CS 354 - Machine Organization & Programming
Tuesday, November 19, 2019

Project p5 (4.5%): DUE at 10 pm on Monday, December 2nd

Project p6 (4.5%): Assigned on Tuesday, November 26th

Homework hw6 (1.5%): DUE TOMORROW at 10 pm on Wednesday, November 20th

Today is last chance to pick up exams from me at lecture.

Last Time

Stack Allocated Arrays in Assembly
Stack Allocated Multidimensional Arrays
Stack Allocated Structs
Alignment
Alignment Practice
Unions

Today

Unions (from last time)
Pointers
Function Pointers
Buffer Overflow & Stack Smashing

Flow of Execution
Exceptional Events
Kinds of Exceptions

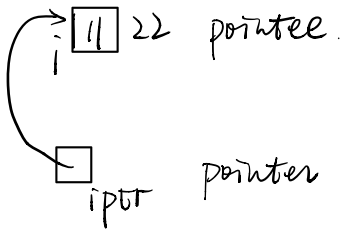
Next Time

More Exceptions, Processes, Context Switches
Read: B&O 8.1 - 8.3, 8.4 through p. 719

Pointers

Recall Pointer Basics in C

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



pointer type `int *`

pointee type used at compiler to determine scaling factors.

pointer value `0x2A300F87`, 0x00000000 (NULL)

addr used with addr modes to specify an effecting addr.

address of `&i`

becomes leal instruction, just calcs the effective address. addr.

dereferencing `*iptr`

becomes MOVL instruction, which accesses mem of the effective addr.

Recall Casting in C

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

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

A casting changed the scalling factor

It does not change the pointers' value.

Function Pointers

What? A *function pointer*

- is a pointer to code.
- stores the addr of the first instruction of a function.

Why? enables function to be

- passed to and return from funcs.
- store in arrays and other data structures.

How?

```
int func(int x) { ...}      //1. implement some functions.

int (*fptr)(int);           //2. declare the function pointer.

fptr = func;                //3. assign function pointer to its function.
     = &func;
int x = fptr(11);           //4. use function pointer to make call.
      = (*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;
}
```

(1) (2) (3) (4)

initialize jump table.

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? intermittent error.

✵ *The lack of bounds checking array accesses* is one of c's main vulnerabilities.

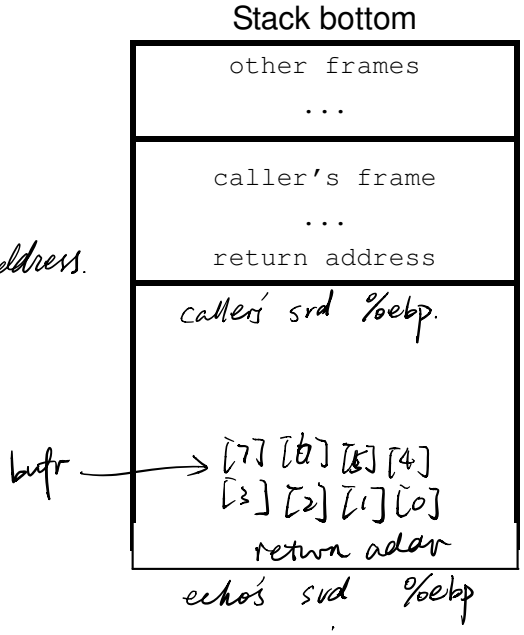
Buffer Overflow

- is exceeding bound of allocated mem.
- particularly dangerous with stack allocated arrays.

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

→ data can be overwritten outside the buffer.

→ but also the state of execution – return address.



Stack Smashing

1. Get "exploit code" in
   Enter input crafted to be machine instruction
2. Get "exploit code" to run
   over write return address with addr of buffer containing exploit code.
3. Cover your tracks
   restore state to execution continues as expected.

Flow of Execution

What?

control transfer a transition from one instruction to another.

control flow a sequence of control transfers.

- 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 normal execution that follows look of the codes.

exceptional control flow special execution that enables a program to react to urgent/unusual/anomalous events.

event a change in a program's state that may or may not be related to current instruction.

processor state a processor's internal mem elements including registers, flags, signals, etc.

Some Uses of Exceptions

process ask for kernel services
share info among processes.
send & receive signals

OS communicate with process & hardware
switch execution among processes.
deal with memory padding

hardware indicate device status.

Exceptional Events

What? An *exception*

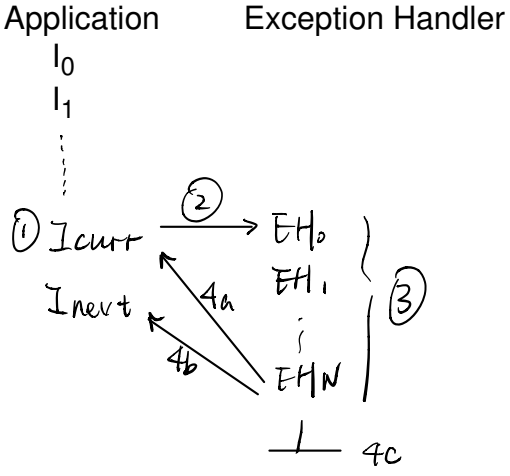
- is an event that sidesteps the logical flag.
- can originate from hardware or software.
- response is an indirect func call.
  that abruptly changes flow of execution

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

asynchronous results from some event unrelated to current flow of execution.

synchronous results from current instruction.

General Exceptional Control Flow



0. normal flow

1. exceptional event occurs.

2. transfer control to appropriate exception handler.

3. exception handler runs

4. return control to
   a. Icurr (page fault)
   b. Inext (file read)
   c. kernel - abort process (seg. fault)
      (return ctrl to OS)

Kinds of Exceptions

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

1. interrupt - enable a device to notify that it needs attention

signal from external device
asynchronous
returns to Inext

How? Generally:

1. Device indicate interrupt
2. finish current instr
3. transfer control to appropriate exception handler
4. transfer control back to interrupted process's next instruction

vs. polling

2. Trap - enables process to interact with os

intentional exception
synchronous
returns to Inext

How? Generally:

1. process indicate what service it wants and then does `int` IA-32 interrupt instruction.
2. transfer control to the OS system call handler which calls the func to do desired service
3. transfer control back to process's next instruction possibly returning a result via %eax

3. Fault - handle problems with current instructions

potentially recoverable error — page fault, seg fault
synchronous
might return to Icurr and re-execute it

4. abort - cleanly ends the process

nonrecoverable fatal errors
synchronous
doesn't return