BiL220 Recitation – 2

Assembly, Stacks, and Registers

Oguzhan Guclu

02/04/2012

Acknowledgement: The slides are adapted from the slides prepared by Kevin C. Su

Registers

- x86
  - EIP
  - EBP
  - ESP

Arguments

- x86
  - Argument 1: %ebp+8
  - Argument 2: %ebp+12
  - …

Today

- Assembly
- Stacks
  - EBP / ESP
- Stack Discipline
- Buffer Overflow
- BufLab

- Summary

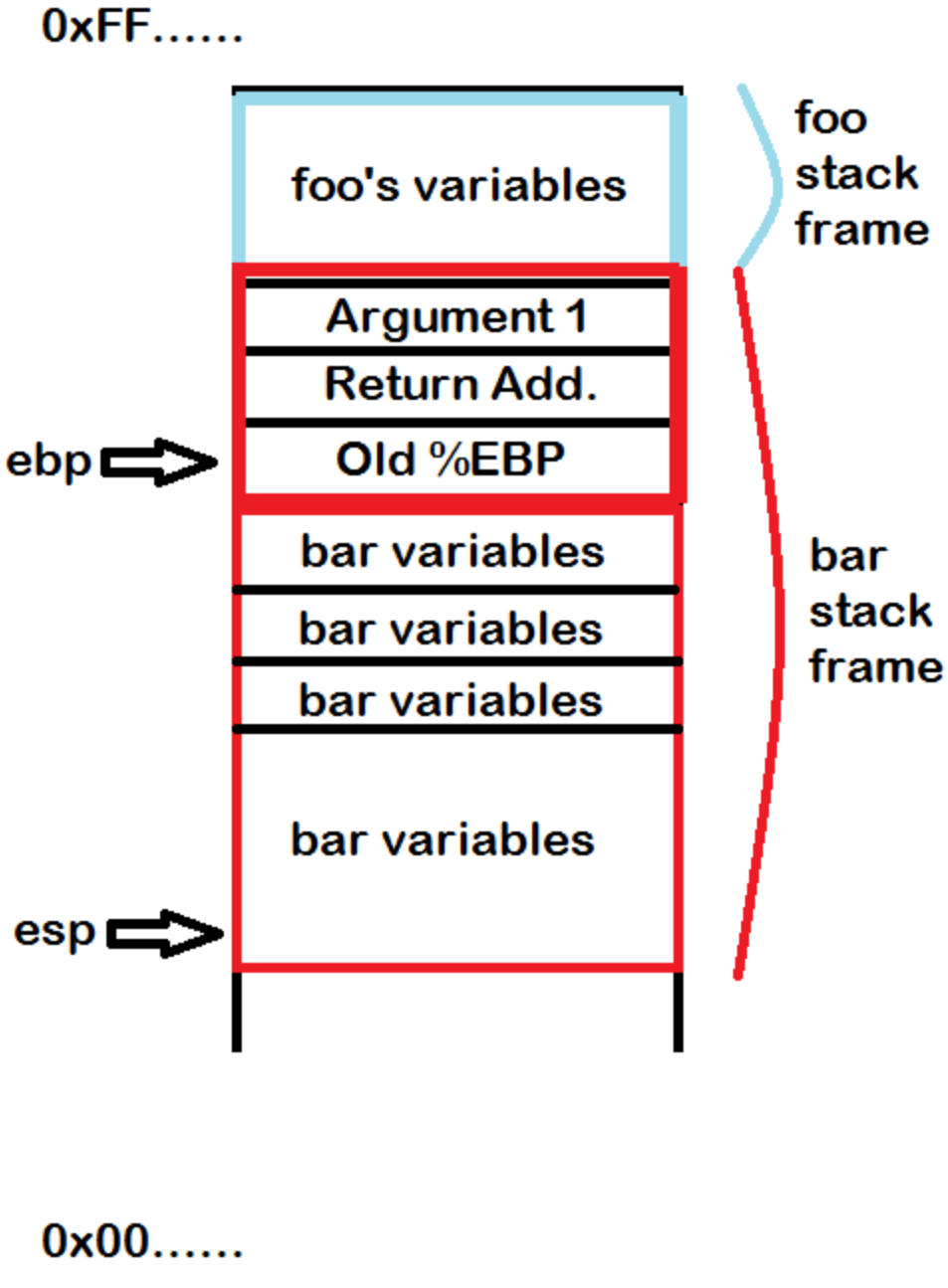
Stacks

- Vital role in handling procedure calls
- Similar to “Stack” data structure
- FILO

- %esp => points to the top of the stack
- %ebp => points to the base of the stack

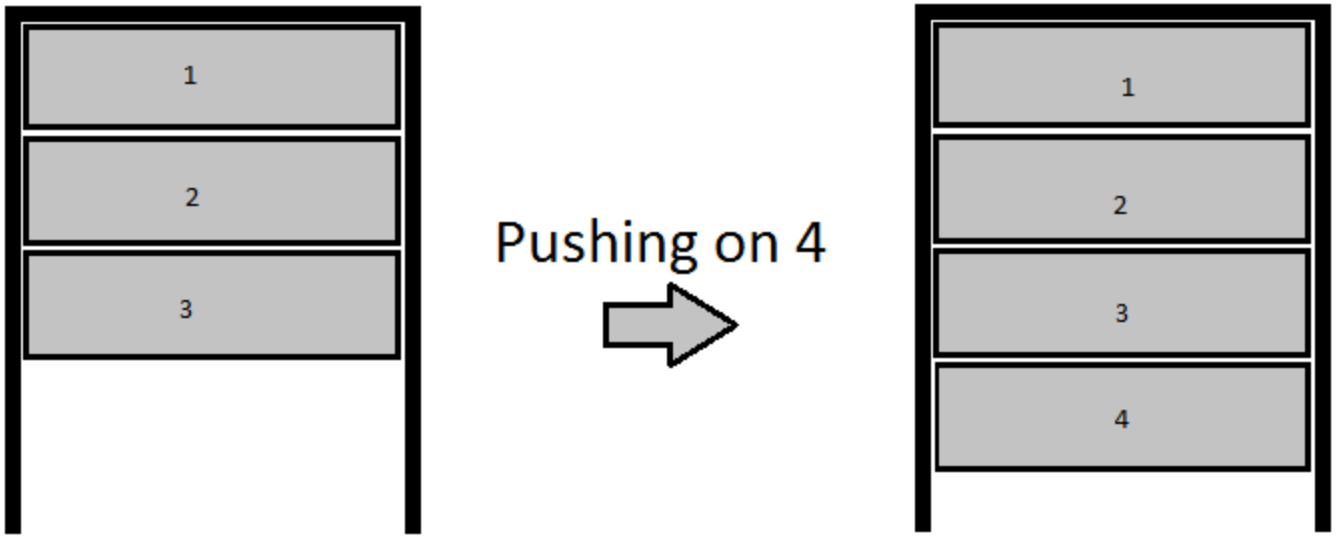
Example

- Example stack
  - foo calls:
    - bar(argument 1)



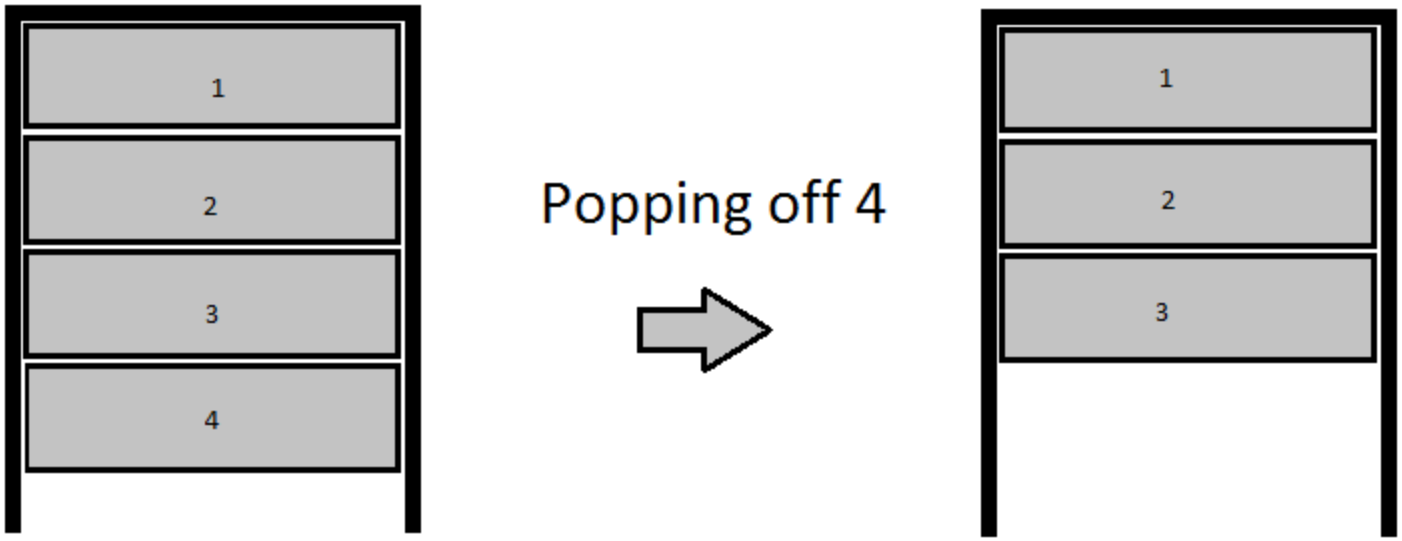
Operations on Stacks

- **PUSH** – pushes an element onto the stack



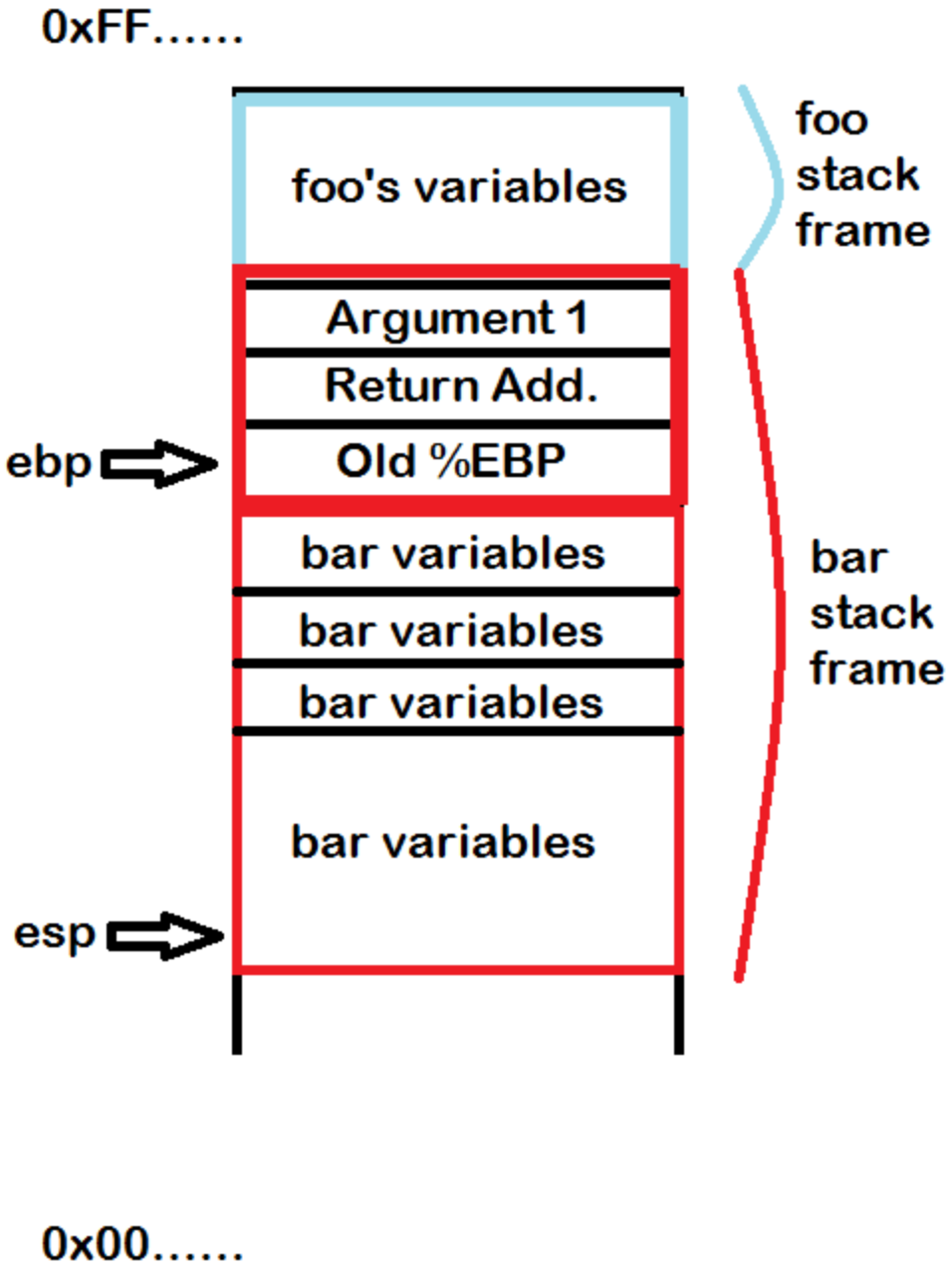
Operations on Stacks (2)

- **POP** – pops an element off of the stack



Function Calls

- Stack layout for a function call



Function Calls (2)

- Function Parameters
  - Pushed on by the calling function
- First parameter starts at %EBP + 8
  - Why?
- Calling foo(x, y, z)
  - In what order do we push the arguments on the stack and why?

Function Calls (3)

- Return address
  - What is it's address in terms of %EBP?
- For the called function to return
- (This will be a target for buflab)

Function Calls (4)

- Saved %EBP
  - Positioned above the last stack frame
- Remember,
  - %ESP = %EBP
  - %EBP = popped old %EBP
  - Pop the return address
- %EBP and %ESP are back to their old values

Today

- BombLab
- Assembly Review
- Stacks
  - EBP / ESP
- **Stack Discipline**
- Buffer Overflow
- BufLab

- Summary

Stack Discipline

- %ebp
  - Where does it point?
  - What happens during a function call?
- %esp
  - Where does it point?
  - What happens during a function call?

Stack Discipline (2)

- Order of objects on the stack
  - Argument 2
  - Argument 1
  - Return Address
  - Saved %ebp
  - Local variables for called function
- Grows downwards!

Stack Discipline (3)

- Calling a function
  - Push arguments
  - Push return address
  - Jump to new function
  - Save old %ebp on stack
  - Subtract from stack pointer to make space

Stack Discipline (4)

- Returning
  - Pop the old %ebp
  - Pop the return address and return to it
    - Think eip = stack.pop()

Stack Discipline (5)

- Useful things
  - Return address
    - %ebp + 4
  - Old %ebp
    - %ebp
  - Argument 1
    - %ebp + 8
  - Argument 2
    - %ebp + 12

Today

- BombLab
- Assembly Review
- Stacks
  - EBP / ESP
- **Buffer Overflow**
- BufLab

- Summary

Buffer Overflows

- A method of gaining control over a program
- Actual exploitation
  - Server is running a program
  - Buffer Overflow vulnerability
  - Take control of program => Take control of server

Back to the Stack

- Calling the function foo(1, 2)
  - Note how the stack is set up (useful for BufLab)

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
BUF[11] BUF[10] BUF[9] BUF[8]	Last 4 bytes of BUF
BUF[7] BUF[6] BUF[5] BUF[4]	Middle 4 bytes of BUF
BUF[3] BUF[2] BUF[1] BUF[0]	First 4 bytes of BUF
…	

Back to the Stack (2)

- strcpy(BUF, userInput) //char BUF[12]
- Let user input = 0x1234567890ABCDEFDEADBEEEF

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
BUF[11] BUF[10] BUF[9] BUF[8]	Last 4 bytes of BUF
BUF[7] BUF[6] BUF[5] BUF[4]	Middle 4 bytes of BUF
BUF[3] BUF[2] BUF[1] BUF[0]	First 4 bytes of BUF
…	

Back to the Stack (3)

- Let user input = 0x1234567890ABCDEFDEADBEEEF
- First 4 copied in

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
BUF[11] BUF[10] BUF[9] BUF[8]	Last 4 bytes of BUF
BUF[7] BUF[6] BUF[5] BUF[4]	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (4)

- Let user input = 0x1234567890ABCDEFDEADBEEEF
- Next 4

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
BUF[11] BUF[10] BUF[9] BUF[8]	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (5)

- Let user input = 0x1234567890ABCDEFDEADBEEF
- Last 4 **available** bytes

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (6)

- Let user input = 0x1234567890ABCDEFDEADBEEF
- What if the user entered in 8 more bytes?

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0xFF FF FF C8	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (7)

- Let user input = 0x1234567890ABCDEFDEADBEEF
- Concatenate 0x**11223344**55667788

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x08 06 5A AD	Return Address
0x**44 33 22 11**	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (7)

- Let user input = 0x1234567890ABCDEFDEADBEEF
- Concatenate 0x1122334455667788

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x**88 77 66 55**	Return Address
0x**44 33 22 11**	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (8)

- Oh no! We've overwritten the return address
- What happens when the function returns?

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x**88 77 66 55**	Return Address
0x**44 33 22 11**	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (9)

- Function will return to 0x55667788
  - Controlled by user

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x**88 77 66 55**	Return Address
0x**44 33 22 11**	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Back to the Stack (10)

- Instead of entering garbage, we could've entered arbitrary code. Then we'd have control of the program.

0x00 00 00 02	Argument 2
0x00 00 00 01	Argument 1
0x**88 77 66 55**	Return Address
0x**44 33 22 11**	Old %EBP
0xEF BE AD DE	Last 4 bytes of BUF
0xEF CD AB 90	Middle 4 bytes of BUF
0x78 56 34 12	First 4 bytes of BUF
…	

Today

- BombLab
- Assembly Review
- Stacks
  - EBP / ESP
- Buffer Overflow
- BufLab

- Summary

BufLab

- Buffer Overflows are the premise of BufLab
- You will inject code, then make the program execute your code
- You can use this to branch to other existing functions, set arbitrary values in variables, or execute anything you want!

Today

- BombLab
- Assembly Review
- Stacks
  - EBP / ESP
- Buffer Overflow
- BufLab

- Summary

Summary – Assembly/Stacks

- Purpose of %ebp
- Purpose of %esp

Summary – Stacks

- Essential for function calls
- 3 important things stored on stack:
  - Arguments
  - Return address
  - Old %EBP
- Which would be a target for BufLab?

Summary – Buffer Overflows

- Unbounded string copy
- Allow a user to overwrite any part of the stack
- Can execute arbitrary code
  - Set variables
  - Call functions

The End

- Questions?
- Good luck on BufLab