HO# 3.2: Recap of x86-64 Assembly & Debuggers

A Bit of Word about Assembly Programming

Assembly language is a critical component in the realm of computer programming, reverse engineering, malware development/analysis and systems development. Assembly language is a low-level programming language that provides a symbolic representation of a computer's machine code. It serves as an intermediary between high-level programming languages and the raw binary instructions executed by a computer’s CPU.

- **Brief History of Assembly Language**
  1. **Early Beginnings (1940s-1950s):**
     a. **Origins**: Assembly language has roots in the early days of computing. In the 1940s and 1950s, computers were programmed using machine code, the lowest-level language consisting of binary instructions.
     b. **First Assemblers**: To simplify programming, early assemblers were developed. These tools translate assembly language, which uses mnemonics and symbolic names, into machine code. This abstraction made programming more manageable than directly using binary.
  2. **Development Through the 1960s-1980s:**
     a. **Improved Assemblers**: Throughout the 1960s and 1970s, assembly language evolved with better assemblers and debuggers. It became standard for writing system software and performance-critical applications.
     b. **High-Level Languages**: While high-level programming languages like C, Fortran, and COBOL began to dominate, assembly language remained important for tasks requiring fine-grained control over hardware.
  3. **Modern Era (1990s-Present):**
     a. **Microprocessors and Assemblers**: As microprocessors became ubiquitous in the 1980s and 1990s, assembly language continued to be used for low-level programming, particularly in embedded systems, device drivers, and operating systems.
     b. **Optimizations**: In the 2000s and beyond, assembly language was primarily used for performance optimization, reverse engineering, and systems programming.

- **Features/Characteristics of Assembly Language**
  1. **Low-Level Control:**
     - **Direct Hardware Interaction**: Assembly language provides direct control over hardware, allowing precise manipulation of processor registers, memory addresses, and I/O ports.
     - **Mnemonics**: Instead of using binary or hexadecimal numbers, assembly language uses mnemonics to represent machine instructions. For example, `MOV` to move data, `ADD` to add numbers, and `SUB` to subtract.
     - **Symbolic Addresses**: Assembly language allows the use of symbolic names for memory addresses and constants. For instance, `var1` might be used instead of a numerical address.

2. **Efficiency:**
    - **High Performance**: Code written in assembly can be highly optimized for performance and space, often achieving better performance than code written in higher-level languages.
    - **Compact Code**: Assembly code can be more compact and efficient, making it suitable for resource-constrained environments.
3. **Platform-Specific:**
    - **Architecture Dependent**: Assembly language is specific to a particular CPU architecture (`e.g., x86, ARM`). Each architecture has its own assembly language with unique instructions and registers. This means assembly code is generally not portable between different types of processors.
4. **Debugging and Optimization:**
    - **Detailed Debugging**: Assembly language allows for detailed debugging at the instruction level, which can be crucial for troubleshooting low-level issues.
    - **Optimizations**: It enables optimization of critical code sections, especially where performance is paramount, such as in embedded systems or high-performance computing.

- <u>**Historical and Modern Uses**</u>
    1. **Past Uses:**
        a. **Early Computer Programming**: Assembly language was widely used in the early days of computing for writing operating systems, compilers, and system utilities.
        b. **Embedded Systems**: Used extensively in embedded systems where direct hardware control and optimization were necessary.
    2. **Present Uses (2024):**
        a. **Embedded Systems**: Still used in embedded systems for microcontrollers and processors where efficiency and direct hardware access are crucial.
        b. **Performance Optimization**: Employed for performance-critical sections of software where high performance and low overhead are required.
        c. **Reverse Engineering and Security**: Used in reverse engineering and security research to understand malware, exploit vulnerabilities, and analyze compiled binaries.
        d. **Educational Purposes**: Taught as a foundational subject to understand computer architecture and low-level programming concepts.

- <u>**Importance and Uses**</u>
    - **Performance Optimization**: Assembly language enables programmers to write highly optimized code that can outperform high-level language implementations, especially in performance-critical applications.
    - **System Programming**: It is used for developing system software like operating systems, device drivers, and embedded firmware.
    - **Embedded Systems**: In embedded systems, where resources are limited, and performance is critical, assembly language helps in creating efficient and compact code.
    - **Reverse Engineering and Security**: Assembly language is vital in reverse engineering and cybersecurity for understanding and analyzing executable binaries, discovering vulnerabilities, and developing exploits or patches.
    - **Educational Value**: Learning assembly language provides deep insights into computer architecture and low-level programming concepts, helping developers understand how high-level languages interact with hardware.

Assembly Languages are Processor Specific

Assembly language is inherently tied to the architecture of the processor on which it runs. This means that assembly languages are specific to different CPU architectures, each with its own set of instructions, registers, and addressing modes. Here's a detailed explanation:

- **Key Concepts**
  1. **Architecture-Specific Instructions**:
     - **Instruction Set**: Each CPU architecture has a unique instruction set, which is the collection of all the instructions that the CPU can execute. For instance, Intel's x86 processors have a different set of instructions compared to ARM processors.
     - **Instruction Format**: The format of instructions, including how they are encoded and how operands are specified, varies between architectures. This affects how assembly language code is written and interpreted.
  2. **Registers**:
     - **Register Names and Sizes**: Different architectures have different sets of registers with varying names, sizes, and purposes. For example, x86 processors have registers like `EAX, EBX, ECX,` while ARM processors use `R0, R1, R2`, etc.
     - **Usage and Functions**: Registers in one architecture may serve different functions compared to those in another. For example, general-purpose registers in x86 might differ in their usage compared to ARM registers.
  3. **Addressing Modes**:
     - **Memory Access**: Different architectures have different methods for addressing memory. For instance, x86 architecture supports complex addressing modes such as base-plus-index, while ARM may use simpler or different modes.
  4. **Instruction Semantics**:
     - **Operation Behavior**: The behavior of instructions can vary. For example, an `ADD` instruction in x86 and ARM might operate differently or have different effects depending on the architecture's design.

- **Examples of Processor-Specific Assembly Languages**
  1. **x86 Assembly**:
     - **Architecture**: Developed by Intel, the x86 assembly language is used for Intel and compatible CPUs (e.g., AMD).
     - **Instructions**: Includes instructions like `MOV, ADD, SUB, JMP, CALL.`
     - **Registers**: Includes registers like `EAX, EBX, ECX, EDX` (32-bit), and `RAX, RBX, RCX, RDX` (64-bit in x86-64).

```
mov eax, 1     ; Move 1 into register EAX
add eax, 2     ; Add 2 to EAX
```

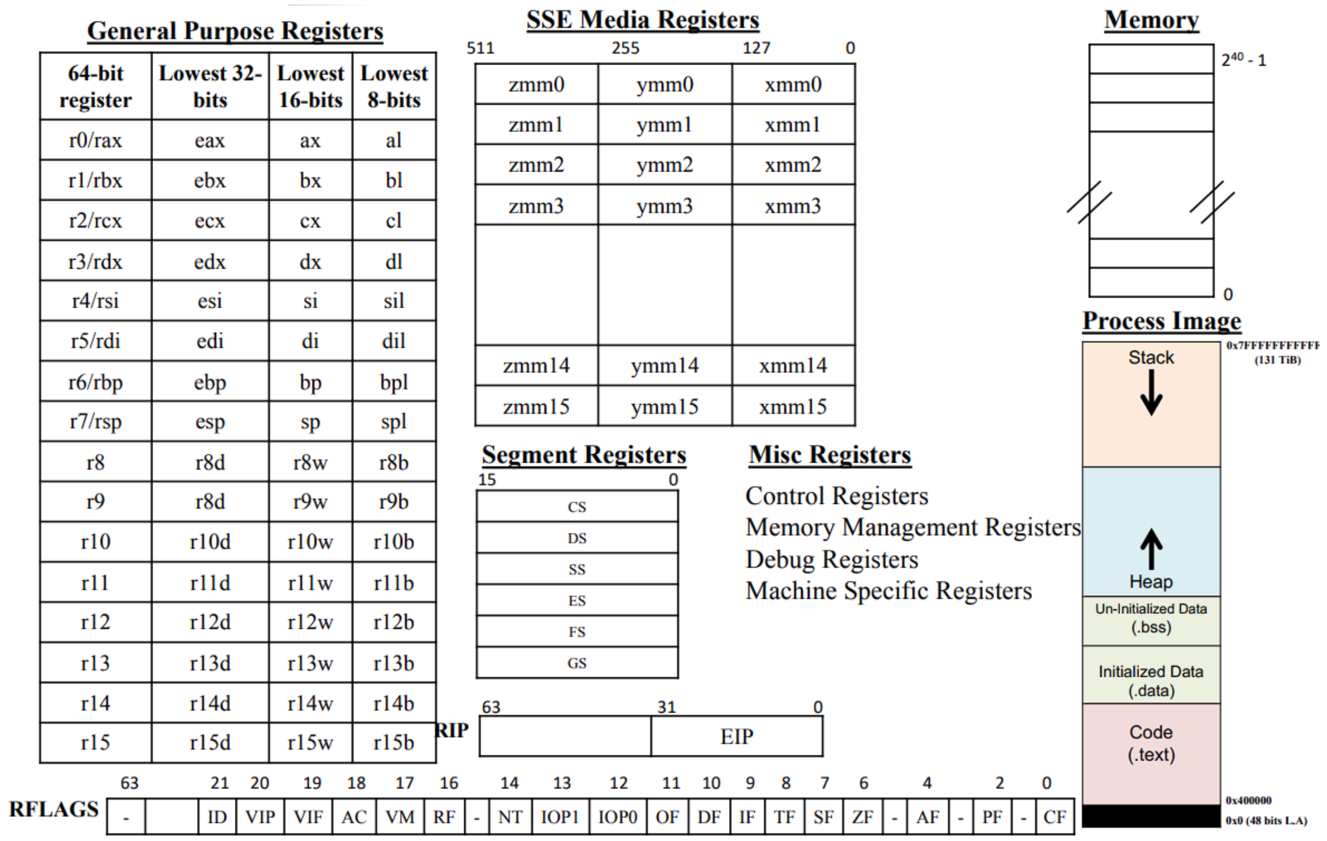
  2. **ARM Assembly**:
     - **Architecture**: Used in ARM processors, common in mobile devices and embedded systems.
     - **Instructions**: Includes instructions like `MOV, ADD, SUB, B` (branch).
     - **Registers**: Uses registers like `R0, R1, R2, R3.`

```
MOV R0, #1     ; Move the value 1 into register R0
ADD R0, #2     ; Add 2 to R0
```

- **Compatibility and Portability Issues**
  1. **Incompatibility**: Code written in assembly language for one architecture cannot be directly executed on another due to differences in instruction sets, registers, and memory models. For example, an assembly program written for an x86 processor will not work on an ARM processor without modification.
  2. **Porting**: To run software on different architectures, it often needs to be ported. This involves translating or rewriting the code to be compatible with the target architecture. This can be done manually by rewriting the assembly code or by using high-level languages with cross-compilation.

- **Workarounds for Cross-Architecture Execution**
  1. **Cross-Compilation**:
     - **Toolchains**: Cross-compilers can translate high-level code written in languages like C or C++ into assembly code for different architectures. This allows software to be compiled for different platforms without manually writing architecture-specific assembly code.
  2. **Emulation and Virtualization**:
     - **Emulators**: Emulators can simulate a different CPU architecture on the current hardware, allowing software designed for one architecture to run on another.
     - **Virtual Machines**: Virtual machines can provide an abstraction layer that allows software to run on different hardware platforms.
  3. **Binary Translation**:
     - **Dynamic Binary Translation**: Some systems use dynamic binary translation to convert executable code from one architecture to another at runtime, allowing for execution of binaries across different platforms.

AMD x86-64 Processor Architecture



General Purpose Registers

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
r0/rax	eax	ax	al
r1/rbx	ebx	bx	bl
r2/rcx	ecx	cx	cl
r3/rdx	edx	dx	dl
r4/rsi	esi	si	sil
r5/rdi	edi	di	dil
r6/rbp	ebp	bp	bpl
r7/rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r8d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

- **Registers of x86-64 Processor**
  Registers are essentially places that the processor can store data. You can think of them as buckets which the processor can store information in. There are sixteen registers in x86-64 processor, and usage of some of the important registers are given below:
  - **rbp:** Base Pointer, points to the bottom of the current stack frame
  - **rsp:** Stack Pointer, points to the top of the current stack frame
  - **rip:** Instruction Pointer, points to the instruction to be executed
  - Arguments to a function are also passed via registers (rdi, rsi, rdx, rcx, r8, r9)
  - Return value from a function is passed in the rax register.
  - Flags Register: The **`rflags`** register is used for status and CPU control information. Out of the 64 bits, mostly are unused and reserved for future use. These flags are divided into three categories namely status flags, control flags (DF) and system flags (IF, TF, RF). A brief description of some important status flags is given below:
    - **Carry flag (CF)** holds the carry out after addition or the borrow in after subtraction out/in of msb (Identify an unsigned overflow)
    - **Parity flag (PF)** is the count of one bits in a number, expressed as odd or even, represented by 0 or 1 respectively
    - **Auxiliary flag (AF)** holds the carry out after addition or the borrow in after subtraction between bit position 3 and 4 of the result (BCD)
    - **Zero flag (ZF)** is set if the previous operation resulted in a zero result
    - **Sign flag (SF)** holds the msb of the result (sign bit) after an arithmetic or logic op
    - **Overflow flag (OF)** is set if the previous signed arithmetic operation resulted in an overflow

Watch: https://www.youtube.com/watch?v=sg3GIXvS36w&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=30

Categories of x86-64 Assembly Instructions

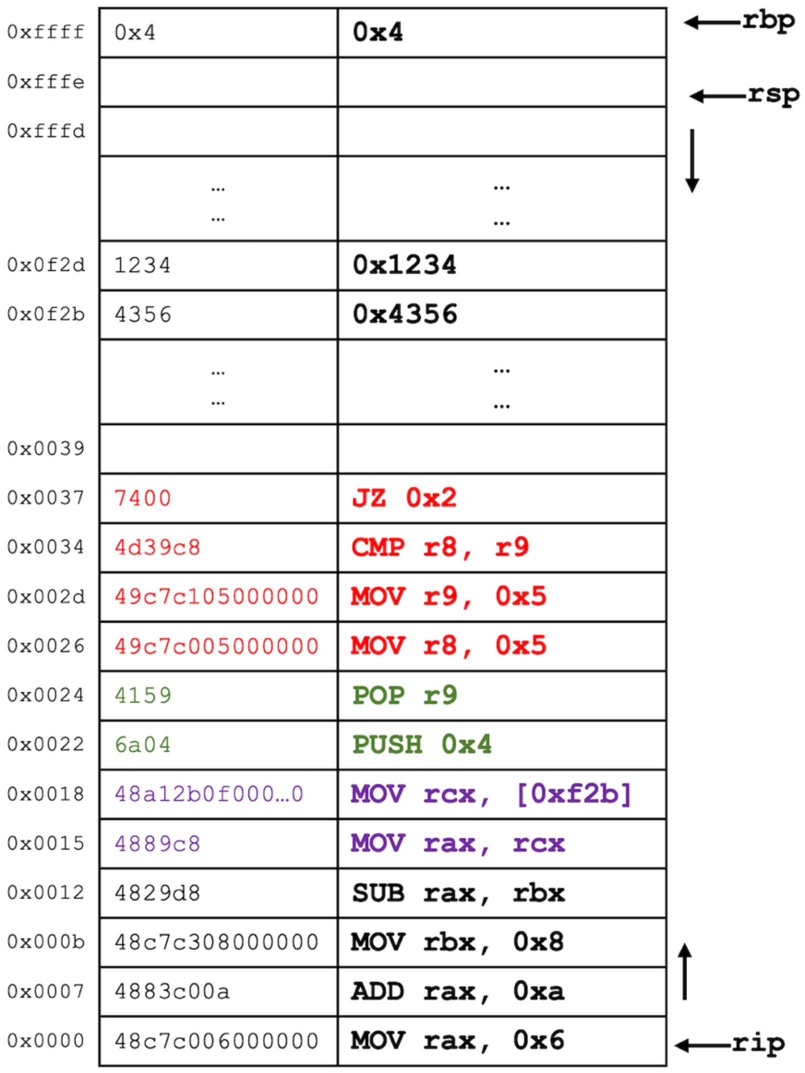
Category	Description	Examples
Data Transfer	Move from source to destination	mov, movzx, movsx, lea, lds, lss, xchg, push, pop, pusha, popa, pushf, popf
Arithmetic	Arithmetic on integer	add, addc, sub, subb, mul, imul, div, idiv, neg, inc, dec, cmp
Bit Manipulation	Logical & bit shifting operations	and, or, not, xor, test, shl/sal, shr, sar, ror, rol, rcr, rcl
Control Transfer	Conditional and unconditional jumps, and procedure calls	jmp jcc(jz,jnz,jg,jge,jl,jle,jc,jnc,...) call, ret
String	Move, compare, input and output	movsb, movsw, lodsb, lodsw, stosb, stosw, rep, repz, repe, repnz, repne
Floating Point	Arithmetic	fld, fst, fstp, fadd, fsub, fmul, fdiv
Conversion	Data type conversions	cbw, cwd, cdq, xlat
Input Output	For input and output	in, out
Miscellaneous	Manipulate individual flags	clc, stc, cld, std, sti

Note: A discussion on the working of all of the assembly instructions is beyond the scope of this handout. Interested students are advised to go through related Video Lectures (26 – 46) from the x86-64 Assembly Programming course at the following link:

Video URL: https://www.youtube.com/playlist?list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz

