

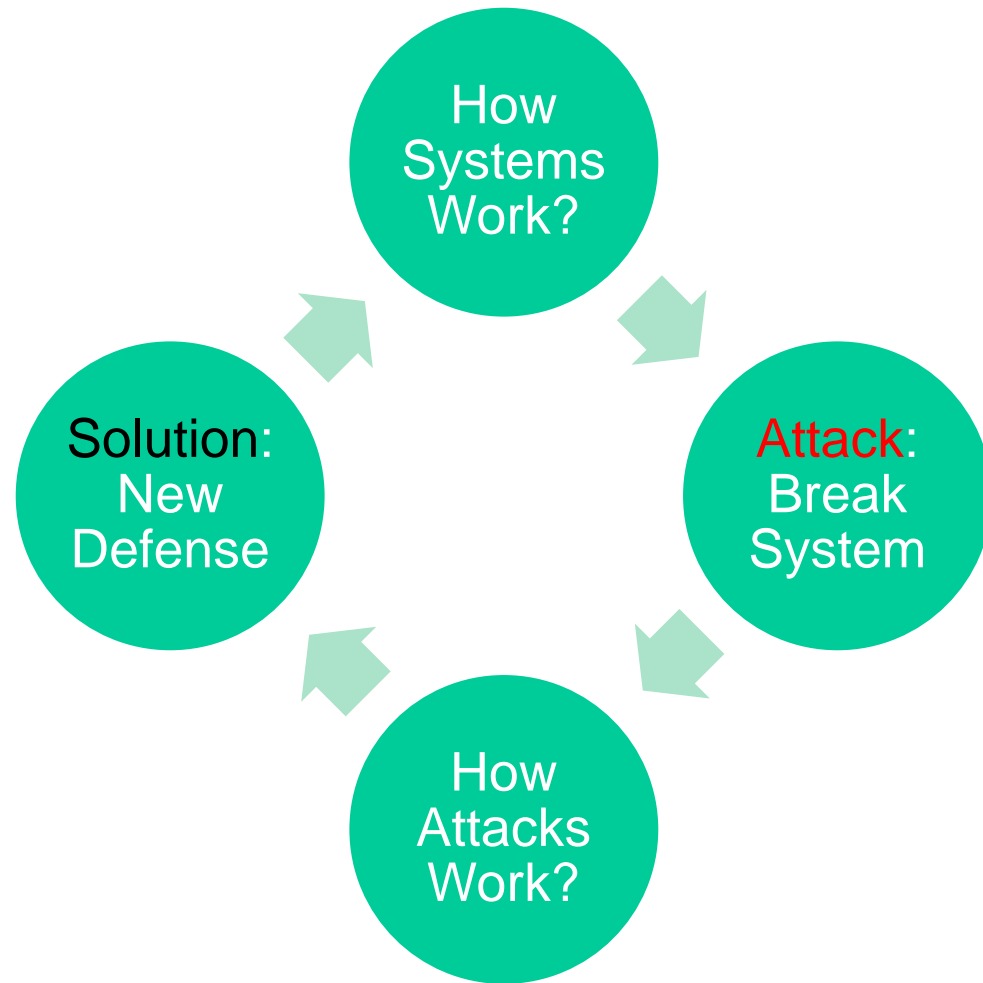
CS5231: Systems Security

Lecture 2: Buffer Overflow Attacks

Buffer Overflow

- Buffer overflows are extremely common.
- They offer attackers a way to access and control the victim system in a significant degree.
 - Attackers run malicious code discussed in last lecture.
- This vulnerability is caused by failure in input checking, allowing attackers to put more data than the memory allocated for it.

