# Lecture 23: Protection and Security

**Instructor: Mitch Neilsen**

**Office: N219D**

# Quote of the Day

"I needed a password eight characters long, so I picked Snow White and the Seven Dwarves."

-- Nick Helm, comedian

# Chapter 14: Protection

- Goals of Protection
- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems
- Language-Based Protection

# Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an access matrix are used to specify the resources a process may access
- Examine capability and language-based protection systems

# Core Components

- Authentication
  - Verify that we really know who we are talking to
- Authorization
  - Check that user X is allowed to do Y
- Access Enforcement
  - Ensure that authorization decision is respected
- Hard to do because every system has holes
  - Social vs. technical enforcement

# Authentication

- Passwords
  - Weakest form, and most common
  - Subject to dictionary attacks
  - Passwords should not be stored in clear text, instead, use a **one-way hash function** to generate shadow passwords.
- Badge or Keycard
  - Should not be (easily) forgeable
  - Problem: how to invalidate?
- Biometrics
  - Problem: ensure trusted path to device

# Now that we've authenticated, Authorization

- Operating system consists of a collection of objects, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem - ensure that each object is accessed correctly and only by those processes/users that are authorized to do so.

# Principles of Protection

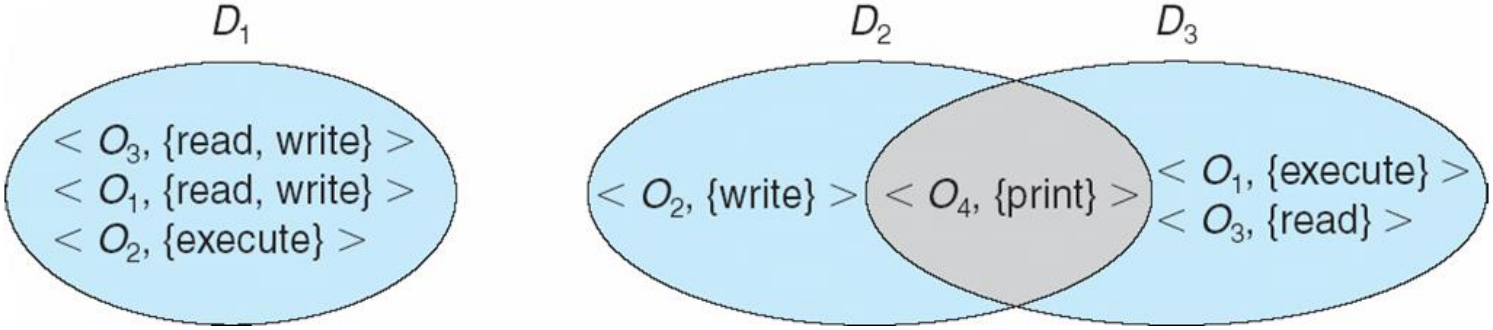
Guiding principle – **principle of least privilege**

- Programs, users and systems should be given just enough privileges to perform their tasks
- Example: SELinux - Security-Enhanced Linux

# Domain Structure

Access-right = <*object-name*, *rights-set*>
where *rights-set* is a subset of all valid operations that can be performed on the object.

Domain = set of access-rights



# Domain Implementation (UNIX)

System consists of 2 domains:

- User
- Supervisor

UNIX

```
ls -l /usr/bin/passwd
-rwsr-xr-x  root  root  .. passwd
```

- Domain = user-id
- Domain switch accomplished via file system.

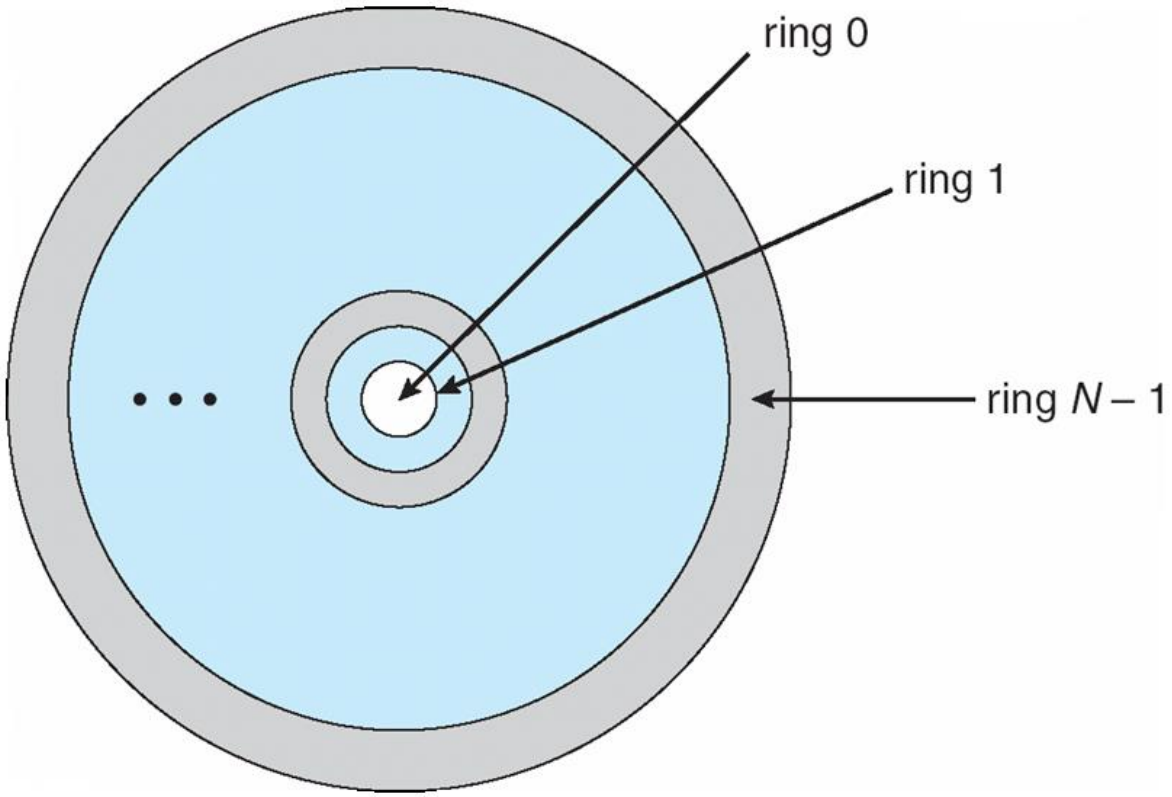
  Each file has associated with it a domain bit (setuid bit (**s**)).

  When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.

# Domain Implementation (MULTICS)

Let $D_i$ and $D_j$ be any two domain rings.

If $j < i \Rightarrow D_i \subseteq D_j$



# Access Matrix

- View protection as a matrix (*access matrix*)
- Rows represent domains
- Columns represent objects
- **Access(i, j)** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

# Access Matrix

| object / domain | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read<br>write | | read<br>write | |

# Use of Access Matrix

If a process in Domain $D_i$ tries to do "op" on object $O_j$, then "op" must be in the access matrix.

Can be expanded to dynamic protection.

