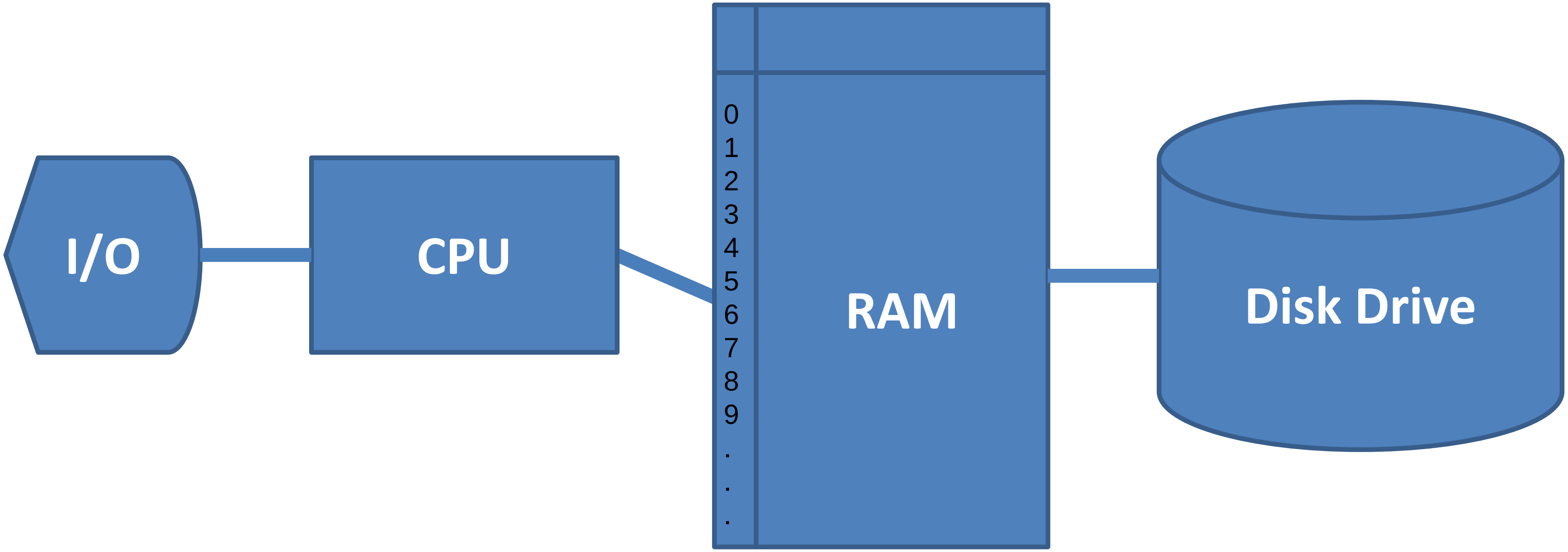
Operating Systems Security

UNIVERSITY OF
MARYLAND

A Computer Model

- An operating system has to deal with the fact that a computer is made up of a CPU, random access memory (RAM), input/output (I/O) devices, and long-term storage.

