Function Calls and Calling Conventions 2

CS449 Fall 2016
#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
  - **leave** translates to:
    
    movl %ebp, %esp  
    popl %ebp  
  
  (Pop the current frame)

  - **call f** translates to:
    
    pushl %eip+4  
    movl f, %eip  
  
  (Push return address and jump to f)

  - **ret** translates to:
    
    popl %eip  
  
  (Pop and jump to return address)
Stack

$EBP

Old $EBP

$ESP

3

Old $EIP

$EBP and $ESP

Old $EBP

main

f
Function Call, 2 params

```c
#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
Observation

- Parameters are pushed right to left onto the stack
- Why?
`printf` function:

```c
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)
#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); // start args traversal after ‘a’
    while (a > 0 && argno < MAXSIZE) {
        array[argno++] = a;
        a = va_arg(ap, int); //return next arg, advance ap
    }
    array[argno] = -1;
    va_end(ap); // end args traversal
    return array;
}
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

• Java:

```java
public static void printArray(Object... objects) {
    for (Object o : objects)
        System.out.println(o);
}

printArray(3, 4, "abc");
```
Stack Allocated Array

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

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

```
f:    pushl  %ebp
    movl  %esp, %ebp
    subl  $56, %esp
    leal  -40(%ebp), %eax
    movl  %eax, 4(%esp)
    movl  $.LC0, (%esp)
    call  scanf
    leave
    ret

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

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

$EBP

$ESP

input

main

f

scanf
Buffer Overrun

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

$EBP

$ESP

input

main

f

scanf
Buffer Overrun Vulnerability

- Old $EBP
- Address of input
- Old $EBP
- Code!
- Address of input
- Return address to f

$EBP

$ESP

input

Code!

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 `a + 112` (the return address) emulates buffer overrun attack during `scanf` call with input string longer than 100 bytes
- Hijack code would be written inside `a[100]` in typical attack

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