How CPU Executes an Assembly/Machine Language Code

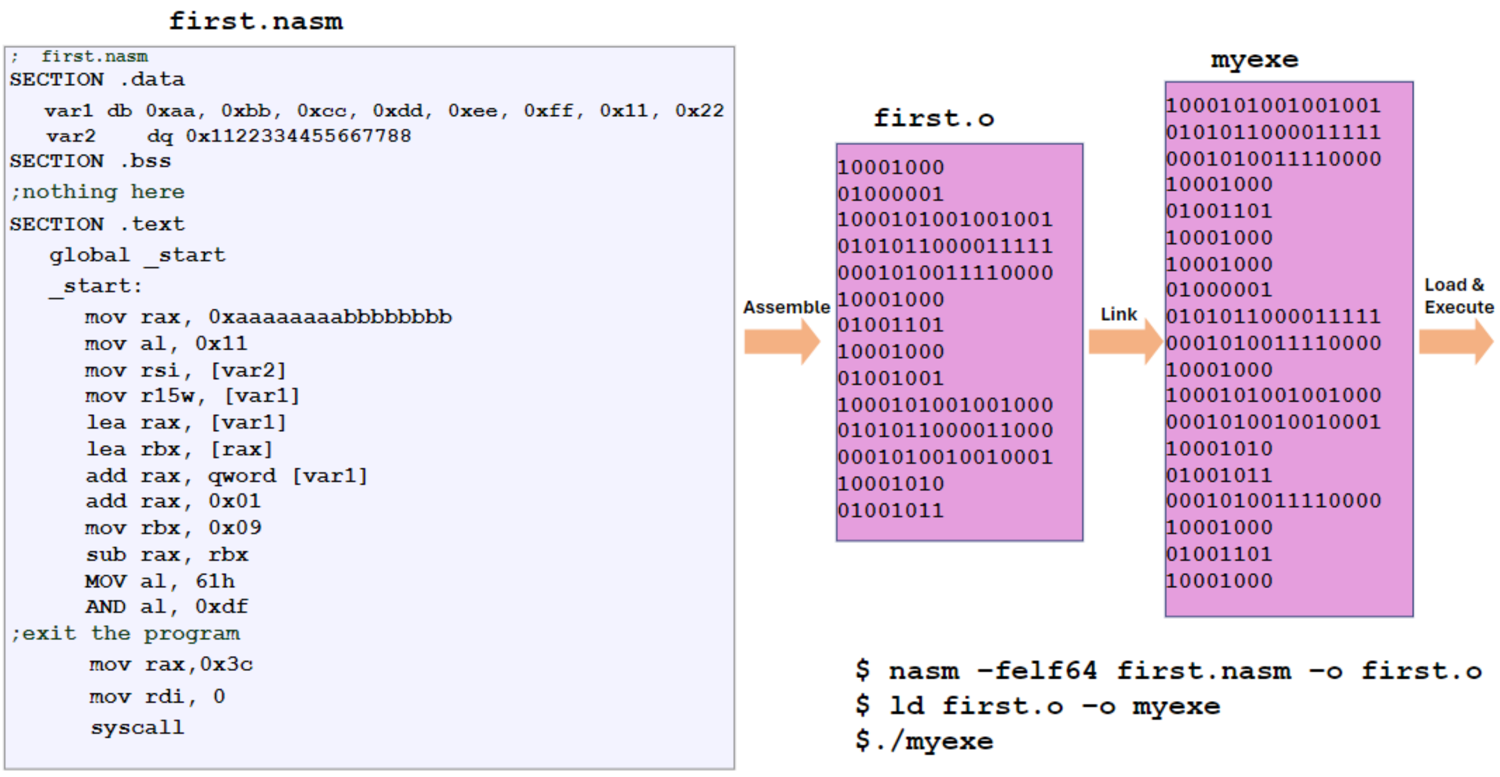
- The screenshot shows a hypothetical layout of memory containing the machine/assembly instruction at the very bottom, then we have two data values in the middle and at the very top we have the stack which grows down. The `x86_64` processor has **sixteen general-purpose 64-bit registers**. You can think of them as variables in programming languages, used to store temporary data and perform different **operations** on them (like `add, sub, mul, div, and, or, xor, cmp`, etc).
- In x86_64, the **Instruction Pointer** (`ip/eip/rip`) is a register which contains the address of the next instruction to be executed by the CPU. Every time an instruction is fetched and executed the Program Counter gets incremented automatically to the number of bytes specific to the size of that instruction. In RISC architectures (ARM, MIPS, SPARC, PowerPC) the size of each instruction is same/fixed, while in CISC architectures (x86, x86_64/AMD64, Motorola 68000, VAX) the size of each instruction is of variable length.
- In real programs, we may need many more variables to work with as compared to the number of registers available. Moreover, we may need to store data that might not fit inside a single register. Therefore, what doesn't fit in registers lives in memory. Memory is to an assembly program what the disk is to a Python program: you pull things out of memory into variables, do things with them, and eventually put them back into memory. An assembly programmer can access memory with loads and stores at addresses. In x86_64 we have the famous MOV instruction that is used to load/store data to and from the memory as if it were a big array.
- The top most addresses in the screenshot shows the process stack that is used to keep temporary variables by functions. The `rbp` (**base pointer**) is a register that points to the bottom of the current stack frame, `rsp` (**stack pointer**) points to the top of the current stack frame. In x86_64, the **process stack** grows from top to bottom, or from higher addresses to lower addresses. Assembly programmers, use `push` and `pop` instructions to write and remove data from the top of the stack, where the `rsp` is pointing. With each `push` the `rsp` is decremented (moves down) and with each `pop` the `rsp` is incremented (moves up). This increment and decrement is as per the width of the stack which is 8 bytes in x86_64.
- Control flow is done via GOTOs (**jumps**, **branches**, or **calls**), and these instructions actually alter the program counter directly. This is required if a programmer want to perform task1, if a condition evaluates to true and perform task2 if the condition evaluates to false. It is also required if a programmer wants to repeat a task multiple times. For control flow, the `rflags` register plays a vital role, where the (C)arry, (P)arity, (A)uxiliary, (Z)ero, (S)igned, (O)verflow are set as per the previous operation.
  - A `jmp` instruction is just an unconditional GOTO.
  - The conditional GOTO instruction in x86_64 are `jz, jnz, jg, jge, jl, jle,...,` based on some status flag, e.g., "GOTO this address only if the last arithmetic operation resulted in zero.
  - A **caller** is a function that calls/invokes another function (**callee**). In x86_64 the caller function prepares arguments, saves registers if needed, pushes the next instruction address on the stack, and transfer the control of execution to the first instruction of the callee. The callee creates its FSF on the stack, executes its code and may use the stack to store its local variables. Finally, the last instruction of the callee is a `ret` instruction, that pop the saved return address from the stack and place it in `rip`. This transfers the control of execution to the caller function.



0xffff	0x4	0x4	←rbp
0xfffe			←rsp
0xfffd			↓
	... ...	... ...	
0x0f2d	1234	0x1234	
0x0f2b	4356	0x4356	
	... ...	... ...	
0x0039			
0x0037	7400	JZ 0x2	
0x0034	4d39c8	CMP r8, r9	
0x002d	49c7c105000000	MOV r9, 0x5	
0x0026	49c7c005000000	MOV r8, 0x5	
0x0024	4159	POP r9	
0x0022	6a04	PUSH 0x4	
0x0018	48a12b0f000…0	MOV rcx, [0xf2b]	
0x0015	4889c8	MOV rax, rcx	
0x0012	4829d8	SUB rax, rbx	
0x000b	48c7c308000000	MOV rbx, 0x8	↑
0x0007	4883c00a	ADD rax, 0xa	
0x0000	48c7c006000000	MOV rax, 0x6	←rip

Structure of x86-64 Assembly Program

The following figure describes the structure of an x86-64 assembly program in a text file named `first.nasm`. We can assemble it using `nasm`, to get an object file name `first.o`. Finally, we need to link the object file with the standard C to make an executable named `myexe`, ready to be loaded inside the memory and executed.



```
; first.nasm
SECTION .data
   var1 db 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x11, 0x22
   var2    dq 0x1122334455667788
SECTION .bss
;nothing here
SECTION .text
   global _start
   _start:
      mov rax, 0xaaaaaaaabbbbbbbb
      mov al, 0x11
      mov rsi, [var2]
      mov r15w, [var1]
      lea rax, [var1]
      lea rbx, [rax]
      add rax, qword [var1]
      add rax, 0x01
      mov rbx, 0x09
      sub rax, rbx
      MOV al, 61h
      AND al, 0xdf
;exit the program
      mov rax,0x3c
      mov rdi, 0
      syscall
```

```
10001000
01000001
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
01001001
1000101001001000
0101011000011000
0001010010010001
10001010
01001011
```

```
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
10001000
01000001
0101011000011111
0001010011110000
10001000
1000101001001000
0001010010010001
10001010
01001011
0001010011110000
10001000
01001101
10001000
```

```
$ nasm -felf64 first.nasm -o first.o
$ ld first.o -o myexe
$./myexe
```

An assembly program is normally divided into three sections:

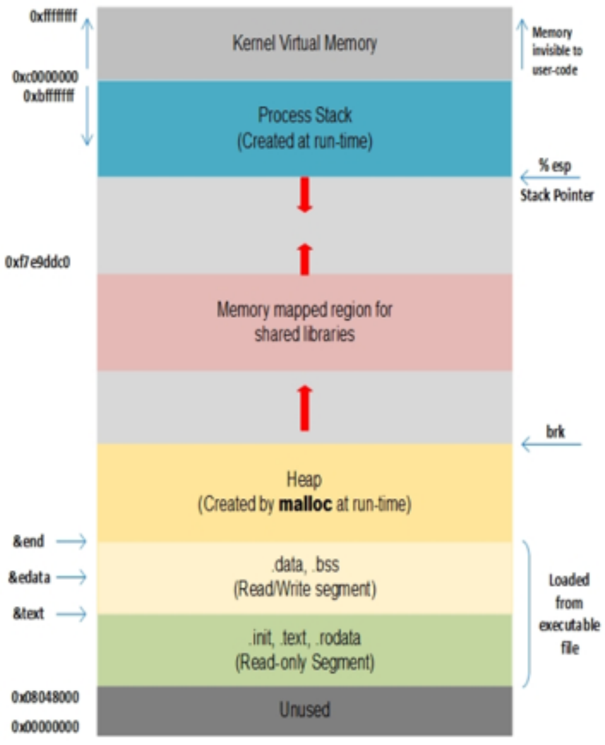
- **SECTION `.data`:** All initialized data like variables and constants are placed in the `.data` section
- **SECTION `.bss`:** All uninitialized data is declared in the `.bss` section (Block Storage Start)
- **SECTION `.text`:** This is actually the code section, and it will always include at least one label named `_start` or `main`, that defines the initial program entry point. The Linux linker `ld(1)`, expect the program entry point label with the name of `_start`, while `gcc(1)` expect the program entry point label with the name of `main`. The `global` directive is used to define a symbol, which is expected to be used by another module using the `extern` directive. The `extern` directive is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module.

There are three types of statements in assembly language programming:

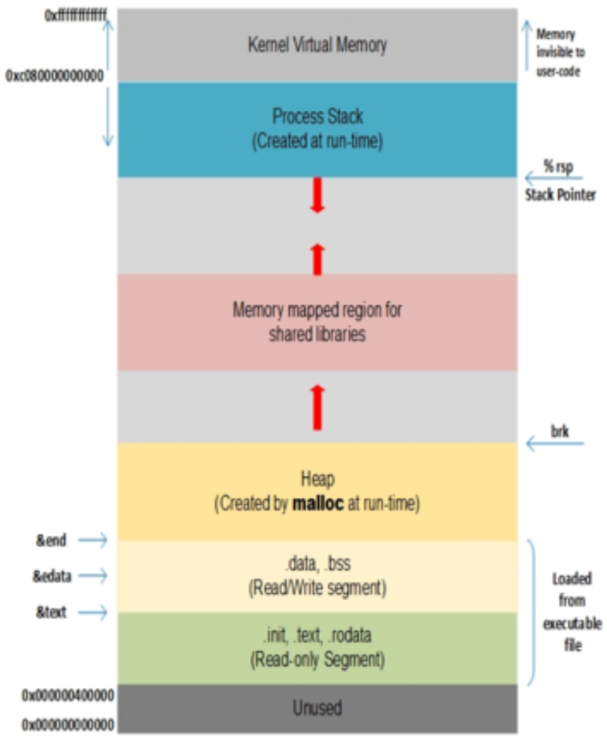
- **x86-64 Assembly Instructions:** These instructions are converted into machine code, and when executed, instruct the processor what to do. Some x86 specific assembly instructions are `mov, add, sub, syscall`
- **Pseudo Instruction:** These are not real x86 machine instructions but are normally used in the real instruction field. Some NASM specific pseudo instructions are `DB, DW, RESB, RESW, EQU`
- **Assembler Directives:** Assembly directives are the statements that direct the assembler to do something. The specialty of these statements is that they are effective only during the assembly of a program and they do not generate any machine executable code. Some NASM specific directives are `SECTION, EXTERN, GLOBAL, BITS`

Process Image Model of x86-64 Process

The layout of various segments of a process running on a Linux system on x86-64 is also shown in the above figure. The x86-64 CPU chips that you can buy today support physical address of 40 bits, so a physical memory of 1 TiB. The processor support 48-bit logical address, which can be broken down as:


0xffffffff
0xc0000000
0xbfffffff
0xf7e9ddc0
&end
&edata
&text
0x08048000
0x00000000
Kernel Virtual Memory
Process Stack (Created at run-time)
Memory mapped region for shared libraries
Heap (Created by **malloc** at run-time)
.data, .bss (Read/Write segment)
.init, .text, .rodata (Read-only Segment)
Unused
Memory invisible to user-code
% esp Stack Pointer
brk
Loaded from executable file

(a) Process Address space (32-bit)

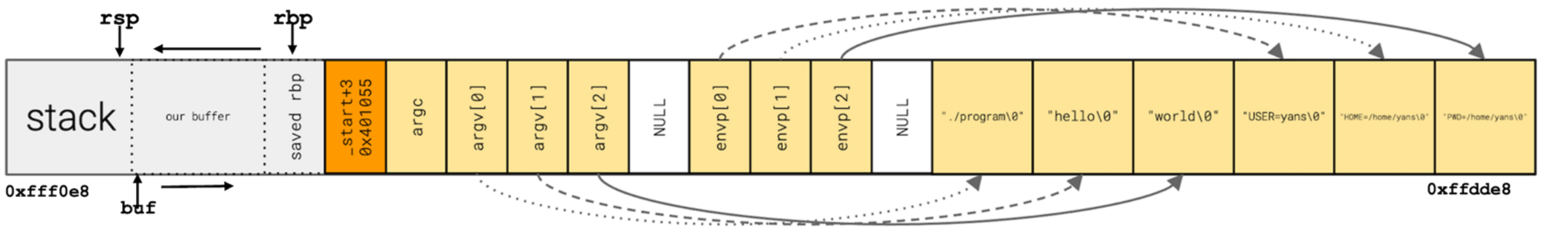

0xffffffffffff
0xc08000000000
&end
&edata
&text
0x000000400000
0x000000000000
Kernel Virtual Memory
Process Stack (Created at run-time)
Memory mapped region for shared libraries
Heap (Created by **malloc** at run-time)
.data, .bss (Read/Write segment)
.init, .text, .rodata (Read-only Segment)
Unused
Memory invisible to user-code
% rsp Stack Pointer
brk
Loaded from executable file

(b) Process Address space (64-bit)

Process Stack

Historically stack grows from higher address to lower addresses (You can think of stack horizontally or vertically). Here is the screenshot of a process stack, which is executed from the shell:

```
$ ./program hello world
      int main(int argc, char *argv[], char* envp[]){
            extern char **environ;
```



Environment variables are globally accessible to any function in your program, either by passing `envp` explicitly from `main()`, or using the global environ pointer. However, command-line arguments are only passed and accessible to the `main()` function.

Example 1: Displaying Hello World using System Calls

First, compile your assembly code to generate an object file. You can use an assembler like **nasm**, and then link it to produce an executable or object file. The NASM (Netwide Assembler) is a popular assembler for the x86 architecture, known for its straightforward syntax and support for various output formats. You can install **nasm** if not already installed on your system via following command:

```
$ sudo apt-get install nasm
```

```
;3.2/syscalls.nasm
SECTION .data
msg  db "Learning is fun with Arif", 0

SECTION .bss
   ;nothing here

SECTION .text
global _start
_start:
; display message on screen
      mov rax, 1   ;ID of write syscall
      mov rdi, 1   ;file descriptor
      mov rsi, msg ;message
      mov rdx, 26  ;size
      syscall      ;write(1,msg,26)
; exit the program gracefully
   mov rax, 60   ;ID of exit syscall
   xor rdi, rdi  ;exit status
   syscall       ;exit(0)
```

The two methods using which a program can request the operating system to perform a service like printing on screen or reading from keyboard are making a system call or making a library call. The `syscalls.nasm` file contains a basic assembly code that displays a message on screen using the `write()` system call. The `.data` section contains initialized data, having just one variable `msg` with a null terminator (0). The `.bss` section contains nothing, while the `.txt` section contains the assembly code. The `global` directive inside the `.text` section is used to define symbols, which are expected to be used by another module.
For making a system call on **`x86-64 running Linux`**, you need to place the system call ID in the `rax` register, first six integer system call arguments inside `rdi, rsi, rdx, r10, r8, r9` registers and remaining (if any) are pushed on the stack, finally you make the `syscall` instruction to transition to kernel mode. The fourth argument in system calling convention is stored in `r10` instead of `rcx` because `rcx` is used to store the return address, when invoking `syscall`. Similarly for making a system call on **`x86-64 running MS Windows`**, you need to place the system call ID in the `rax` register, first four integer arguments are passed via `rcx, rdx, r8,` and `r9` registers and remaining are pushed on the stack. For both Linux and Windows, the return value of a system call is placed inside the `rax` register. Unlike Linux, Windows system calls are not stable and may change across versions. So, we normally call `ntdll.dll`, which internally invokes the correct system call. For both Linux and Window, in case of system calls, floating point values are NOT passed via `xmm` registers, rather converted into integers before passing. In case if you want to pass floating point values, they are placed in memory and their address is passed via registers mentioned above. Every operating system has its own set of system calls and every system call has an associated ID. To check the available system calls and their IDS on Linux, you can view `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` file.
We will use **nasm** to assemble this file to an object file for either 32-bit or 64-bit architecture using following commands:

```
$ nasm -f elf64 syscalls.nasm
$ nasm -f elf32 syscalls.nasm
```

In order to link this object file with the standard C library to make an executable, we can use either **ld** or **gcc**. Since in this assembly program, the starting point is mentioned using the `_start` symbol, so we are using `ld`. If it contains `main` instead of `_start` then `gcc` can also be used for linking purpose instead of `ld`. By default, the linker `ld` will link and create a 64-bit binary as shown below:

```
$ ld syscalls.o -o myexe                 //will create a 64-bit executable
$ ld -m elf_i386 syscalls.o -o myexe     //will create a 32-bit executable
$ ./myexe
Learning is fun with Arif
$ echo $?
```

Example 2: Displaying Hello World using Library Calls

```
;3.2/libcalls.nasm
SECTION .data
msg db "A hello to C library functions", 0

SECTION .bss
   ;nothing here

SECTION .text
global main
extern printf, exit
main:
; display message on screen
      lea rdi, [msg]  ;first arg to printf
      xor rax, rax
      call printf
; exit the program gracefully
   mov rdi, 0   ;return value of exit
   call exit    ;exit(0)
```

An **x86-64** machine **running Linux** make a library call (user space function) using the **System V AMD64 ABI**. The first six integer arguments are placed inside rdi, rsi, rdx, rcx, r8, r9 registers and remaining (if any) are pushed on the stack, finally you make the call instruction to shift the control of execution to the library function. The return value is stored in rax register. For library calls on Linux, the first eight floating-point arguments are passed via xmm0 – xmm7, rest are passed via stack and the return value is stored in xmm0 register. Similarly, an **x86-64** machine running **MS Windows** make a library call (user space function) using the **Microsoft x64 ABI**. The first four integer arguments are passed via rcx, rdx, r8, and r9 registers and remaining are pushed on the stack. For library calls, the first four floating-point arguments are passed via xmm0 – xmm3, rest are passed via stack and the return value is stored in xmm0 register.