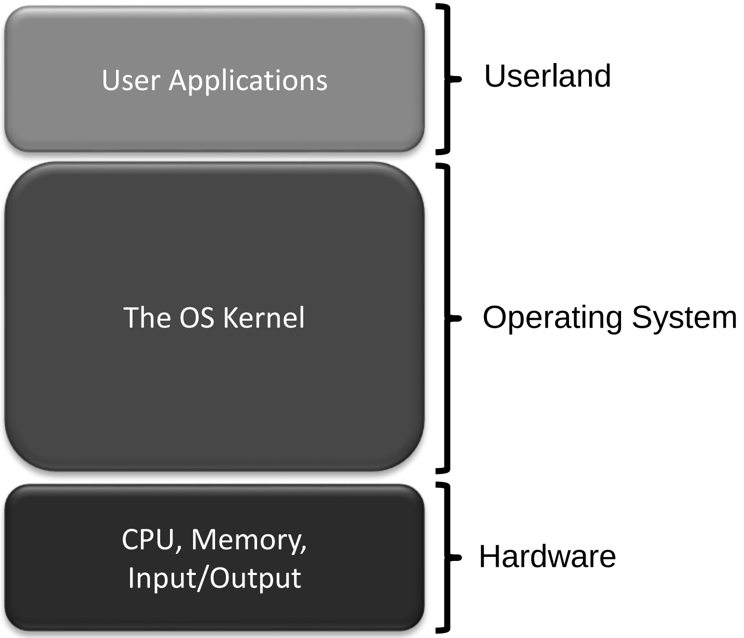


OS Concepts

- An **operating system (OS)** provides the interface between the users of a computer and that computer's hardware.
  - An operating system manages the ways applications access the resources in a computer, including its disk drives, CPU, main memory, input devices, output devices, and network interfaces.
  - An operating system manages multiple users.
  - An operating system manages multiple programs.

The Kernel

- The **kernel** is the core component of the operating system. It handles the management of low-level hardware resources, including memory, processors, and input/output (I/O) devices, such as a keyboard, mouse, or video display.
- Most operating systems define the tasks associated with the kernel in terms of a **layer** metaphor, with the hardware components, such as the CPU, memory, and input/output devices being on the bottom, and users and applications being on the top.



System Calls

- User applications don’t communicate directly with low-level hardware components, and instead delegate such tasks to the kernel via **system calls**.
- System calls are usually contained in a collection of programs, that is, a **library** such as the C library (libc), and they provide an interface that allows applications to use a predefined series of APIs that define the functions for communicating with the kernel.
  – Examples of system calls include those for performing file I/O (open, close, read, write) and running application programs (exec).

Processes

- A **process** is an instance of a program that is currently executing.
- The actual contents of all programs are initially stored in persistent storage, such as a hard drive.
- In order to be executed, a program must be loaded into random-access memory (RAM) and uniquely identified as a process.
- In this way, multiple copies of the same program can be run as different processes.
  - For example, we can have multiple copies of MS Powerpoint open at the same time.

Process IDs

- Each process running on a given computer is identified by a unique nonnegative integer, called the **process ID (PID).**
- Given the PID for a process, we can then associate its CPU time, memory usage, user ID (UID), program name, etc.

File Systems

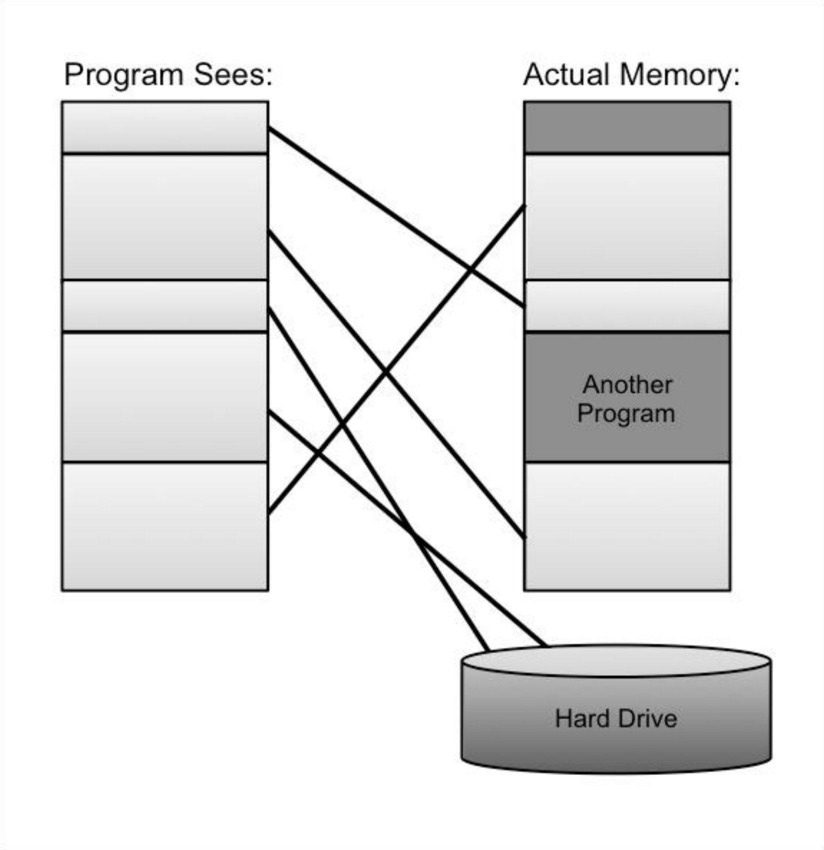
- A **filesystem** is an abstraction of how the external, nonvolatile memory of the computer is organized.
- Operating systems typically organize files hierarchically into **folders,** also called **directories.**
- Each folder may contain files and/or subfolders.
- Thus, a volume, or drive, consists of a collection of nested folders that form a **tree**.
- The topmost folder is the **root** of this tree and is also called the root folder.

