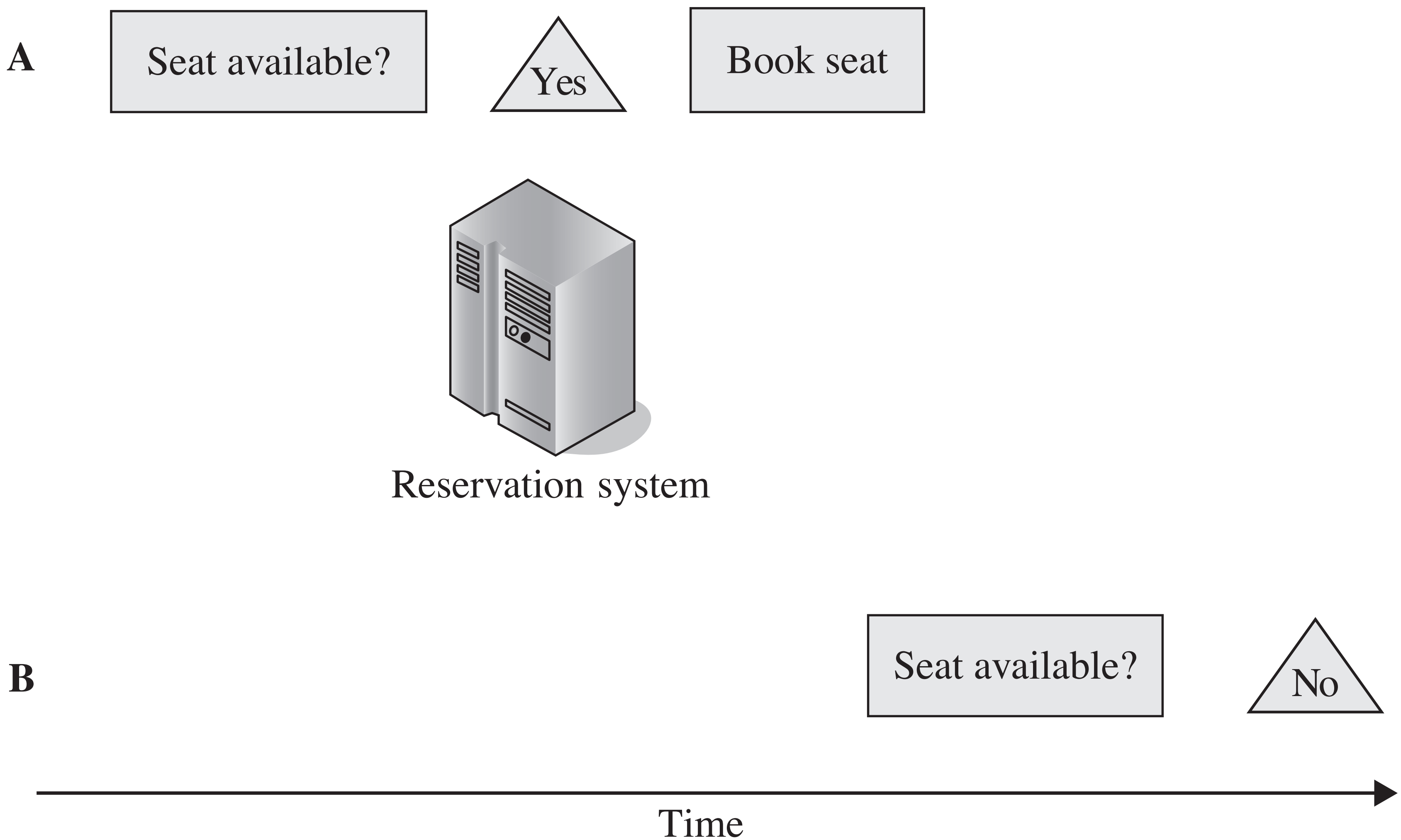
Time-of-Check to Time-of-Use (TOCTTOU) vulnerability

- Software bugs caused by changes in a system
  - Between the checking of a condition (such as a security credential) and the use of the results of that check
- An example of a race condition