We have seen the use of system calls in Example1, now let us repeat the same using C printf and exit library calls. This time the entry point in the .txt section is the symbol main instead of _start because we will be using gcc instead of ld to link. The extern directive is used to declare symbols which are not defined anywhere in the module being assembled, but are assumed to be defined in some other module. Before calling the printf function, we need to place the first argument to printf inside the rdi register. Similarly, before calling the exit function, we need to place its first argument inside the rdi register. The printf is a variadic function, and moreover, it can be passed integer arguments as well as floating point arguments. In a variadic function like printf, floating point arguments are duplicated, i.e., stored in xmm registers and copied into general purpose registers as well. In the sample code, we are clearing rax with the xor rax, rax instruction because, in the **x86-64 System-V** calling convention, the rax register is used to specify the number of floating-point arguments passed to a function in vector registers (xmm0, xmm1, etc.). So, for the printf function here, since there are no floating-point arguments, rax must be set to 0. This ensures that the printf function knows that it doesn't need to fetch any values from the xmm registers.

First, assemble your assembly code to generate an object file using nasm as shown below:

```
$ nasm -f elf64 libcalls.nasm
```

In order to link this object file with the standard C library to make an executable, we will use gcc, and that is why we have mentioned the starting point using the main symbol. Remember, by default gcc generates a Position Independent Executable, which can be loaded at any memory address, which enhances security features like Address Space Layout Randomization (ASLR). The -no-pie flag of gcc explicitly disables the creation of a PIE executable.

```
$ gcc -no-pie libcalls.o -o myexe
```

You may get a warning saying that the stack is set as executable. To remove this use -z noexecstack option of gcc. More on this later… ☺

```
$ gcc -no-pie -z noexecstack libcalls.o -o myexe
```

Let us execute the executable now:

```
$ ./myexe
A hello to C library functions
$ echo $?
0
```

Example 3: Unconditional Jumps

This example program show the usage of unconditional jump. The **jmp _end** instruction shown in bold, unconditionally shifts the control of flow of instruction to the label **_end**. Understand the code, and then assemble, link and execute the program:

```
$ nasm -felf64 uncondjump.nasm

$ gcc -no-pie -z noexecstack uncondjump.o -o myexe

$ ./myexe
Study Cyber Security
```

```
;3.2/uncondjump.nasm
SECTION .data
msg1 db “Study Cyber Security”, 0
msg2 db “Play Cricket”, 0

SECTION .text
global main
extern printf, exit
main:
; display msg1 on screen
      lea rdi, [msg1]
      xor rax, rax
      call printf
      jmp _end
; display msg2 on screen
      lea rdi, [msg2]
      xor rax, rax
      call printf
; exit the program gracefully
   mov rdi, 0
   call exit
```

Example 4: Conditional Jumps

This example program show the usage of conditional jump. The conditional jump instructions are mostly used after a `cmp op1, op2` instruction, which will subtract `op2` from `op1` without storing the result and just update the relevent flags. In the given code, since the result will be positive five, so the relevent flags will be updated. After the compare instruction, the **jge _positive** instruction shown in bold will execute, and since the result is positive five (greater than or equal to zero), therefore, the control of execution will be transferred to the label `_positive`. Let us assemble, link and execute the program:

```
$ nasm -felf64 condjump.nasm

$ gcc -no-pie -z noexecstack condjump.o -o myexe

$ ./myexe
Negative Number!
```

```
;3.2/condjump.nasm
SECTION .data
msg1 db “Negative Number!”, 0
msg2 db “Positive Number!”, 0

SECTION .text
global main
extern printf, exit
main:
   mov ax, -5d
   cmp ax, 0
   jge _positive
; display msg1 on screen
      lea rdi, [msg1]
      xor rax, rax
      call printf
      jmp _end
   _positive:
; display msg2 on screen
      lea rdi, [msg2]
      xor rax, rax
      call printf
; exit the program gracefully
_end:
   mov rdi, 0
   call exit
```

Example 5: Defining and Calling a User Defined Function

In computer programming languages, a procedure, function, sub-routine, or method is a named piece of code (set of instructions) that can be called from a program in order to perform some specific tasks, thus making a program more structural, easier to understand and manageable. An assembly procedure is defined as a set of logically related instructions having a name that:

- is meant to be called from different places
- can accept parameters (via registers, global memory locations, stack)
- do some processing (e.g., add numbers, print string, get input, and so on)
- may return some value to its caller (via register, global memory location)

Following screenshot describes the syntax of defining a user defined function for x86 assembly to be assembled using **`nasm`** or `masm` assembler:

Defining a Procedure in NASM

```
        <procname>:
0xf70     <1st instr>
0xf71     <2nd instr>
0xf72     <3rd instr>
            …


0xf8a       ret
```

Defining a Procedure in MASM

```
        <procname> proc
0xf70     <1st instr>
0xf71     <2nd instr>
0xf72     <3rd instr>
            …

0xf8a      ret
        <procname> endp
```

Here is a hello world assembly program which uses the `call` and the `ret` instruction to transfer and return the control of execution to and from a function.

```
;3.2/funccalling.nasm
SECTION .data
msg db "Cyber Security Course is fun", 0
SECTION .text
global main
extern printf, exit
main:
   call printmsg
   mov rdi, 0
   call exit
printmsg:
   lea rdi, [msg]
   xor rax, rax
   call printf
   ret
```

Let us assemble, link and execute the program:

```
$ nasm -felf64 funccalling.nasm

$ gcc -no-pie funccalling.o -o myexe

$ ./myexe
Cyber Security Course is fun
```

Function Calling Convention and Use of Stack in Function Calls

The **function calling convention** is a set of rules that dictate *how functions receive parameters, return values, manage the stack, and how to share the CPU registers between the caller and the callee*. These rules ensure that functions can correctly interact with each other and with the operating system, enabling compatibility between different pieces of code and across different programming languages. Understanding these conventions is crucial in various areas, including *software development*, *reverse engineering* and *exploitation*. Here's a detailed look at their importance in these contexts:

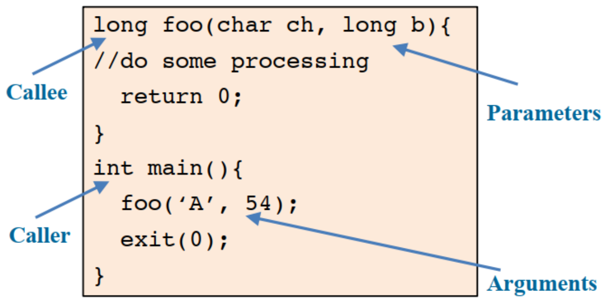
- **Software Development:**
  - Interoperability: Different components or modules of a program, potentially written in different languages or by different teams, must adhere to the same calling conventions to interact correctly.
  - Debugging: Understanding function calling conventions helps in troubleshooting and debugging complex issues.
  - Optimization: Compilers optimize code by understanding calling conventions.

- **Reverse Engineering**
  - Understanding Binary Code: When doing reverse engineering, analysts decompile binary code to reconstruct the source code in order to understand how functions are called and how arguments are passed.
  - Static Analysis: Analyzing the assembly code or disassembled binaries requires knowledge of calling conventions to correctly interpret function calls, parameters, and returns.
  - Dynamic Analysis: During dynamic analysis, tools like debuggers rely on calling conventions to correctly step through code, track function calls, and inspect memory.

- **Exploitation**
  - Exploiting Stack Overflows: Exploiting vulnerabilities such as buffer overflows often involve manipulating the stack to overwrite return addresses or function pointers to gain control over program execution. Knowledge of the calling convention helps in crafting payloads that correctly manipulate the stack.
  - Return-Oriented Programming (ROP): ROP exploits involve chaining together small pieces of code (gadgets) that end in return instructions. Knowing the calling convention helps in crafting ROP chains by understanding how the stack is organized and how gadgets are invoked.

Key Components of a Function Calling Convention

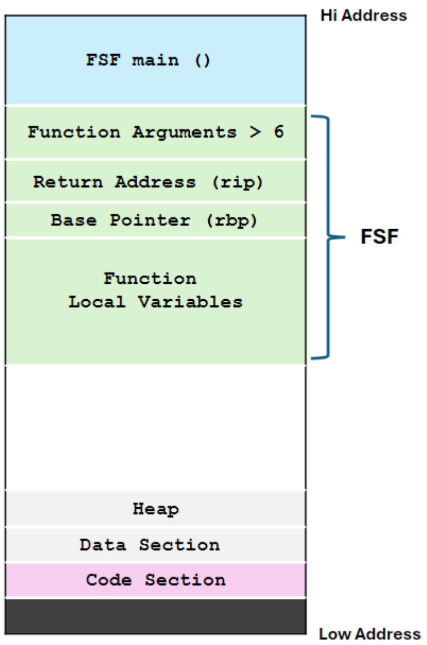
1. **Argument Passing and Returning Values**:
   - A function may have arguments/parameters, which might be integer/floating point values as well as addresses pointing to data. This enables a function to operate on different data with each call.
   - In high level programming languages like C and C++, the values passed by the caller to the callee are called arguments. When the values are received by the called subroutine, they are called parameters. In programming, the terms "caller" and "callee" refer to the relationship between functions or procedures in the context of function calls:
     - ➢ Caller Function: The function that initiates a call to another function to perform a task or compute a result.
     - ➢ Callee Function: This is the function being called by another function. It is the one that gets executed as a result of the call.



```
long foo(char ch, long b){
//do some processing
  return 0;
}
int main(){
  foo('A', 54);
  exit(0);
}
```

   - In the 16-bit and 32-bit days, since there were only eight general purpose registers in x-86 architecture, therefore, all the arguments were passed by the caller to the callee by pushing the arguments on the stack. On x86-64 processor, Linux, Solaris and Mac Operating Systems use a function call protocol called the System-V AMD64 ABI. In which first six integer parameters are passed via registers: **rdi, rsi, rdx, rcx, r8, r9**, and first eight floating point parameters via xmm0 to xmm7 registers (rest on the runtime stack). On the contrary MS Windows Operating System use MS X64 Calling Convention, in which first four integer parameters are passed via registers and first four floating point parameters via xmm0 to xmm3 registers (rest on the runtime stack)
   - Both Linux and MS Windows use **`rax`** register to return integer values and **`xmm0`** register to return floating point values. For larger return values or complex data structures, the return value might be passed via the stack.

2. **Stack Management**:
   - The stack plays a crucial role in function calling conventions, providing a structured way to manage function calls, local variables, and return addresses. The diagram shows the logical process address space of a process where the Code section contains machine code instructions of your program. Above code section we have initialized and uninitialized data sections for global variables. Then we have heap, which is used for dynamic memory allocation, and it grows towards higher addresses. Finally, the stack is at the top of virtual memory below the kernel code and grows from higher memory addresses to lower memory addresses in architectures like x86, MIPS, Motorola, and SPARC.
   - Each function call typically creates a Function Stack Frame (FSF) that holds space for function arguments, `rip`, `rbp`, and local variables.

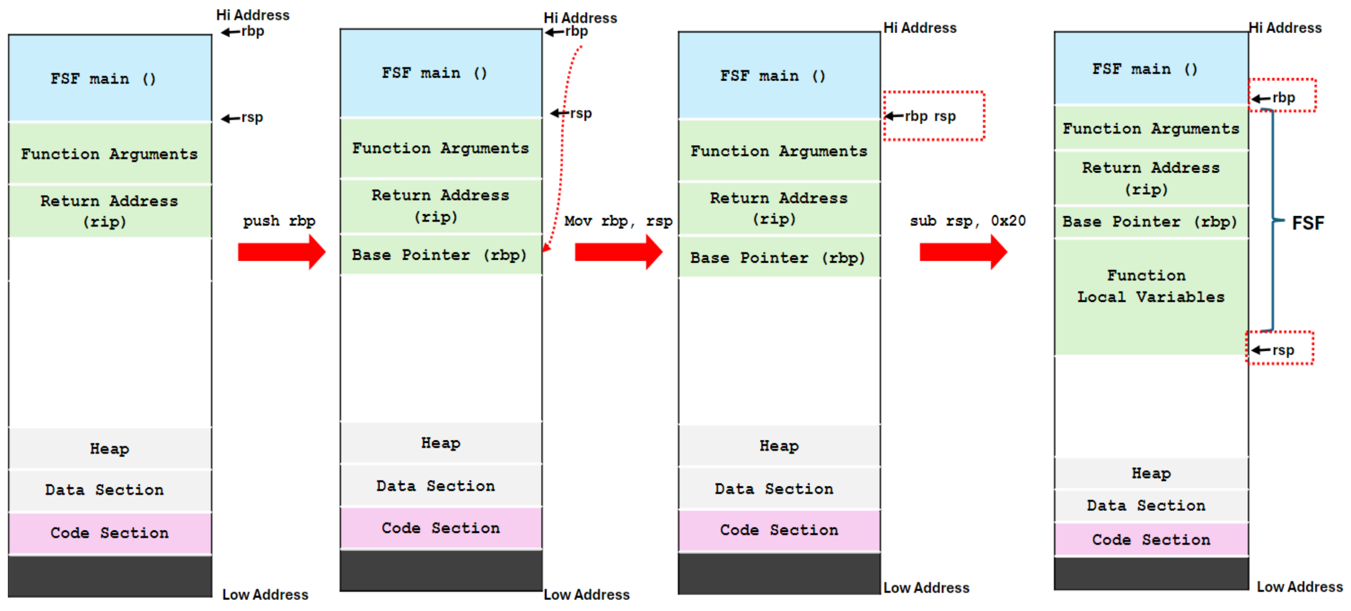


- In x86-64 assembly language, `rsp` and `rbp` are two important registers used for stack management and function calls.
  - **`rsp`** (Stack Pointer) is used to point to the current top of the stack. In x86-64 architecture, the stack grows downward, meaning that pushing data onto the stack decreases the value of `rsp`, and popping data from the stack increases the value of `rsp`.
  - **`rbp`** (Base Pointer) is used to point to the base of the stack frame for the current function, so the address of each argument and local variable can be calculated using this register and an offset. It helps manage local variables and function parameters.

