

Exam 2 topics

Lectures

Programming with Memory
x86 Basics
x86 Control Flow
x86 Procedures, Call Stack
Representing Data Structures
Buffer Overflows
Processes Model
Shells
Memory Hierarchy, Caching

Labs

Pointers in C
x86 Assembly
x86 Stack
Data structures in memory
Buffer overflows
Processes

Topics

C programming: pointers, dereferencing, arrays, structs, cursor-style programming, using malloc
x86: instruction set architecture, machine code, assembly language, reading/writing x86, basic program translation
Procedures and the call stack, data layout, security implications
Processes, shell, fork, wait
Basics of: memory hierarchy, caching

Assignments

Commands
x86
Buffer
Concurrency

Exam 2: ISA
April 22 (during lab, 2 hr)



Practice problems

For Exam 2: ISA

Which practice problem should we do first?

 0

Short answer

x86 2-D array

x86 arithmetic

Processes

x86 recursive

C programming

x86 structs

x86 short answer practice problems

The icon consists of the letters 'ex' in a bold, orange font, centered within a rounded square orange border.

1. Which x86 instructions implicitly (e.g., without naming it) change the stack pointer? How do they change it?
2. What are some things defined by the *word size*? What is the word size we have been using for x86 in class?
3. Describe the general idea of a buffer overflow exploit in C code compiled to x86.
4. Describe how a child process's memory is related to the memory of the parent process.

2-D array practice problem

ex

```
long a[2][3];
```

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `a[0][0]` at offset +0);

Recall: $index = C * r + c$
scale by element size

```
long get_elem_1_2(long a[2][3]) {  
    return a[1][2];  
}
```

2. Write x86 assembly code to implement this function.

x86 arithmetic practice problem

ex

```
long funmath0(long x, long y) {  
    return x + 4*y + 21;  
}
```

```
long funmath1(long x, long y) {  
    return 2*x + 4*y + 21;  
}
```

```
long funmath2(long x, long y) {  
    return 6*x + 5*y + 21;  
}
```

Implement the above functions in x86 *without* `addq` or `mulq`.
You can use `leaq` and any other x86 instruction.

Recall: addressing modes can only multiply by 1, 2, 4, or 8.

Processes practice problem

ex

```
void f(int s) {
    printf("A\n");
    int pid = fork();
    if (pid != 0) {
        printf("B\n");
        waitpid(pid, &s, 0);
        printf("C\n");
    } else {
        printf("D\n");
    }
}
```

In how many different orders could the print statements print to the terminal? List them.

Fun attendance question: what snacks/drinks would you like for the last CS240 class?

 24

Calpico or aloe water



x86 recursive procedure practice problem

ex

```
mystery:
401106 mov     $0x0,%eax
40110b test    %edi,%edi
40110d jne    401110 <mystery+0xa>
40110f ret
401110 push   %rbx
401111 mov    %esi,%ebx
401113 sub    $0x1,%edi
401116 call   401106 <mystery>
40111b movslq %ebx,%rsi
40111e add    %rsi,%rax
401121 pop    %rbx
401122 ret
```

