

CMSC 313 COMPUTER ORGANIZATION & ASSEMBLY LANGUAGE PROGRAMMING

LECTURE 16,

TOPICS TODAY

- **Project 6**
- **Perils & Pitfalls of Memory Allocation**
- **C Function Call Conventions in Assembly Language**

PERILS & PITFALLS



Memory-Related Perils and Pitfalls

Dereferencing bad pointers

Reading uninitialized memory

Overwriting memory

Referencing nonexistent variables

Freeing blocks multiple times

Referencing freed blocks

Failing to free blocks

Dereferencing Bad Pointers

The classic `scanf` bug.

Typically reported as an error by the compiler.

```
int val;  
  
...  
  
scanf("%d", val);
```

Reading Uninitialized Memory

Assuming that heap data is initialized to zero

```
/* return y = A times x */
int *matvec(int A[N][N], int x[N]) {
    int *y = malloc( N * sizeof(int));
    int i, j;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            y[i] += A[i][j] * x[j];
    return y;
}
```

Overwriting Memory

Allocating the (possibly) wrong sized object

```
int i, **p;  
  
p = malloc(N * sizeof(int));  
  
for (i = 0; i < N; i++) {  
    p[ i ] = malloc(M * sizeof(int));  
}
```

Overwriting Memory

Not checking the max string size

```
char s[8];  
int i;  
  
gets(s); /* reads "123456789" from stdin */
```

Modern attacks on Web servers
AOL/Microsoft IM war

Overwriting Memory

Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {  
    while (*p != NULL && *p != val)  
        p += sizeof(int);  
  
    return p;  
}
```

Referencing Nonexistent Variables

Forgetting that local variables disappear when a function returns

```
int *foo () {  
    int val;  
  
    return &val;  
}
```

Freeing Blocks Multiple Times

Nasty!

```
x = malloc(N * sizeof(int));  
    <manipulate x>  
free(x);  
  
y = malloc( M * sizeof(int));  
    <manipulate y>  
free(x);
```

Referencing Freed Blocks

Evil!

```
x = malloc(N * sizeof(int));  
  <manipulate x>  
free(x);  
  ...  
y = malloc(M * sizeof(int));  
for (i = 0; i < M; i++)  
  y[ i ] = x[ i ]++;
```

Failing to Free Blocks (Memory Leaks)

Slow, long-term killer!

```
foo() {  
    int *x = malloc(N * sizeof(int));  
    ...  
    return;  
}
```

Failing to Free Blocks (Memory Leaks)

Freeing only part of a data structure

```
struct list {  
    int val;  
    struct list *next;  
};  
  
foo() {  
    struct list *head = malloc(sizeof(struct list));  
    head->val = 0;  
    head->next = NULL;  
    <create and manipulate the rest of the list>  
    ...  
    free(head) ;  
    return;  
}
```

Adapted from Dennis Frey CMSC 313 Spring 2011

Dealing With Memory Bugs

Conventional debugger (gdb)

Good for finding bad pointer dereferences

Hard to detect the other memory bugs

Some malloc implementations contain checking code

Linux glibc malloc: `setenv MALLOC_CHECK_ 2`

Dealing With Memory Bugs (cont.)

Binary translator: **valgrind** (Linux)

- Powerful debugging and analysis technique

- Rewrites text section of executable object file

- Can detect all errors as debugging `malloc`

- Can also check each individual reference at runtime

 - Bad pointers

 - Overwriting

 - Referencing outside of allocated block

Garbage collection (Boehm-Weiser Conservative GC)

- Let the system free blocks instead of the programmer.

C FUNCTION CALL CONVENTIONS IN ASSEMBLY LANGUAGE



Linux/gcc/i386 Function Call Convention

- **Parameters pushed right to left on the stack**
 - ◇ first parameter on top of the stack
- **Caller saves EAX, ECX, EDX if needed**
 - ◇ these registers will probably be used by the callee
- **Callee saves EBX, ESI, EDI**
 - ◇ there is a good chance that the callee does not need these
- **EBP used as index register for parameters, local variables, and temporary storage**
- **Callee must restore caller's ESP and EBP**
- **Return value placed in EAX**

A typical stack frame for the
function call:

```
int foo (int arg1, int arg2, int arg3) ;
```

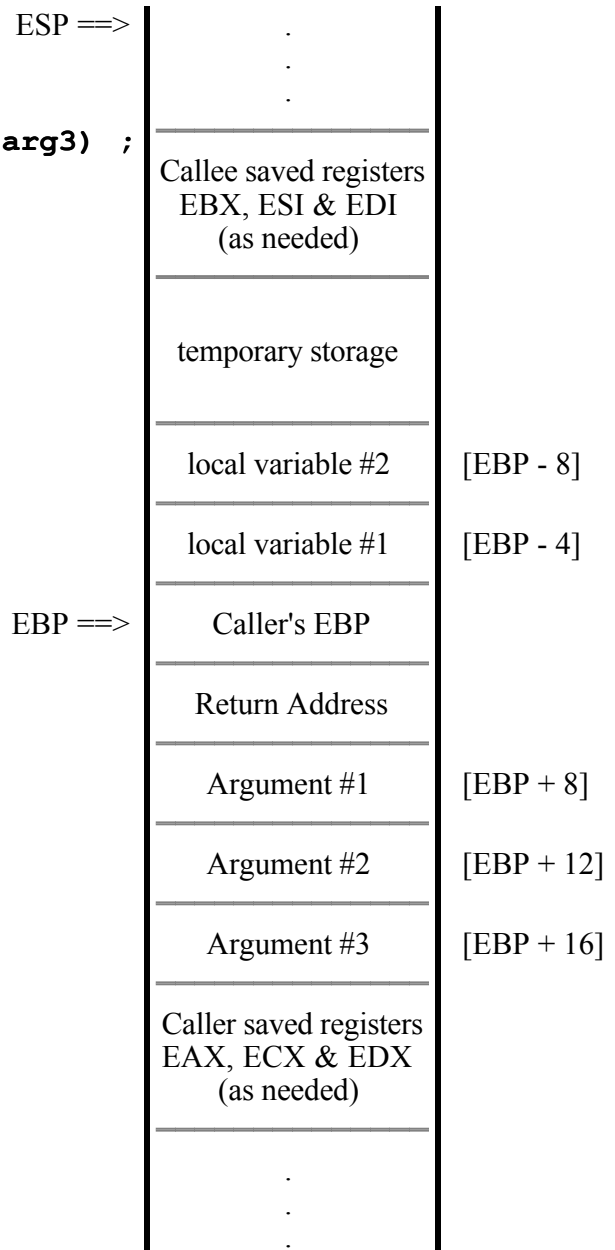


Fig. 1

The caller's actions before the function call

- Save EAX, ECX, EDX registers as needed
- Push arguments, last first
- CALL the function

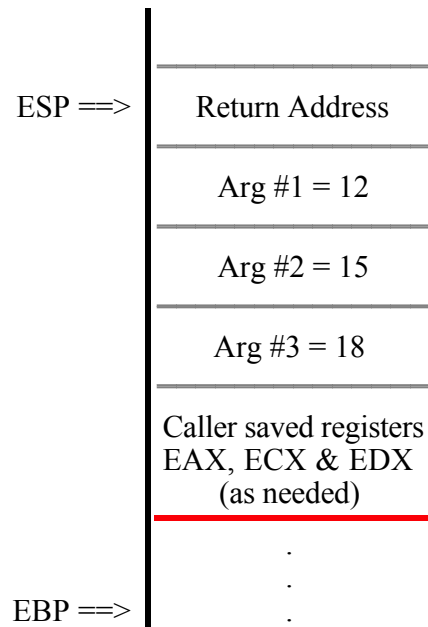


Fig. 2

The callee's actions after function call

- Save main's EBP, set up own stack frame