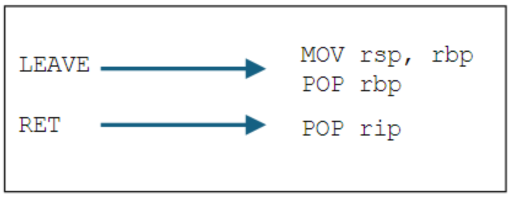
- The stack frame is set up using a piece of code called procedure **prologue** and torn down using a piece of code called procedure **epilogue** and is the responsibility of the Callee in x86 arch.
  - **Procedure Prolog:** On x86-64 running Linux Operating System, the FSF for a function is created by the following sequential steps:
    - The function arguments (>6) are pushed on the stack by the caller.
    - The return address (`rip`) is pushed on the stack.
    - After that, control is shifted to the first instruction of the callee, which performs a procedure prolog having three lines of assembly code shown below:

```
; Function prologue

push rbp               ; Save the old base pointer
mov rbp, rsp           ; Set the new base pointer
sub rsp, 0x20         ; Allocate space for local variables
```

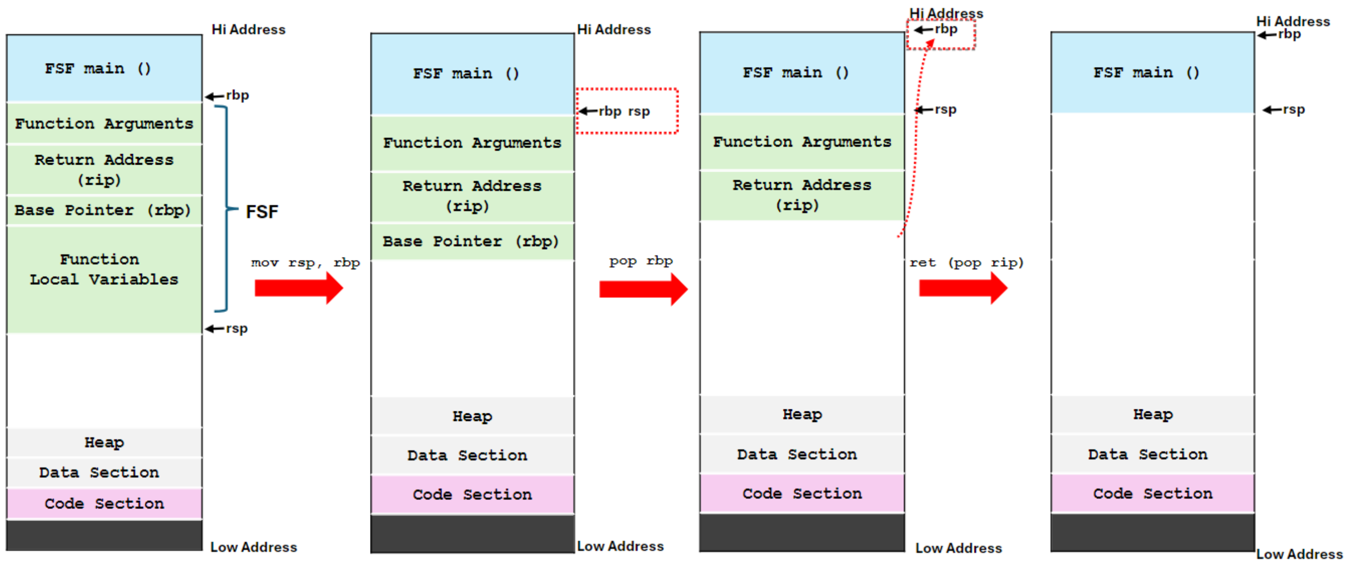
- The creation of FSF by the function prolog is described in the following images:



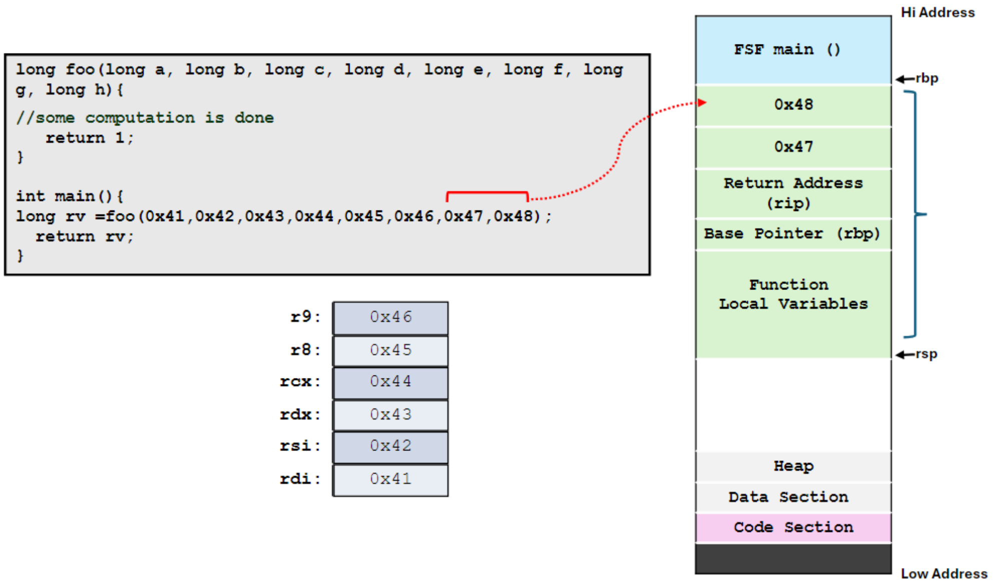
- **Procedure Epilog:** When callee is done with its execution, it first cleans up the FSF and then calls the return statement to transfer control to its caller by performing a procedure epilog:
    - Restore the old stack pointer (`mov rsp, rbp`).
    - Restore the old base pointer (`pop rbp`).
    - Return to the caller using `ret`.



- The removal of FSF by the function epilog is described in the following images:



- An illustration of the FSF of a C program is shown:



```
long foo(long a, long b, long c, long d, long e, long f, long
g, long h){
//some computation is done
  return 1;
}

int main(){
long rv =foo(0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48);
  return rv;
}
```

Overview of Debuggers and Installing GDB

Debugging is the science and art of finding and eliminating bugs in a computer program. A debugger is a program running another program allowing you to see what is going on inside another program while it executes, or what another program was doing at the moment it crashed. There exist different types of debuggers like `GNU gdb (PEDA/GEF), IDA Pro, radare2, cutter, ghidra, OllyDbg, binaryninja, ptrace, strace, ltrace, ftrace, bpftrace` and so on. Using a debugger, a programmer can:

- Start a program, specifying anything that might affect its behavior.
- Make a program stop on specified conditions.
- Examine what has happened, when a program has stopped.
- Change things in a program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- Last but not the least, can be used for run time analysis of binaries, disassembly, reverse engineering and cracking binaries.

We will be using `GDB`, the GNU Project debugger that can debug a program running on the same machine as `GDB` (native), on may be another machine (remote), or may be on a simulator. `GDB` is a portable debugger that can run on the most popular UNIX and Microsoft Windows variants, as well as on Mac OS X. The target processors include `IA-32, x86-64, arm, mips, powerpc, sparc, alpha` and many others. GDB works for many programming languages including Assembly, C/C++, Objective C, OpenCL, Go, Modula-2, Fortran, Pascal and Ada.

GDB Installation on Linux

From Binary:

```
$ sudo apt update
$ sudo apt install gdb
$ gdb -version
GNU gdb (Debian 15.1-1) 15.1
```

From Source:

```
$ sudo apt install build-essential texinfo
$ wget http://ftp.gnu.org/gnu/gdb/gdb-<version>.tar.gz
$ tar -xvzf gdb-<version>.tar.gz
$ cd gdb-<version>
$ ./configure
$ make
$ sudo make install
$ gdb -version
```

Basic Commands of GDB

Commands	**Description**

`$ nasm -g -felf64 prog1.nasm`
 	
`$ gcc -ggdb -c prog1.c`
	In order to load and properly analyze a program in `gdb` you need to compile it with `-g` or `-ggdb` option, to instruct the compiler to keep debugging symbols, source file names and line numbers in the object files

`$ gdb`
`(gdb) file myexe`
OR
`$ gdb myexe`
`(gdb)`
	There are two ways to load a binary inside `gdb` by either running `gdb` command and then specifying the binary name with the file command. Or by specifying the binary name as an argument to `gdb`.

`(gdb) quit`	Exits the current session of `gdb`.

`(gdb) help`
 	
`(gdb) help <classname>`
 	
`(gdb) help <command>`
	The `help` command of `gdb` is used to display the listing of twelve different classes in which `gdb` commands are categorized. You can also specify the classname (breakpoints, running, stack, …) or the command to get help about it.

`(gdb) run [arg1 arg2 …]`
OR
`(gdb) set args arg1 arg2 …`
`(gdb) run`
	Once the program is loaded and `gdb` is running, you can pass command line arguments to the binary using the `run` command of `gdb`. Or can use the `set` command instead and later use the `run` command.

`(gdb) attach <PID>`	If you want to debug a process that is already running, you can attach GDB to it using its process ID (PID).

`(gdb) info sources/functions/variables/locals`
`(gdb) info registers/all`
`(gdb) info sharedlibrary`
`(gdb) info address <function name>`
	Once a program *loaded* inside `gdb`, you can use the `info` command to display the name of all the source files from which symbols have been read in, name of functions, global variables, name of local variables inside a FSF, and the CPU registers.

`(gdb) list [1,12]`
 	
`(gdb) list <filename>:<line#>`
 	
`(gdb) list <filename>:<function name>`
	The `list` command of `gdb` is used to display the source code (provided if the source file is there in the pwd)
	.

`(gdb) disassemble`
`(gdb) disassemble <function name>`
 	
`(gdb) set disassembly-flavor intel`
	Disassembles the current function or code segment. By default, `gdb` disassembles in AT&T format, to change the format to intel, use the `set disassembly-flavor` command.

`(gdb) break <filename>:<line#>`
 	
`(gdb) break <filename>:<function name>`
 	
`(gdb) break <filename>:*0x2xfff0500`
	Breakpoint is the LOC in your program where you want to stop the execution. You can set as many break points as you feel like using the `break` command of `gdb` by mentioning the line#, function name, or by virtual address

`(gdb) info break`
`(gdb) disable <breakpoint#>`
`(gdb) enable <breakpoint#>`
`(gdb) delete <breakpoint#>`
`(gdb) clear <breakpoint#>`
	To get the information about the existing breakpoints already set in your program, you can use the `info` command. Moreover, you can disable/enable/delete/ and clear breakpoints.

(gdb)`watch <variable name>` **(gdb)**`info watch` **(gdb)**`disable <watchpoint#>` **(gdb)**`enable <watchpoint#>` **(gdb)**`delete <watchpoint#>` **(gdb)**`clear <watchpoint#>`	Like breakpoints, we can set watchpoints on variables. Whenever the value of that variable will change, `gdb` will interrupt the program and print out the old and the new value.
(gdb)`continue / c / ci` **(gdb)**`next / n / ni` **(gdb)**`step / s / si` **(gdb)**`finish`	Once a breakpoint is hit, you can do the following: o **`c:`** Continue till the next breakpoint or end of program. o **`n:`** Execute and move to next instruction, but don't dive into functions. o **`s:`** Execute and move to next instruction, by diving into functions. o **`finish:`** Continue until the current function returns.
(gdb)`print /format-char <var-name>`	Once a breakpoint is hit during execution of a program, you can inspect/modify contents of variables, CPU registers as well as different memory addresses. The print command is the most common command to check the contents of variables in the specified format o **`/d`** is for signed decimal o **`/u`** is for unsigned decimal o **`/x`** for printing as hex o **`/o`** for printing as octal o **`/t`** for printing as binary o **`/f`** for floating point number o **`/s`** for C-string o **`/a`** for address Note: Unlike `print` the `display` command is used to display the value of variable, each time the program stops.
(gdb)`set variable <var-name> = <value>`	The `set` command is used to modify the value of a variable.
(gdb)`x/12cb <address>` **(gdb)**`x/12db &var1` **(gdb)**`x/4xb *0x601000` **(gdb)**`x/32b $rsp`	The `examine` command or its alias `x` is passed a memory address to display its contents. It is optionally followed by a forward slash (/) and then a: o Count field, which is a number in decimal. o Format field, which is a single letter with 'd' for decimal, 'x' for hex, 't' for binary and 'c' for ASCII. o Size field, which is single letter with 'b' for byte, 'h' for 16-bit word, and 'w' for 32-bit word.
(gdb)`backtrace`	The `backtrace` command or its alias b displays the call trace of a program.
(gdb)`! clear`	To run the OS shell commands inside `gdb`, you can precede the command with a **`!`** symbol.

A discussion on detailed commands of `gdb` is beyond the scope of this handout. Interested students are advised to go through the Video Lecture of the Assembly course at the following link:
https://www.youtube.com/watch?v=2x-pkzSmsD8&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=31

Example 6: Running/Debugging Assembly Program inside GDB

Let us write down a basic x86-64 assembly program on our Kali Linux machine using some text editor like `vim`, `nano`, or `gedit`, assemble using `nasm` (by keeping the symbols), link using `gcc` (by keeping the symbols and making it non position independent executable), finally load the binary inside `gdb` in quite mode:

```
;3.2/condjump.nasm
SECTION .data
msg1 db “Negative Number!”, 0
msg2 db “Positive Number!”, 0

SECTION .text
global main
extern printf, exit
main:
   mov ax, -5d
   cmp ax, 0
   jge _positive
; display msg1 on screen
      lea rdi, [msg1]
      xor rax, rax
      call printf
      jmp _end
_positive:
; display msg2 on screen
      lea rdi, [msg2]
      xor rax, rax
      call printf
; exit the program gracefully
_end:
   mov rdi, 0
   call exit
```

```
$ nasm -f elf64 -g condjump.nasm
$ gcc -no-pie -g condjump.o -o myexe
$ gdb -q ./myexe
Reading symbols from ./myexe …
(gdb) help
List of classes of commands:
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
support -- Support facilities.
text-user-interface -- TUI is the GDB text interface.
tracepoints -- Tracing of program execution w/o stopping the program.
user-defined -- User-defined commands.
(gdb) set disassembly-flavor intel
(gdb) disassemble main
0x0401140 <+0>:       mov ax, 0xfffb
0x0401144 <+4>:       cmp ax, 0x0
0x0401148 <+8>:       jge 0x40115c <_positive>
0x040114A <+10>:      lea rdi, ds:0x404020
0x0401152 <+18>:      xor rax, rax
0x0401155 <+21>:      call 0x401030 <printf@plt>
0x040115A <+26>:      jmp 0x40116c <_end>
(gdb) break main
(gdb) run
(gdb) info registers | functions | breakpoints   [To delete a breakpoint delete 1]
(gdb) print $<register-name>
(gdb) x <addr>
(gdb) x/8xb <addr>      [count can be a decimal value] [format can be x|t|d|c] [size can be b|h|w|g]
(gdb) si | ni | c
Negative Number! [Inferior 1 (process 208589) exited normally]
(gdb) quit
```

Practice all the commands of `gdb` mentioned in the previous table and play around with this program by executing it step by step and making changes to the code as it executes. ☺

Dis-assemblers & De-compilers (Dynamic Analysis Tools)

GDB with PEDA & GEF

In our previous handout we have used GNU GDB to run/debug sample C as well as assembly programs. GNU GDB is too good a debugger, however, it lacks intuitive interface, do not have a smart context display, do not have commands for exploit development, and has weak scripting support. So, to enhance the fire power of `gdb` for analyzing, exploiting and doing reverse engineering on executables, hackers use:

- a gdb plug-in called **PEDA** (Python Exploit Development Assistance)
- a gdb plug-in called **GEF** (GDB Enhanced Features)

PEDA is a fantastic tool that provides commands to make the exploitation development process smoother. However, it has limitations:

- PEDA code is too fundamentally linked to Intel architectures (x86-32 and x86-64)
- PEDA development has been quite idle for a few years now, and many new interesting features a debugger can provide simply do not exist.

On the other hand, GEF not only supports all the architecture supported by GDB (currently x86, ARM, AARCH64, MIPS, PowerPC, SPARC) but is designed to integrate new architectures as well. Moreover, GEF provides a suite of powerful commands to assist with binary exploitation tasks. Whether you're dealing with buffer overflows, format string vulnerabilities, ROP chains, or heap exploitation, these commands allow for better memory inspection, breakpoint management, and code analysis.

Installation of PEDA: https://github.com/longld/peda

PEDA is available only on Linux and supported by `gdb 7.x` and `Python 2.6` onwards. In order to install PEDA plugin for `gdb`, you simply have to download or clone its repository and then update the `.gdbinit` file in your home directory as shown below:

```
$ git  clone https://github.com/longld/peda.git    ~/peda
$ echo  "source ~/peda/peda.py"   >>    ~/.gdbinit
```

Installation of GEF: https://github.com/hugsy/gef.git

On the same grounds, if you want to install GEF plugin for `gdb`, you simply have to download it and then update the `.gdbinit` file in your home directory as shown below:

```
$ git clone https://github.com/hugsy/gef.git  ~/gef
$ echo "source ~/gef/gef.py" >> ~/.gdbinit
```

Running/Debugging C Program inside GDB with GEF Plugin

Let us now debug the following C program. In the source file, the `main()` function creates two long variables `main_var1` and `main_var2` and character pointer `*main_str2` and calls a function `f1()` and passing 8 parameters to that function. The function `f1()` receives 8 parameters and further creates two local variables and then calls another function `f2()` and passes one parameter to it. The `f2()` function receives a single a parameter, performs some operations and returns a value to `f1()` that further returns 1 to parent function which is `main()` and finally `main()` returns 0 to its parent which is the shell program.

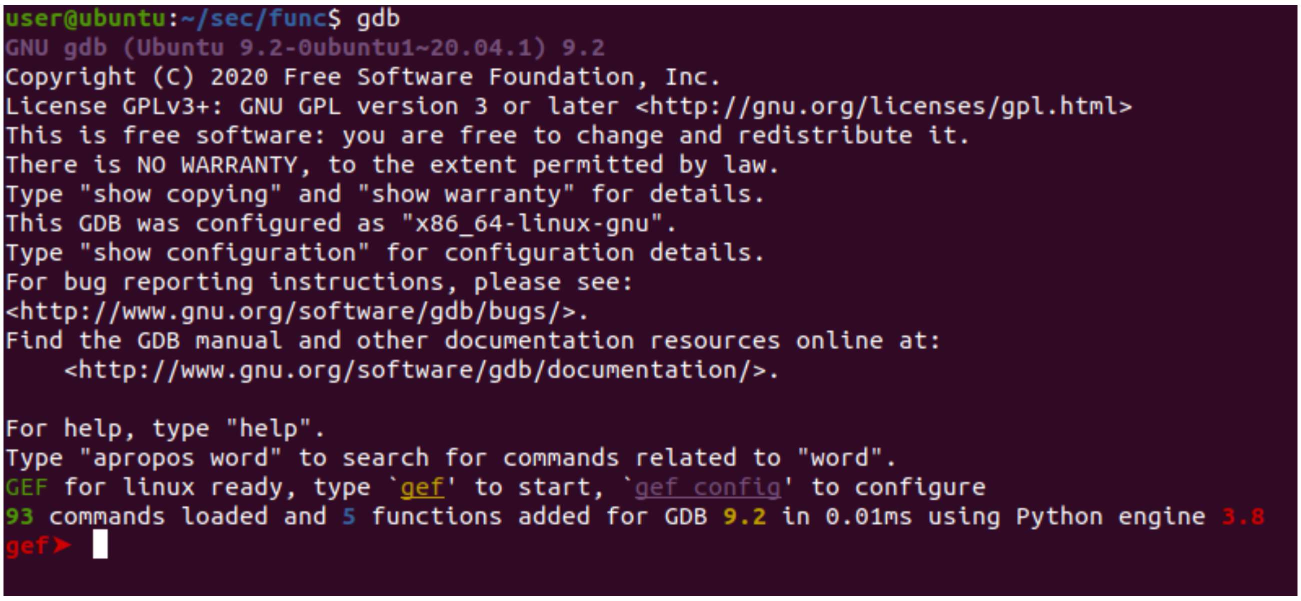
```c
//3.2/cprogs/gef/debugme.c
#include <stdio.h>
#include <stdlib.h>
   int f2(int a){
   int b = a +1;
   return b;
}

int f1(long a, long b, long c, long d, long e, long f, long g, long h){
   unsigned long f1_var1 = 0x123456789;
   unsigned long f1_var2 = 0x0abcdef;
   int rv = f2(5);
   return 1;
}

int main(int argc, char *argv[]){
   unsigned long main_var1 = 0x1122334455667788;
   unsigned long main_var2 = 0x99aabbccddeeff00;
   char *main_str2  = "Arif";
   int rv_f1 = f1(0x11111111, 0x22222222, 0x33333333, 0x44444444, 0x55555555,
0x66666666, 0x77777777, 0x88888888);
   return 0;
}
```