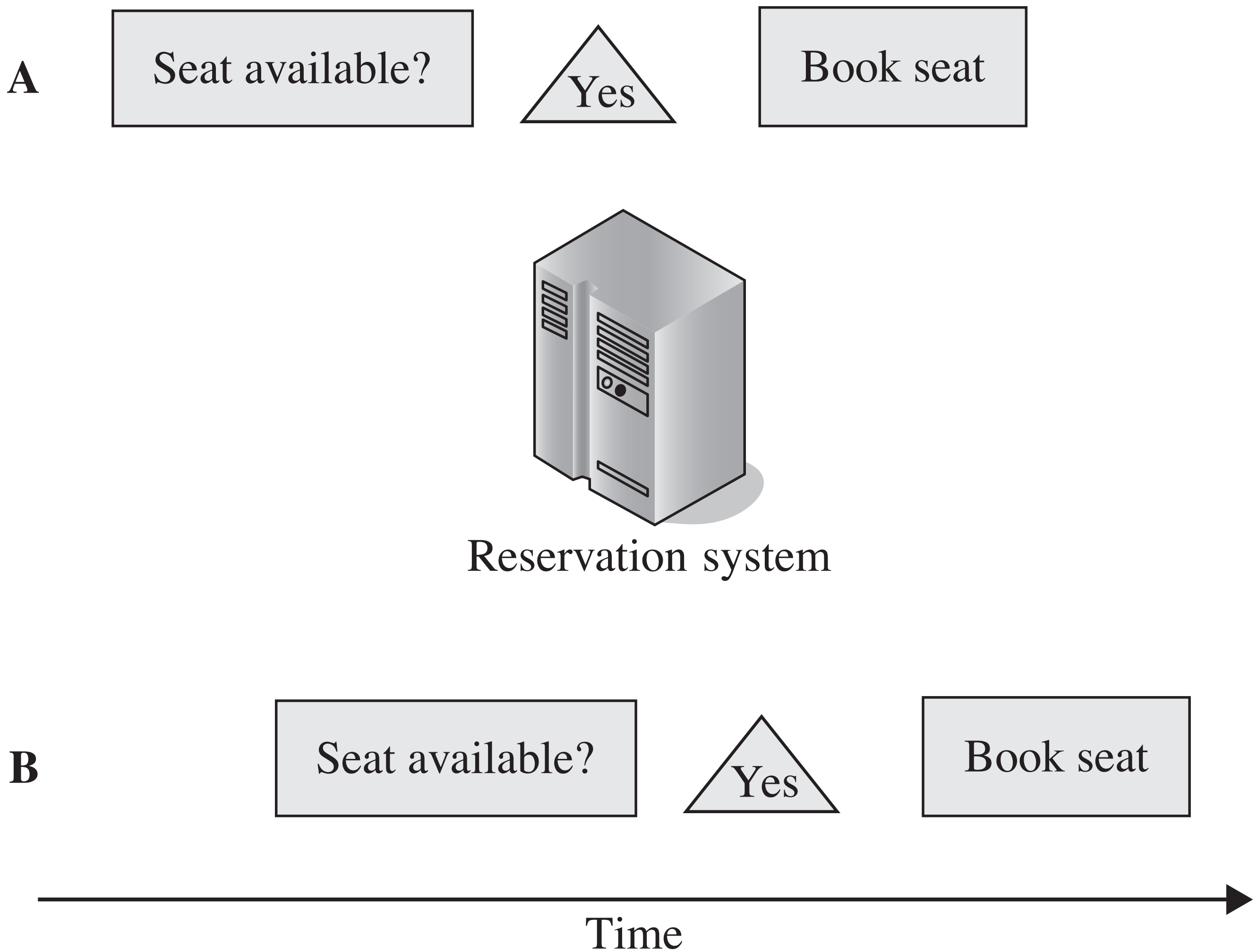
Race Condition Examples:

- Booking a seat on a place:
  - No race condition:
    - A booker books the last seat on the plane
    - thereafter the system shows no seat available
  - Race condition:
    - Before the first booker can complete the booking for the last available seat, a second booker looks for available seats
    - This system has a race condition, where the overlap in timing of the requests causes errant behavior

Example: No Race Condition



Example: Race Condition Exists



Other Programming Oversights

- Undocumented access points (backdoors)
- Off-by-one errors
  - Occurs when an iterative loop iterates one time too many or too few
  - May occur when:
    - Using “<=" where “<" should have been used in a comparison
    - A sequence starts at zero rather than one (as with array indices in many languages)

Other Programming Oversights

- Integer overflows
  - Attempt to create a numeric value outside of the range that can be represented with a given number of digits
    - either larger than the maximum or lower than the minimum representable value
- Unterminated null-terminated string
- Parameter length, type, or number errors
- Unsafe utility libraries

Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …
  - New software versions are constantly released
    - Keeping up with other competitive applications, OS changes, add new features.
- What’s hard about patching?
  - Can break crucial functionality inadvertently
  - Management burden:
    - It never stops (the “patch treadmill”) …

- Questions?