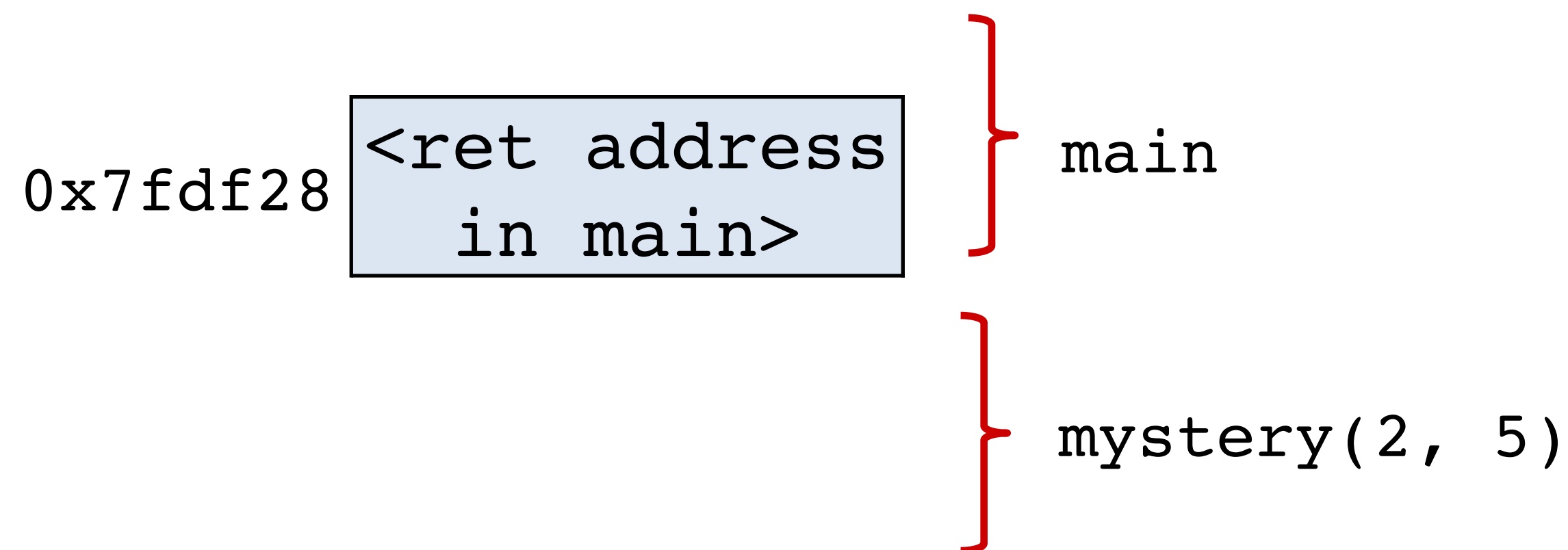
1. What register is being saved to the stack? Why?

2. What instruction address gets saved to the stack? Why?

3. What is this function computing?

4. Fill in the top of this stack up to when the function returns to main for `mystery(2, 5)`.

What is each value returned, in order?



C programming practice problem

Alice attempted to write the following C code to implement the functionality “return a null-terminated string with 3 copies of the given char”

1. What is incorrect about this implementation?

Hint: consider the stack vs. heap

2. Implement a correct version of this function using `malloc`

```
/* Return a string with 3 copies of c */
char* thriceChar(char c) {
    char s[4];
    for (int i = 0; i < 3; i++) {
        s[i] = c;
    }
    s[n] = '\0';
    return s;
}
```

Memory layout, C programming

```
typedef struct {
    char *name;
    int number[10];
} Contact;

int main() {
    Contact contact1;
    Contact *contact2 = (Contact *)malloc(sizeof(Contact));
    // TODO: use contacts
    return 0;
}
```

Give examples of which data in this program is laid out in instructions (text), on the stack, in the heap, and in registers

Give example code that fills in data for both contacts. After filling in the data, clean up the memory used as needed.

x86 struct/LinkedList practice problem

ex

```
nodeFunc2:
    pushq    %rbp
    pushq    %rbx
    subq    $8, %rsp
    movl    %esi, %ebx
    movslq   %esi, %rax
    testq   %rdi, %rdi
    je     .L1
    movq   %rdi, %rbp
    movl   8(%rdi), %esi
    cmpl  %esi, %ebx
    jb    .L5
.L3:    movq   0(%rbp), %rdi
        movl  %ebx, %esi
        call nodeFunc2
.L1:    addq   $8, %rsp
        popq  %rbx
        popq  %rbp
        ret
.L5:    movl  %esi, %ebx
        jmp  .L3
```

```
typedef struct Node {
    struct Node* next;
    unsigned int value;
} Node;

long nodeFunc2(Node* node, unsigned int x) {
    // ???
}

long nodeFunc1(Node* node) {
    nodeFunc2(node, 0);
}
```

Consider the above function that calculates something useful about a linked list of unsigned integers using a helper function.

1. Identify which pieces of x86 refer to `next` and `value`.
2. Identify the base case of the recursive function `nodeFunc2`. What is returned in this case?
3. Identify the recursive case of `nodeFunc2`. What is the argument passed to the recursive call?
4. What is `nodeFunc1` calculating with helper `nodeFunc2`?

Struct practice problem

ex

```
struct s {  
    long x;  
    char y;  
    int p[5];  
    char z;  
    short w;  
};
```

Recall: a short is
2 bytes in C

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with a at offset +0);
2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).
3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.