CMSC 449
Malware Analysis

Lecture 7

x86 Assembly

General Purpose Registers

EAX (AL, AH, AX)	Stores return value
EBX (BL, BH, BX)	
ECX (CL, CH, CX)	Loop counter
EDX (DL, DH, DX)	Used with EAX in multiplication, division

More General Purpose Registers

ESI	Source pointer
EDI	Destination pointer
ESP	Stack pointer
EBP	Base pointer

Other Registers

- EIP Instruction pointer

- EFLAGS Status register
  - ZF Zero Flag
  - CF Carry Flag
  - OF Overflow Flag

MOV

- MOV EAX, EBX
- MOV EAX, 0x0
- MOV EAX, [0x400000]
- MOV EAX, [EBX + ESI * 4]

LEA

- "Load Effective Address"
- Moves a pointer into a register, does not dereference

LEA	EAX, [EBX + 8]	Puts EBX + 8 into EAX
MOV	EAX, [EBX + 8]	Dereferences EBX + 8 and puts value into EAX

LEA vs MOV

```
_start:   mov       ebx, message
          lea       eax, [ebx]
          mov       ecx, [ebx]

          section   .data

message:  db        "Hello, World", 10
```

Arithmetic Instructions

- ADD EAX, 0x10
- SUB EAX, EBX
- INC EAX
- DEC EAX

More Arithmetic Instructions

-
  ```
  MOV     EAX,  0x2
  MUL     0x4
  ```
  Multiples EAX by 4, stores upper 32 bits in EDX and lower 32 bits in EAX

-
  ```
  MOV     EDX, 0x0
  MOV     EAX, 0x9
  DIV     0x3
  ```
  Divides EDX:EAX by 3, stores result in EAX and remainder in EDX

Logical Operator Instructions

- XOR EAX, EAX

- AND EAX, 0xFF

- OR EAX, EBX

Bit Shifting Instructions

- SHL EAX, 0x2
- SHR EAX, EBX
- ROL EAX, 0x4
- ROR EAX, EBX

Conditional Instructions

- CMP EAX, EBX
- TEST EAX, 0x10
- TEST EAX, EAX

Branching Instructions

JMP	LOC	Unconditional jump
JZ / JE	LOC	Jump if ZF == 1
JNZ / JNE	LOC	Jump if ZF == 0
JG / JA	LOC	Jump if DST > SRC
JL / JB	LOC	Jump if DST < SRC
JGE / JAE	LOC	Jump if DST >= SRC
JLE / JBE	LOC	Jump if DST <= SRC

REP Instructions

- Used for making common loop constructions more efficient
  - Increment ESI and EDI pointers, decrement ECX in a loop
- REP -> Stop when ECX = 0
- REPE (Repeat equal) -> Stop when ECX = 0 or ZF = 0
- REPNE (Repeat not equal) -> Stop when ECX = 0 or ZF = 1

Common REP Instructions

REPE	CMPSB	Compare ESI and EDI buffers
REP	STOSB	Initialize all bytes of EDI buffer to the value stored in AL
REP	MOVSB	Copy contents of ESI to EDI
REPNE	SCASB	Search EDI for the byte in AL

PUSH in Assembly Language

- What does PUSH actually do?

- **`PUSH myVal`**
  - **`SUB ESP, 4`**

    Subtract 4 from the stack pointer (“make room” on the stack)

  - **`MOV [ESP], myVal`**

    Copy the value into that new space on the stack

POP in Assembly Language

- What does POP actually do?

- **`POP myRegister`**
  - **`MOV myRegister, [ESP]`**
  - **`ADD ESP, 4`**

Copy the value off the stack into the register

Add 4 to the stack pointer (move the stack back "up")

CALL in Assembly Language

- What does CALL actually do?

- **CALL myFunc**
  - **PUSH &nextInstruction**
    - **SUB ESP, 4**
    - **MOV [ESP], &nextInstruction**

  - **JMP myFunc**

Push the address in memory you’ll want to return to

Jump to where the function you’re calling resides in memory

RET in Assembly Language

- What does RET actually do?

- **`RET`**
  - **`POP EIP`**

Pop the return address into EIP

- Trusting that whatever's at the top of the stack is the return address
  - When you execute the next instruction it looks at EIP to see what to do next

What is Cdecl?