Compile the `debugme.c` program using `gcc` (for 64-bit and 32-bit), load it inside GDB with GEF to practically understand all the concepts discussed in this handout specially the function calling convention, stack growing and shrinking etc. Happy Learning ☺

GEF Interface

After successful installation of the `gef` plug-in, when you run `gdb`, you get the following prompt:

$ gdb



```
user@ubuntu:~/sec/func$ gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 9.2 in 0.01ms using Python engine 3.8
gef➤ 
```

Note the prompt is not **(gdb)**, rather is **gef>**, that means `gdb` with enhanced features. Inside `gef`, you can give the `gef` command, which will display brief description of different `gef` commands:

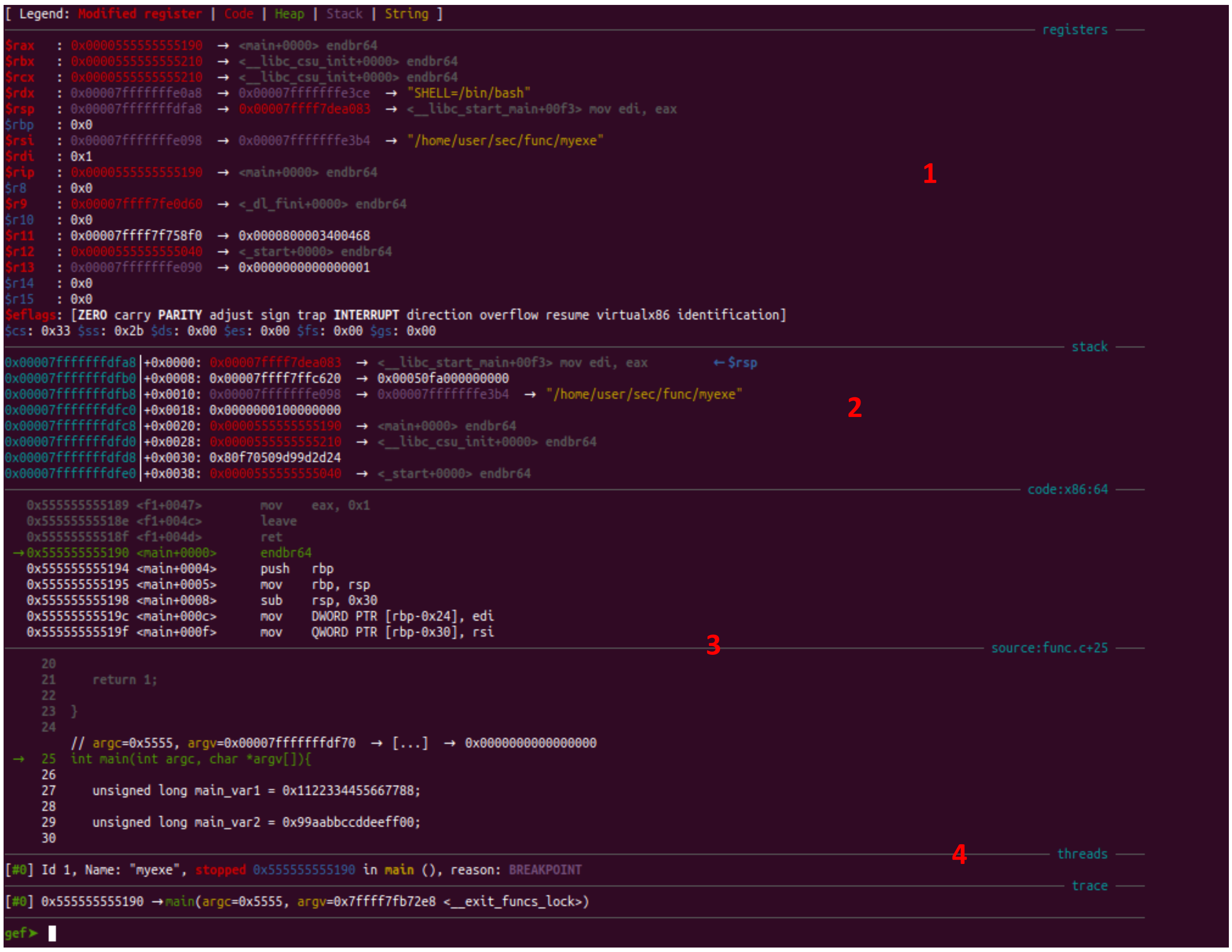
```
gef> gef
```

Let us load the binary named `debugme` (`3.2/cprogs/debugme.c`) from the current working directory, set a breakpoint at `main`, and run the program:

```
gef> file debugme
Reading symbols from debugme …
gef> break main
Breakpoint 1 at 0x1197: file debugme.c, line 17.
gef> run
```

When you run a binary inside `gef`, you get six panels, showing different information about the running process:

- Registers:
- Stack:
- Code:
- Source:
- Threads:
- Trace



```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0000555555555190  →  <main+0000> endbr64
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rcx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rdx   : 0x00007fffffffe0a8  →  0x00007fffffffe3ce  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffdfa8  →  0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax
$rbp   : 0x0
$rsi   : 0x00007fffffffe098  →  0x00007fffffffe3b4  →  "/home/user/sec/func/myexe"
$rdi   : 0x1
$rip   : 0x0000555555555190  →  <main+0000> endbr64
$r8    : 0x0
$r9    : 0x00007ffff7fe0d60  →  <_dl_fini+0000> endbr64
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  →  0x0000800003400468
$r12   : 0x0000555555555040  →  <_start+0000> endbr64
$r13   : 0x00007fffffffe090  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdfa8│+0x0000: 0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax       ←$rsp
0x00007fffffffdfb0│+0x0008: 0x00007ffff7ffc620  →  0x00050fa000000000
0x00007fffffffdfb8│+0x0010: 0x00007fffffffe098  →  0x00007fffffffe3b4  →  "/home/user/sec/func/myexe"
0x00007fffffffdfc0│+0x0018: 0x0000000100000000
0x00007fffffffdfc8│+0x0020: 0x0000555555555190  →  <main+0000> endbr64
0x00007fffffffdfd0│+0x0028: 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
0x00007fffffffdfd8│+0x0030: 0x80f70509d99d2d24
0x00007fffffffdfe0│+0x0038: 0x0000555555555040  →  <_start+0000> endbr64
────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555555189 <f1+0047>        mov    eax, 0x1
   0x55555555518e <f1+004c>        leave
   0x55555555518f <f1+004d>        ret
 →0x555555555190 <main+0000>       endbr64
   0x555555555194 <main+0004>      push   rbp
   0x555555555195 <main+0005>      mov    rbp, rsp
   0x555555555198 <main+0008>      sub    rsp, 0x30
   0x55555555519c <main+000c>      mov    DWORD PTR [rbp-0x24], edi
   0x55555555519f <main+000f>      mov    QWORD PTR [rbp-0x30], rsi
──────────────────────────────────────────────────────────────── source:func.c+25 ────
     20
     21        return 1;
     22
     23    }
     24
           // argc=0x5555, argv=0x00007fffffffdf70  → [...]  → 0x0000000000000000
 →   25    int main(int argc, char *argv[]){
     26
     27        unsigned long main_var1 = 0x1122334455667788;
     28
     29        unsigned long main_var2 = 0x99aabbccddeeff00;
     30
─────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "myexe", stopped 0x555555555190 in main (), reason: BREAKPOINT
───────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555555190 →main(argc=0x5555, argv=0x7ffff7fb72e8 <__exit_funcs_lock>)
────────────────────────────────────────────────────────────────────────────────────────
gef➤ 
```

1. **Registers Panel:** The Registers Panel in GEF displays the current values of the CPU registers/flags, providing an organized and easily readable view. It helps in analyzing the state of the CPU, tracking changes in register values, and debugging at a lower level. It does not show the floating-point registers, however, you can view the contents of all registers, use the `info all` command of `gdb`.
2. **Stack Panel:** The Stack Panel displays top of the call stack, which includes a list of function calls that are currently active. This is really beneficial to understand the current Function Stack Frame of a function. Remember, the top of the stack is displayed at the top of this panel, where the `rsp` register is pointing.
3. **Code Panel:** The Code Panel displays the assembly code along with the virtual addresses. The line currently being executed or where the breakpoint is set is typically highlighted or marked to provide a clear point of focus.
4. **Source Panel:** This panel displays the corresponding high level language code, with the current LOC highlighted. This way you can corelate the high-level code with its corresponding assembly.
5. **Threads & Trace Panels:** This provides information about the threads in a multithreaded program, including their states and stack traces.

Note: To configure the panels to be displayed, you can use the following command inside `gef`:

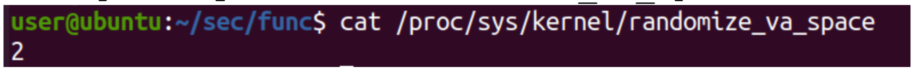
```
gef> gef config context.layout "regs stack code source"
```

Loading and Running a Program inside GDB with GEF

- **Disable ASLR:** Before performing any step let's just check if ASLR (Address Space layout randomization) is enabled on our machine, and if yes then we need to disable it. ASLR is a security feature of the operating system that randomizes the memory addresses used by system and application processes, making it harder for attackers to predict memory locations. On Linux systems, the ASLR setting can have following three values, which can be changed as well:
  - **0**: No randomization. Everything is static.
  - **1**: Conservative randomization. Shared libraries, stack, mmap(), heap, and VDSO are randomized.
  - **2**: Full randomization.

To check the current state of ASLR, you can view the contents of `randomize_va_space` file:

```
$ cat /proc/sys/kernel/randomize_va_space
```



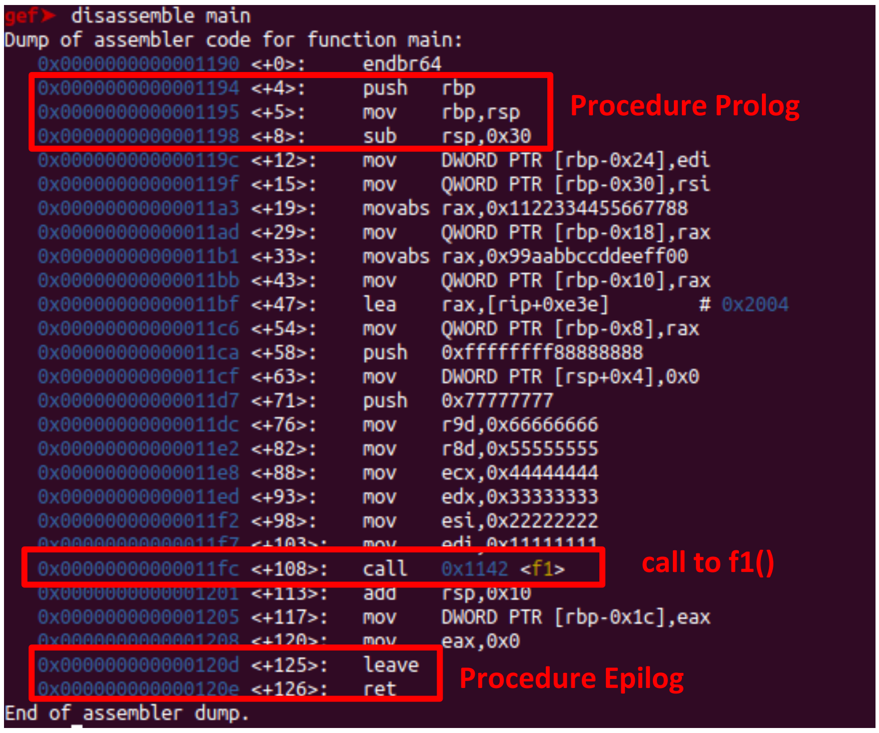
```
user@ubuntu:~/sec/func$ cat /proc/sys/kernel/randomize_va_space
2
```

To change the current state of ASLR, you can use any of the following commands:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
$ sudo sysctl -w kernel.randomize_va_space=0
```

- **Disassemble `main()` in GEF:** To show the disassembly of the `main` function in GDB with GEF (GDB Enhanced Features) without running the program, you can use GDB's disassembly commands directly after loading the binary.

```
Gef➤ disassemble main
```



```
gef➤ disassemble main
Dump of assembler code for function main:
   0x0000000000001190 <+0>:     endbr64
   0x0000000000001194 <+4>:     push   rbp
   0x0000000000001195 <+5>:     mov    rbp,rsp
   0x0000000000001198 <+8>:     sub    rsp,0x30
   0x000000000000119c <+12>:    mov    DWORD PTR [rbp-0x24],edi
   0x000000000000119f <+15>:    mov    QWORD PTR [rbp-0x30],rsi
   0x00000000000011a3 <+19>:    movabs rax,0x1122334455667788
   0x00000000000011ad <+29>:    mov    QWORD PTR [rbp-0x18],rax
   0x00000000000011b1 <+33>:    movabs rax,0x99aabbccddeeff00
   0x00000000000011bb <+43>:    mov    QWORD PTR [rbp-0x10],rax
   0x00000000000011bf <+47>:    lea    rax,[rip+0xe3e]        # 0x2004
   0x00000000000011c6 <+54>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000000011ca <+58>:    push   0xffffffff88888888
   0x00000000000011cf <+63>:    mov    DWORD PTR [rsp+0x4],0x0
   0x00000000000011d7 <+71>:    push   0x77777777
   0x00000000000011dc <+76>:    mov    r9d,0x66666666
   0x00000000000011e2 <+82>:    mov    r8d,0x55555555
   0x00000000000011e8 <+88>:    mov    ecx,0x44444444
   0x00000000000011ed <+93>:    mov    edx,0x33333333
   0x00000000000011f2 <+98>:    mov    esi,0x22222222
   0x00000000000011f7 <+103>:   mov    edi,0x11111111
   0x00000000000011fc <+108>:   call   0x1142 <f1>
   0x0000000000001201 <+113>:   add    rsp,0x10
   0x0000000000001205 <+117>:   mov    DWORD PTR [rbp-0x1c],eax
   0x0000000000001208 <+120>:   mov    eax,0x0
   0x000000000000120d <+125>:   leave
   0x000000000000120e <+126>:   ret
End of assembler dump.
```

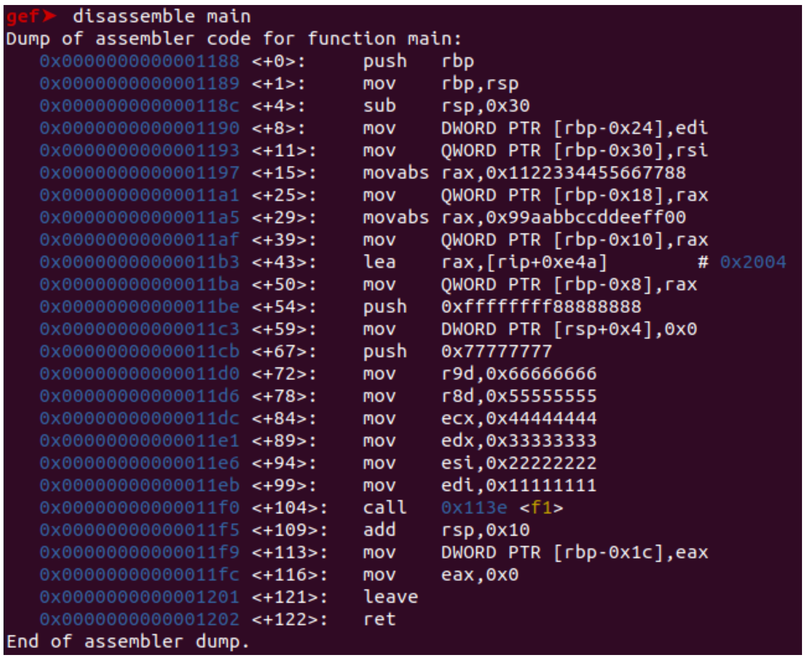
Similarly, you can check the disassembly of `f1()` and `f2()` functions as well.

- **Disable CET:** You may have noticed the `endbr64` instruction before procedure prolog in the above screenshot. The `endbr64` instruction is part of the Intel Control-flow Enforcement Technology (CET), specifically the Indirect Branch Tracking (IBT) feature, which is designed to enhance security by protecting against certain types of control-flow attacks such as Return Oriented Programming (ROP) and Jump Oriented Programming (JOP). It helps ensure that indirect branches (such as calls and jumps) are redirected to valid locations. This instruction is used to mark valid targets for indirect branches, ensuring that the control flow cannot be hijacked by malicious code. Excluding or removing the `endbr64` instruction from binaries generally involves manipulating the binary code, which can be done for various purposes such as reverse engineering, debugging, or modifying software behavior. You can experiment with turning it off to disable CET. Thus, compile your source file again with `-fcp-protection=none` and generate executable. After that load it in GDB.

```
$ gcc -ggdb -fcf-protection=none debugme.c -o debugme
$ gdb debugme
```

If you view the disassembly again, you can note that CET has been excluded or disabled.