Methodology

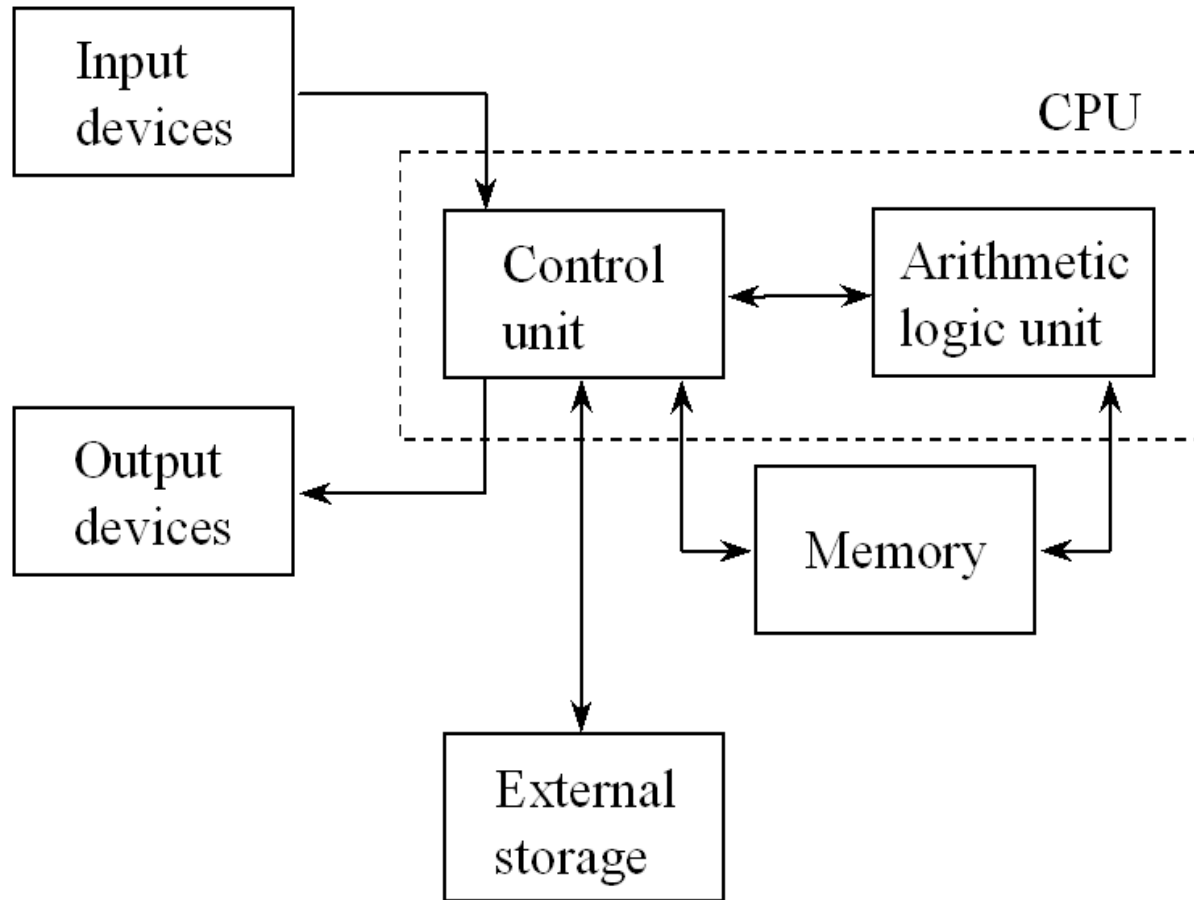


Lecture Outline

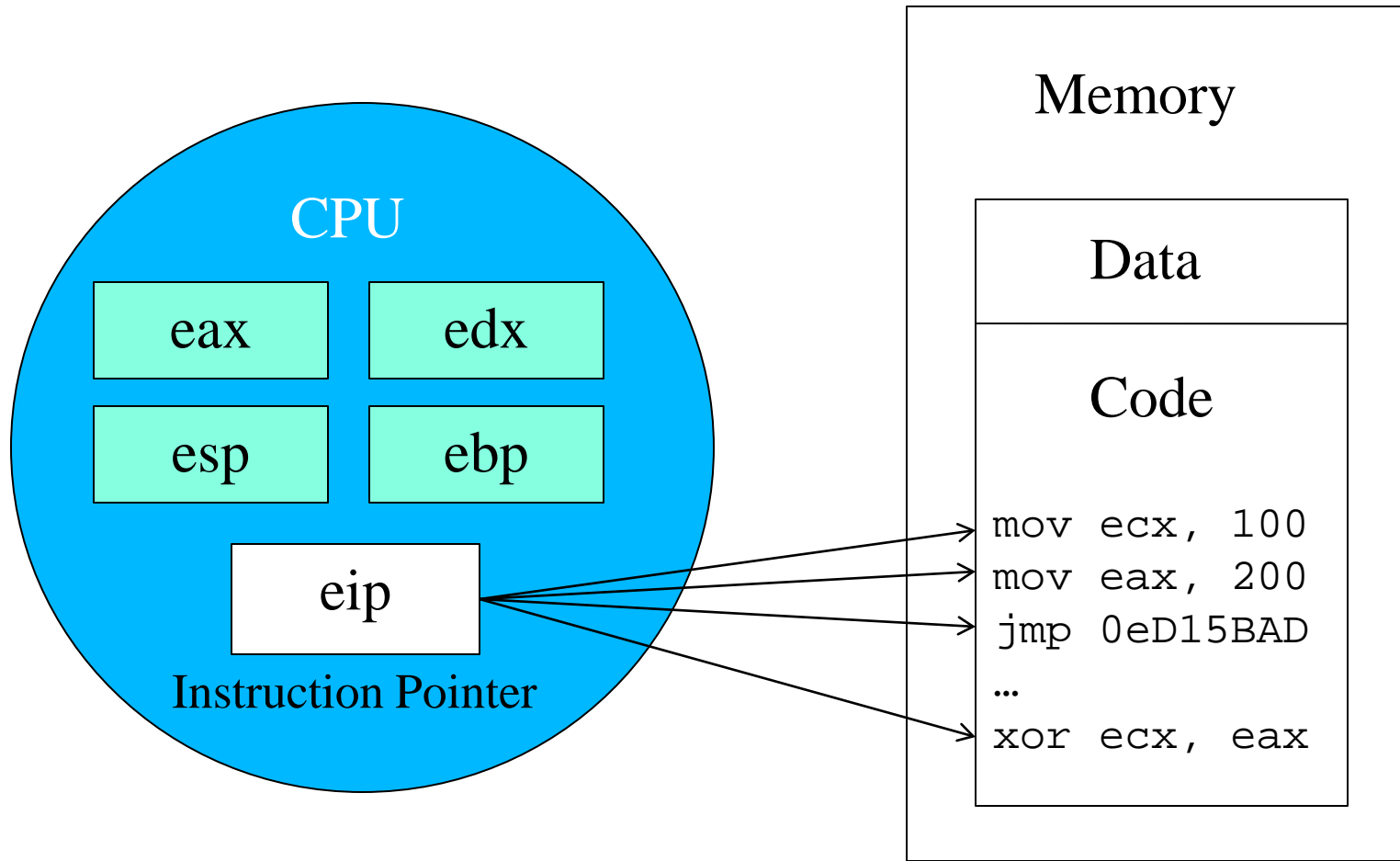
- Function call mechanism
- Buffer overflow attacks
- Defense mechanisms of buffer overflow