- The calling convention for the C programming language

- Calling conventions determine
  - Order in which parameters are placed onto the stack
  - Which registers are used/preserved for the caller
  - How the stack in general is handled

Simple Cdecl Example – Code

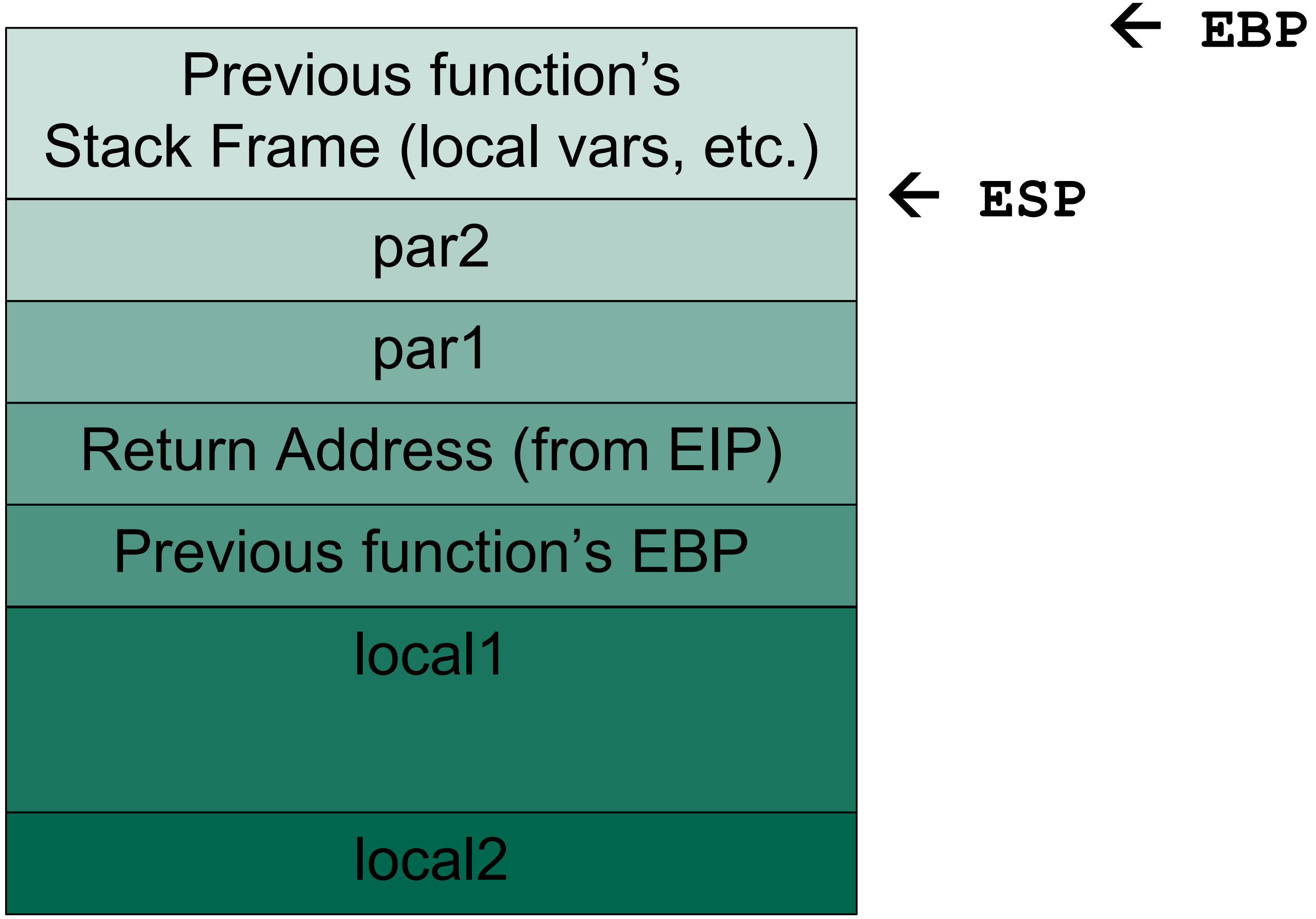
```
int myFunc(char *par1, int par2)
{
    char local1[64];
    int  local2;
    return 0;
}


int main(int argc, char **argv)
{
    myFunc(argv[1], atoi(argv[2]);
    return 0;
}
```

- What actually happens on the stack when this program is run?
  - What variables are allocated first?
  - How does the stack grow?

