

The x86 Software Architecture

Computer Architecture Exploitation and Security

Table of Contents

[Objectives 2](#_Toc74089873)

[Background Reading 2](#_Toc74089874)

[Problem 1 Opcode and Intel Instructions \_\_/14 3](#_Toc74089875)

[A. In the Intel Software Developer’s Manual (SDM), use: 3](#_Toc74089876)

[B. Explore Intel Manual Vol 2 Chapters 3 and 4 “Instruction Set Reference” to find the opcode of the following instructions: 3](#_Toc74089877)

[Problem 2 - PUSH and POP Instructions \_\_\_ 10 4](#_Toc74089878)

[Problem 3 - CALL, RET and the Stack \_\_\_\_/14 5](#_Toc74089879)

[Problem 4 - Stack Analysis \_\_\_/14 7](#_Toc74089880)

[Problem 5 - Stack analysis (near call) \_\_\_\_/10 8](#_Toc74089881)

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

Signature: \_\_Coleton Sanheim\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EVALUATION**:

|  |  |  |
| --- | --- | --- |
| Opcode and Intel Instructions | 14 |  |
| Push and Pop Instructions | 10 |  |
| CALL , RET instructions and the stack | 14 |  |
| Stack analysis (System V 64bits) | 14 |  |
| Stack analysis (32bits CDECL) | 10 |  |
| TOTAL MARK | 62 |  |

Computer Architecture Exploitation and Security

The x86 Software Architecture

Objectives

This lab focuses on the following objectives:

* Describe Intel instruction format.
* Describe addressing modes.
* Explain stack structure.
* Explain calling conventions.
* Explain interrupts and exceptions.
* Explain procedure calls and frame pointers for x86 architecture.
* Describe instruction encoding.

Background Reading

The following manual may be available at SAIT (check with your instructor). Otherwise, download a free copy of the manual below. Specific figures in the manual are referred to in this lab.

*Intel 64 and IA-32 Architectures Software Developer’s Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C*

<https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>

# Problem 1 Opcode and Intel Instructions \_\_/14

## In the Intel Software Developer’s Manual (SDM), use:

1. The **Opcode Map** (Vol. 2D, A7-A8) pg 2673-2674
2. 32-Bit Addressing Forms (*Vol. 2A 2-6*) pg 532, **Table 2-2**
3. ModR/M Byte (*Vol. 2A 2-3*) pg 529 to decode the mnemonic for the following instructions:

|  |  |
| --- | --- |
| **Opcodes** | **Instruction** |
| CC | INT3 |
| 89 EC | mov esp, ebp |
| 8B 45 08 | mov eax, ecx |
| 55 | PUSH rbp |
| C3 | near RET |
| 01 D8 | ADD eax, ebx |
| 8D 83 F8 FE FF FF |  |

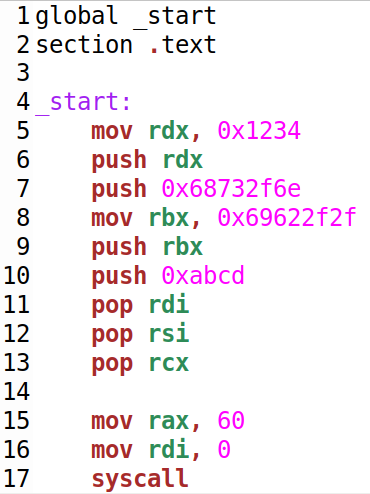
## Explore Intel Manual Vol 2 Chapters 3 and 4 “Instruction Set Reference” to find the opcode of the following instructions:

The page numbers vary so look for the instructions as describes in class.

|  |  |
| --- | --- |
| **Instruction** | **Opcode** |
| One byte NOP | 90 |
| syscall | 0F 05 |
| Call eax | FF D0 |
| pop esi | 07 |
| leave | C9 |
| xor eax,eax | 3 |
| sub esp,0x10 | 2D C4 10 |

# Problem 2 - PUSH and POP Instructions \_\_\_ 10

1. Use the following instructions <https://github.com/hugsy/gef> to Install **Gef** (GDB Enhanced Features). To learn more about gef and commands read <https://gef.readthedocs.io/en/master/>
2. Create the following assembly code in a text editor on your linux Virtual Machine



1. Use the commands **nasm** and **ld** to create the executable
2. Use gef commands:
   1. **break** **\_start**
   2. **step (s)** or **next (n)**

commands to analyze the **stack region** while pushing and popping data to/from the stack.