Memory Management

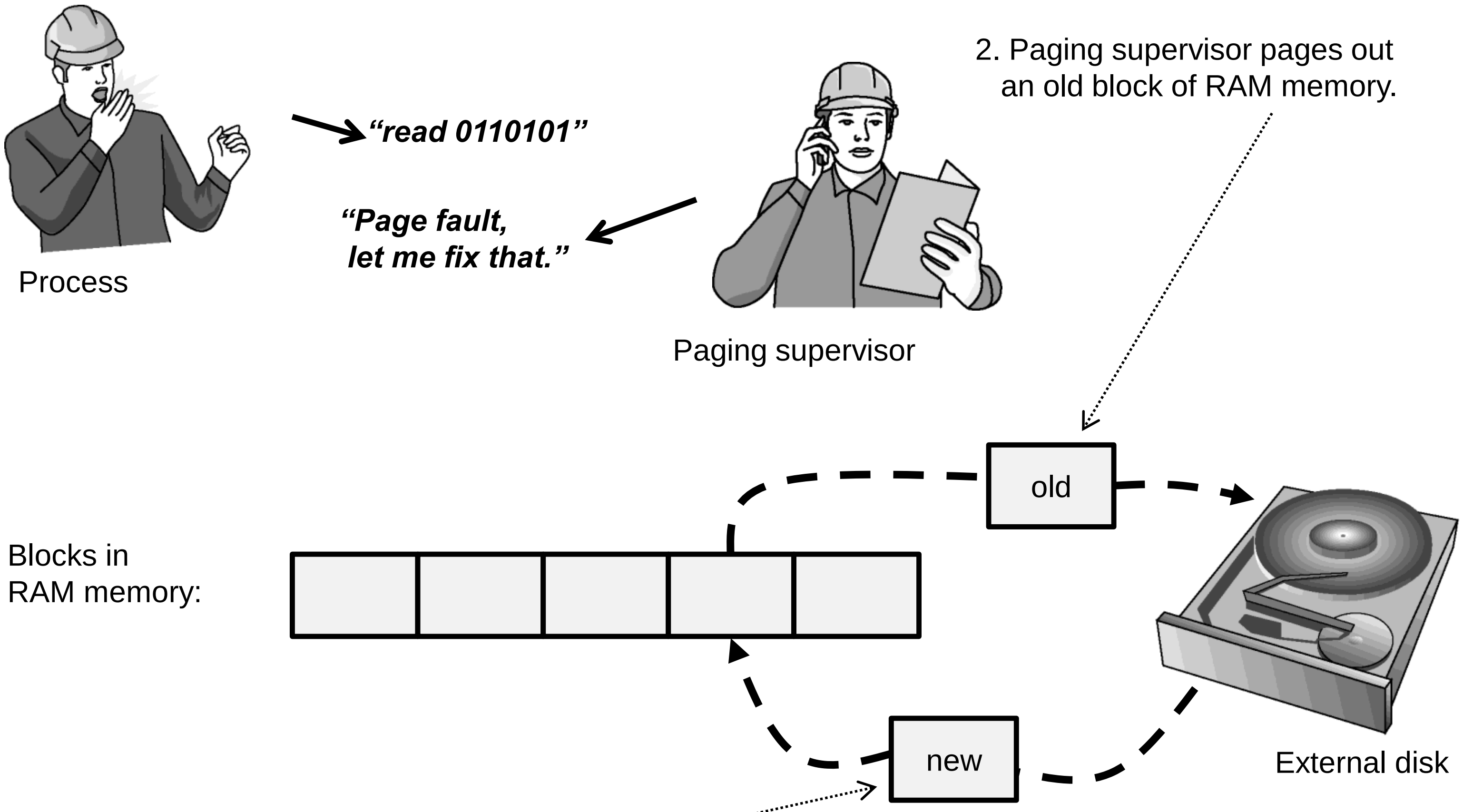
- The RAM memory of a computer is its **address space.**
- It contains both the code for the running program, its input data, and its working memory.
- For any running process, it is organized into different segments, which keep the different parts of the address space separate.
- As we will discuss, security concerns require that we never mix up these different segments.