Function Call Mechanism

Von Neumann Architecture



CPU and Memory

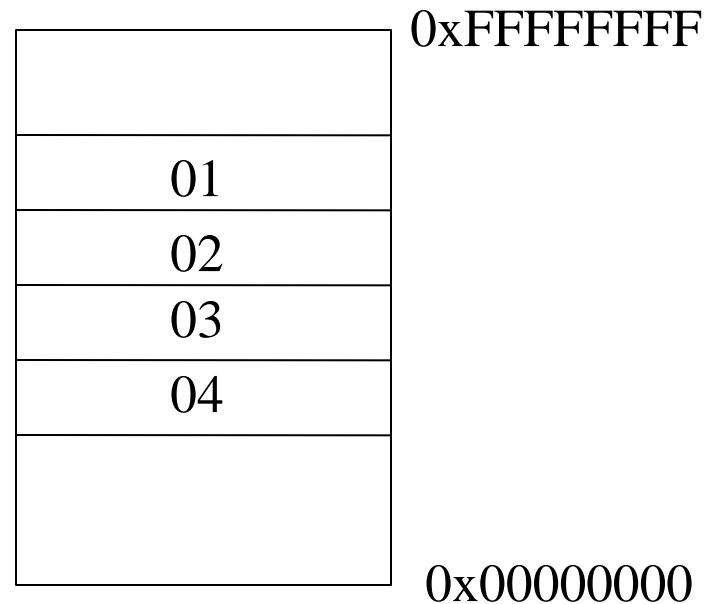


Program Representation in Memory

- Both code and data are represented as numbers
- Code
 - `lea ecx, [esp+4]` represented as `0x8d 0x4c 0x24 0x04`

- Data

- On Intel CPUs, least significant bytes is put at lower addresses
- It is called little endian
- For example, `0x01020304`



OS Memory Management

- Terminology
 - Physical Address vs. Logical Address
 - Address Binding
 - Page Table
 - TLB
- Virtual Memory Management
 - Page Fault / Demand Paging

Function Calls

- Functions break code into smaller pieces
 - Facilitating modular design and code reuse
- A function can be called in many program locations
- How does it know where it should continue after it finishes?

```
void sample_function(void)
{
    char buffer[10];
    printf("Hello!\n");
    return;
}

→ main()
{
    sample_function();
    printf("Loc 1\n");
    sample_function();
    printf("Loc 2\n");
}
```

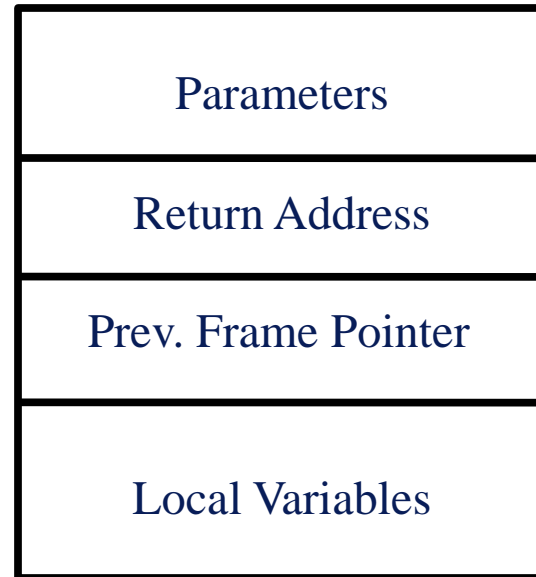
Stack

- A data structure storing important information for each process running on a computer
- Last in, first out (LIFO)
- Stack operations
 - push
 - pop
 - top
- In Intel systems, stack grows from high address to low address



Activation Record

- Each call of a function has an activation record (stack frame)
 - Local variables
 - Previous frame pointer
 - Return address
 - Parameters