1. Observe how **$rsp** instruction address changes as data is pushed into the stack.
2. What is your observation? **Hint**: You can talk about: (**3pts**)
   1. What happens to the address in RSP after pushing

**The address decreases by 0x8**

* 1. What happens to the address in RSP after popping

**The address increases by 0x8**

* 1. What happens to the registers RDI, RSI and RCX

**When a value is popped it goes into the rdi, then rsi, then rcx registers**

1. What strings, were pushed into the stack? (**1pt**)
   1. When GDB interprets the information is there an English word or letters?

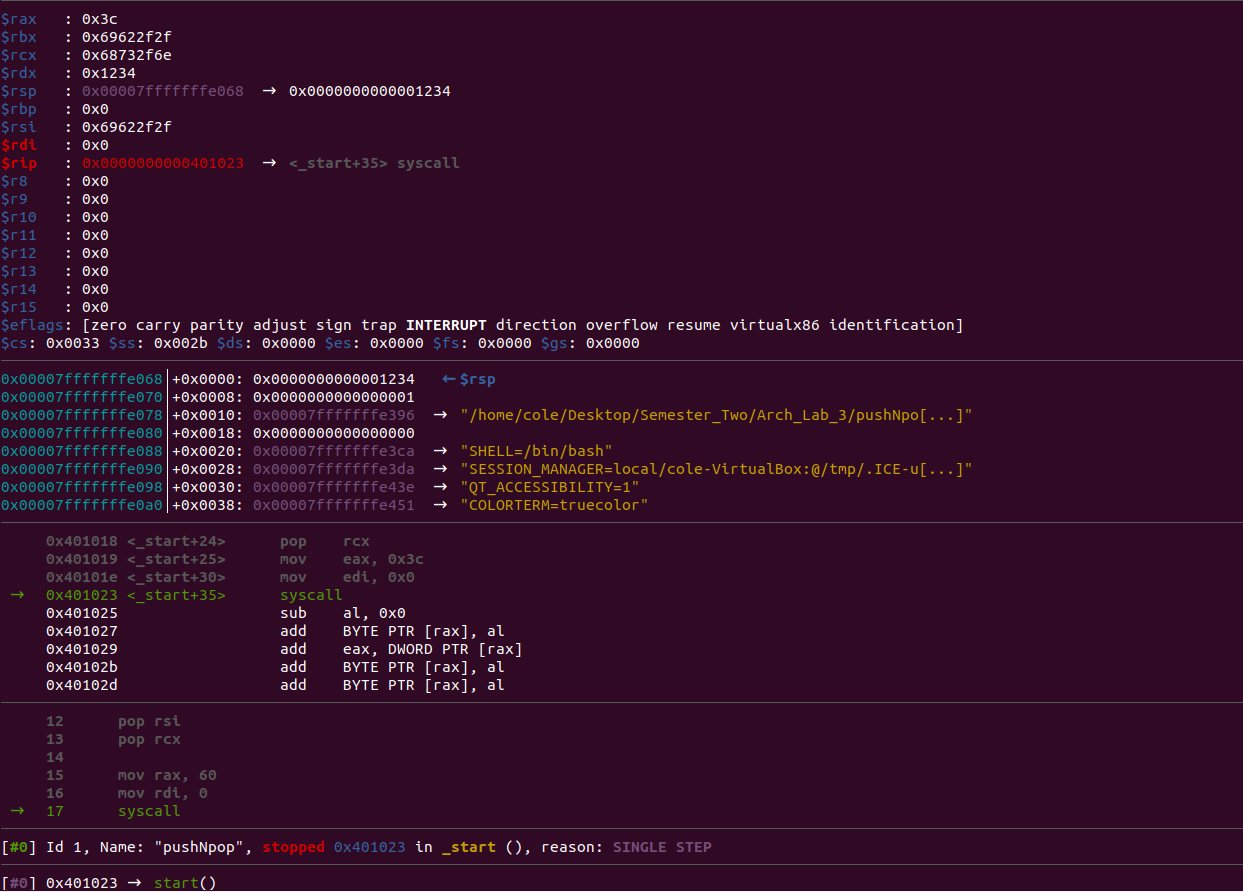
**Rip appears to contain the current instruction**