Virtual Memory

- There is generally not enough computer memory for the address spaces of all running processes.
- Nevertheless, the OS gives each running process the illusion that it has access to its complete (contiguous) address space.
- In reality, this view is **virtual**, in that the OS supports this view, but it is not really how the memory is organized.
- Instead, memory is divided into **pages**, and the OS keeps track of which ones are in memory and which ones are stored out to disk.



Page Faults



1. Process requests virtual address not in memory, causing a page fault.

Process

"read 0110101"

"Page fault, let me fix that."

Paging supervisor

2. Paging supervisor pages out an old block of RAM memory.

old

Blocks in
RAM memory:

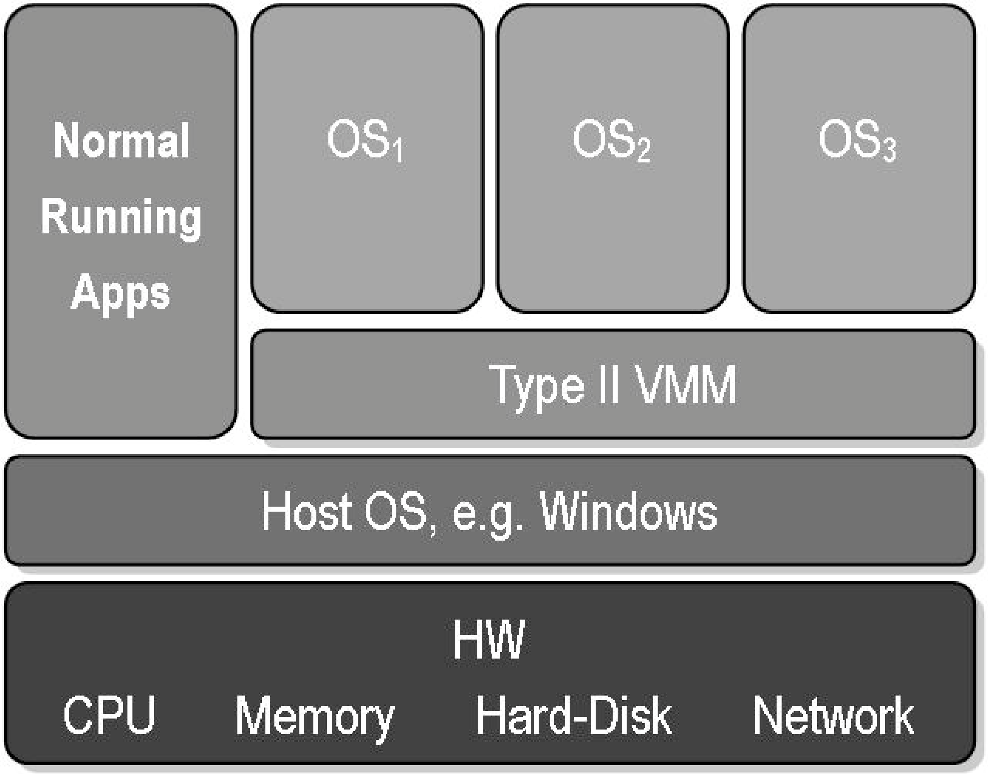
new

External disk

3. Paging supervisor locates requested block on the disk and brings it into RAM memory.

Virtual Machines

- **Virtual machine:** A view that an OS presents that a process is running on a specific architecture and OS, when really it is something else. E.g., a windows emulator on a Mac.
- **Benefits:**
  - **Hardware Efficiency**
  - **Portability**
  - **Security**
  - **Management**



Public domain image from http://commons.wikimedia.org/wiki/File:VMM-Type2.JPG

Buffer Overflow Attacks

What is an Exploit?