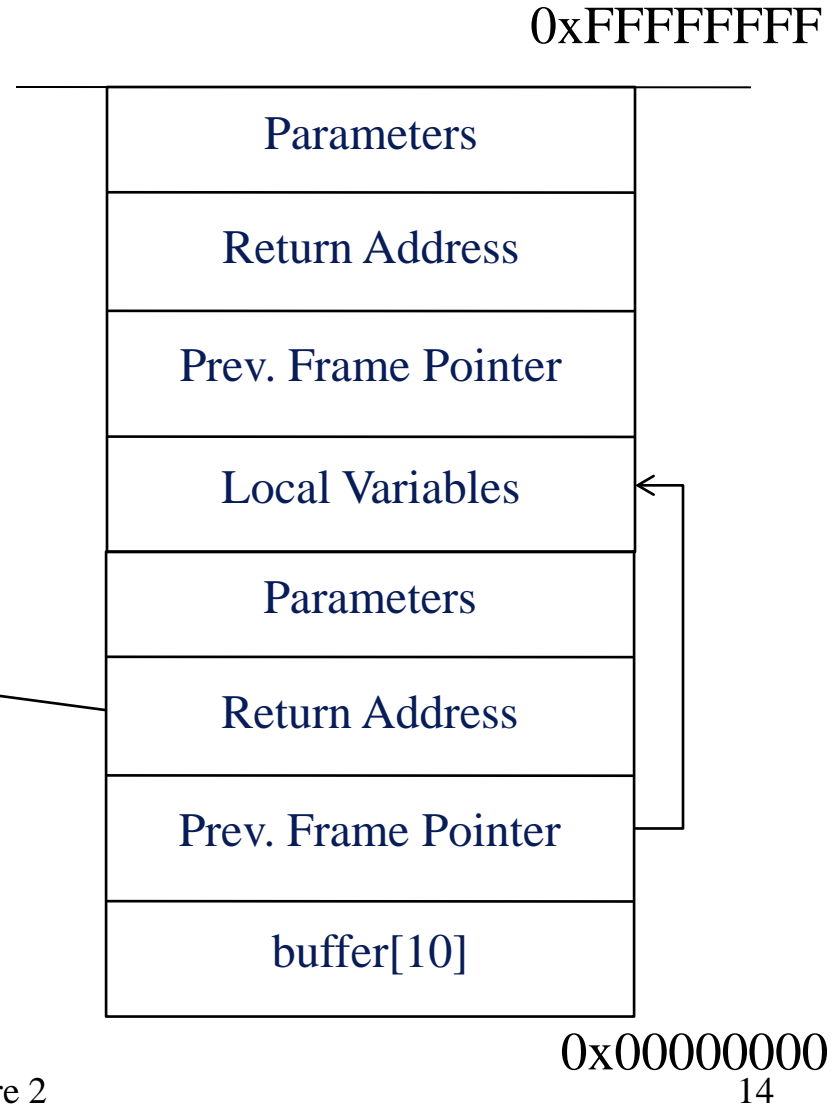
Steps of Call and Return

- Caller
 - Save registers
 - Push parameters on stack
 - Push return address on stack
 - Jump to the beginning of function
- Callee
 - (Optional) Save frame pointer (EBP), and set frame pointer to stack top
 - Allocate local variables
- Callee
 - Set return values
 - Deallocate local variables
 - (Optional) restore frame pointer
 - Jump to the return address on stack
- Caller
 - Get return values
 - Pop parameters from stack
 - Restore saved registers

Stack Action Illustrated

```
void sample_function(void)
{
    char buffer[10];
    → printf("Hello!\n");
    return;
}

main()
{
    → sample_function();
    → printf("Loc 1\n");
    sample_function();
    printf("Loc 2\n");
}
```



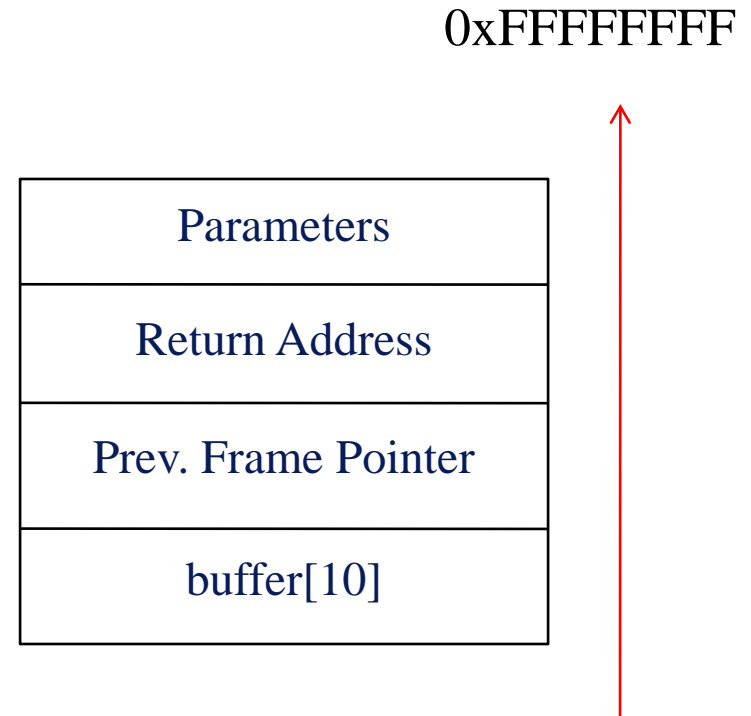
See It Using gdb

- Compile program and run gdb
 - `gcc -o sample -g sample.c`
 - `gdb sample`
- gdb commands
 - Set break point: `break <functionname>`
 - Check register values: `info registers`
 - Check variables: `print <variablename>`
 - Inspect memory: `x/b <variable_or_address>`

Buffer Overflow Exploits

Observation

- Buffer grows toward return address
- If we more than 10 bytes for array buffer, the content will spill into adjacent memory region, previous frame pointer, then return address