1. Can the push instruction be used to push more than 4 bytes into the stack at a time? Explain how this can be done. (**2pts**)

**As long as the stack is wide enough then you can use a 64-bit instruction to move 8 bytes**

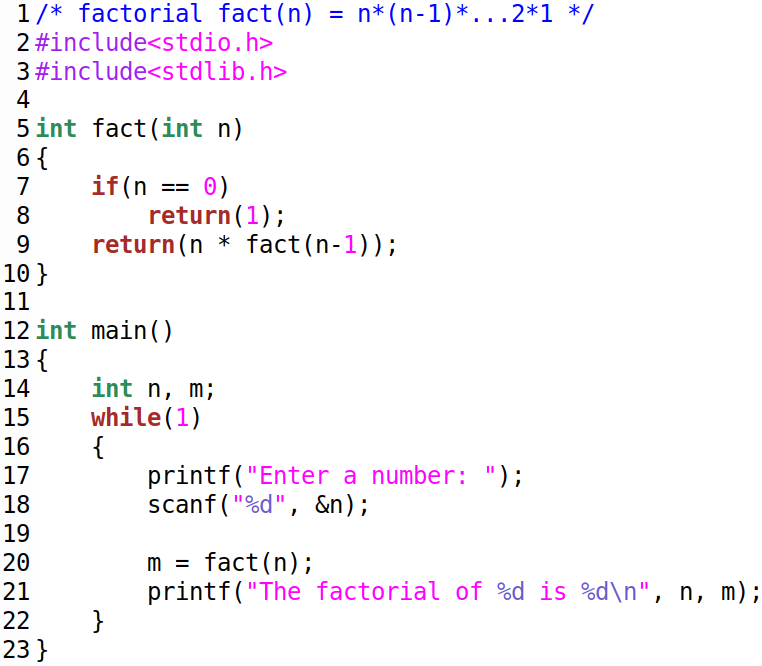
**Hint**: you may need to look at the intel manual to see if there are multiple forms of the push instruction.

1. Observe the registers content as data is pushed and popped off the stack.
2. Attach the screen capture that demonstrates data pushed and pop off the stack and the final content of the registers used (**4pts**)