- An exploit is any input (i.e., a piece of software, an argument string, or sequence of commands) that takes advantage of a bug, glitch or vulnerability in order to cause an attack
- An attack is an unintended or unanticipated behavior that occurs on computer software, hardware, or something electronic and that brings an advantage to the attacker

Buffer Overflow Attack

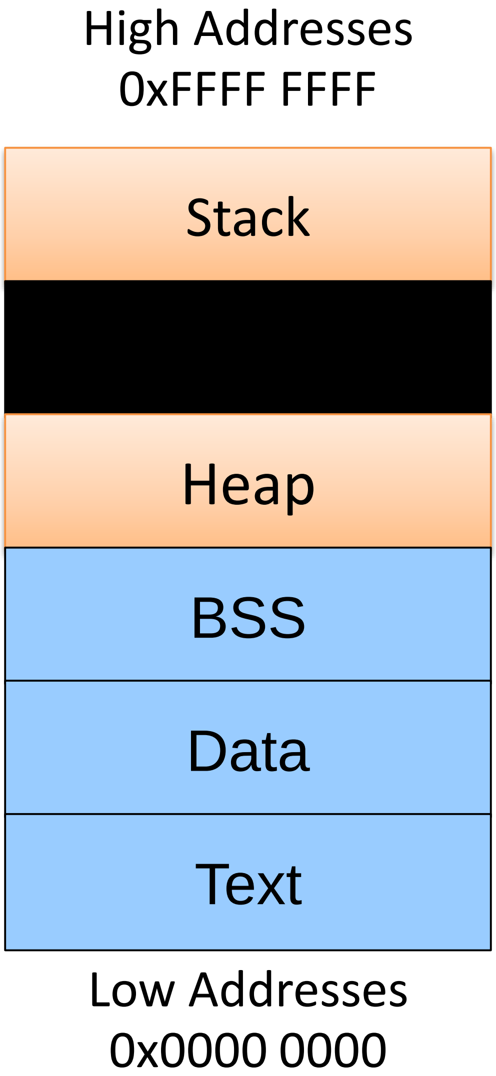
- One of the most common OS bugs is a buffer overflow
  - The developer fails to include code that checks whether an input string fits into its buffer array
  - An input to the running process exceeds the length of the buffer
  - The input string overwrites a portion of the memory of the process
  - Causes the application to behave improperly and unexpectedly
- Effect of a buffer overflow
  - The process can operate on malicious data or execute malicious code passed in by the attacker
  - If the process is executed as root, the malicious code will be executing with root privileges

Address Space

- Every program needs to access memory in order to run
- For simplicity sake, it would be nice to allow each process (i.e., each executing program) to act as if it owns all of memory
- The address space model is used to accomplish this
- Each process can allocate space anywhere it wants in memory
- Most kernels manage each process' allocation of memory through the virtual memory model
- How the memory is managed is irrelevant to the process

Unix Address Space

- Text: machine code of the program, compiled from the source code
- Data: static program variables initialized in the source code prior to execution
- BSS (block started by symbol): static variables that are uninitialized
- Heap : data dynamically generated during the execution of a process
- Stack: structure that grows downwards and keeps track of the activated method calls, their arguments and local variables


High Addresses
0xFFFF FFFF

Stack

Heap

BSS

Data

Text

Low Addresses
0x0000 0000

Vulnerabilities and Attack Method

- Vulnerability scenarios
  - The program has root privileges (setuid) and is launched from a shell
  - The program is part of a web application
- Typical attack method
  1. Find vulnerability
  2. Reverse engineer the program
  3. Build the exploit