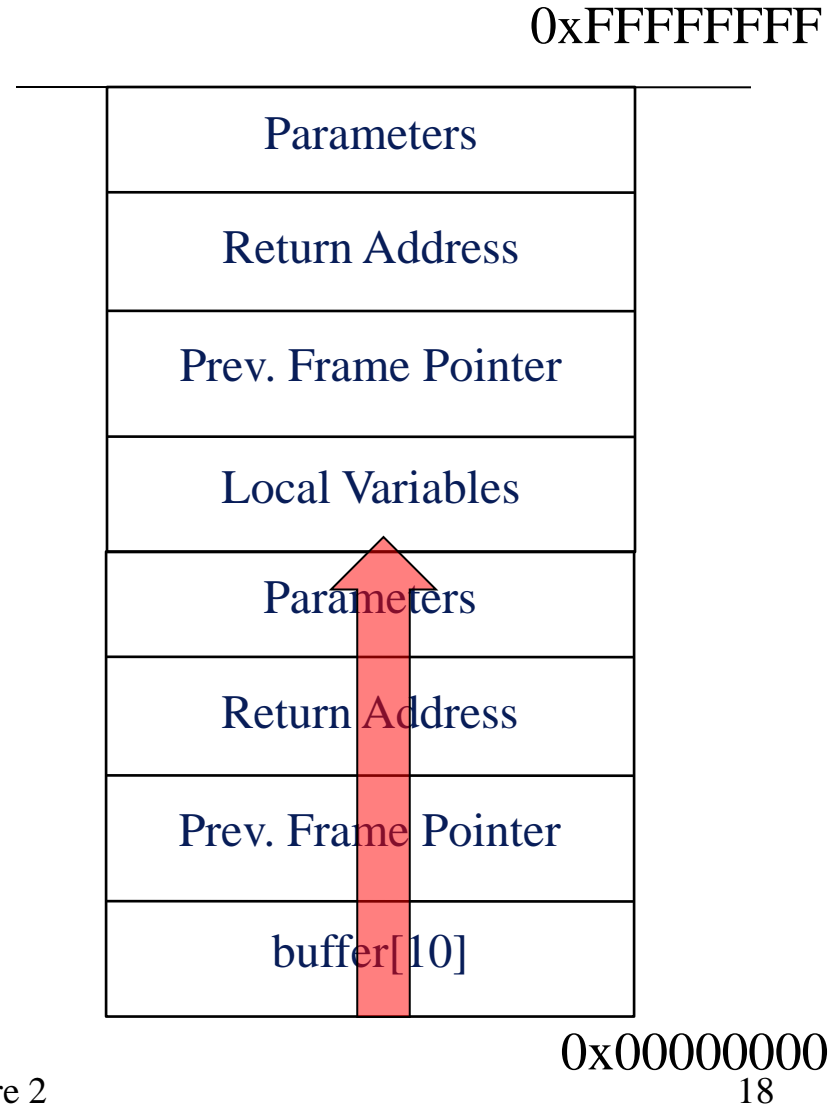


0x00000000
17

Buffer Overflow

```
void sample_function(void)
{
    char buffer[10];
    → gets(buffer);
    return;
}

main()
{
    sample_function();
    printf("Loc 1\n");
    sample_function();
    printf("Loc 2\n");
}
```

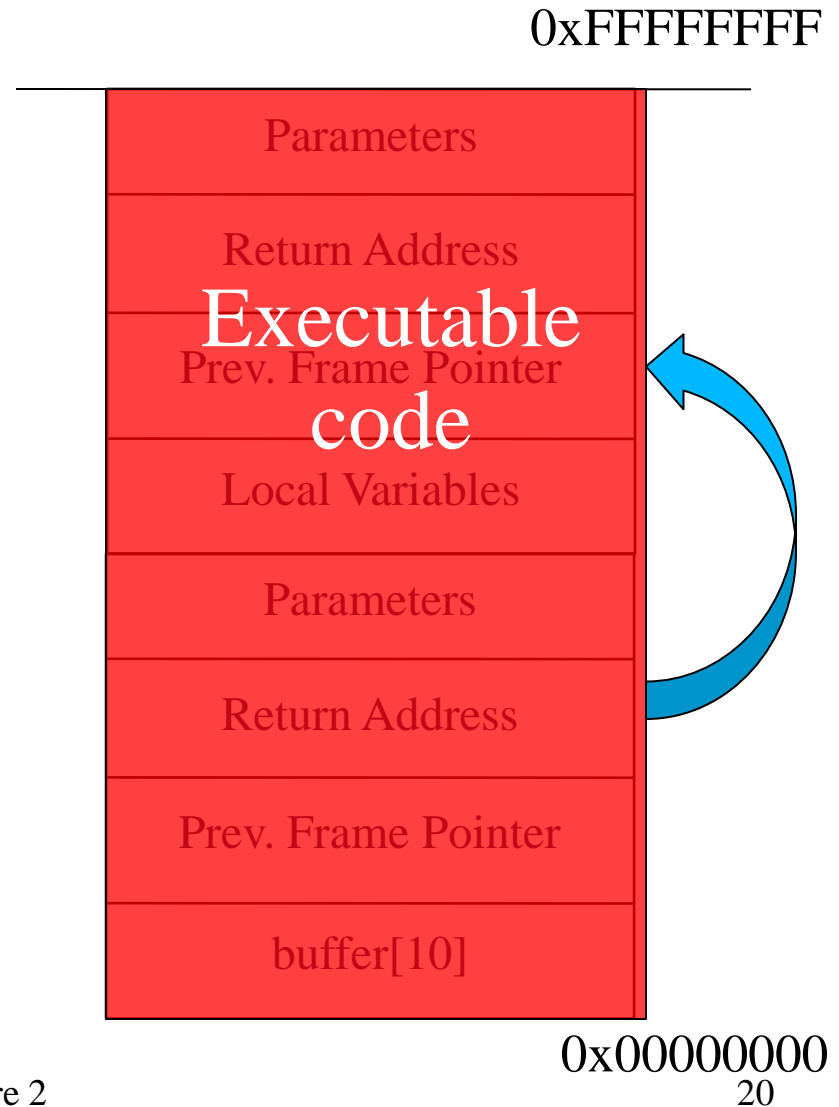


Result of Buffer Overflow

- Return address is overwritten by user inputs
 - Program will “return” to the new address after finishing the function with vulnerable buffer
- If the overwritten return address is an invalid memory address, program will crash.
- Where is attacker’s malicious code?

Malicious Code Injection

- Remember executable code also represented as bytes
- Attackers can include code in the input
 - Called shell code
- They can arrange the return address to point to the injected code

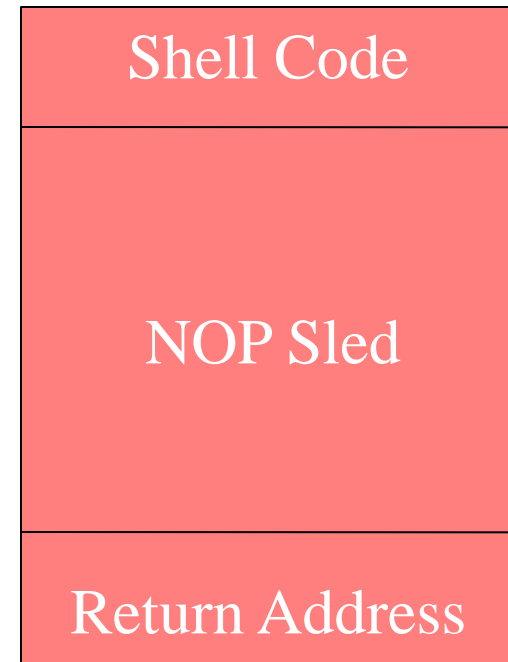


Can They Know the Exact Address of Injected Code?

