
CMSC 426

Principles of Computer Security

Lecture 03

Assembly and Stack Basics

Last Class We Covered

- Security Standards
 - Standards Bodies
- Security Principles
- Security Strategy

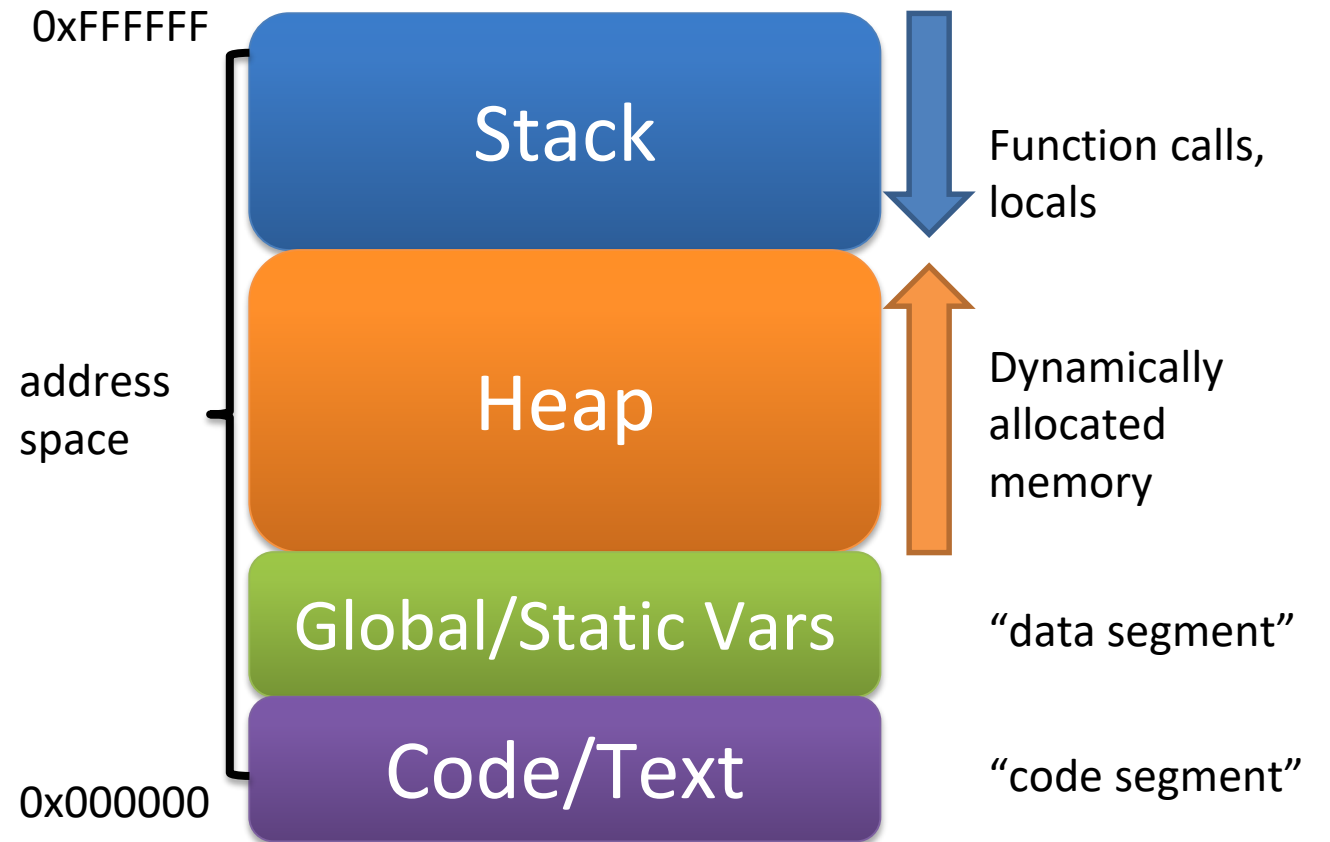
Any Questions from Last Time?