- Operations to add, delete access rights.
- Special access rights:

  *owner of* $O_i$

  *copy op from* $O_i$ *to* $O_j$

  *control –* $D_i$ *can modify* $D_j$ *access rights*

  *transfer – switch from domain* $D_i$ *to* $D_j$

# Use of Access Matrix (Cont.)

Access matrix design separates mechanism from policy.

- **Policy - what should be done**

  User dictates policy.

  Who can access what object and in what mode.

- **Mechanism - how it should be done**

  Operating system provides access-matrix + rules.

  If ensures that the matrix is only manipulated by authorized agents and that rules (policy) are strictly enforced.

# Implementation of Access Matrix

Each column = **Access Control List** for one object --
Defines who can perform what operation.

Domain 1 = read

Domain 4 = read, write

| object / domain | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

Each Row = **Capability List** (like keys or tickets)
For each domain, what operations are allowed on what objects. E.g., doman $D_3$:

Object $F_2$ – read

Object $F_3$ – execute

| object / domain | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

# Access Control List

- Store list of <user (domain), set of privileges> for each object
  - Example: files. For each file store who is allowed to access it (and how: read, write, execute)
  - Most contemporary file systems support it.
- Groups can be used to compress the information:
  - Traditional Unix permissions rwxr-xr-x; chmod 755 myDir.
- Where in the filesystem should you store ACLs/permissions?

# Capabilities

- Store (capability) list of <object, set of privileges> for each user (domain)
  - Typically used in systems that must be very secure
  - Default is empty capability list
- Capabilities also often function as names
  - Can access something if you know the name
- Must make names unforgeable, or must have system monitor who holds what capabilities each user (domain) has (e.g., by storing them in a protected area)

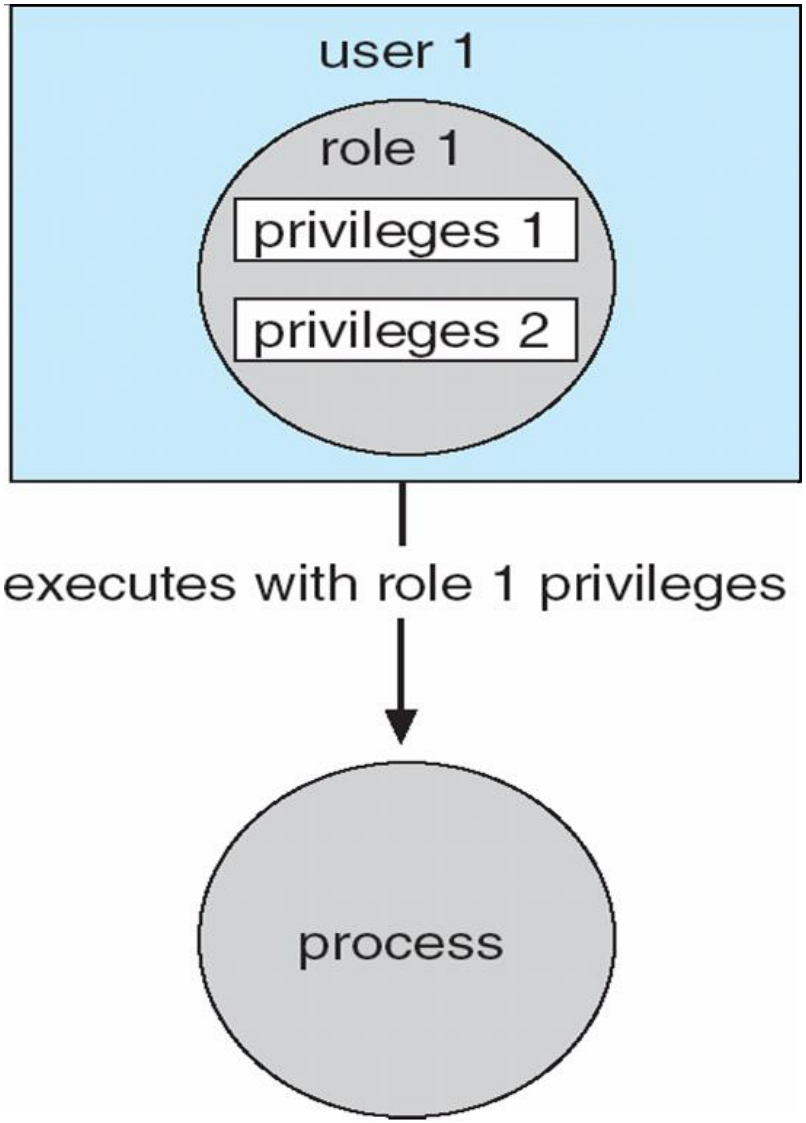
# Access Control

Protection can be applied to non-file resources, too

Solaris 10 provides **role-based access control** to implement least privilege

- Privilege is right to execute system call or use an option within a system call
- Can be assigned to processes
- Users assigned roles granting access to privileges and programs

# Role-based Access Control in Solaris 10



# Revocation of Access Rights

***Access List*** – Delete access rights from access list. (e.g., chmod)

- Simple
- Immediate

***Capability List*** – Scheme required to locate capability in the system before the capability can be revoked.

- Reacquisition
- Back-pointers
- Indirection
- Keys

# Capability-Based Systems

Hydra

- Fixed set of access rights known to and interpreted by the system.
- Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.

Cambridge CAP System

- Data capability - provides standard read, write, execute of individual storage segments associated with object.
- Software capability - interpretation left to the subsystem, through its protected procedures.