- Attackers can analyze the vulnerable program using a debugger to find out the address of target stack frame.
- Directly jump into system libraries
 - E.g., `system()` will execute a command
 - Where is its arguments?
- Sometimes, attacks only know the possible range of the code they injected

NOP Sled

- Instruction NOP, No Operation.
 - Tell CPU to do nothing and fetch the next instruction
- Including a large block of NOP instructions in the injected code as *landing area*
- Execution will reach shell code as long as return address pointing to somewhere in the NOP sled



Shell Code Example

```
int main(int argc,  
char*argv[])  
{  
    char *sh;  
    char *args[2];  
  
    sh = "/bin/bash";  
    args[0] = sh;  
    args[1] = NULL;  
    execve(sh, args, NULL);  
}
```

- Shell Code

```
90 90 eb 1a 5e 31 c0 88  
46 07 8d 1e 89 5e 08  
89 46 0c b0 0b 89 f3  
8d 4e 08 8d 56 0c cd  
80 e8 e1 ff ff ff 2f  
62 69 6e 2f 73 68 20  
20 20 20 20 20
```

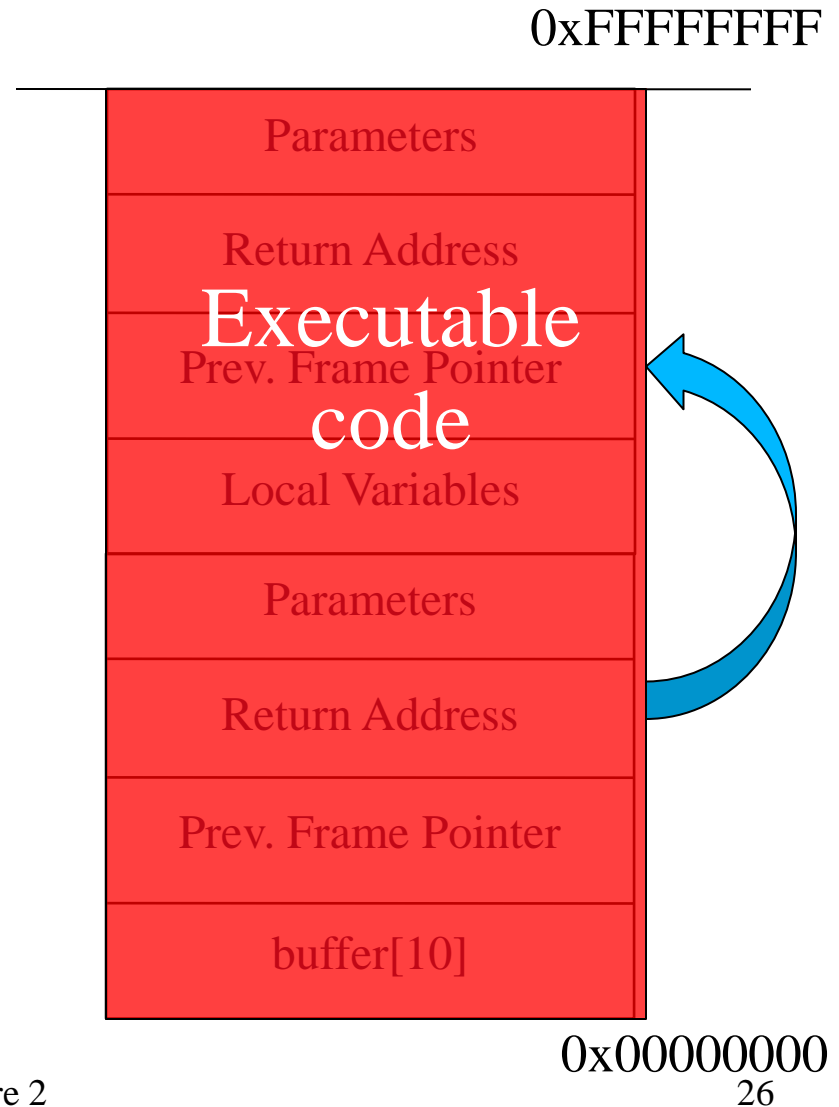
Targets of Buffer Overflow Attack

- Memories that control program execution
 - Return address
 - Function pointer
 - Virtual function table
- Important variables in program
 - Variable storing user id
 - Current balance of bank application
 - ...

Buffer Overflow Defense

Requirements of BO

- Existence of vulnerability
- Overwriting important data
- Known location of injected code
- Executable code in input



Essential Steps of Buffer Overflow

- Three essential steps
 - Attack Code Injection
 - How did you inject the attack code?
 - Control Flow Hijacking
 - How did you guess the address of attack code?
 - Attack Code Execution
 - How did you write the shellcode?

Buffer Overflow Defense (1)

- Existence of vulnerability
- Overwriting important data
- Known location of injected code
- Executable code in input

Safe Language and Coding

- Choose a safe programming language
 - Strong notion of variable types, such as Java
- Safe coding techniques
 - Pay attention to loops
 - Explicitly specify size of destination buffer
- Use safe libraries

Some Unsafe C Lib Functions

- strcpy(char *dest, const char *src)
- strcat(char *dest, const char *src)
- gets(char *s)
- sprintf(const char *format, ...)