```
push    ebp
mov     ebp, esp
```

- Allocate space for local variables and temporary storage
- Save EBX, ESI and EDI registers as needed

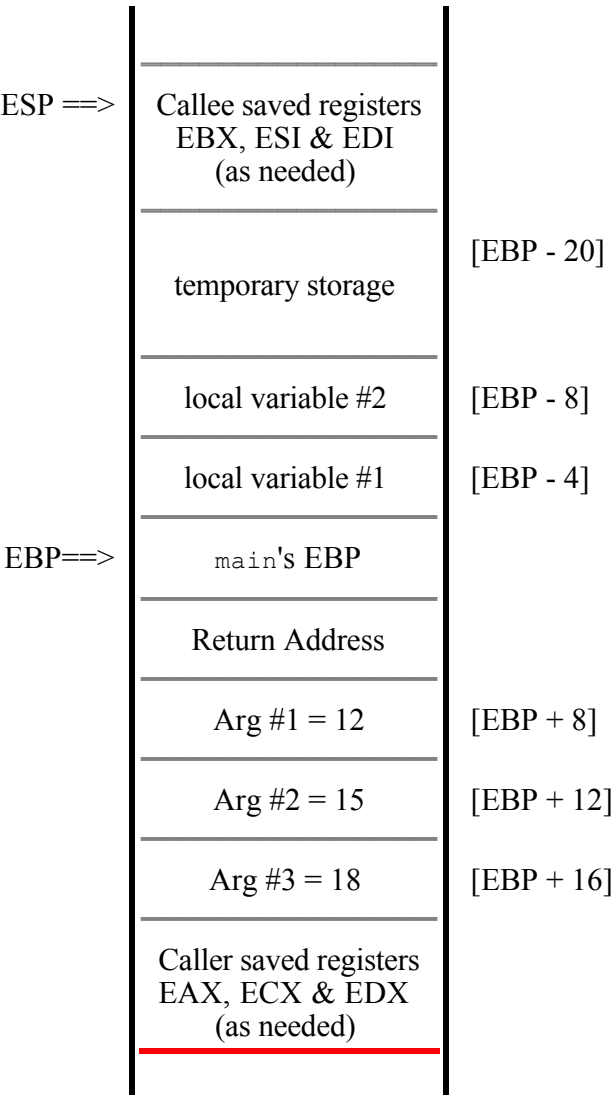


Fig. 4

The callee's actions before returning

- Store return value in EAX
- Restore EBX, ESI and EDI registers as needed
- Restore main's stack frame

```
mov    esp, ebp
pop    ebp
```

- RET to main

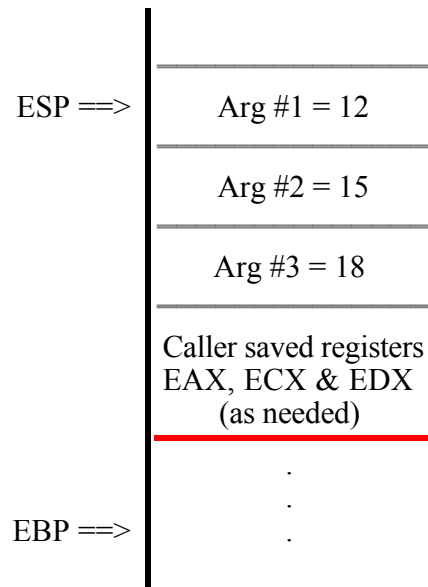


Fig. 5

The caller's actions after returning

- POP arguments off the stack
- Store return value in EAX
- Restore EAX, ECX and EDX registers as needed

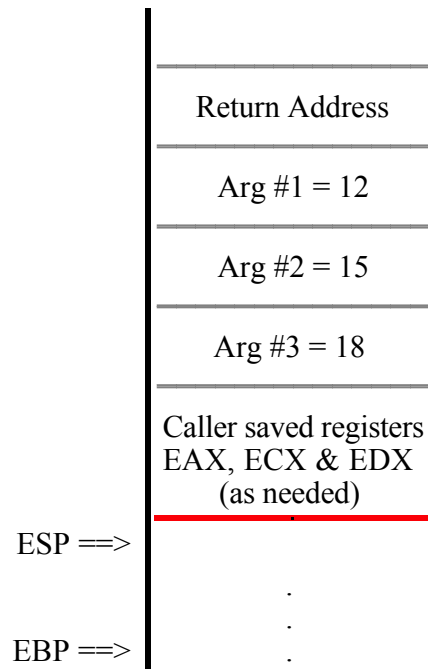


Fig. 6

NEXT TIME

- **Finish C Function Call Conventions**
- **Function Pointers**

