Function Calls and Calling Conventions 2

CS449 Fall 2017
#include <stdio.h>

int f(int x)
{
    return x;
}

int main()
{
    int y;
    y = f(3);
    return 0;
}
Function Call, 1 param

- Try drawing the stack at each instruction, with the following pseudo-code translations in mind
- `call f` translates to:
  ```assembly
  pushl %eip+4
  movl f, %eip
  ```
  (Push return address and jump to f)
- `leave` translates to:
  ```assembly
  movl %ebp, %esp
  popl %ebp
  ```
  (Pop the current frame)
- `ret` translates to:
  ```assembly
  popl %eip
  ```
  (Pop and jump to return address)
Stack

- $EBP
- $ESP
- Old $EBP
- Old $ESP
- Old $EIP

Variables:
- 3

Function Call:
- main
- f
Function Call, 2 params

```
#include <stdio.h>

int f(int x, int y)
{
    return x+y;
}

int main()
{
    int y;

    y = f(3, 4);

    return 0;
}
```

```
f:       pushl  %ebp
    movl   %esp, %ebp
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    leave
    ret

main:    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    andl   $-16, %esp
    subl   $16, %esp
    movl   $4, 4(%esp)
    movl   $3, (%esp)
    call   f
    movl   %eax, -4(%ebp)
    movl   $0, %eax
    leave
    ret
```
Stack

$EBP

Old $EBP

$ESP

4

3

Old $EIP

Old $EBP

main

f

$EBP and $ESP
Observation

• Parameters are pushed right to left onto the stack
• Why?
int printf(const char *format,...);

- “…” means variable number of arguments
- format must be pushed last
  - printf must first parse format to discover the number of arguments
  - Pushing format last fixes its location relative to EBP (base pointer)
```c
#include <stdarg.h>

int *makearray(int a, ...) {
    va_list ap; // a ‘pointer’ to the stack frame
    int *array = (int *)malloc(MAXSIZE * sizeof(int));
    int argno = 0;
    va_start(ap, a); // ap = address of first arg after ‘a’
    while (a > 0 && argno < MAXSIZE) {
        array[argno++] = a;
        a = va_arg(ap, int); // advance ap by sizeof(int)
    }
    array[argno] = -1;
    va_end(ap); // end args traversal
    return array;
}

• Compiler knows stack layout so can translate easily
• E.g. va_start(ap, a) ➔ ap = (char*)&a + sizeof(a)
• E.g. va_arg(ap, int) ➔ ap += sizeof(int)
```
Variable Arguments Usage

```c
int main()
{
    int *p;
    int i;
    p = makearray(1,2,3,4,-1);

    for(i=0; i<5; i++)
    {
        printf("%d\n", p[i]);
    }

    return 0;
}
```
Other Notes

• Also called a *Variadic* function
• Reason for format specifiers (e.g. `%d`) in format string of `printf`
  • Only way to find the types and sizes of arguments
• Java: Does not allow stack traversal (For safety and security reasons)

```java
public static void printObjects(Object... objects) {
    for (Object o : objects)
        System.out.println(o);
}
printObjects(3, 4, "abc");
⇒ Passed in the form of a list object
```
How Arrays are Allocated

```c
void f()
{
    char input[30];
    scanf(“%s”, input);
}

int main()
{
    f();
    return 0;
}
```
Stack at Beginning of `scanf`

- Old EBP
- Return address to main
- Old EBP
- Input[29]
- Input[0]
- Address of input
- Format String
- Return address to f

```c
main
```
```c
f
```
```c
scanf
```
Buffer Overrun Attack

- Old $EBP
- Hijacked $EIP
- Old $EBP
- Input[29]
  ...
- Input[0]
- Address of input
- Format String
- Return address to f
  ...

$EBP

$input$

$ESP$

main

f

scanf
Buffer Overrun Attack

Old $EBP
Base address of input
Old $EBP
Code!

Code!
Address of input
Format String
Return address to f

$EBP
input
$ESP

main
f
scanf
Buffer Overrun Example

```c
#include <stdio.h>

void hijack() {
    printf("Hijacked!\n");
}

int main() {
    char a[100];
    scanf("%s", a);
    *(int*)(a + 112) = hijack;
    printf("%s\n", a);
    return 0;
}
```

- Write to *(int *)(a + 112) emulates buffer overrun attack during scanf call where input string is 116 bytes long, overrunning char a[100]
- Bytes 112 – 115 of string overwrites old $EIP in main() stack frame
- On return from main(), code would jump to hijack()
- In real attack, code for hijack() would be part of input string

`>> gcc -m32 ./main.c`  
`>> ./a.out`  
Hello  
Hello  
Hijacked!  
Segmentation fault (core dumped)`