```
gef➤ disassemble main
```

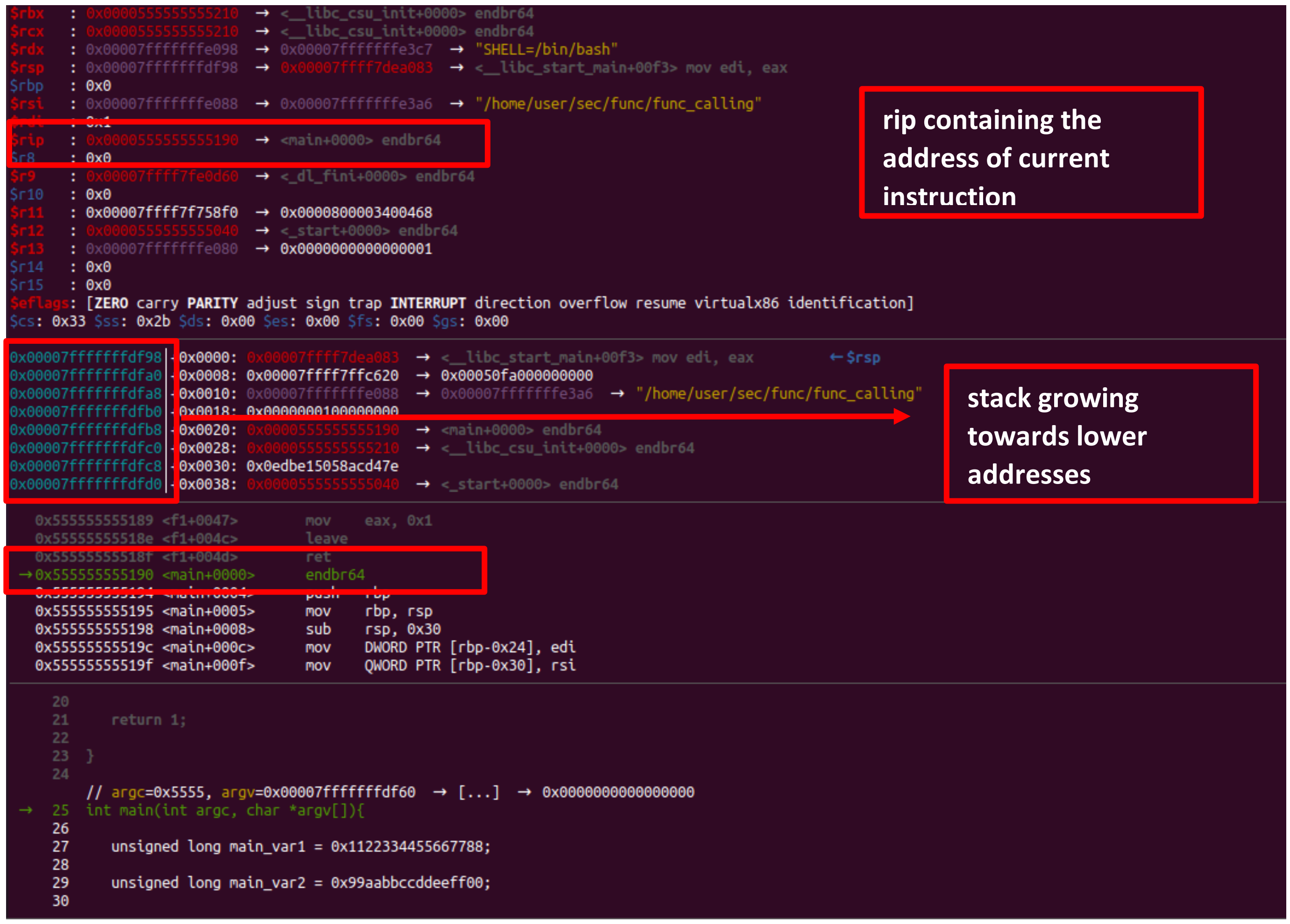


```
gef➤ disassemble main
Dump of assembler code for function main:
   0x0000000000001188 <+0>:     push   rbp
   0x0000000000001189 <+1>:     mov    rbp,rsp
   0x000000000000118c <+4>:     sub    rsp,0x30
   0x0000000000001190 <+8>:     mov    DWORD PTR [rbp-0x24],edi
   0x0000000000001193 <+11>:    mov    QWORD PTR [rbp-0x30],rsi
   0x0000000000001197 <+15>:    movabs rax,0x1122334455667788
   0x00000000000011a1 <+25>:    mov    QWORD PTR [rbp-0x18],rax
   0x00000000000011a5 <+29>:    movabs rax,0x99aabbccddeeff00
   0x00000000000011af <+39>:    mov    QWORD PTR [rbp-0x10],rax
   0x00000000000011b3 <+43>:    lea    rax,[rip+0xe4a]        # 0x2004
   0x00000000000011ba <+50>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000000011be <+54>:    push   0xffffffff88888888
   0x00000000000011c3 <+59>:    mov    DWORD PTR [rsp+0x4],0x0
   0x00000000000011cb <+67>:    push   0x77777777
   0x00000000000011d0 <+72>:    mov    r9d,0x66666666
   0x00000000000011d6 <+78>:    mov    r8d,0x55555555
   0x00000000000011dc <+84>:    mov    ecx,0x44444444
   0x00000000000011e1 <+89>:    mov    edx,0x33333333
   0x00000000000011e6 <+94>:    mov    esi,0x22222222
   0x00000000000011eb <+99>:    mov    edi,0x11111111
   0x00000000000011f0 <+104>:   call   0x113e <f1>
   0x00000000000011f5 <+109>:   add    rsp,0x10
   0x00000000000011f9 <+113>:   mov    DWORD PTR [rbp-0x1c],eax
   0x00000000000011fc <+116>:   mov    eax,0x0
   0x0000000000001201 <+121>:   leave
   0x0000000000001202 <+122>:   ret
End of assembler dump.
```

- **Run the Program:** Now you can apply break point and run the program:

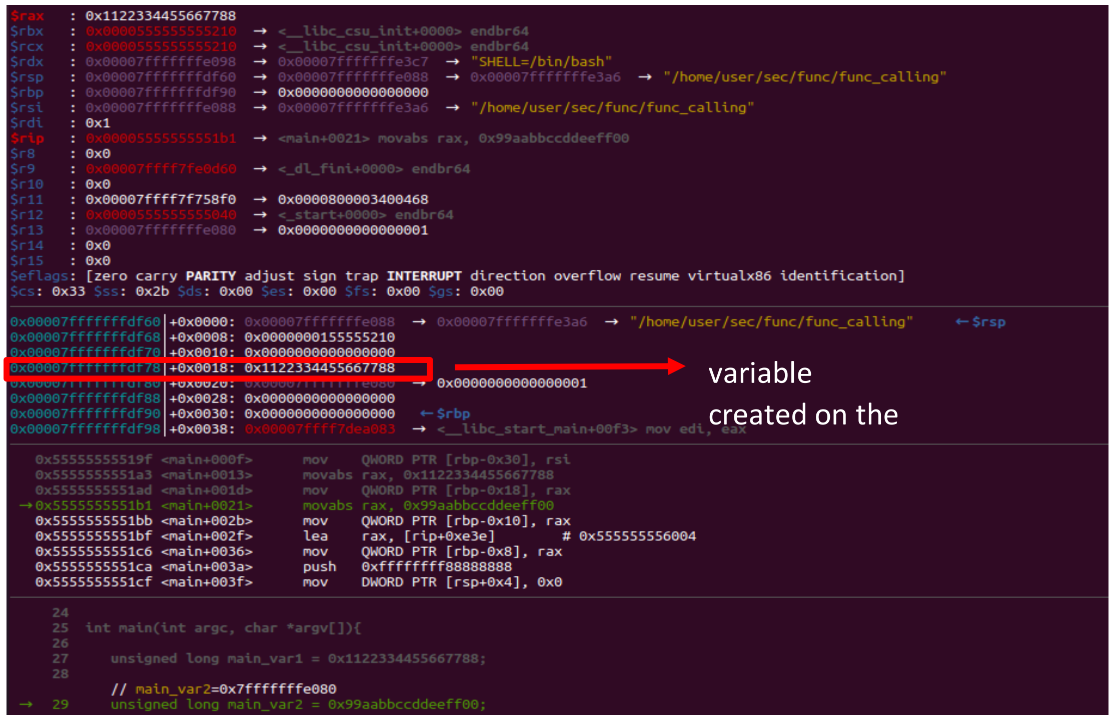
```
gef➤ break main
gef➤ run
```

- Since we are running the program in GDB with GEF, it shows the output in different sections including registers, stack, code section, threads etc, as we have discussed earlier. Here you need to understand multiple things as shown in the screenshot:


rip containing the address of current instruction
stack growing towards lower addresses

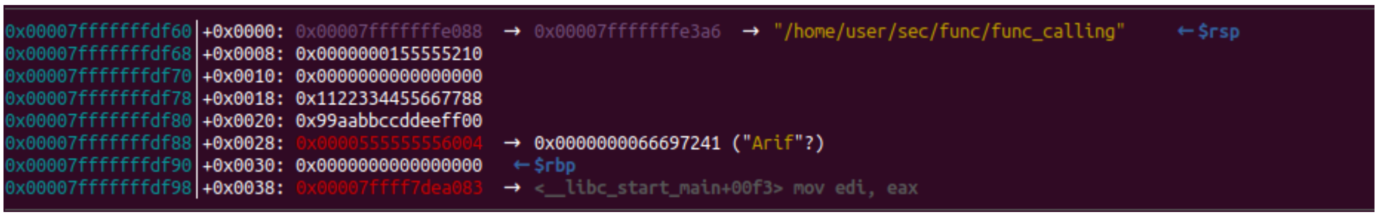
```
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rcx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rdx   : 0x00007fffffffe098  →  0x00007fffffffe3c7  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffdf98  →  0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax
$rbp   : 0x0
$rsi   : 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"
$rdi   : 0x1
$rip   : 0x0000555555555190  →  <main+0000> endbr64
$r8    : 0x0
$r9    : 0x00007ffff7fe0d60  →  <_dl_fini+0000> endbr64
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  →  0x0000800003400468
$r12   : 0x0000555555555040  →  <_start+0000> endbr64
$r13   : 0x00007fffffffe080  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x00007fffffffdf98│+0x0000: 0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax      ←$rsp
0x00007fffffffdfa0│+0x0008: 0x00007ffff7ffc620  →  0x00050fa000000000
0x00007fffffffdfa8│+0x0010: 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"
0x00007fffffffdfb0│+0x0018: 0x0000000100000000
0x00007fffffffdfb8│+0x0020: 0x0000555555555190  →  <main+0000> endbr64
0x00007fffffffdfc0│+0x0028: 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
0x00007fffffffdfc8│+0x0030: 0x0edbe15058acd47e
0x00007fffffffdfd0│+0x0038: 0x0000555555555040  →  <_start+0000> endbr64

   0x555555555189 <f1+0047>        mov    eax, 0x1
   0x55555555518e <f1+004c>        leave
   0x55555555518f <f1+004d>        ret
 →0x555555555190 <main+0000>       endbr64
   0x555555555194 <main+0004>      push   rbp
   0x555555555195 <main+0005>      mov    rbp, rsp
   0x555555555198 <main+0008>      sub    rsp, 0x30
   0x55555555519c <main+000c>      mov    DWORD PTR [rbp-0x24], edi
   0x55555555519f <main+000f>      mov    QWORD PTR [rbp-0x30], rsi


        return 1;

    }

    // argc=0x5555, argv=0x00007fffffffdf60  →  [...]  →  0x0000000000000000
 →  int main(int argc, char *argv[]){

        unsigned long main_var1 = 0x1122334455667788;

        unsigned long main_var2 = 0x99aabbccddeeff00;

```

- If you give the `step` command, you can see that `main_var1` has been created on the function stack frame of the main function:



```
$rax   : 0x1122334455667788
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rcx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rdx   : 0x00007fffffffe098  →  0x00007fffffffe3c7  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffdf60  →  0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"
$rbp   : 0x00007fffffffdf90  →  0x0000000000000000
$rsi   : 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"
$rdi   : 0x1
$rip   : 0x00005555555551b1  →  <main+0021> movabs rax, 0x99aabbccddeeff00
$r8    : 0x0
$r9    : 0x00007ffff7fe0d60  →  <_dl_fini+0000> endbr64
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  →  0x0000800003400468
$r12   : 0x0000555555555040  →  <_start+0000> endbr64
$r13   : 0x00007fffffffe080  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x00007fffffffdf60│+0x0000: 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"      ←$rsp
0x00007fffffffdf68│+0x0008: 0x0000000155555210
0x00007fffffffdf70│+0x0010: 0x0000000000000000
0x00007fffffffdf78│+0x0018: 0x1122334455667788
0x00007fffffffdf80│+0x0020: 0x00007fffffffe080  →  0x0000000000000001
0x00007fffffffdf88│+0x0028: 0x0000000000000000
0x00007fffffffdf90│+0x0030: 0x0000000000000000   ←$rbp
0x00007fffffffdf98│+0x0038: 0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax

   0x55555555519f <main+000f>      mov    QWORD PTR [rbp-0x30], rsi
   0x5555555551a3 <main+0013>      movabs rax, 0x1122334455667788
   0x5555555551ad <main+001d>      mov    QWORD PTR [rbp-0x18], rax
 →0x5555555551b1 <main+0021>       movabs rax, 0x99aabbccddeeff00
   0x5555555551bb <main+002b>      mov    QWORD PTR [rbp-0x10], rax
   0x5555555551bf <main+002f>      lea    rax, [rip+0xe3e]        # 0x555555556004
   0x5555555551c6 <main+0036>      mov    QWORD PTR [rbp-0x8], rax
   0x5555555551ca <main+003a>      push   0xffffffff88888888
   0x5555555551cf <main+003f>      mov    DWORD PTR [rsp+0x4], 0x0

    int main(int argc, char *argv[]){

        unsigned long main_var1 = 0x1122334455667788;

        // main_var2=0x7fffffffe080
 →      unsigned long main_var2 = 0x99aabbccddeeff00;
```

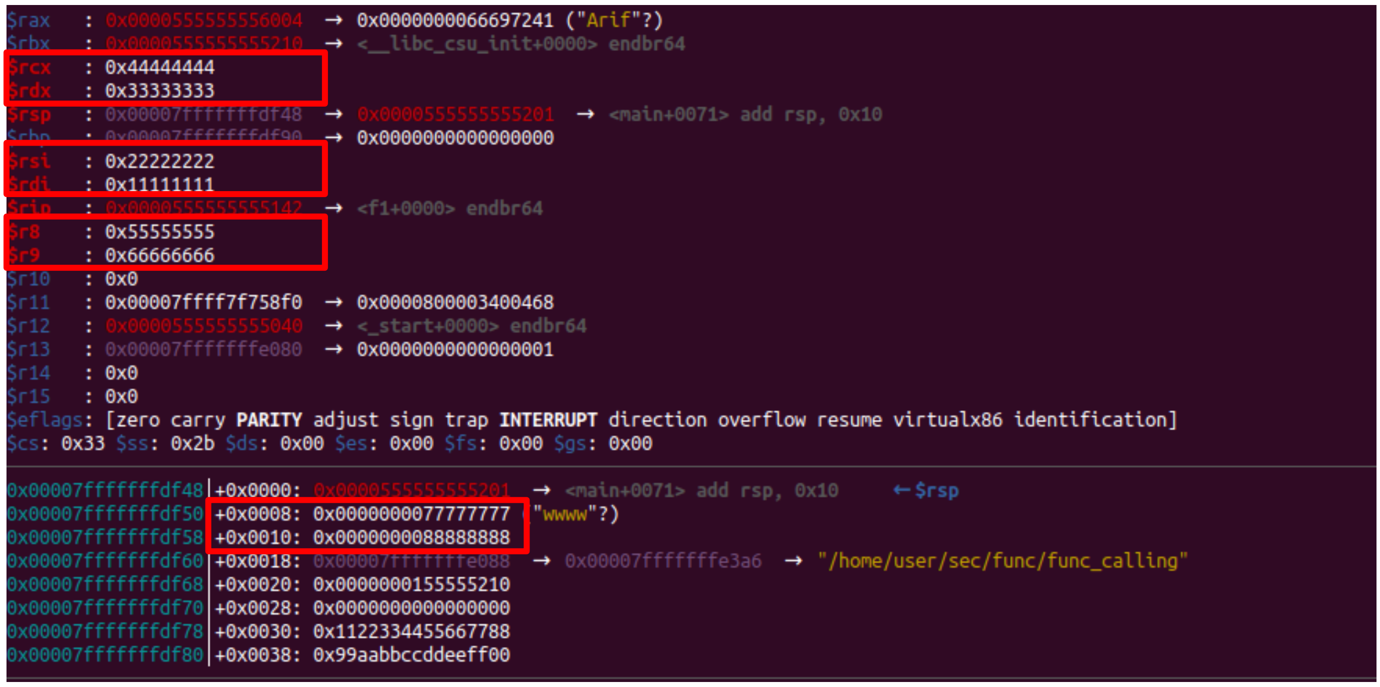
- Similarly, after giving the `step` command multiple times, you can see `main_var1, main_var2 and *main_str2` have been created.