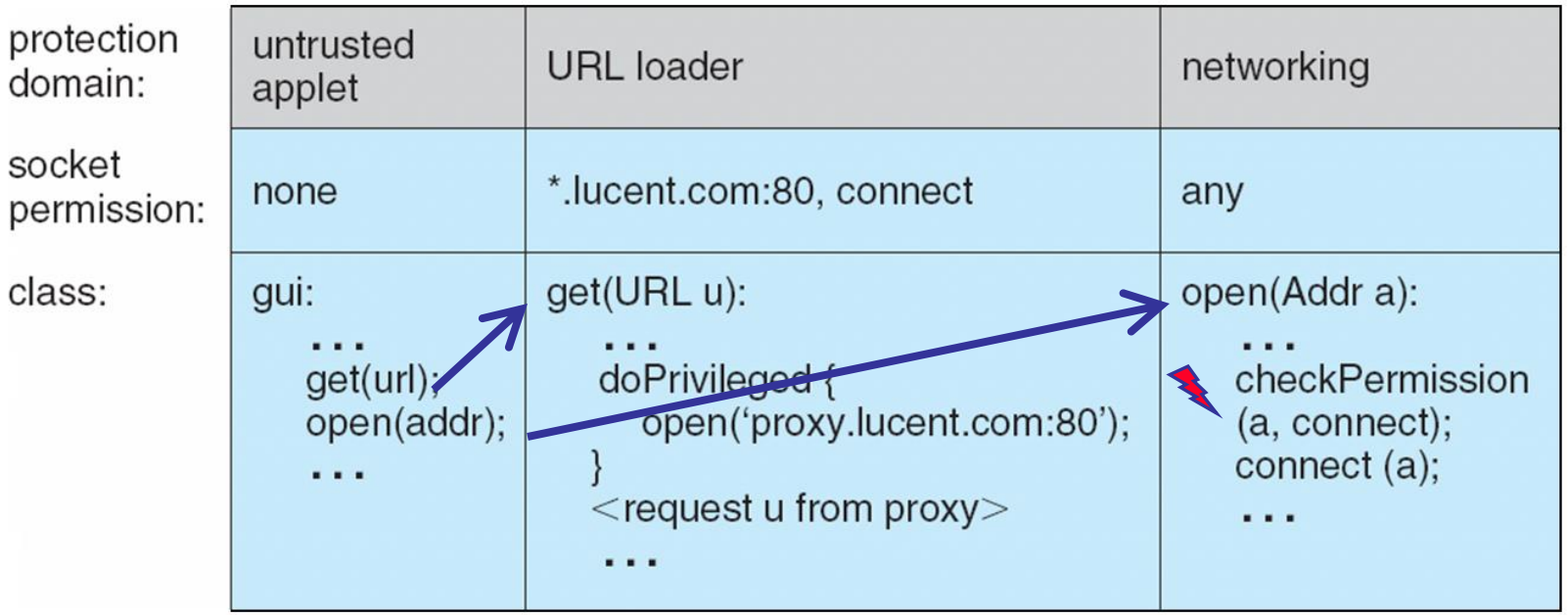
# Language-Based Protection

- Specification of protection in a programming language allows the high-level description of policies for the allocation and use of resources.
- Language implementation can provide software for protection enforcement when automatic hardware-supported checking is unavailable.
- Interpret protection specifications to generate calls on whatever protection system is provided by the hardware and the operating system.
- Based on type-safe languages (Java, C#, etc.)
- Do not allow direct memory access.
- Include access modifiers (private/public, etc.)
- Verify code before they execute it with respect to these safety properties.
- Build security systems on top of type-safe language run-times which associate code with sets of privileges.

# Example: Protection in Java

- Protection is handled by the Java Virtual Machine (JVM)
- A class is assigned a protection domain when it is loaded by the JVM.
- The protection domain indicates what operations the class can (and cannot) perform.
- If a library method is invoked that performs a privileged operation, the stack is inspected to ensure the operation can be performed by the library.

# Stack Inspection




Note that a process must not be allowed to modify annotations on its own stack frame or do other manipulations to bypass stack inspection.

# Chapter 15: Security

- The Security Problem
- Program Threats
- System and Network Threats
- Cryptography as a Security Tool
- User Authentication
- Implementing Security Defenses
- Firewalls to Protect Systems and Networks
- Computer-Security Classifications
- An Example: Windows XP

# Objectives

- To discuss security threats and attacks
- To explain the fundamentals of encryption, authentication, and hashing
- To examine the uses of cryptography in computing
- To describe the various countermeasures to security attacks

# The Security Problem

- Security must consider external environment of the system, and protect the system resources.
- Intruders (crackers) attempt to breach security.
- A **threat** is potential security violation.
- An **attack** is attempt to breach security.
- Attacks can be accidental or malicious.
- Easier to protect against accidental than malicious misuse.

# Security Violations

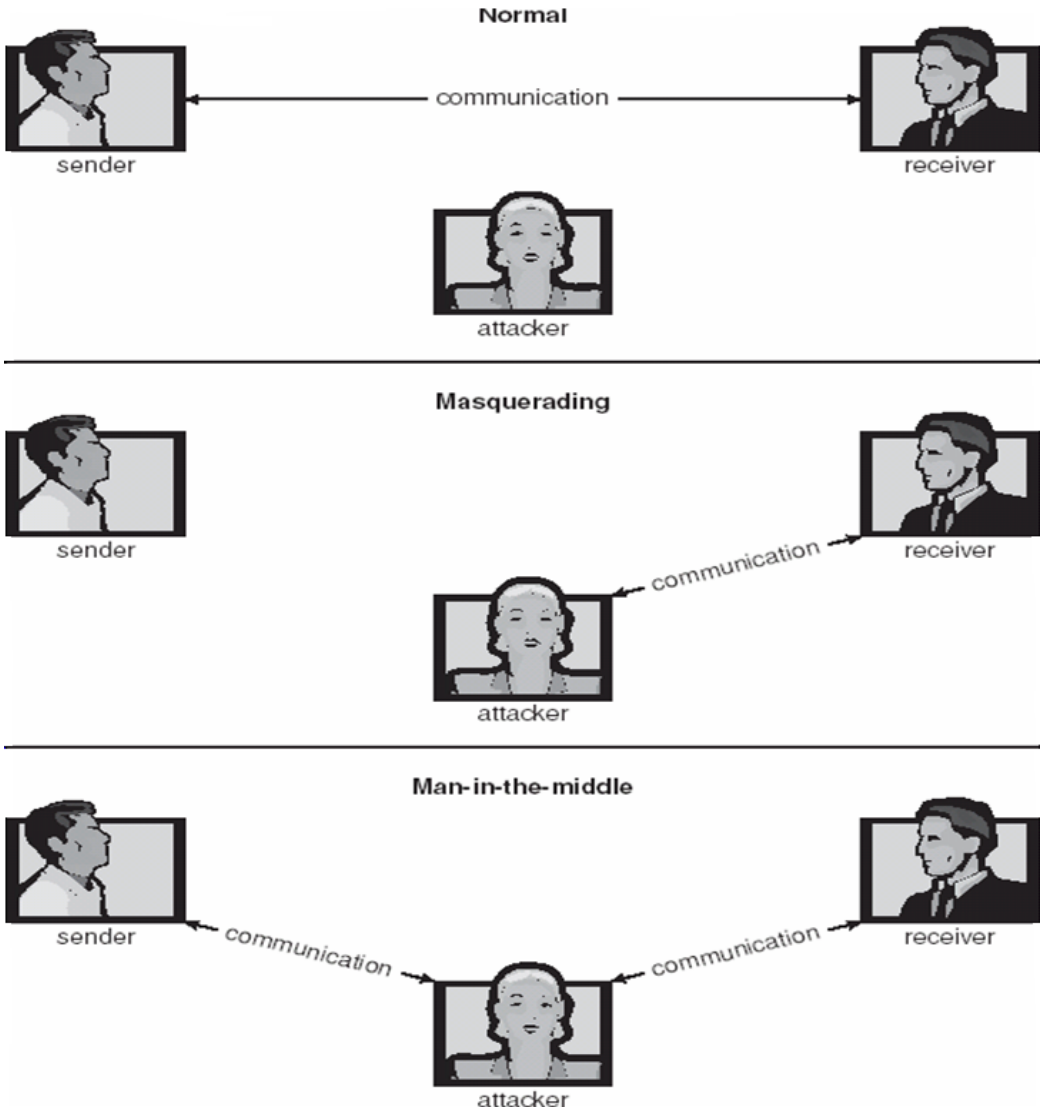
Categories

- Breach of confidentiality
- Breach of integrity
- Breach of availability
- Theft of service
- Denial of service

Methods

- Masquerading (breach authentication)
- Replay attack
- Message modification
- Man-in-the-middle attack
- Session hijacking