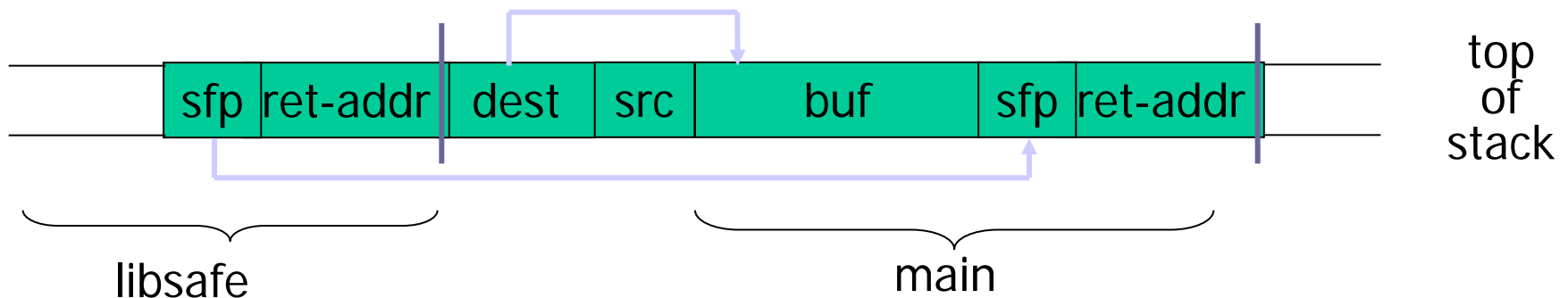
- Safe versions:
- strncpy, strncat, fgets, snprintf

Code Checking Tools

- Tools checking for vulnerabilities using static source code analysis
 - ITS4 (It is the Software, Stupid --- Security Scanner)
 - RATS (Rough Auditing Tool for Security)
 - Flawfinder

Security Extension -- Libsafe

- Idea: Making unsafe functions safe!
 - Intercepts calls to `strcpy (dest, src)`
 - Validates sufficient space in current stack frame:
 $|\text{frame-pointer} - \text{dest}| > \text{strlen}(\text{src})$
 - If so, does `strcpy`.
Otherwise, terminates application.



Buffer Overflow Defense (2)

- Existence of vulnerability
- **Overwriting important data**
- Known location of injected code
- Executable code in input

MemGuard

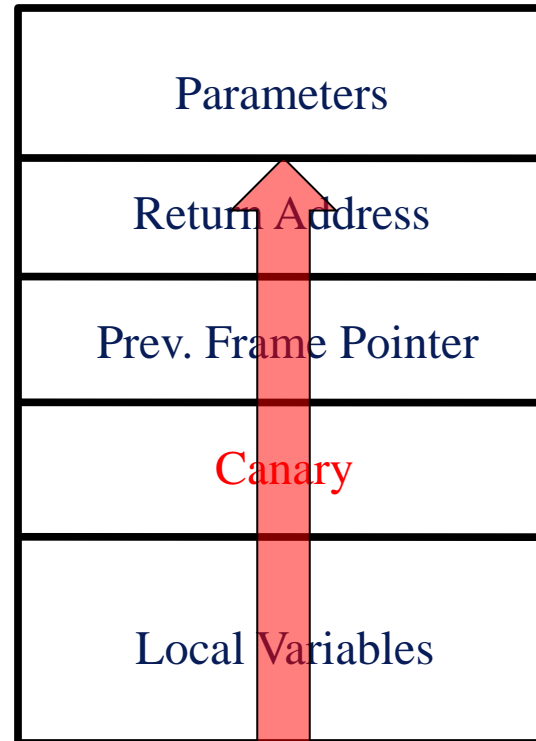
- Main idea: prevent return address from changing via mark it as **read-only**.
 - Extend **VM** model to protect user pages (such as return address in stack).
 - Ex: GCC's `function_prologue` and `function_epilogue`
- Flaw:
 - *Performance penalties*
 - Loading VM hardware is a privileged operation, and so the application process must trap to kernel mode to protect a word.

StackGuard

- Main idea: the technique used to smash the stack currently always involve **sequential memory writing**.
 - If the return address in stack was destroyed, the content before the return address must be destroyed, too.
 - Keep a “canary word” **before** return address and check this word before function returns
- Simple Demo
 - <http://nsfsecurity.pr.erau.edu/bom/StackGuard.html>

StackGuard

- Put a value below saved frame pointer upon entering the function, called canary
- Check canary value before function exit
- Changed canary value indicates an overflow
- Turned on by default in current gcc, try it out!



Flaw in static canary

- What if the attacker can easily guess the canary value?
- Workaround?
 - **randomize** canary-word

Buffer Overflow Defense (3)

- Existence of vulnerability
- Overwriting important data
- **Known location of injected code**
- Executable code in input

Why Randomization?