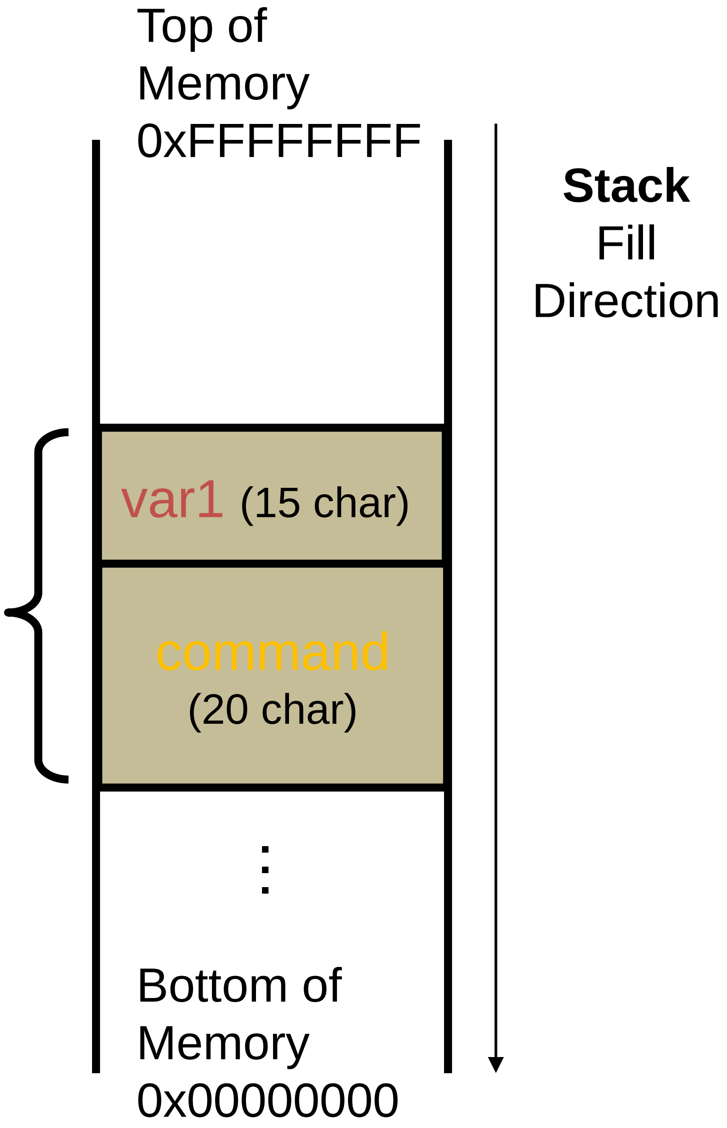
Buffer Overflow Attack in a Nutshell

- First described in

  Aleph One. Smashing The Stack For Fun And Profit. e-zine www.Phrack.org #49, 1996
- The attacker exploits an unchecked buffer to perform a buffer overflow attack
- The ultimate goal for the attacker is getting a shell that allows to execute arbitrary commands with high privileges
- Kinds of buffer overflow attacks:
  - Heap smashing
  - Stack smashing