```
0x00007fffffffdf60│+0x0000: 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"     ←$rsp
0x00007fffffffdf68│+0x0008: 0x0000000155555210
0x00007fffffffdf70│+0x0010: 0x0000000000000000
0x00007fffffffdf78│+0x0018: 0x1122334455667788
0x00007fffffffdf80│+0x0020: 0x99aabbccddeeff00
0x00007fffffffdf88│+0x0028: 0x0000555555556004  →  0x0000000066697241 ("Arif"?)
0x00007fffffffdf90│+0x0030: 0x0000000000000000  ←$rbp
0x00007fffffffdf98│+0x0038: 0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax
```

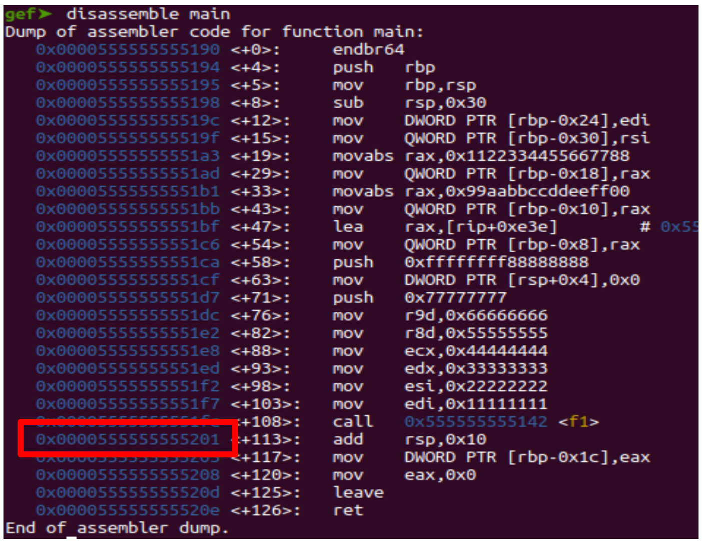
- **Function Call:** The next instruction is the function call. Before the control transfers, the 8th and the 7th arguments to the functions are pushed on the stack (from right to left). Then the remaining six arguments will be placed inside the registers (rdi, rsi, rdx, rcx, r8, r9). This is shown in the screenshot below:

`int rv_f1 = f1(0x11111111, 0x22222222, 0x33333333, 0x44444444, 0x55555555, 0x66666666, `**`0x77777777`**`, `**`0x88888888`**`);`



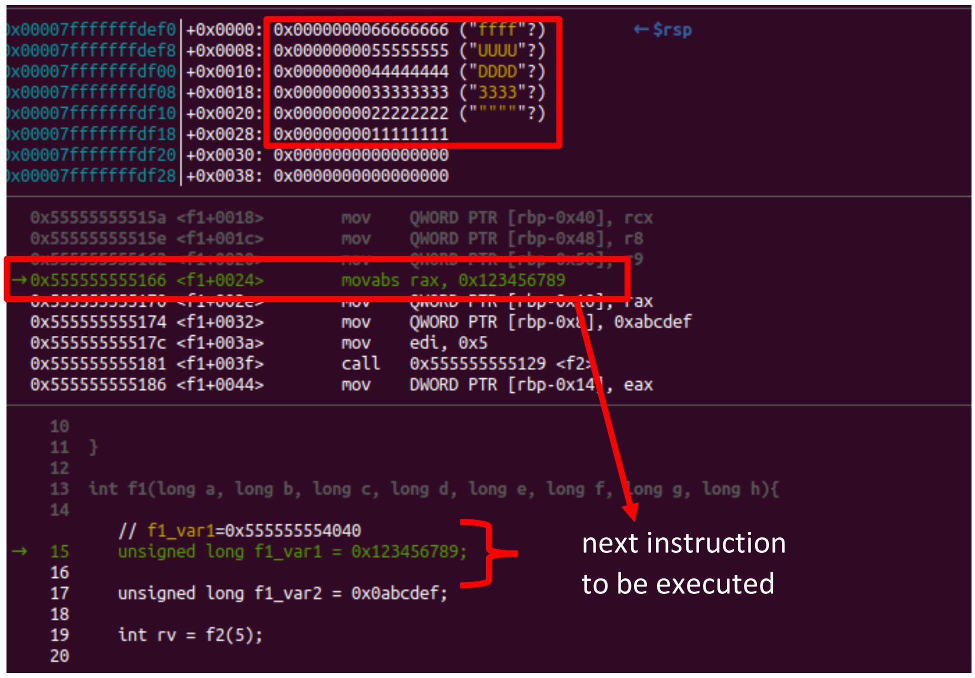
```
$rax   : 0x0000555555556004  →  0x0000000066697241 ("Arif"?)
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rcx   : 0x44444444
$rdx   : 0x33333333
$rsp   : 0x00007fffffffdf48  →  0x0000555555555201  →  <main+0071> add rsp, 0x10
$rbp   : 0x00007fffffffdf90  →  0x0000000000000000
$rsi   : 0x22222222
$rdi   : 0x11111111
$rip   : 0x0000555555555142  →  <f1+0000> endbr64
$r8    : 0x55555555
$r9    : 0x66666666
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  →  0x0000800003400468
$r12   : 0x0000555555555040  →  <_start+0000> endbr64
$r13   : 0x00007fffffffe080  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

0x00007fffffffdf48│+0x0000: 0x0000555555555201  →  <main+0071> add rsp, 0x10     ←$rsp
0x00007fffffffdf50│+0x0008: 0x0000000077777777 ("wwww"?)
0x00007fffffffdf58│+0x0010: 0x0000000088888888
0x00007fffffffdf60│+0x0018: 0x00007fffffffe088  →  0x00007fffffffe3a6  →  "/home/user/sec/func/func_calling"
0x00007fffffffdf68│+0x0020: 0x0000000155555210
0x00007fffffffdf70│+0x0028: 0x0000000000000000
0x00007fffffffdf78│+0x0030: 0x1122334455667788
0x00007fffffffdf80│+0x0038: 0x99aabbccddeeff00
```

You can also observe that before the control is actually transferred to the function `f1()`, the address of the next instruction (`0x55201`) after the `call` instruction is pushed at the top of the stack.



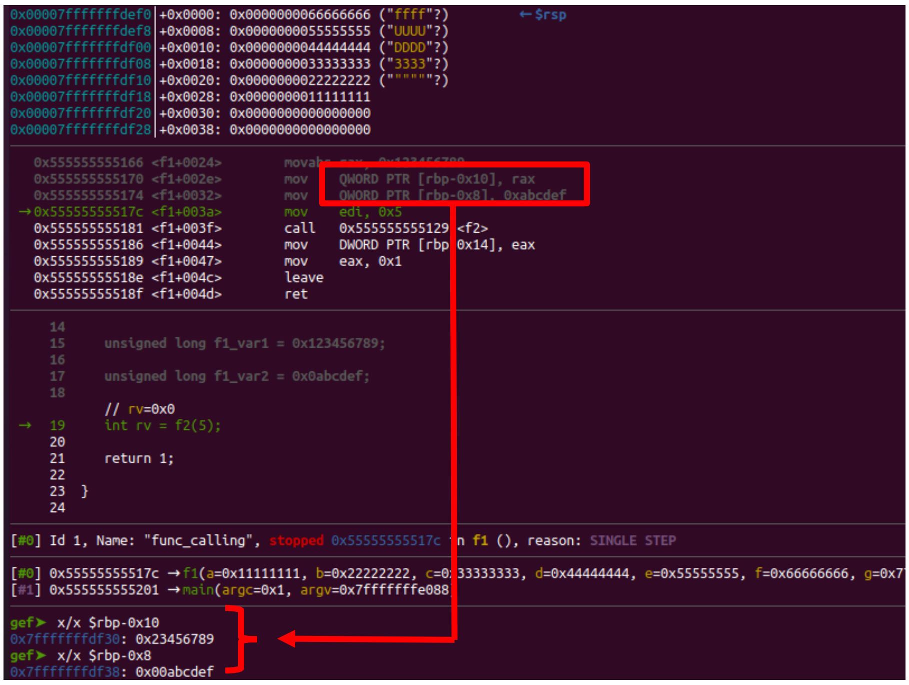
```
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000555555555190 <+0>:     endbr64
   0x0000555555555194 <+4>:     push   rbp
   0x0000555555555195 <+5>:     mov    rbp,rsp
   0x0000555555555198 <+8>:     sub    rsp,0x30
   0x000055555555519c <+12>:    mov    DWORD PTR [rbp-0x24],edi
   0x000055555555519f <+15>:    mov    QWORD PTR [rbp-0x30],rsi
   0x00005555555551a3 <+19>:    movabs rax,0x1122334455667788
   0x00005555555551ad <+29>:    mov    QWORD PTR [rbp-0x18],rax
   0x00005555555551b1 <+33>:    movabs rax,0x99aabbccddeeff00
   0x00005555555551bb <+43>:    mov    QWORD PTR [rbp-0x10],rax
   0x00005555555551bf <+47>:    lea    rax,[rip+0xe3e]        # 0x55
   0x00005555555551c6 <+54>:    mov    QWORD PTR [rbp-0x8],rax
   0x00005555555551ca <+58>:    push   0xffffffff88888888
   0x00005555555551cf <+63>:    mov    DWORD PTR [rsp+0x4],0x0
   0x00005555555551d7 <+71>:    push   0x77777777
   0x00005555555551dc <+76>:    mov    r9d,0x66666666
   0x00005555555551e2 <+82>:    mov    r8d,0x55555555
   0x00005555555551e8 <+88>:    mov    ecx,0x44444444
   0x00005555555551ed <+93>:    mov    edx,0x33333333
   0x00005555555551f2 <+98>:    mov    esi,0x22222222
   0x00005555555551f7 <+103>:   mov    edi,0x11111111
   0x00005555555551fc <+108>:   call   0x555555555142 <f1>
   0x0000555555555201 <+113>:   add    rsp,0x10
   0x0000555555555205 <+117>:   mov    DWORD PTR [rbp-0x1c],eax
   0x0000555555555208 <+120>:   mov    eax,0x0
   0x000055555555520d <+125>:   leave
   0x000055555555520e <+126>:   ret
End of assembler dump.
```

- Once the control goes inside the function `f1()`, as two arguments are already on the stack, the remaining six arguments which are there in the registers are also moved on the stack (space for local arguments).



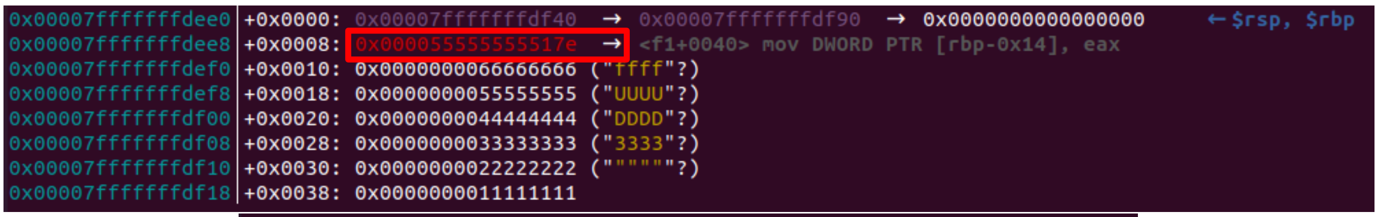
```
x00007fffffffdef0│+0x0000: 0x0000000066666666 ("ffff"?)       ←$rsp
x00007fffffffdef8│+0x0008: 0x0000000055555555 ("UUUU"?)
x00007fffffffdf00│+0x0010: 0x0000000044444444 ("DDDD"?)
x00007fffffffdf08│+0x0018: 0x0000000033333333 ("3333"?)
x00007fffffffdf10│+0x0020: 0x0000000022222222 (""""?)
x00007fffffffdf18│+0x0028: 0x0000000011111111
x00007fffffffdf20│+0x0030: 0x0000000000000000
x00007fffffffdf28│+0x0038: 0x0000000000000000

   0x55555555515a <f1+0018>       mov    QWORD PTR [rbp-0x40], rcx
   0x55555555515e <f1+001c>       mov    QWORD PTR [rbp-0x48], r8
   0x555555555162 <f1+0020>       mov    QWORD PTR [rbp-0x50], r9
 →0x555555555166 <f1+0024>        movabs rax, 0x123456789
   0x555555555170 <f1+002e>       mov    QWORD PTR [rbp-0x10], rax
   0x555555555174 <f1+0032>       mov    QWORD PTR [rbp-0x8], 0xabcdef
   0x55555555517c <f1+003a>       mov    edi, 0x5
   0x555555555181 <f1+003f>       call   0x555555555129 <f2>
   0x555555555186 <f1+0044>       mov    DWORD PTR [rbp-0x14], eax

    }

    int f1(long a, long b, long c, long d, long e, long f, long g, long h){

       // f1_var1=0x555555554040
 →     unsigned long f1_var1 = 0x123456789;

       unsigned long f1_var2 = 0x0abcdef;

       int rv = f2(5);

```

- However, after those two local variables have been created, we can't see them on stack. They have been created on the stack, but some other location that isn't visible in our stack panel. So, just to verify that they have been created let's copy address from the assembly instruction:

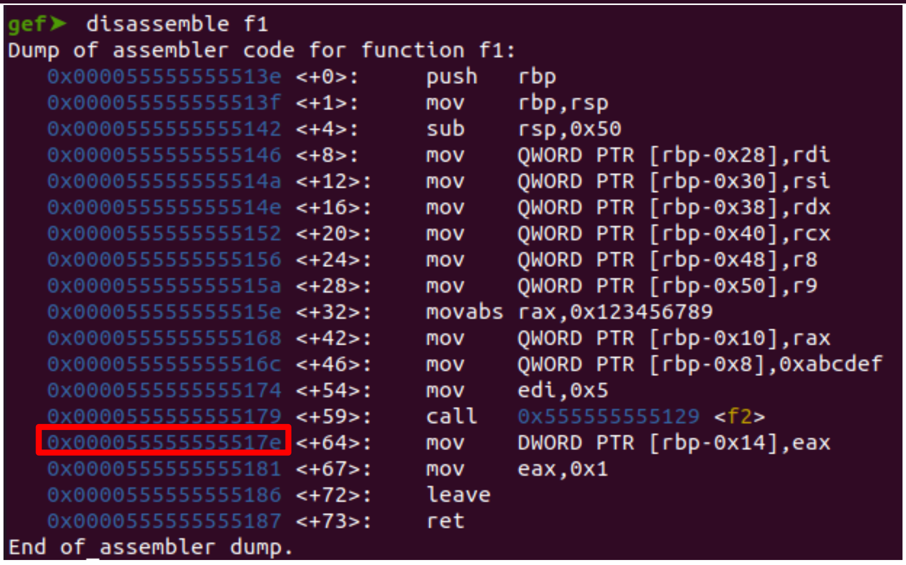


```
0x00007fffffffdef0│+0x0000: 0x0000000066666666 ("ffff"?)       ←$rsp
0x00007fffffffdef8│+0x0008: 0x0000000055555555 ("UUUU"?)
0x00007fffffffdf00│+0x0010: 0x0000000044444444 ("DDDD"?)
0x00007fffffffdf08│+0x0018: 0x0000000033333333 ("3333"?)
0x00007fffffffdf10│+0x0020: 0x0000000022222222 (""""?)
0x00007fffffffdf18│+0x0028: 0x0000000011111111
0x00007fffffffdf20│+0x0030: 0x0000000000000000
0x00007fffffffdf28│+0x0038: 0x0000000000000000

   0x555555555166 <f1+0024>       movabs rax, 0x123456789
   0x555555555170 <f1+002e>       mov    QWORD PTR [rbp-0x10], rax
   0x555555555174 <f1+0032>       mov    QWORD PTR [rbp-0x8], 0xabcdef
 →0x55555555517c <f1+003a>        mov    edi, 0x5
   0x555555555181 <f1+003f>       call   0x555555555129 <f2>
   0x555555555186 <f1+0044>       mov    DWORD PTR [rbp-0x14], eax
   0x555555555189 <f1+0047>       mov    eax, 0x1
   0x55555555518e <f1+004c>       leave
   0x55555555518f <f1+004d>       ret


       unsigned long f1_var1 = 0x123456789;

       unsigned long f1_var2 = 0x0abcdef;

       // rv=0x0
 →     int rv = f2(5);

       return 1;

    }

[#0] Id 1, Name: "func_calling", stopped 0x55555555517c in f1 (), reason: SINGLE STEP

[#0] 0x55555555517c →f1(a=0x11111111, b=0x22222222, c=0x33333333, d=0x44444444, e=0x55555555, f=0x66666666, g=0x7
[#1] 0x555555555201 →main(argc=0x1, argv=0x7fffffffe088)

gef➤  x/x $rbp-0x10
0x7fffffffdf30: 0x23456789
gef➤  x/x $rbp-0x8
0x7fffffffdf38: 0x00abcdef
```

- After stepping in multiple times, let's get into function `f2()`, where you can see that the return address of the very next instruction of `f1()` has been pushed on the stack.

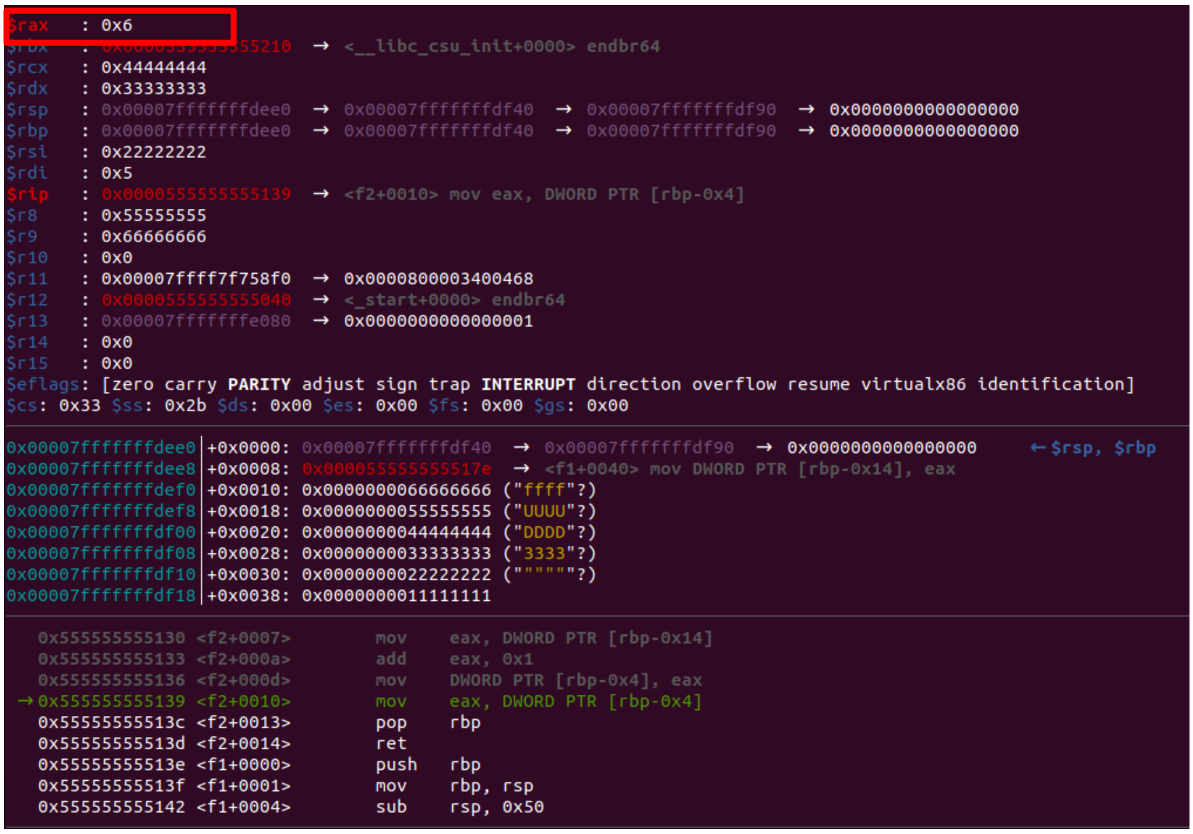