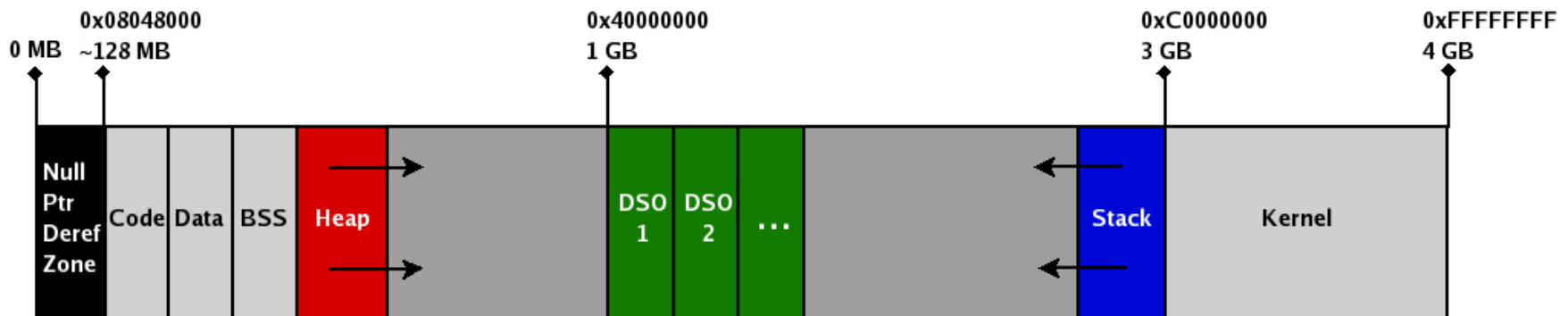
- Buffer overflow and **return-into-libc** exploits need to know the (virtual) address to which pass control
 - Address of attack code in the buffer
 - Address of a standard kernel library routine
- Same address is used on many machines
 - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce **artificial diversity**
 - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

Address Space Layout Randomization (ASLR)

- ASLR renders exploits *which depend on predetermined memory addresses* useless by randomizing the layout of the virtual memory address space.
 - Base addresses of stack, heap, and code segment
- Randomization can be done at compile- or link-time, or by rewriting existing binaries
- Several implementations available
 - E.g., PaX ASLR

An Example: Linux VM System

- Each process has its own 32-bit address space
- Regions are page aligned
- Code & data regions fixed
- Multiple instances, same memory layout
- Traditional locations of heap, user stack, mmap
 - Randomization moves these three regions



ASLR in Linux

```
liangzk@precision:~  
File Edit View Terminal Tabs Help  
[liangzk@precision ~]$ sleep 100 &  
[1] 28834  
[liangzk@precision ~]$ cat /proc/28834/maps  
004fa000-00514000 r-xp 00000000 fd:00 9568320 /lib/ld-2.5.so  
00514000-00515000 r-xp 00019000 fd:00 9568320 /lib/ld-2.5.so  
00515000-00516000 rwxp 0001a000 fd:00 9568320 /lib/ld-2.5.so  
00518000-00656000 r-xp 00000000 fd:00 9569531 /lib/libc-2.5.so  
00656000-00658000 r-xp 0013e000 fd:00 9569531 /lib/libc-2.5.so  
00658000-00659000 rwxp 00140000 fd:00 9569531 /lib/libc-2.5.so  
00659000-0065c000 rwxp 00659000 00:00 0  
00f1e000-00f1f000 r-xp 00f1e000 00:00 0 [vdso]  
08048000-0804c000 r-xp 00000000 fd:00 13598780 /bin/sleep  
0804c000-0804e000 rw-p 00003000 fd:00 13598780 /bin/sleep  
09f3a000-09f5b000 rw-p 09f3a000 00:00 0 [heap]  
b7c16000-b7c17000 r--p 00949000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7c17000-b7cee000 r--p 00191000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7cee000-b7eee000 r--p 00000000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7eee000-b7ef0000 rw-p b7eee000 00:00 0  
bfe1a000-bfe2f000 rw-p bffea000 00:00 0 [stack]  
[liangzk@precision ~]$
```

```
liangzk@precision:~  
File Edit View Terminal Tabs Help  
[liangzk@precision ~]$ sleep 100 &  
[1] 29196  
[liangzk@precision ~]$ cat /proc/29196/maps  
00110000-0024e000 r-xp 00000000 fd:00 9569531 /lib/libc-2.5.so  
0024e000-00250000 r-xp 0013e000 fd:00 9569531 /lib/libc-2.5.so  
00250000-00251000 rwxp 00140000 fd:00 9569531 /lib/libc-2.5.so  
00251000-00254000 rwxp 00251000 00:00 0  
004fa000-00514000 r-xp 00000000 fd:00 9568320 /lib/ld-2.5.so  
00514000-00515000 r-xp 00019000 fd:00 9568320 /lib/ld-2.5.so  
00515000-00516000 rwxp 0001a000 fd:00 9568320 /lib/ld-2.5.so  
0052b000-0052c000 r-xp 0052b000 00:00 0 [vdso]  
08048000-0804c000 r-xp 00000000 fd:00 13598780 /bin/sleep  
0804c000-0804e000 rw-p 00003000 fd:00 13598780 /bin/sleep  
09c48000-09c69000 rw-p 09c48000 00:00 0 [heap]  
b7c97000-b7c98000 r--p 00949000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7c98000-b7d6f000 r--p 00191000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7d6f000-b7f6f000 r--p 00000000 fd:00 1599402 /usr/lib/locale/locale-archive  
b7f6f000-b7f71000 rw-p b7f6f000 00:00 0  
bfc47000-bfc5c000 rw-p bffea000 00:00 0 [stack]  
[liangzk@precision ~]$
```

Buffer Overflow Defense (4)

- Existence of vulnerability
- Overwriting important data
- Known location of injected code
- **Executable code in input**

Non-Executable Stack

- Using CPU's memory management unit to mark stack as non-executable
- Vulnerable program crashes if it jump to the stack for execution
- But there are legitimate reasons to put code on stack
 - Self-modifying code, e.g., Skype
 - Linux signal handlers

	Buffer Overflow Essential Steps		
	Attack Code injection	Control Flow Hijacking	Attack Code Execution
Non-Executable Stack	X	X	✓
MemGuard	✓	✓	—
StackGuard	X	✓	—
LibSafe	✓	✓	—
ASLR	X	✓	—

Summary

- Function call mechanism
 - Basic concepts
 - Using debugger to observe it
- Buffer overflow attack
- Buffer overflow defense