Buffer Overflow

```
domain.c
Main(int argc, char *argv[ ])
/* get user_input */
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```



Top of
Memory
0xFFFFFFFF

Stack
Fill
Direction

var1 (15 char)

command
(20 char)

⋮

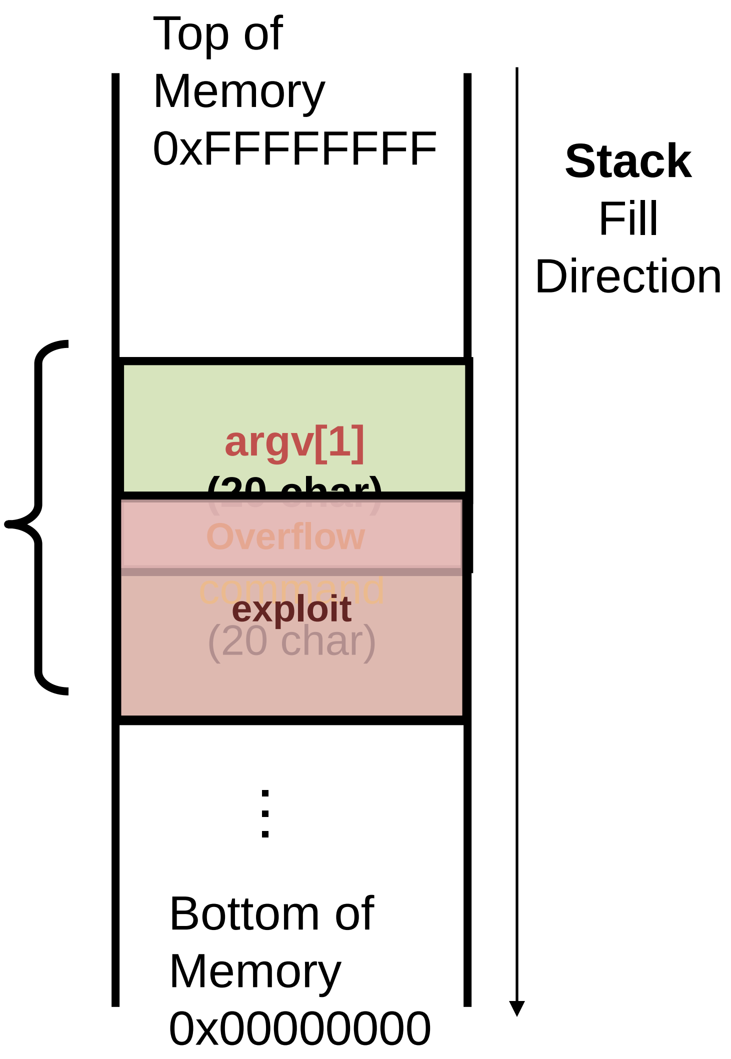
Bottom of
Memory
0x00000000

strcpy() Vulnerability

domain.c

```
Main(int argc, char *argv[])
/*get user_input*/
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```

- argv[1] is the user input
- strcpy(dest, src) does not check buffer
- strcat(d, s) concatenates strings



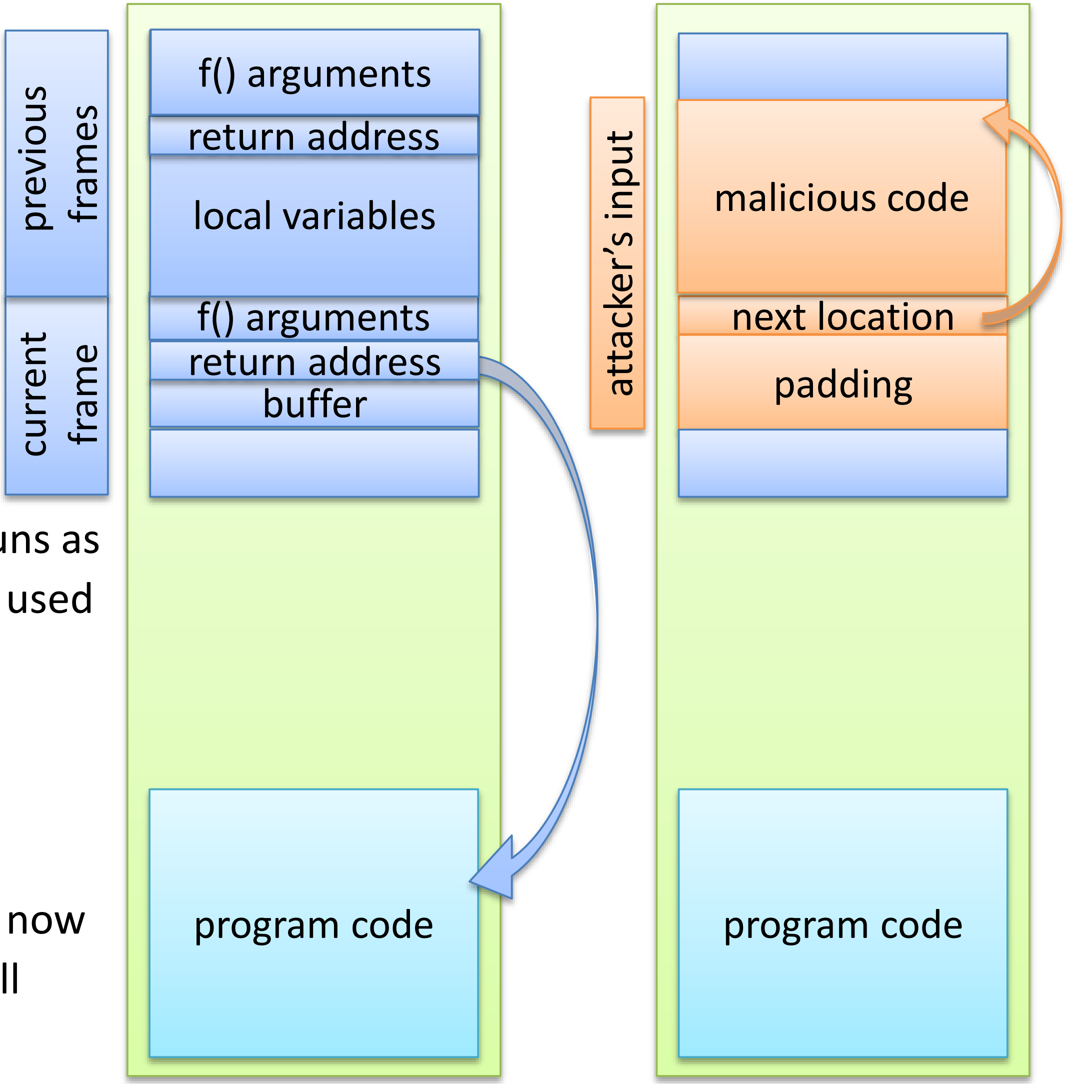
strcpy() vs. strncpy()