# Standard Security Attacks



# Security Measure Levels

- Security must occur at four levels to be effective:
  1. Physical
  2. Human
  3. Operating System
  4. Network
- Security is as weak as the weakest chain

# Program Threats

Stack and Buffer Overflow

- Exploits a bug in a program (overflow either the stack or memory buffers)

Trojan Horse

- Code segment that misuses its environment
- Exploits mechanisms for allowing programs written by users to be executed by other users
- **Spyware, pop-up browser windows, covert channels**

Trap Door

- Specific user identifier or password that circumvents normal security procedures;  could be included in a compiler

Logic Bomb

- Program that initiates a security incident under certain circumstances

# Buffer Overflow Attacks

November, 1988

- First Internet Worm spread over then-new Internet
- Many university machines compromised
- No malicious effect

Today

- Buffer overflow is still the initial entry for over 50% of network-based attacks

# String Library Code

- Implementation of Unix function `gets()`

  No way to specify limit on number of characters to read

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

  Similar problems exist with other Unix functions:

  `strcpy`: Copies string of arbitrary length

  `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

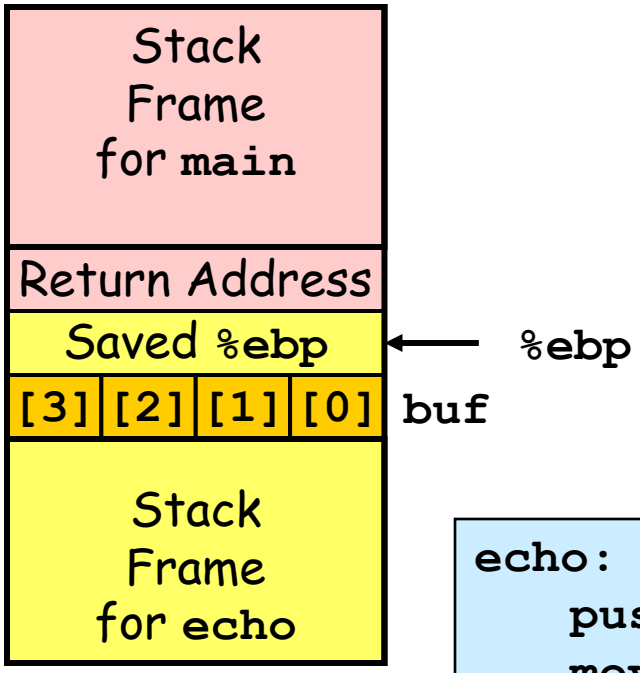
# Buffer Overflow Executions

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
12345
→ note valid output, bad input
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