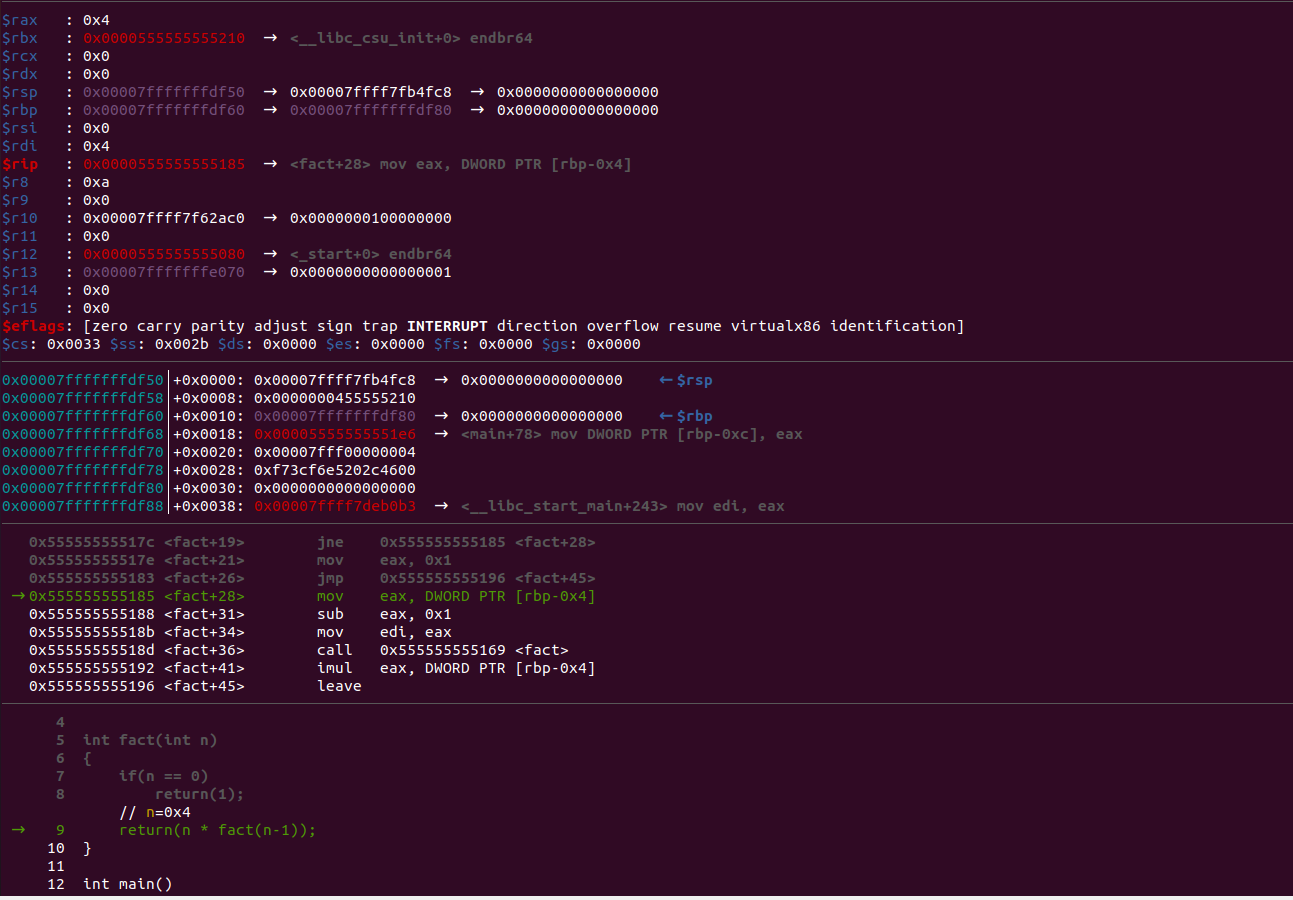


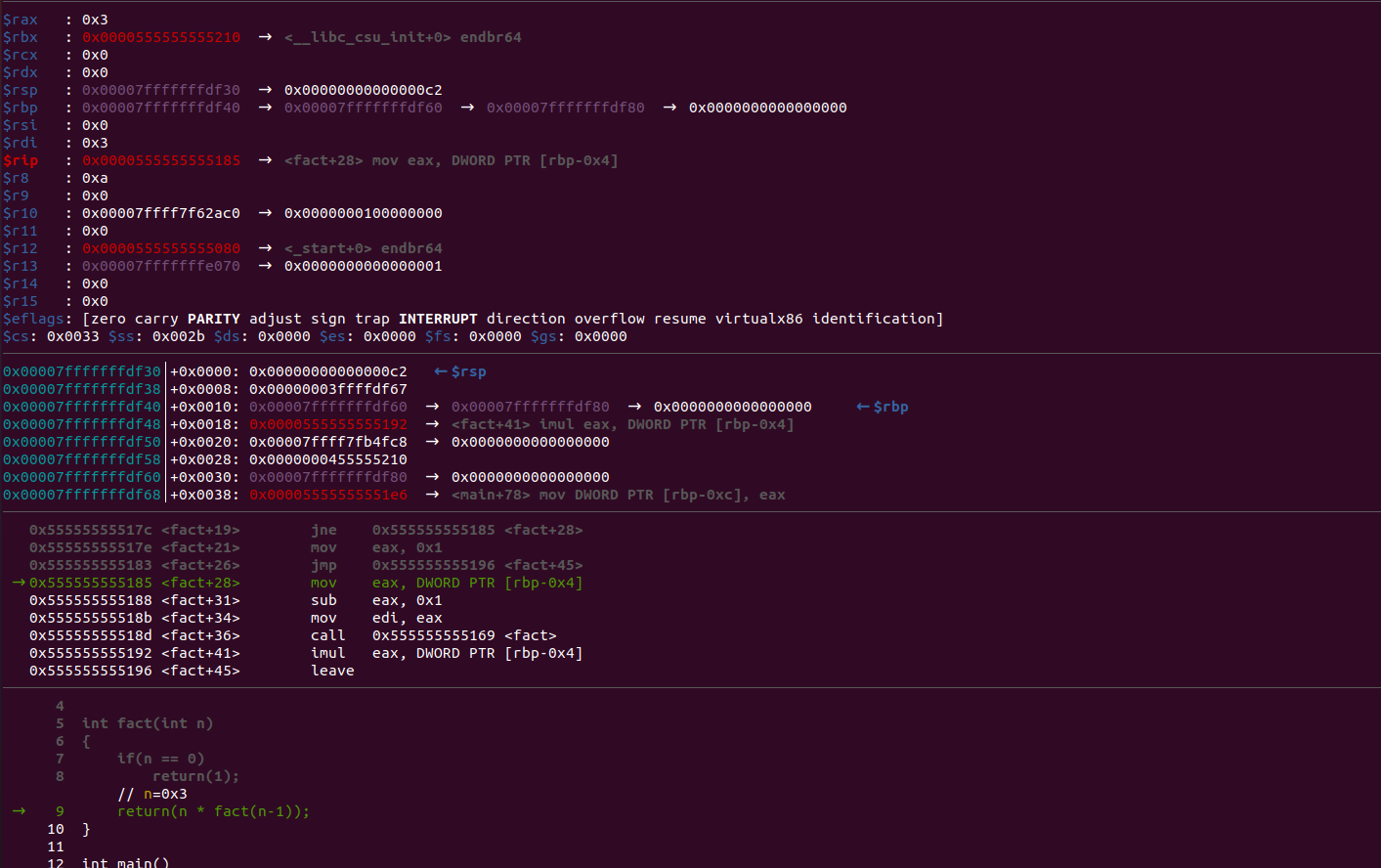
# Problem 3 - CALL, RET and the Stack \_\_\_\_/14

1. Create the following C code in your text editor and save it as **factorial.c**



1. Compile the code so that you can get debugging symbols that make debugging easier.
2. Use a debugger to disassembly the code and analyze instructions
   1. **disass** or **disass /r**
3. Create 2 break points
   1. **main** function
   2. **fact** function
4. Run the program
   1. **run**
5. Use **next** or **stepi** to move to next instruction and analyze the stack in each step needed to calculate the factorial of a provided integer.
6. Provide your observations as images with some explanation text. (**3pts**)
   1. Show images of the registers and stack for 2 iterations of the loop calculating the factorial





* 1. In your image highlight the register change and observe the relationship between registers and variables in C.

**It appears that the rbp register displays each recursion of the function, and it appears that registers store the values that variables contain**

1. Analyze the instructions used in the fact function. Which **conditional** jump instruction(s) were used in the code? (**1pt**)

**JNE instruction is used where it jumps IF the Zero Flag (ZF) is cleared (set to 0)**

* 1. Conditional instructions are instructions that perform an action based on a flag status.
     1. ***Example JNA is used for unsigned operations in which the result of a calculation is greater than 0****.*
  2. Unconditional instructions ignore any flag conditions and always executes.

1. What was the purpose of **cmp**? (**1pt**)

**In this instance it is used to calculate the flag for JNE**

1. Use **disas main** to disassembly main. How many call instructions are found in the main function? (**1pt**)

**four**

1. What is the address of the fact function? (**1pt**)

**0x555555555169**

1. Now disassemble the fact function. (**3pts**)
   1. What is the purpose of **ret** instruction?

**Returns control back to the main function**

* 1. After the instruction executes what is the address of **RIP** register?

**0x00005555555551e9**

* 1. What instruction is directly above this instruction?