- Function strcpy() copies the string in the second argument into the first argument
  - e.g., strcpy(dest, src)
  - If source string > destination string, the overflow characters may occupy the memory space used by other variables
  - The null character is appended at the end automatically
- Function strncpy() copies the string by specifying the number **n** of characters to copy
  - e.g., strncpy(dest, src, n); dest[n] = ‘\0’
  - If source string is longer than the destination string, the overflow characters are discarded automatically
  - You have to place the null character manually

Return Address Smashing

```
void fingerd (…) {
            char buf[80];
            …
            get(buf);
            …
}
```

- The Unix fingerd() system call, which runs as root (it needs to access sensitive files), used to be vulnerable to buffer overflow
- Write malicious code into buffer and overwrite return address to point to the malicious code
- When return address is reached, it will now execute the malicious code with the full rights and privileges of root



Shellcode Injection

- An exploit takes control of attacked computer so injects code to “spawn a shell” or “shellcode”
- A shellcode is:
  - Code assembled in the CPU’s native instruction set (e.g. x86 , x86-64, arm, sparc, risc, etc.)
  - Injected as a part of the buffer that is overflowed.
- We inject the code directly into the buffer that we send for the attack
- A buffer containing shellcode is a “payload”

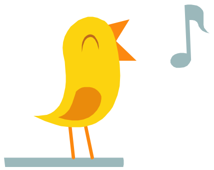
Buffer Overflow Mitigation

- We know how a buffer overflow happens, but why does it happen?
- This problem could not occur in Java; it is a C problem
  - In Java, objects are allocated dynamically on the heap (except ints, etc.)
  - Also cannot do pointer arithmetic in Java
  - In C, however, you can declare things directly on the stack
- One solution is to make the buffer dynamically allocated
- Another (OS) problem is that fingerd had to run as root
  - Just get rid of fingerd's need for root access (solution eventually used)
  - The program needed access to a file that had sensitive information in it
  - A new world-readable file was created with the information required by fingerd

Stack-based buffer overflow detection using a random canary

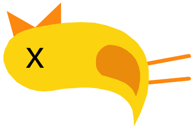
Normal (safe) stack configuration:

Buffer	**Other local variables**	**Canary (random)**	**Return address**	**Other data**



Buffer overflow attack attempt:

Buffer	**Overflow data**	**Corrupt return address**	**Attack code**



- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.