# Buffer Overflow Stack (IA32)

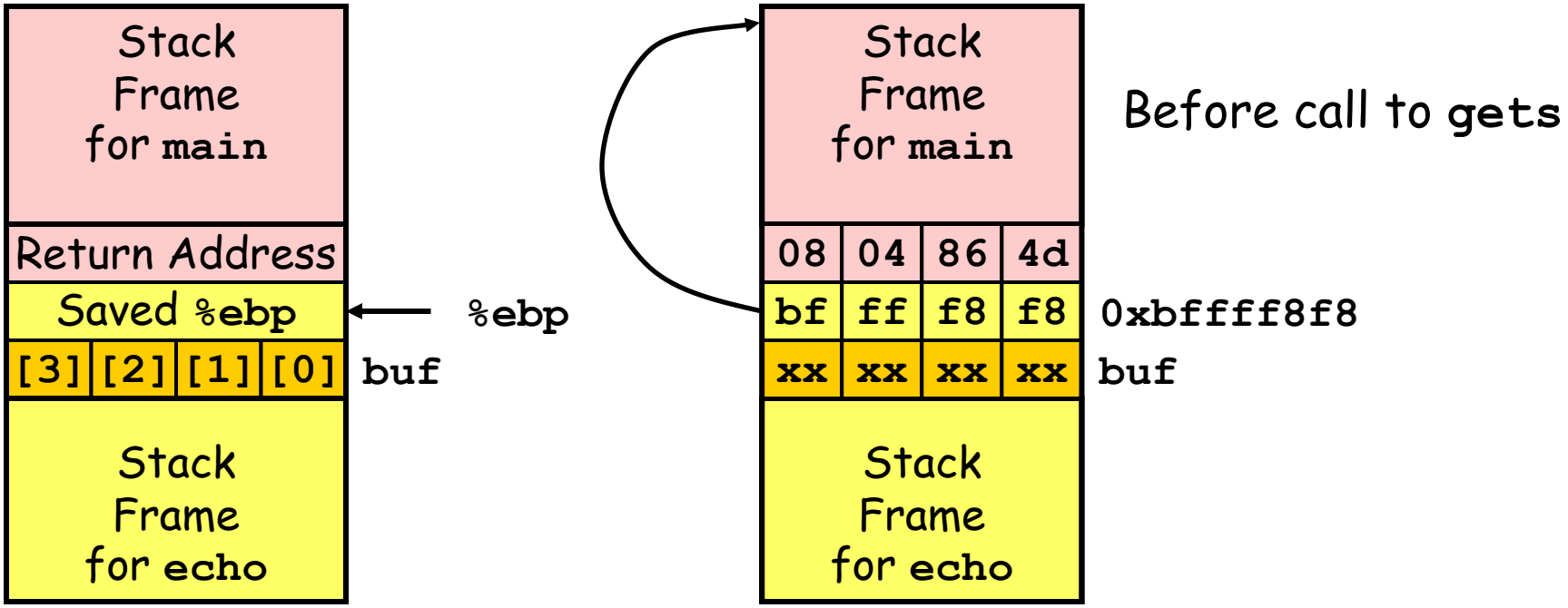



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl %esp,%ebp
    subl $20,%esp           # Allocate stack space
    pushl %ebx              # Save %ebx
    addl $-12,%esp          # Allocate stack space
    leal -4(%ebp),%ebx      # Compute buf as %ebp-4
    pushl %ebx              # Push buf on stack
    call gets               # Call gets
    . . .
```

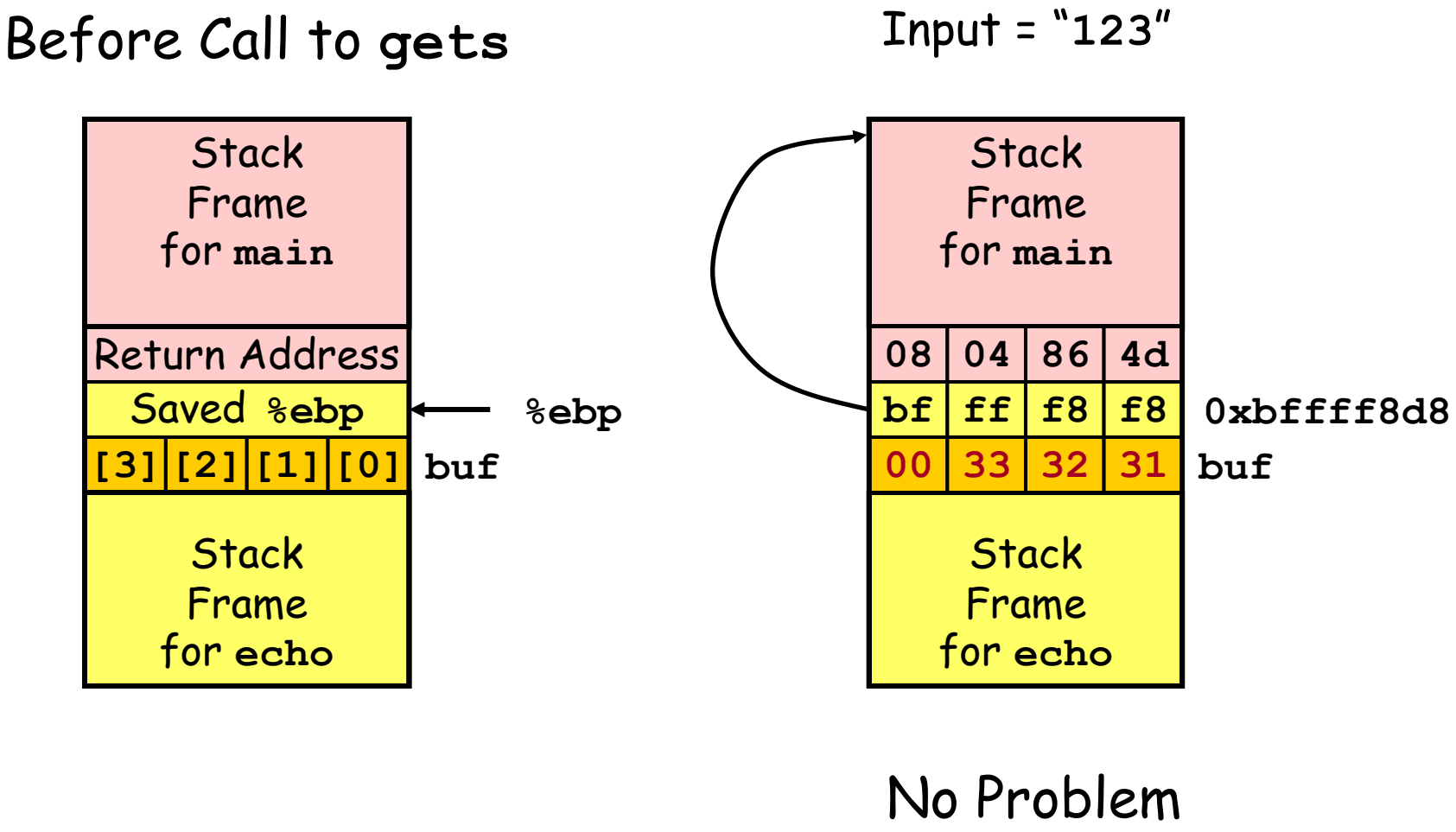
# Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

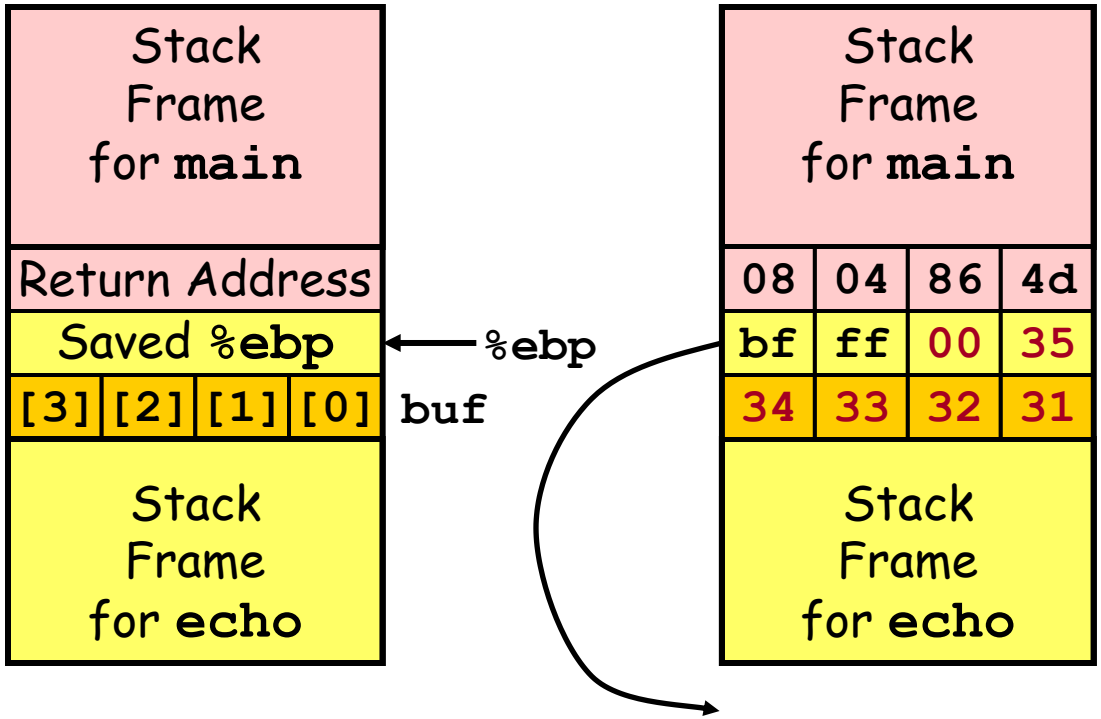



```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

# Buffer Overflow Example #1



# Buffer Overflow Stack Example #2




Input = "12345"