Today's Topics

- Memory allocation in programs
- Assembly language review
 - Registers
 - **PUSH, POP, CALL, RET**
- `cdecl`
 - Code example
- Vulnerable code
 - Finding and avoiding

Stacks, Heaps, and More

- Processes get their own address space when run
- Address space is divided into smaller pieces, each with a specific purpose
- Stack grows “down” to lower addresses



Stack Allocation

- Memory allocated by the program as it runs
 - Local variables
 - Function calls
 - Parameters passed
 - Function-local variables
 - Return addresses
- Grows “down” from the top
 - As things are pushed onto the stack, the address decreases



Heap Allocation

- Dynamically allocated memory
 - Memory explicitly allocated by the user
 - Using `malloc()`, `calloc()`, `new`, etc.
 - Creation and deletion (freeing) is controlled by the user
- Grows “up” towards the stack



Writing Data to Buffers

- Programs constantly write data to areas of memory (buffers)
- Higher level languages (Java, Python, etc.) do a lot of user hand-holding and won't allow unsafe use of the language
- Lower level languages (C, etc.) do a minimal amount of checking, and can only be used safely if we assume that the programmer knows what they're doing
 - Even if they do (they often don't) they still have to be careful

Buffer Overflows

- When a buffer has data written to it that exceeds the size of the buffer, a buffer overflow occurs
 - The excess data continues to write, overflowing into nearby variables and other areas of memory
- When this happens inside the stack, it's called a stack overflow
- But why would a stack overflow be more dangerous than something else, like a heap overflow?
 - Let's talk assembly language for a bit...

Assembly Language Review

x86 Registers

- EAX, EBX, ECX, EDX
 - Used for general data storage
- ESI, EDI
 - Source and destination registers
 - (Mostly used for string and buffer operations)
- ESP, EBP
 - Stack and base pointer
 - (Used for keeping track of stack frames and operations)
- EIP
 - Instruction pointer (points to current instruction being executed)

x86 Registers

- EAX, EBX, ECX, EDX
 - Used for general data storage
- ESI, EDI
 - Source and destination registers
 - (Mostly used for string and buffer operations)
- ESP, EBP
 - Stack and base pointer
 - (Used for keeping track of stack frames and operations)
- EIP
 - Instruction pointer (points to current instruction being executed)

what matters for our purposes

Assembly Language Instructions

- **PUSH**

- PUSH adds data to the top of the stack

- **POP**

- POP removes an item from the top of the stack

- **CALL**

- Call a subroutine (*i.e.*, a function)

- **RET**

- Return from a subroutine


With thanks to Dr. Jennifer Rexford's on Function Calls in Assembly Language for some much-needed clarification:
<https://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/15AssemblyFunctions.pdf>

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 where ESP points:
the "new room" made on the stack

POP in Assembly Language

- What does `POP` actually do?


- `POP myRegister`

- `MOV myRegister, [ESP]`



Copy the value off the stack into the register

- `ADD ESP, 4`



Add 4 to the stack pointer
(move the stack back “up”)

Reminder – Stack Growth

- The stack grows down
- The ESP is the “stack pointer”
 - Keeps track of the “top” of the stack (actually the lowest valid address)
 - The boundary between actual data and junk on the stack
- When the ESP is incremented, we are going UP the stack
 - This means something is being removed from the stack
- When the ESP is decremented, we are going DOWN the stack
 - This means space is being “added” to the stack for new information

CALL in Assembly Language

- What does **CALL** actually do?

- **CALL myFunc**

- **PUSH params**
- **PUSH EIP**
 - **SUB ESP, 4**
 - **MOV [ESP], EIP**
- **PUSH EBP**
- **MOV ESP, EBP**
- **JMP myFunc**

Push any function parameters onto the stack; they'll be referenced in relation to EBP

Push the address in memory you'll want to return to (*i.e.*, where the next instruction is stored, in EIP)

Push the base pointer onto the stack, then move it to the current "edge," where the new function is

Have EIP jump to where the function you're calling resides in memory

RET in Assembly Language

- What does **RET** actually do?