Simple Cdecl Example – Calling

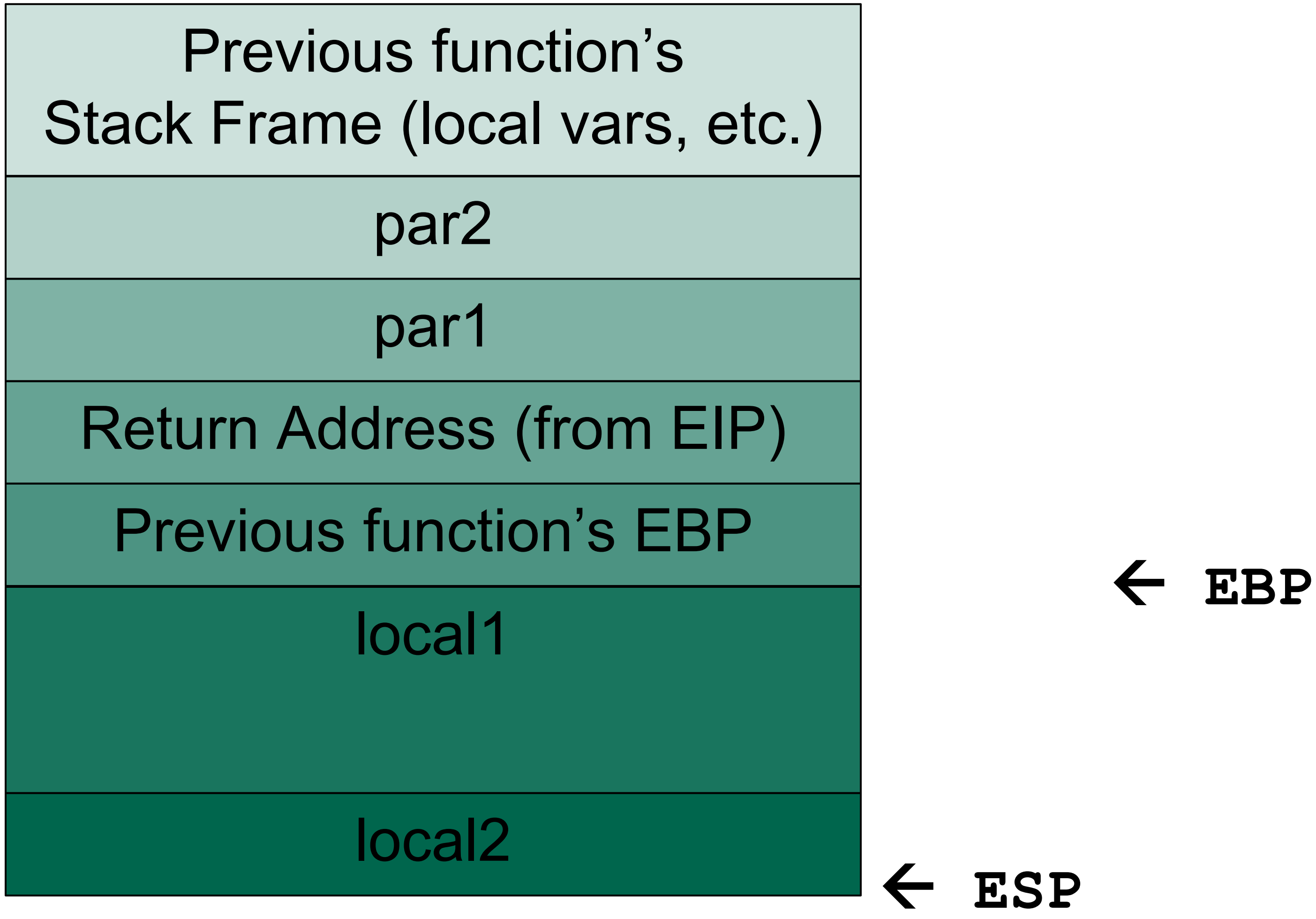
- PUSH par2
- PUSH par1
- PUSH EIP
- PUSH EBP
- MOV EBP, ESP
- SUB ESP, 68
  - 64 bytes for chars
  - 4 bytes for integer


← EBP
Previous function's Stack Frame (local vars, etc.)
← ESP
par2
par1
Return Address (from EIP)
Previous function's EBP
local1
local2

Simple Cdecl Example – Returning

- **`MOV ESP, EBP`**
- **`POP EBP`**
- **`RETN (POP EIP)`**

The caller handles popping parameters upon return.


Previous function's Stack Frame (local vars, etc.)
par2
par1
Return Address (from EIP)
Previous function's EBP
local1
local2
← EBP
← ESP