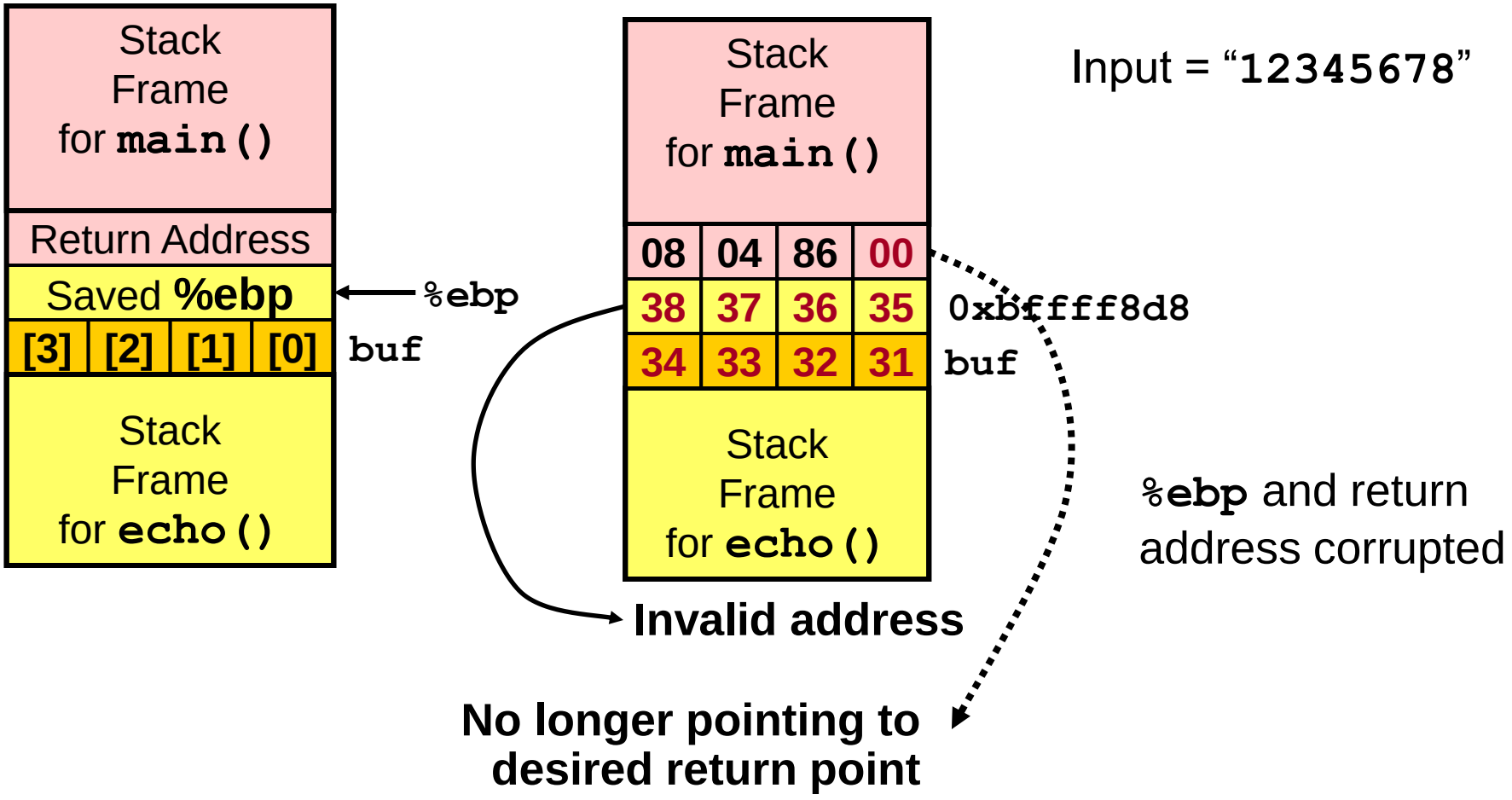
Saved value of %ebp set to 0xbfff0035

Bad news when later attempt to restore %ebp

echo code:

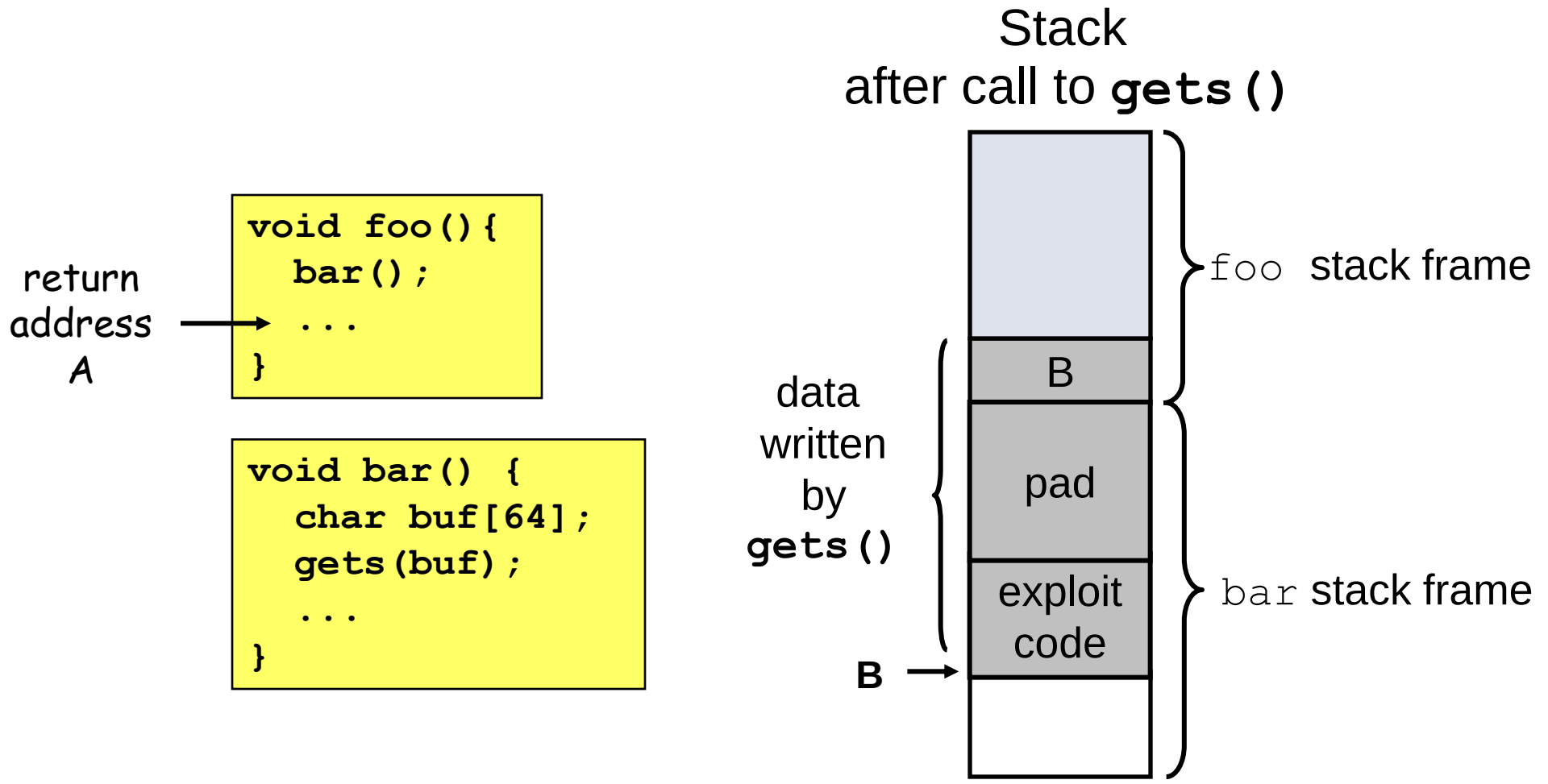
```
8048592:  push   %ebx
8048593:  call   80483e4 <_init+0x50>  # gets
8048598:  mov    0xffffffe8(%ebp),%ebx
804859b:  mov    %ebp,%esp
804859d:  pop    %ebp     # %ebp gets set to invalid value
804859e:  ret
```

# Buffer Overflow Stack Example #3




```
8048648:  call 804857c <echo>
804864d:  mov  0xffffffe8(%ebp),%ebx # Return Point
```

# Malicious Use of Buffer Overflow




- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When **`bar()`** executes **`ret`**, will jump to exploit code

# Exploits Based on Buffer Overflows

*Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines.*

Internet worm

- Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:

  **`finger neilsen@cis.ksu.edu`**

- Worm attacked fingerd server by sending phony argument:

  **`finger “exploit-code padding new-return-address”`**

  exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

# C Call Stack

- When a function call is made, the return address is pushed onto the stack.
- Often the values of parameters passed to the function are put onto the stack (call-by-value).
- Usually the function saves the stack frame pointer (old %ebp) on the stack.
- Local variables are placed on the stack.