```
0x00007fffffffdee0|+0x0000: 0x00007fffffffdf40  → 0x00007fffffffdf90  → 0x0000000000000000     ←$rsp, $rbp
0x00007fffffffdee8|+0x0008: 0x000055555555517e  → <f1+0040> mov DWORD PTR [rbp-0x14], eax
0x00007fffffffdef0|+0x0010: 0x0000000066666666 ("ffff"?)
0x00007fffffffdef8|+0x0018: 0x0000000055555555 ("UUUU"?)
0x00007fffffffdf00|+0x0020: 0x0000000044444444 ("DDDD"?)
0x00007fffffffdf08|+0x0028: 0x0000000033333333 ("3333"?)
0x00007fffffffdf10|+0x0030: 0x0000000022222222 (""""""?)
0x00007fffffffdf18|+0x0038: 0x0000000011111111
```



```
gef➤  disassemble f1
Dump of assembler code for function f1:
   0x000055555555513e <+0>:     push   rbp
   0x000055555555513f <+1>:     mov    rbp,rsp
   0x0000555555555142 <+4>:     sub    rsp,0x50
   0x0000555555555146 <+8>:     mov    QWORD PTR [rbp-0x28],rdi
   0x000055555555514a <+12>:    mov    QWORD PTR [rbp-0x30],rsi
   0x000055555555514e <+16>:    mov    QWORD PTR [rbp-0x38],rdx
   0x0000555555555152 <+20>:    mov    QWORD PTR [rbp-0x40],rcx
   0x0000555555555156 <+24>:    mov    QWORD PTR [rbp-0x48],r8
   0x000055555555515a <+28>:    mov    QWORD PTR [rbp-0x50],r9
   0x000055555555515e <+32>:    movabs rax,0x123456789
   0x0000555555555168 <+42>:    mov    QWORD PTR [rbp-0x10],rax
   0x000055555555516c <+46>:    mov    QWORD PTR [rbp-0x8],0xabcdef
   0x0000555555555174 <+54>:    mov    edi,0x5
   0x0000555555555179 <+59>:    call   0x555555555129 <f2>
   0x000055555555517e <+64>:    mov    DWORD PTR [rbp-0x14],eax
   0x0000555555555181 <+67>:    mov    eax,0x1
   0x0000555555555186 <+72>:    leave
   0x0000555555555187 <+73>:    ret
End of assembler dump.
```

- As we already know, `f2()` performs add operation and returns 6 in rax register, we can also verify this by checking content of `rax` register:



```
$rax   : 0x6
$rbx   : 0x0000555555555210  → <__libc_csu_init+0000> endbr64
$rcx   : 0x44444444
$rdx   : 0x33333333
$rsp   : 0x00007fffffffdee0  → 0x00007fffffffdf40  → 0x00007fffffffdf90  → 0x0000000000000000
$rbp   : 0x00007fffffffdee0  → 0x00007fffffffdf40  → 0x00007fffffffdf90  → 0x0000000000000000
$rsi   : 0x22222222
$rdi   : 0x5
$rip   : 0x0000555555555139  → <f2+0010> mov eax, DWORD PTR [rbp-0x4]
$r8    : 0x55555555
$r9    : 0x66666666
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  → 0x0000800003400468
$r12   : 0x0000555555555040  → <_start+0000> endbr64
$r13   : 0x00007fffffffe080  → 0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
────────────────────────────────────────────────────────────────────────────────────────────────────
0x00007fffffffdee0|+0x0000: 0x00007fffffffdf40  → 0x00007fffffffdf90  → 0x0000000000000000     ←$rsp, $rbp
0x00007fffffffdee8|+0x0008: 0x000055555555517e  → <f1+0040> mov DWORD PTR [rbp-0x14], eax
0x00007fffffffdef0|+0x0010: 0x0000000066666666 ("ffff"?)
0x00007fffffffdef8|+0x0018: 0x0000000055555555 ("UUUU"?)
0x00007fffffffdf00|+0x0020: 0x0000000044444444 ("DDDD"?)
0x00007fffffffdf08|+0x0028: 0x0000000033333333 ("3333"?)
0x00007fffffffdf10|+0x0030: 0x0000000022222222 (""""""?)
0x00007fffffffdf18|+0x0038: 0x0000000011111111
────────────────────────────────────────────────────────────────────────────────────────────────────
   0x555555555130 <f2+0007>        mov    eax, DWORD PTR [rbp-0x14]
   0x555555555133 <f2+000a>        add    eax, 0x1
   0x555555555136 <f2+000d>        mov    DWORD PTR [rbp-0x4], eax
 → 0x555555555139 <f2+0010>        mov    eax, DWORD PTR [rbp-0x4]
   0x55555555513c <f2+0013>        pop    rbp
   0x55555555513d <f2+0014>        ret
   0x55555555513e <f1+0000>        push   rbp
   0x55555555513f <f1+0001>        mov    rbp, rsp
   0x555555555142 <f1+0004>        sub    rsp, 0x50
```

- In the same fashion, you can run this program to completion to practically understand what all concepts we have discussed in this handout ☺**Sample Program Adjusted According to 32-bit Architecture:**

Loading and running a 32 Bit Binary inside GDB with GEF

```c
//3.2/cprogs/debugme_x32.c
#include <stdio.h>
#include <stdlib.h>
int f2(int a) {
    int b = a + 1;
    return b;
}
int f1(int a, int b, int c, int d, int e, int f, int g, int h) {
    unsigned int f1_var1 = 0x12345678; // Adjusted to fit within 32 bits
    unsigned int f1_var2 = 0x0abcdef0; // Adjusted to fit within 32 bits
    int rv = f2(5);
    return 1;
}
int main(int argc, char *argv[]) {
    unsigned int main_var1 = 0x11223344; // Adjusted to fit within 32 bits
    unsigned int main_var2 = 0x99aabbcc; // Adjusted to fit within 32 bits
    char *main_str2 = "Arif";
    int rv_f1 = f1(0x11111111, 0x22222222, 0x33333333, 0x44444444,
0x55555555, 0x66666666, 0x77777777, 0x88888888);
    return 0;
}
```

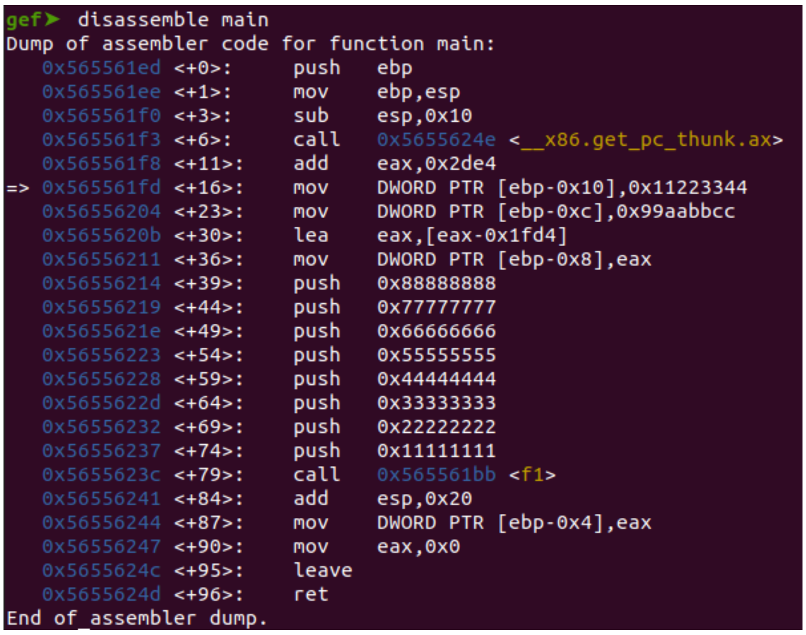
- **Pre-requisites for Creating 32-bit Binary:** We need to use some previous version of GCC such as gcc-7 to compile 32-bit binary on 64-bit architecture, other `ld` linker throws error due to some unknown reasons.

```
$ sudo apt-get update
$ sudo apt-get install gcc-7 g++-7 gcc-multilib g++-7-multilib
```

- **Compiling and Loading Program in GDB with GEF**: Following are the commands to create a 32-bit binary and then loading it inside GDB in quite mode:

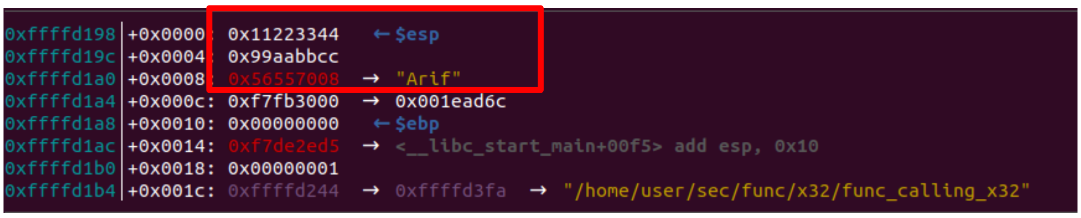
```
$ gcc-7 -ggdb -m32 debugme_x32.c -o debugme_x32
$ gdb -q ./debugme_x32
```

- **View Disassembly:** From the disassembly of the main function, you can observe that all the eight arguments to the `f1()` function are pushed on the stack from right to left instead of passing six via registers and remaining tow via stack.



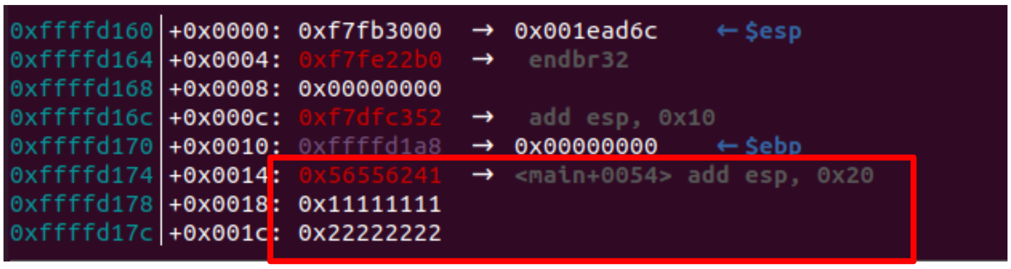
```
gef➤  disassemble main
Dump of assembler code for function main:
   0x565561ed <+0>:     push   ebp
   0x565561ee <+1>:     mov    ebp,esp
   0x565561f0 <+3>:     sub    esp,0x10
   0x565561f3 <+6>:     call   0x5655624e <__x86.get_pc_thunk.ax>
   0x565561f8 <+11>:    add    eax,0x2de4
=> 0x565561fd <+16>:    mov    DWORD PTR [ebp-0x10],0x11223344
   0x56556204 <+23>:    mov    DWORD PTR [ebp-0xc],0x99aabbcc
   0x5655620b <+30>:    lea    eax,[eax-0x1fd4]
   0x56556211 <+36>:    mov    DWORD PTR [ebp-0x8],eax
   0x56556214 <+39>:    push   0x88888888
   0x56556219 <+44>:    push   0x77777777
   0x5655621e <+49>:    push   0x66666666
   0x56556223 <+54>:    push   0x55555555
   0x56556228 <+59>:    push   0x44444444
   0x5655622d <+64>:    push   0x33333333
   0x56556232 <+69>:    push   0x22222222
   0x56556237 <+74>:    push   0x11111111
   0x5655623c <+79>:    call   0x565561bb <f1>
   0x56556241 <+84>:    add    esp,0x20
   0x56556244 <+87>:    mov    DWORD PTR [ebp-0x4],eax
   0x56556247 <+90>:    mov    eax,0x0
   0x5655624c <+95>:    leave
   0x5655624d <+96>:    ret
End of assembler dump.
```

- After running the program in GDB with gef and after multiple step in, here are the variables created on stack:



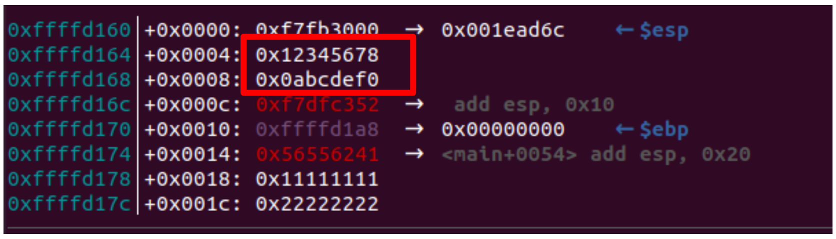
```
0xffffd198│+0x0000: 0x11223344    ← $esp
0xffffd19c│+0x0004: 0x99aabbcc
0xffffd1a0│+0x0008: 0x56557008  →  "Arif"
0xffffd1a4│+0x000c: 0xf7fb3000  →  0x001ead6c
0xffffd1a8│+0x0010: 0x00000000    ← $ebp
0xffffd1ac│+0x0014: 0xf7de2ed5  →  <__libc_start_main+00f5> add esp, 0x10
0xffffd1b0│+0x0018: 0x00000001
0xffffd1b4│+0x001c: 0xffffd244  →  0xffffd3fa  →  "/home/user/sec/func/x32/func_calling_x32"
```

- According to C calling convention function parameters are also passed on the stack instead of registers and return address is also pushed on stack. Some of them are shown below:



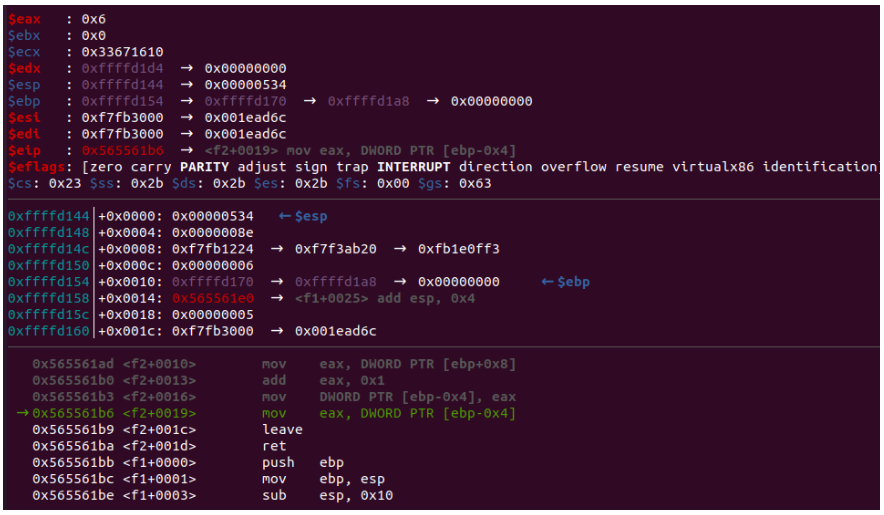
```
0xffffd160│+0x0000: 0xf7fb3000  →  0x001ead6c    ← $esp
0xffffd164│+0x0004: 0xf7fe22b0  →  endbr32
0xffffd168│+0x0008: 0x00000000
0xffffd16c│+0x000c: 0xf7dfc352  →  add esp, 0x10
0xffffd170│+0x0010: 0xffffd1a8  →  0x00000000    ← $ebp
0xffffd174│+0x0014: 0x56556241  →  <main+0054> add esp, 0x20
0xffffd178│+0x0018: 0x11111111
0xffffd17c│+0x001c: 0x22222222
```

- From f1() function, local variables are also created on stack as shown:



```
0xffffd160│+0x0000: 0xf7fb3000  →  0x001ead6c    ← $esp
0xffffd164│+0x0004: 0x12345678
0xffffd168│+0x0008: 0x0abcdef0
0xffffd16c│+0x000c: 0xf7dfc352  →  add esp, 0x10
0xffffd170│+0x0010: 0xffffd1a8  →  0x00000000    ← $ebp
0xffffd174│+0x0014: 0x56556241  →  <main+0054> add esp, 0x20
0xffffd178│+0x0018: 0x11111111
0xffffd17c│+0x001c: 0x22222222
```

- After stepping in through `f2()`, return value which is 6 can also be seen in `eax` register:



```
$eax   : 0x6
$ebx   : 0x0
$ecx   : 0x33671610
$edx   : 0xffffd1d4  →  0x00000000
$esp   : 0xffffd144  →  0x00000534
$ebp   : 0xffffd154  →  0xffffd170  →  0xffffd1a8  →  0x00000000
$esi   : 0xf7fb3000  →  0x001ead6c
$edi   : 0xf7fb3000  →  0x001ead6c
$eip   : 0x565561b6  →  <f2+0019> mov eax, DWORD PTR [ebp-0x4]
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffd144│+0x0000: 0x00000534    ← $esp
0xffffd148│+0x0004: 0x0000008e
0xffffd14c│+0x0008: 0xf7fb1224  →  0xf7f3ab20  →  0xfb1e0ff3
0xffffd150│+0x000c: 0x00000006
0xffffd154│+0x0010: 0xffffd170  →  0xffffd1a8  →  0x00000000      ← $ebp
0xffffd158│+0x0014: 0x565561e0  →  <f1+0025> add esp, 0x4
0xffffd15c│+0x0018: 0x00000005
0xffffd160│+0x001c: 0xf7fb3000  →  0x001ead6c

   0x565561ad <f2+0010>        mov    eax, DWORD PTR [ebp+0x8]
   0x565561b0 <f2+0013>        add    eax, 0x1
   0x565561b3 <f2+0016>        mov    DWORD PTR [ebp-0x4], eax
 → 0x565561b6 <f2+0019>        mov    eax, DWORD PTR [ebp-0x4]
   0x565561b9 <f2+001c>        leave
   0x565561ba <f2+001d>        ret
   0x565561bb <f1+0000>        push   ebp
   0x565561bc <f1+0001>        mov    ebp, esp
   0x565561be <f1+0003>        sub    esp, 0x10
```

- Practice running 32-bit and 64-bit versions of the `debugme.c` program to have a crystal-clear understanding of difference in function calling convention in the two architectures.
Happy Learning ☺

To Do:

- Given the following C program, where the `virus` function is not being called from anywhere inside the code. You are required to compile and load the binary of this source program inside `gdb`, and then execute it in such a way that the `virus` function gets executed and you get the output: "**`Let us Hack Planet Earth with Arif`**". Interested students can watch my video at the following link, where I have performed this task in Video Lecture # 38, from time 57:00 to 1:01:00. Happy Learning ☺

```
void f3(){
   return;
}
void f2(){
   f3();
}
void f1(){
   f2();
}
int main(){
   f1();
   return 0;
}
int virus(){
   printf("Let us Hack Planet Earth with Arif.\n");
   exit(0);
}
```

https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW