CS 354 - Machine Organization & Programming
Tuesday November 21, 2023

Thanksgiving Break: There is no TA Consulting or Peer Mentoring, from 4pm on Wednesday Nov 22 through the weekend until Sunday Nov 26.

Deb will still have regular schedule of office hours this week and next week.

Homework hw6: DUE on or before Monday Nov 20

Homework hw7: DUE on or before Monday Nov 27

Project p5: DUE on or before Friday Nov 24 (do before Wed Nov 23)

Project p6: Assigned soon and Due on last day of classes.

Learning Objectives

- Understand when and how to use function pointers for selecting which function at runtime
- Identify when buffer overflow occurs and be able to eliminate the chance for buffer overflow
- Identify exceptional control flow in C programs
- Understand the default behavior and to define new behaviors for exceptional events
- Trace the control flow that occurs when an exception occurs.
- Name and describe four categories of Exceptions in C.

This Week	**Next Week**
Pointers Function Pointers Buffer Overflow & Stack Smashing Flow of Execution Exceptional Events Kinds of ExceptionsTransferring Control via Exception Table THANKSGIVING BREAK	Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example
Next Week: Signals, and multifile coding, Linking and Symbols B&O 8.5 Signals Intro, 8.5.1 Signal Terminology 8.5.2 Sending Signals 8.5.3 Receiving Signals 8.5.4 Signal Handling Issues, p.745	

Pointers

Recall Pointer Basics in C

```
int i = 11;
int *iptr = &i;
*iptr = 22;
```

pointer type	`int *`
pointer value	`0x2A300F87`, `0x00000000` (NULL)
address of	`&i`
dereferencing	`*iptr`

Recall Casting in C

```
int *p = malloc(sizeof(int) * 11);

... (char *)p + 2
```

✵ *Casting changes*

Function Pointers

What? A *function pointer*

- ◆

- ◆

Why?

enables functions to be

- ◆

- ◆

How?

```
int func(int x) { ...}

int (*fptr)(int);

fptr = func;

int x = fptr(11);
```

Example

```
#include <stdio.h>

void add     (int x, int y) { printf("%d + %d = %d\n", x, y, x+y); }
void subtract(int x, int y) { printf("%d - %d = %d\n", x, y, x-y); }
void multiply(int x, int y) { printf("%d * %d = %d\n", x, y, x*y); }

int main() {
    void (*fptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int choice;
    int i = 22, j = 11; //user should input

    printf("Enter: [0-add, 1-subtract, 2-multiply]\n");
    scanf("%d", &choice);
    if (choice > 2) return -1;
    fptr_arr[choice](i, j);
    return 0;
}
```

Buffer Overflow & Stack Smashing

Bounds Checking

```
int a[5] = {1,2,3,4,5};
printf("%d", a[11]);
```

→ What happens when you execute the code?

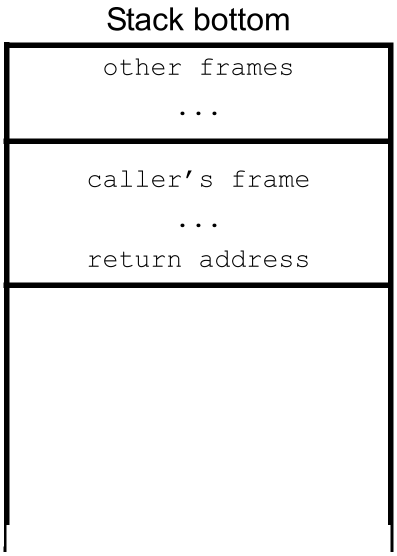
❈ *The lack of bounds checking array accesses*

Buffer Overflow

◆

◆

```
void echo() {
   char bufr[8];
   gets(bufr);
   puts(bufr);
}
```



Stack bottom

other frames
...

caller's frame
...
return address

❈ *Buffer overflow can overwrite*

❈ *It can also overwrite*

Stack Smashing

1. Get "exploit code" in

2. Get "exploit code" to run

3. Cover your tracks

❈ *In 1988 the Morris Worm*

Flow of Execution

What?

control transfer

control flow

- ➢ What control structure results in a smooth flow of execution?
- ➢ What control structures result in abrupt changes in the flow of execution?

Exceptional Control Flow

logical control flow

exceptional control flow

event

processor state

Some Uses of Exceptions

process

OS

hardware

✵

Exceptional Events

What? An *exception*

-
-
-

→ What's the difference between an asychronous vs. a synchronous exception?

asynchronous

synchronous

General Exceptional Control Flow

	Application	Exception Handler
0. normal flow	I_0 I_1	

1.

2.

3.

4.

Kinds of Exceptions

→ Which describes a ***Trap***? ***Abort***? ***Interrupt***? ***Fault***?

1.

signal from external device
asynchronous
returns to Inext

How? Generally:

1.

2.

3. transfer control to appropriate exception handler

4. transfer control back to interrupted process's next instruction

vs. polling

2.

intentional exception
synchronous
returns to Inext

How? Generally:

1.

   `int`

2. transfer control to the OS system call handler

3. transfer control back to process's next instruction

3.

potentially recoverable error
synchronous
might return to Icurr and re-execute it

4.

nonrecoverable fatal errors
synchronous
doesn't retur