# Stack Direction

- On Linux (x86) the stack grows from high addresses to low.
- Pushing something onto the stack moves the Top Of the Stack (%esp) towards address 0.

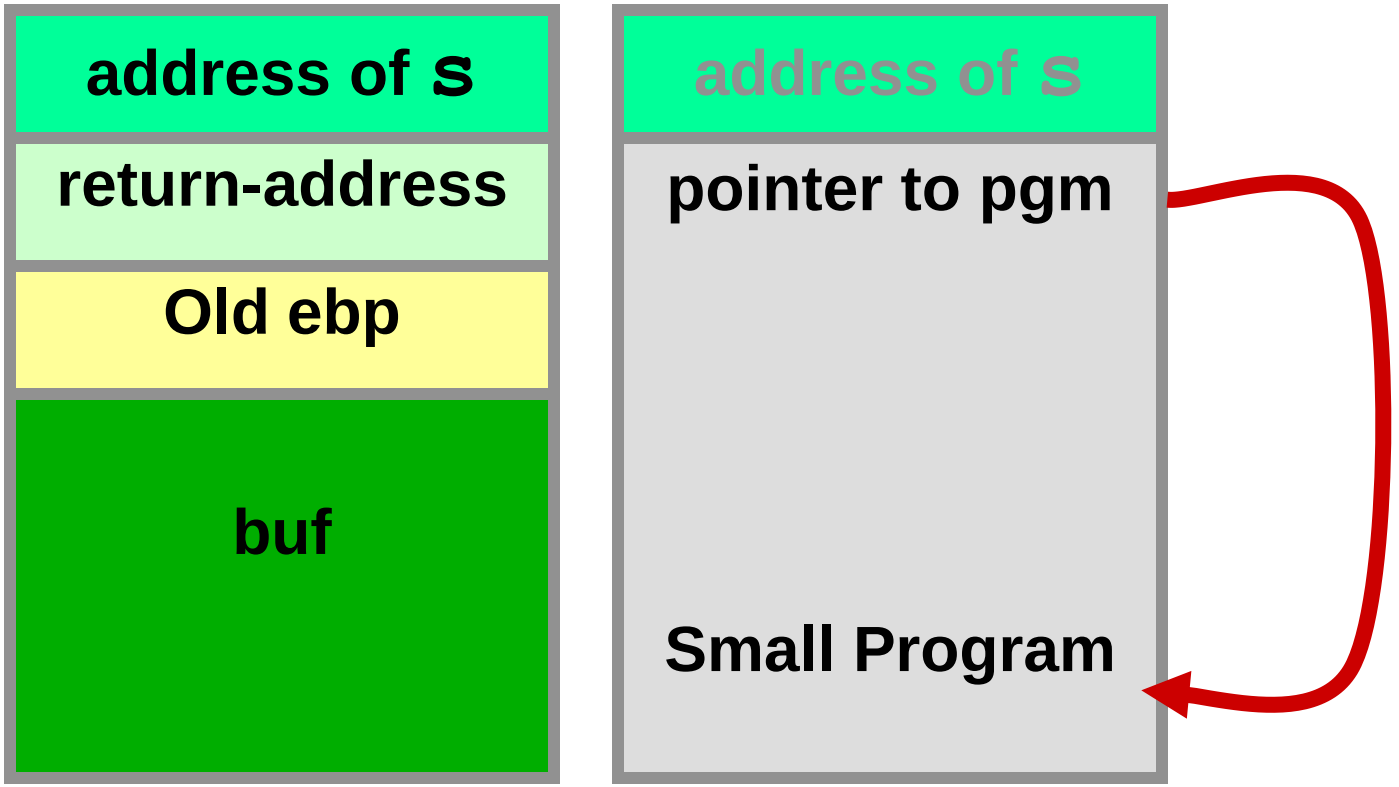
# “Smashing the Stack”*

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

*taken from the title of an article in Phrack 49-7

# Before and After

```
void foo(char *s) {
      char buf[100];
      strcpy(buf,s);
      …
```



# Issues

- How do we know what value the pointer should have (the new “return address”).
- It’s the address of the buffer, but how do we know what address this is?
- How do we build the “small program” and put it in a string?

# Guessing Addresses

- Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.
- An estimate is often good enough!

# Building the small program

- Typically, the small program stuffed in to the buffer does an `exec()`.
- Sometimes it changes the password file or other files…

# exec()

- In Unix, the way to run a new program is with an **`exec()`** system call.
  - There is actually a *family* of **`exec()`** system calls…
  - This doesn't create a new process, it changes the current process to run a new program.
  - To create a new process you need another system call ( e.g., **`fork()`** ).

# `exec()` example

```
#include <stdio.h>

void execls(void) {
  execl("/bin/ls", "ls", NULL);
  printf("Line not printed if execl is
          successful.\n");
}
```

# Generating a String

- You can take code like the previous slide, and generate machine language.
- Copy down the individual byte values and build a string.
- To do a simple exec( ) requires less than 100 bytes.

# Some important issues

- The small program should be position-independent – able to run at any memory location.
- Statically link the libraries to see the code generated for the exec( ) system call; e.g., gcc execExample.c, to see how the exec( ) system call is made. To statically link the libraries, use:

  gcc –static execExample.c

- It can't be too large, or we can't fit the program and the new return-address on the stack!

# A Sample Program/String

Does an exec( ) of /bin/ls:

```
unsigned char cde[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
“\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
“\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/ls";
```

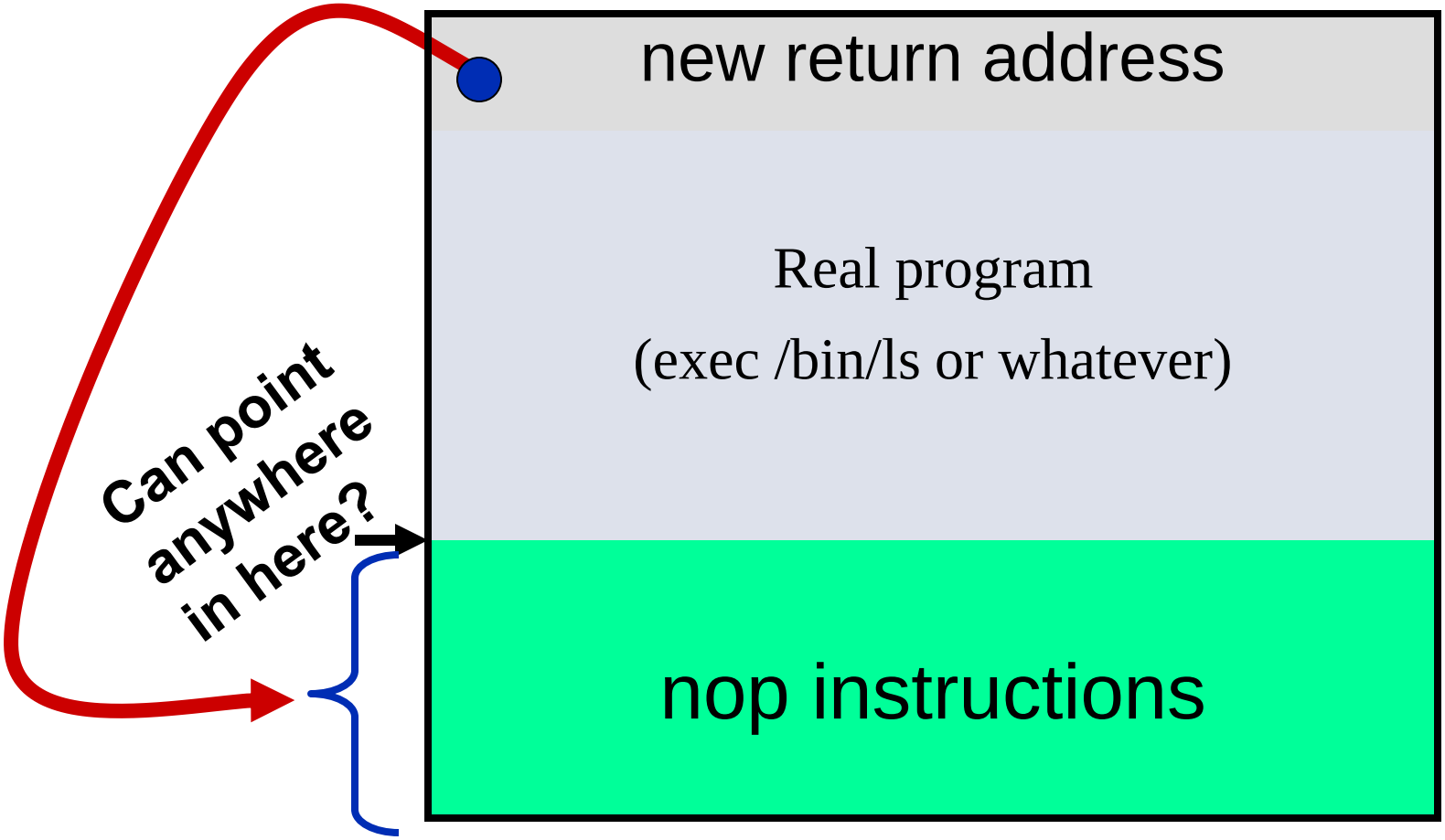
# Attacking a real program

- Recall that the idea is to feed a server a string that is too big for a buffer.
- This string overflows the buffer and overwrites the return address on the stack.
- Assuming we put our small program in the string, we need to know it’s address.

# NOPs

- Most CPUs have a *No-Operation* instruction – it does nothing but advance the instruction pointer.
- Usually we can put a bunch of these ahead of our program (in the string).
- As long as the new return-address points to a NOP we are OK.

# Using NOPs

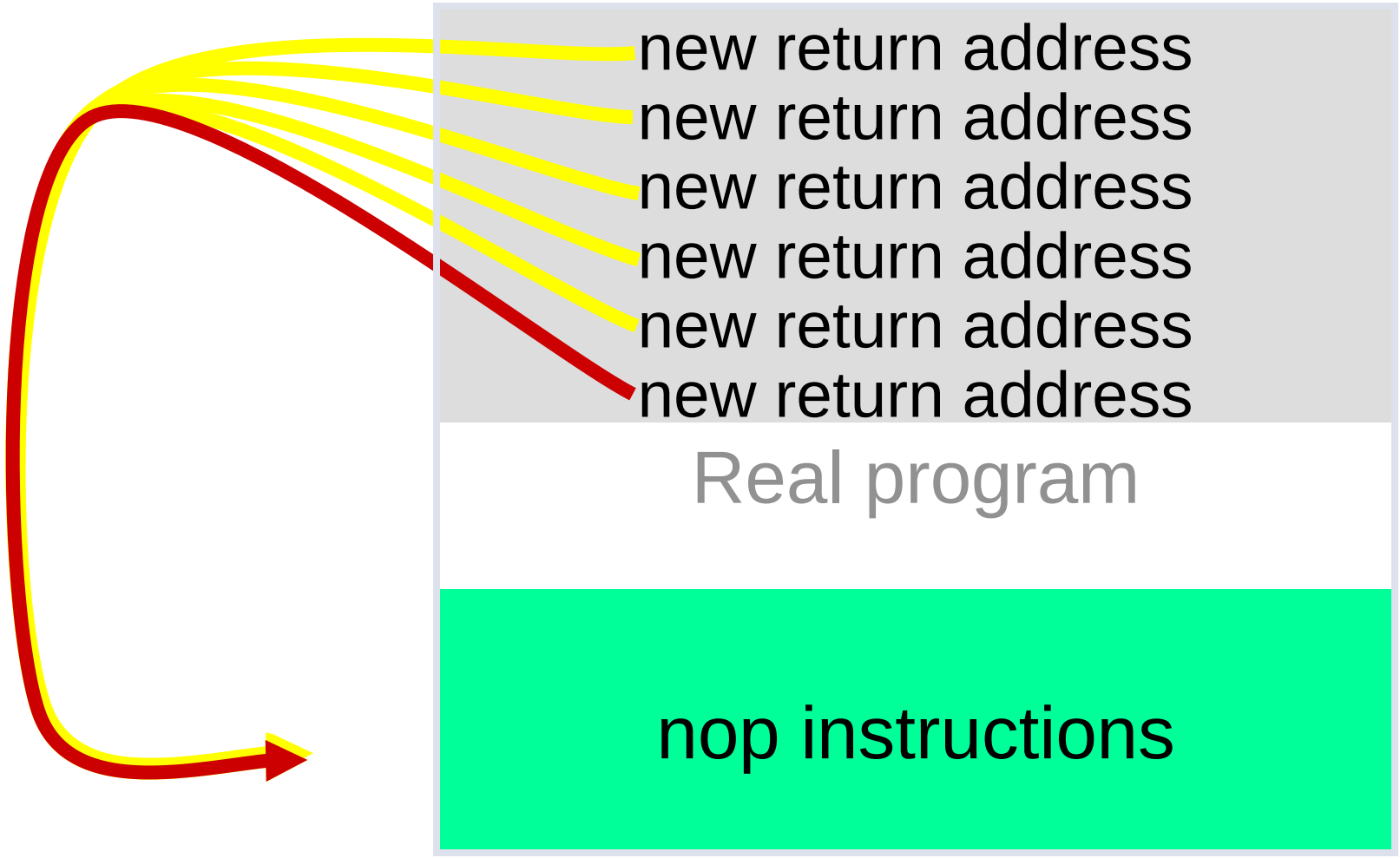

new return address
Real program
(exec /bin/ls or whatever)
Can point anywhere in here?
nop instructions

# Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.

- Put in a bunch of new return addresses!

# Estimating the Location



# Summary

- Read Ch 13-15
- Quiz #2 – Wed., Nov. 20, open-book, open-notes
- Project #3
  - Application
  - Kernel modifications – add new system call