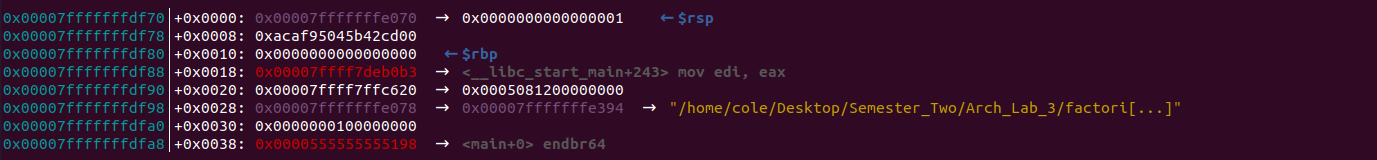
**LEAVE**

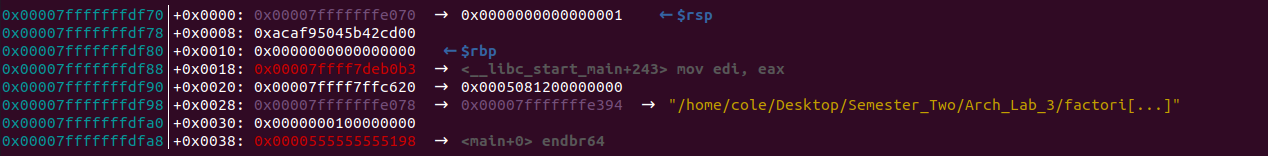
1. What is the purpose of the call <fact> instruction within this function fact? (**1pt**)

**To recursively call this function to complete the calculation**

1. Attach the screen capture that demonstrates three changes you observed in the stack (**3pts**)



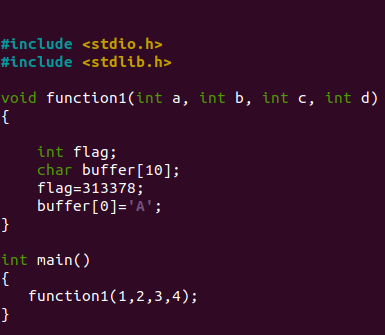




**You can see the RSP register changing value, you can see the RBP register appearing and you can see main being initialized**

# Problem 4 - Stack Analysis \_\_\_/14

1. Create the following C code and save it as **function1.c**



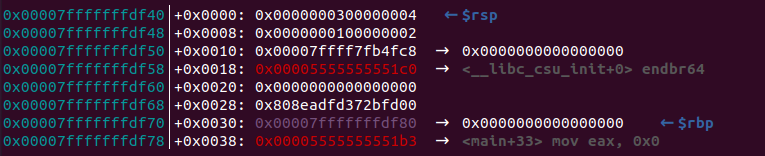
1. Compile the code and run it using the debugger
   1. **gdb -q ./a.out**
2. Create break points for
   1. main
   2. function1
3. Run the program to the beginning of main
4. Verify the current values of registers **RSP**, **RBP** and **RIP­**. Record those values: (**1pt**)

|  |  |
| --- | --- |
| Register | Value stored in the register |
| RSP | 0x00007ffff7deb0b3 |
| RIP | 0x0000555555555192 |
| RBP | 0x0 |

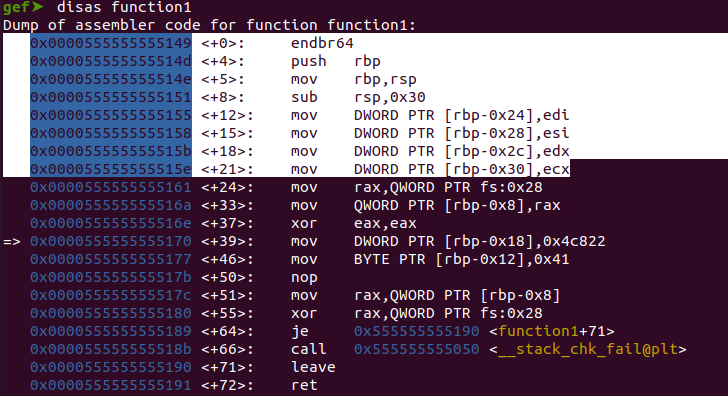
1. Use **x/5i $RIP** - to verify the next 5 instructions to execute