- **RET**

- **MOV EBP, ESP**

- **POP EBP**

- **POP EIP**

Put the stack pointer back to where it was “before” this function was executed

Put the base pointer back to where it was before ...

Pop the return address we previously stored back into EIP

- Trusting that whatever’s at the top of the stack is the return address

x86 Registers

- EAX, EBX, ECX, EDX

- Used for general purpose

The “edge” of the entire stack (lowest address)

- ESI, EDI

- Source and destination registers
- (Mostly used for string and buffer operations)

The “base” of the current context (*i.e.*, the function)

- **ESP, EBP**

- Stack and base pointer
- (Used for keeping track of stack frames and operations)

We’ll want to keep track of these as we talk about the stack

- EIP

- Instruction pointer (points to current instruction being executed)

cdecl

What is cdecl?

- The calling convention for the C programming language is called “cdecl” (short for “C declaration”)
- 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

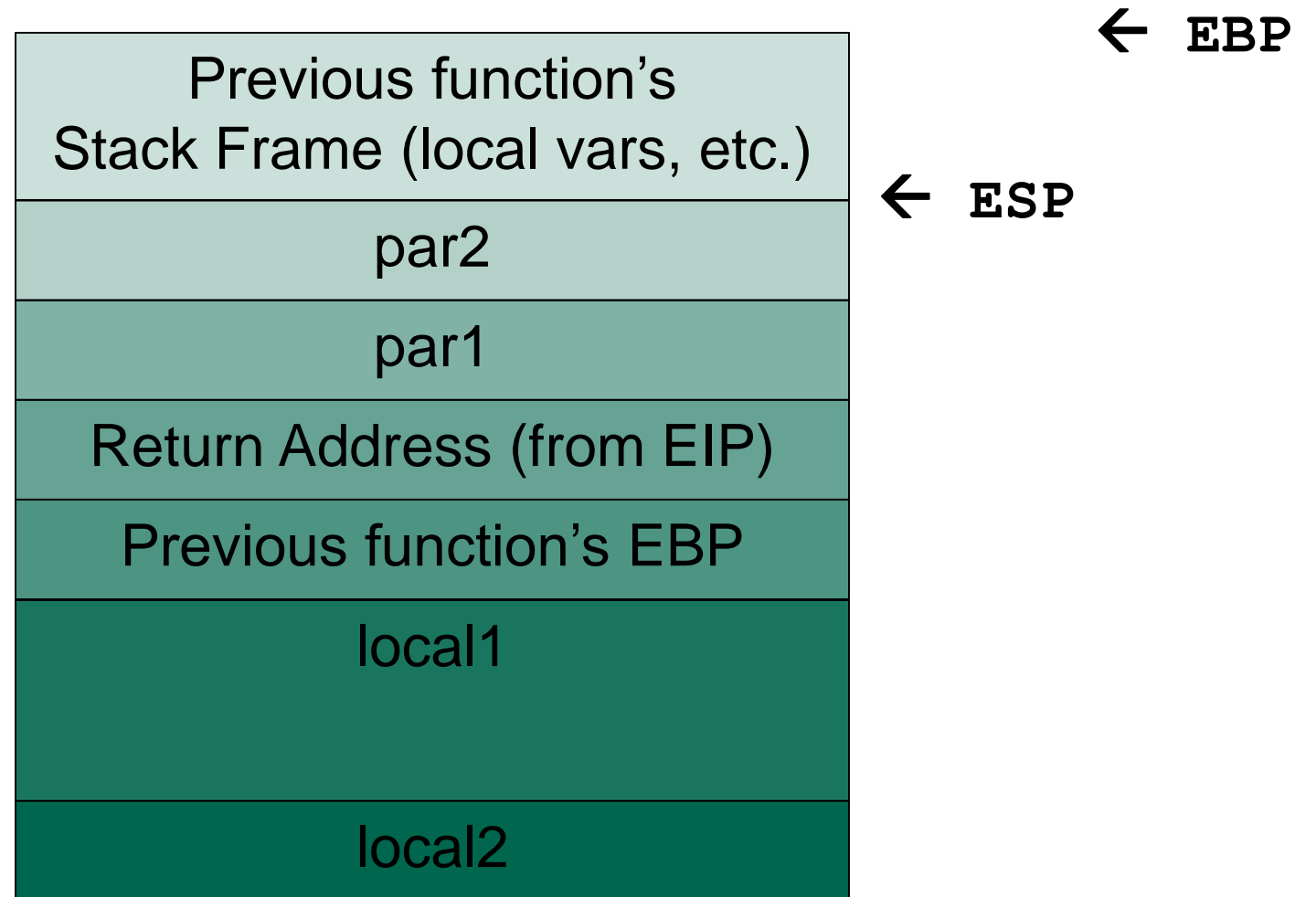
```
int myFunc(char *par1, int par2)
{
    char local1[64];
    int local2;
    return 0;
}
```

```
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

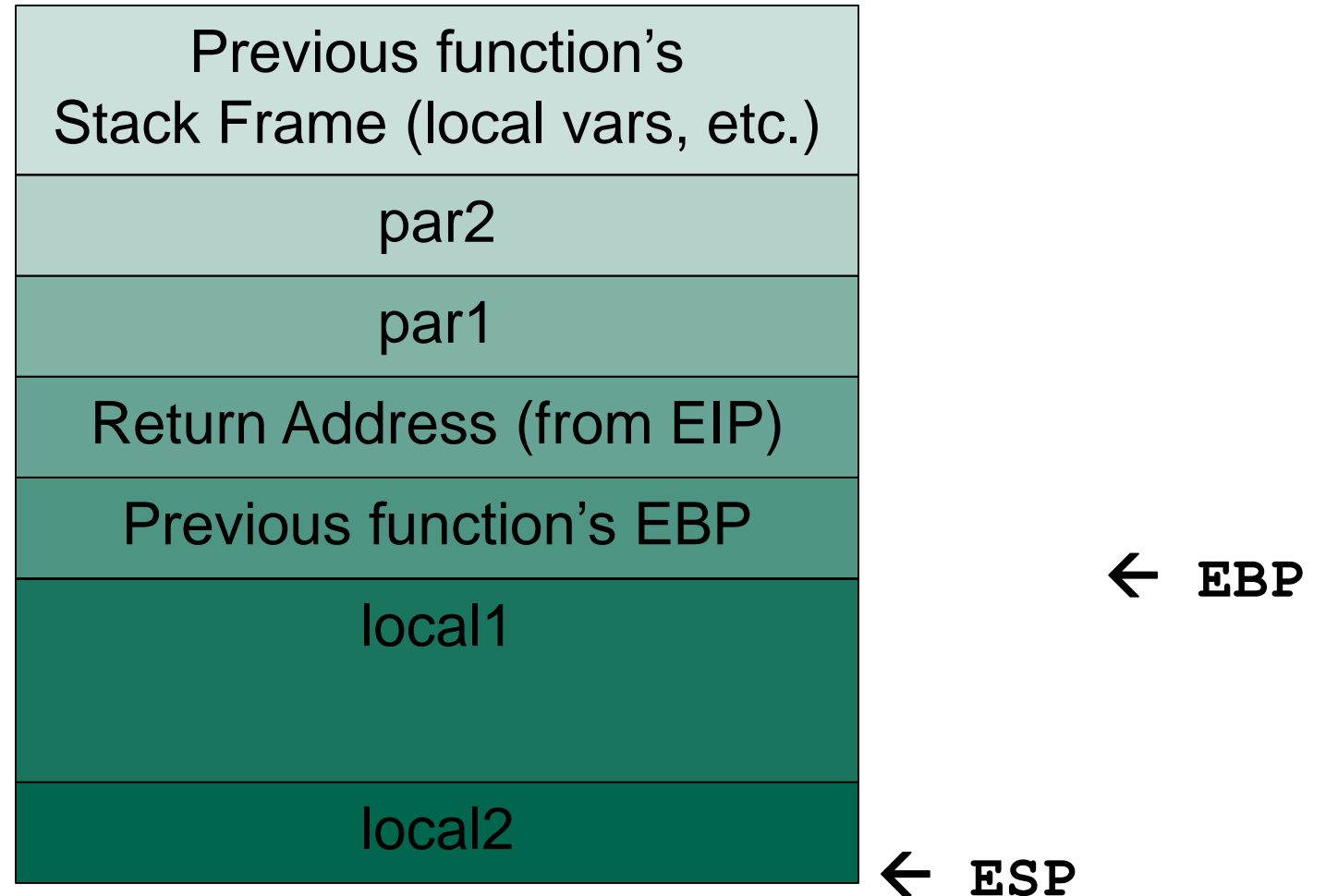
- **PUSH par2**
- **PUSH par1**
- **PUSH EIP**
- **PUSH EBP**
- **MOV EBP, ESP**
- **SUB ESP, 68**
 - 64 bytes for chars
 - 4 bytes for integer



Simple Cdecl Example – Returning

- **MOV ESP, EBP**
- **POP EBP**
- **POP EIP**

The caller handles popping parameters upon return.



Vulnerable Code

Possible Vulnerabilities

- There are a number of vulnerable pieces left on the stack during the process of calling a function and returning from it
- Assumed that all “pieces” are in the correct place and are not changed or tampered with
 - Address popped into EBP: base pointer of calling function
 - Address popped into EIP: address of code to return to
 - Information of all previously called function (EIPs, EBPs, etc.)
- Most exploitable is the EIP, the return address

Finding and Avoiding Vulnerable Code

- Easiest way to find: inspect source code of programs
- Trace the execution of programs with oversized input
 - Brute forcing or “fuzzing” a program with large inputs to see if errors arise
- To avoid vulnerable code: ensure that buffers only take in the amount of data they can actually hold
 - Enforce size limits on inputs from users and files
 - Use a higher-level language when needed
- Don't use bad, outdated functions!

Unsafe Functions and Alternatives

- Set a maximum size/number of characters to handle at once

Unsafe	Safe	Description
<code>gets ()</code>	<code>fgets ()</code>	Read characters from a stream
<code>strcpy ()</code>	<code>strncpy ()</code>	Copy from one string to another
<code>strcat ()</code>	<code>strncat ()</code>	Concatenate one string to another
<code>sprintf ()</code>	<code>snprintf ()</code>	Write data to a string

Safe Programming and Safe Libraries

- Early language designers hoped/assumed that programmers would exercise care and foresight when writing code
 - (and were also limited in what the technology could do at the time)
 - C allows for better performance and space efficiency than Java
 - But, programmers are **not** generally careful or thoughtful
- Standard libraries allow for unsafe actions (like previous slide)
 - Create alternatives to unsafe functions/entire libraries
 - Requires rewriting/updating the source code
 - Create safe versions of type libraries (like strings)

Making a “Memory Safe” C

- Many have tried, many have failed (to have it catch on)
- There are literally dozens of “memory safe” C attempts
 - Cyclone in 2000
 - CCured in 2002
 - Fail-safe C in 2009
 - Safe-C in 2011
 - Others have come up with annotations, semantic restrictions, and modified hardware or compilers to solve the problem
- PROTIP: don't pick this for a dissertation topic

Daily Security Tidbit

- June 2007, Lifelock used CEO Todd Davis's social security number prominently in many of its advertisements
 - Meant to show how good the company was at preventing identity theft
- His identity was stolen 13 times within the year
 - Most of it was small charges (\$100 - \$500), probably done by people showing off that it could be done
- Lifelock was fined by the FTC for deceptive advertising

Information taken from <https://www.wired.com/2010/05/lifelock-identity-theft/>

Announcements

- Sign up for Piazza if you haven't already, as assignments will be starting soon