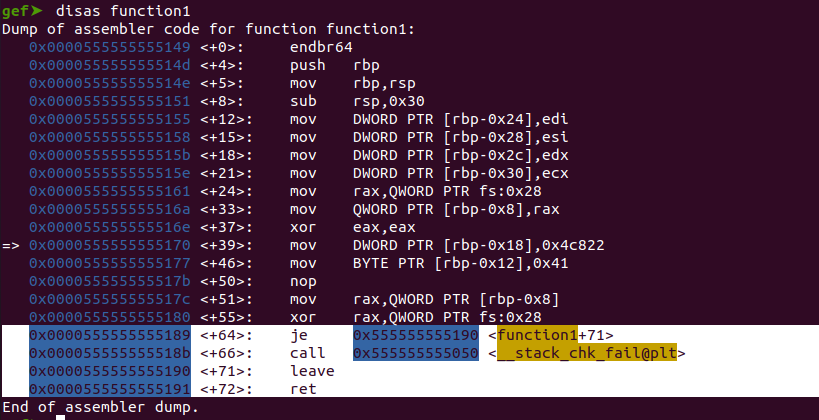
**This command doesn’t work, it only gives errors**

1. Press **c** to continue. You may also need to execute **n** to execute the prolog; creating the stack frame for function1.
2. After executing continue from step 7, document all the state of the stack. Use a table or images. Pay attention to the values on the stack and the new location of the RSP and RBP registers. (**3pts**)



1. Observe the **RSP** and **RBP** registers, the addresses between these registers represents the stack frame for this function
2. **disas function1** and identify the **prolog** and **epilog** of this function. Provide a screenshot clearly **highlighting** the **Prolog** and **Epilog**. (**2pts**)





1. What is the purpose of the prolog and epilog? (**2pts**)

**The prolog sets the stack frame to the current function, the epilog sets the stack frame back to the main function**

1. What is the purpose of the instruction **sub rsp, 0xYY** in the prolog of function1?

**It is allocating space for the function arguments**

**NOTE***: 0xYY for some of you will be* ***0x30*** *and others will get a different value. The idea is still the same.*

1. Use **x/16w $rsp** and identify the following: (**4pts**)
   1. Function arguments

**0x4, 0x3, 0x2, 0x1**

* 1. Function local variables address

**0xf7fb4fc8 0x7fff 0x555551c0 0x5555**

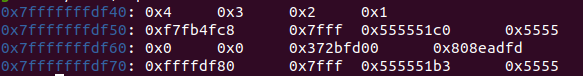
* 1. Return address

**0x555551b3 0x5555**

* 1. Saved frame pointer or old RBP( saved RBP)

**0x0 0x0**

1. Attach screen capture with identified arguments, local variables, ret and saved rbp (**2pts**)



# Problem 5 - Stack analysis (near call) \_\_\_\_/10

For reference read Intel manual Vol. 1 6-4, **Section 6.4** pg 154 and 156

Analyze the functions in the following code and complete the table.

The operations for this stack will follow CDEL calling convention, which uses the stack to pass arguments to the function being called.

int functionA(int arg1, int arg2, int arg3, int arg4, int arg5)

{

int local1;

int local2;

int local3;

local2 = arg2 + arg1;

local1 = arg2 – arg3;

local3 = arg4 \* arg5;

arg5 = functionB(local2,local1,local3, arg1, arg2, arg3);

return (arg5);

}

The table below represents a fictitious stack. The table shows the Stack Pointer (**ESP**) marked location at the moment just before the first instruction of functionA() is executed. Assume that both functionA() and functionB() reside in the **same code segment**, therefore a near call will be executed.

Complete the following table:

1. Indicate, using the variable names, which values will be on the stack at the exact moment before the first instruction of functionB() is executed.
2. The Memory Data column is used for the variable names associated with the data on the stack.

|  |  |  |
| --- | --- | --- |
|  | Address | Memory Data |
| EBP | 124 |  |
|  | 128 |  |
|  | 12c | Local3 |
|  | 130 | Local1 |
|  | 134 | Local2 |
|  | 138 |  |
|  | 13C | Arg1 |
|  | 140 | Arg2 |
|  | 144 | Arg3 |
|  | 148 | Arg4 |
|  | 14c | Arg5 |
| ESP | 150 |  |
|  | 154 |  |
|  | 158 |  |
|  | 15C |  |
|  | 160 |  |