

Handout: 3.2

x86-64 Assembly & Reverse Engineering

A Bit of Word about Assembly Programming

Assembly language is a critical component in the realm of computer programming 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.

• Historical and Modern Uses

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.

• Importance and Uses

- **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.

Processor-Specific Assembly Languages

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
```

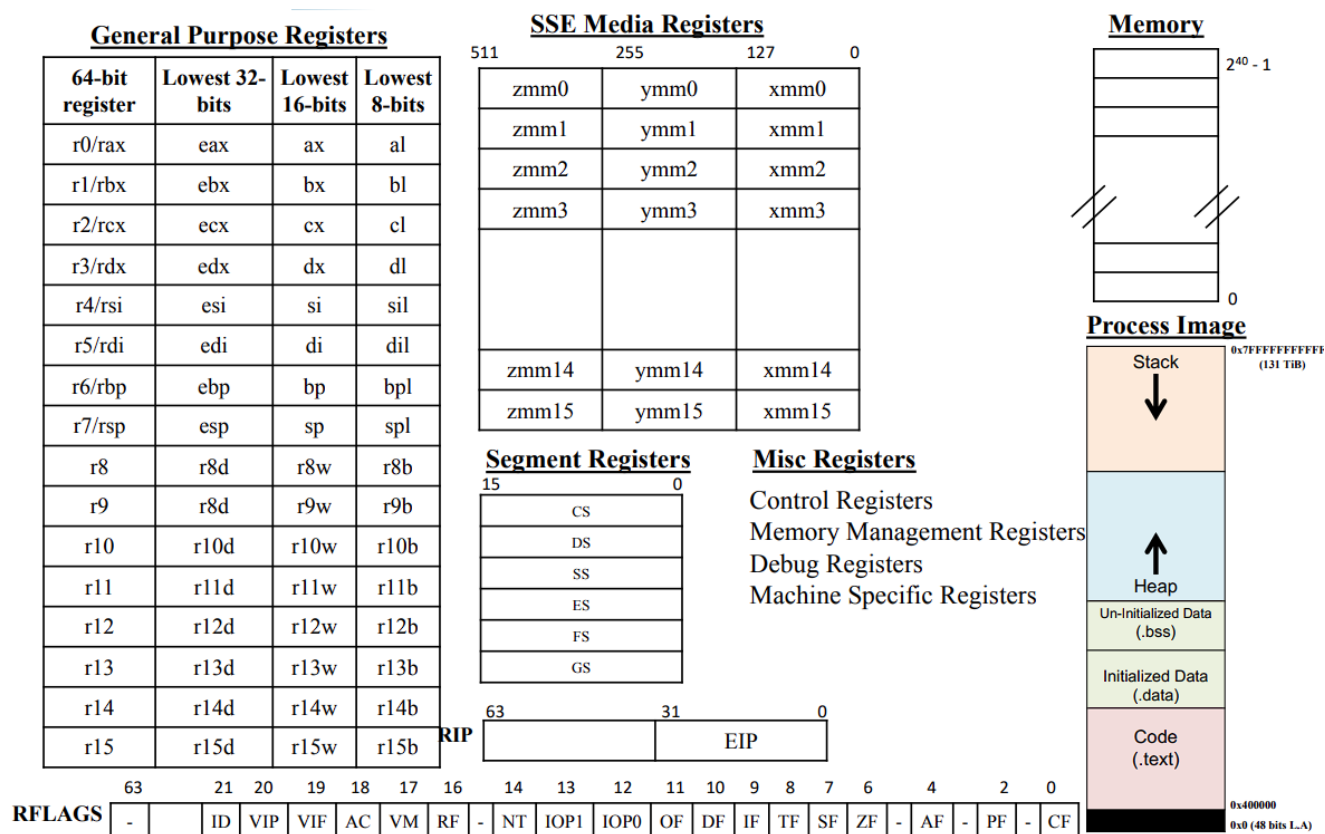
2. **ARM Assembly:**

- **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**
 - **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.
 - **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



- **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:

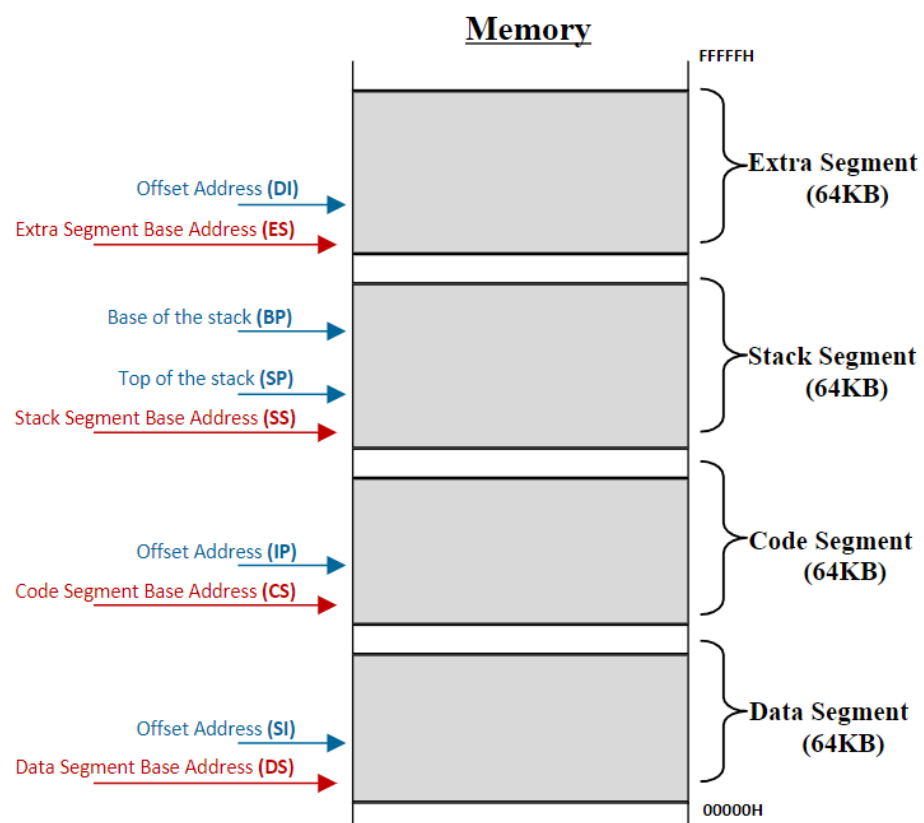
- **rbp**: Base Pointer, points to the bottom of the current stack frame
- **rsp**: Stack Pointer, points to the top of the current stack frame
- **rip**: Instruction Pointer, points to the instruction to be executed
- Arguments to a function are also passed via registers (rdi, rsi, rdx, rcx, r8, r9)
- 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

• Segment Registers of x86-64 Processor

Segment registers are a fundamental feature in x86 architecture, particularly in 16-bit and early 32-bit environments. They were used to manage memory segmentation, which is a technique for dividing memory into segments, each with its own address space. Here's a brief overview of their usage and relevance in modern architectures:

1. Usage in 16-bit Architecture:

- a. **Memory Segmentation (Purpose):** Segment registers were used to implement memory segmentation, allowing for a more organized way to manage memory. This was crucial in the 16-bit architecture where the addressable memory was limited.
 - **Registers:** In 16-bit mode, the x86 architecture includes several segment registers:
 - **CS (Code Segment):** Points to the base address (start) of segment containing executable code.
 - **DS (Data Segment):** Points to the base address (start) of segment used for data storage.
 - **SS (Stack Segment):** Points to the base address (start) of segment used for the stack.
 - **ES (Extra Segment):** Used for additional data segments, often for string operations.



- b. **Address Calculation:** A 16-bit segment register holds the base address of a segment, and a 16-bit offset specifies the location within that segment. The physical address is computed as: $\text{Physical Address} = (\text{Segment Register} * 16) + \text{Offset}$ **Example:** If $CS=0x2000$ and $IP=0x1234$, the physical address of the instruction would be $(0x2000 * 16) + 0x1234 = 0x201234$

2. Usage in 32-bit and 64-bit Architectures

a. 32-bit Architecture:

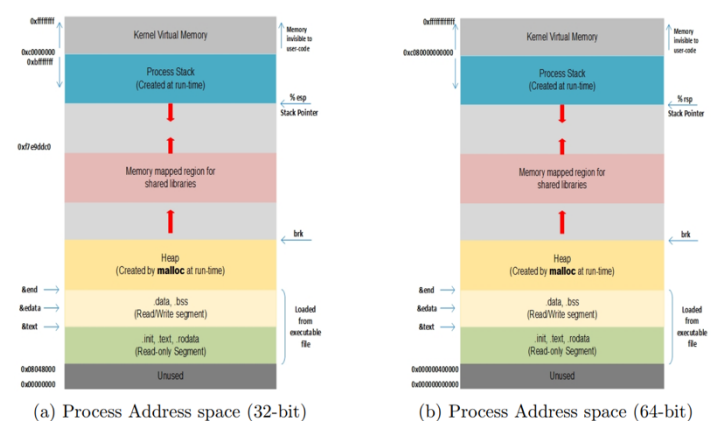
- i. **Flat Memory Model:** In 32-bit mode, the x86 architecture introduced the flat memory model, which reduced the importance of segment registers. Memory segmentation was still present, but most applications and operating systems used a flat address space where segment registers had less impact on address calculation.
- ii. **Segment Registers in 32-bit Mode:** While segment registers still exist, their role in addressing is minimized. The base address of segments is often set to zero, and segment registers mainly help in segment-based operations like protection and privilege levels.

b. 64-bit Architecture:

- i. **x86-64 Mode:** With the transition to 64-bit architecture (x86-64), the use of segment registers became even less prominent. The 64-bit mode uses a flat memory model by default where segmentation is largely ignored for most applications.
- ii. **Segment Registers in 64-bit Mode:** In x86-64 mode, segment registers are still present but are not used for addressing purposes. Most applications use a linear address space where segmentation does not affect address calculation:
- iii. **CS (Code Segment):** Remains important for defining code execution privileges but does not affect address computation.
- iv. **DS, ES, SS:** Typically used in a similar manner as in 32-bit mode but with less emphasis on segmentation.
- v. **FS and GS:** Additional segment registers introduced in 64-bit mode used for specific purposes, such as accessing thread-local storage or kernel data.

• 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:

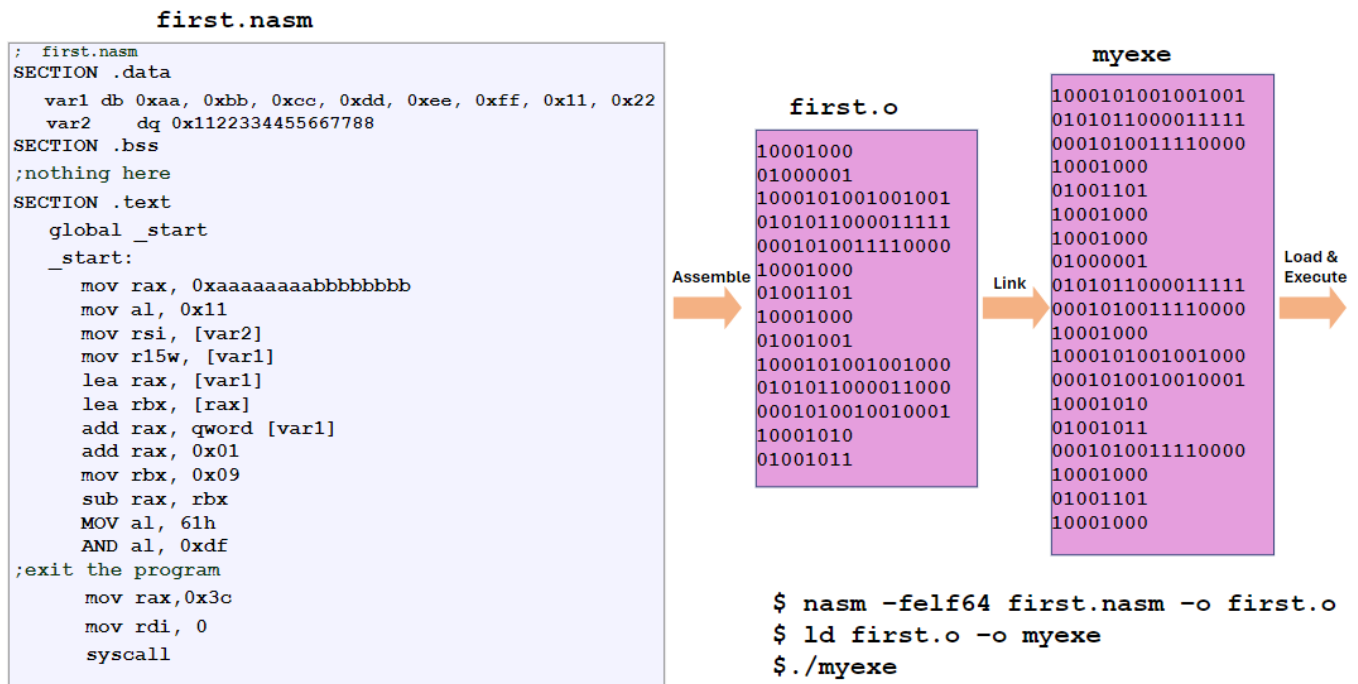


The description of different components of this address is beyond the scope of this handout. The layout of various segments of a process running on a Linux system on x86-64 is also shown in the figure (Code Section, Initialized Data Section, Uninitialized Data Section, Heap, Stack)

63 - 48 47 - 39 38 - 30 29 - 21 20 - 12 11 - 0

Unused	PML4 index	Page directory pointer index	Page directory index	Page table index	Page offset
--------	------------	------------------------------	----------------------	------------------	-------------

Structure of x86-64 Assembly Program



The above figure describes the structure of an x86-64 assembly program and its corresponding object file and final executable. 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`

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.

Categories of x86-64 Assembly Instructions

The following table divides and briefly describes the assembly instructions:

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

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

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
```

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 hello world 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 `.text` 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, depending on your architecture, you need to place the system call ID in the **rax** register. Next step is to place the system call arguments inside registers: **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**. If there are more than six arguments they are pushed on the stack. For floating point arguments, we use the registers **xmm0**, **xmm1**, and so on. After the system call returns, the return value can be found inside **rax** register. 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, 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 -felf64 syscalls.nasm
```

```
$ nasm -felf32 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 $?
```

```
0
```

```
;3.2/assemblyprogs/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)
```

Example 2: Displaying Hello World using Library Calls

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. Before calling `printf` library function, 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 vector (floating-point or SIMD) 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.

```
;3.2/assemblyprogs/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)
```

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

```
$ nasm -felf64 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 shows 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/assemblyprogs/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 shows 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 relevant flags. In the given code, since the result will be positive five, so the relevant 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/assemblyprogs/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.

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

```

```

;3.2/assemblyprogs/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

```

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.

Importance of Function Calling Convention:

Function calling conventions are fundamental in software development and security because they dictate how functions interact with each other and the system. 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.

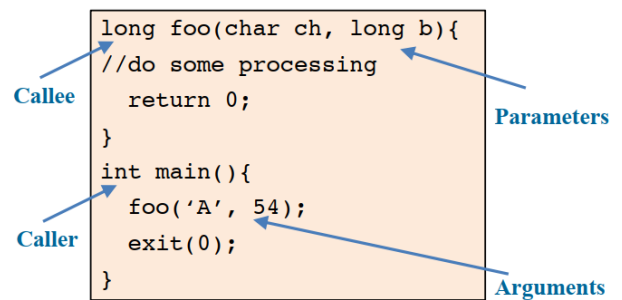
- **Exploitation**

- 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:

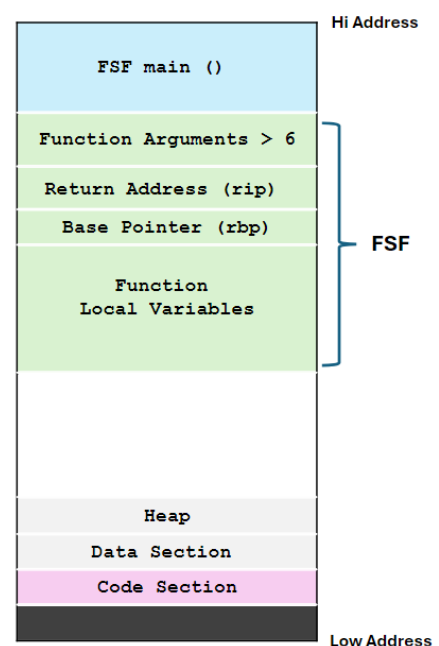
- A function may have arguments/parameters, which might be integer/floating point values as well as addresses pointing to data. This enables a function to operate on different data with each call.
- In high level programming languages like C and C++, the values passed by the caller to the callee are 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.
- 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:

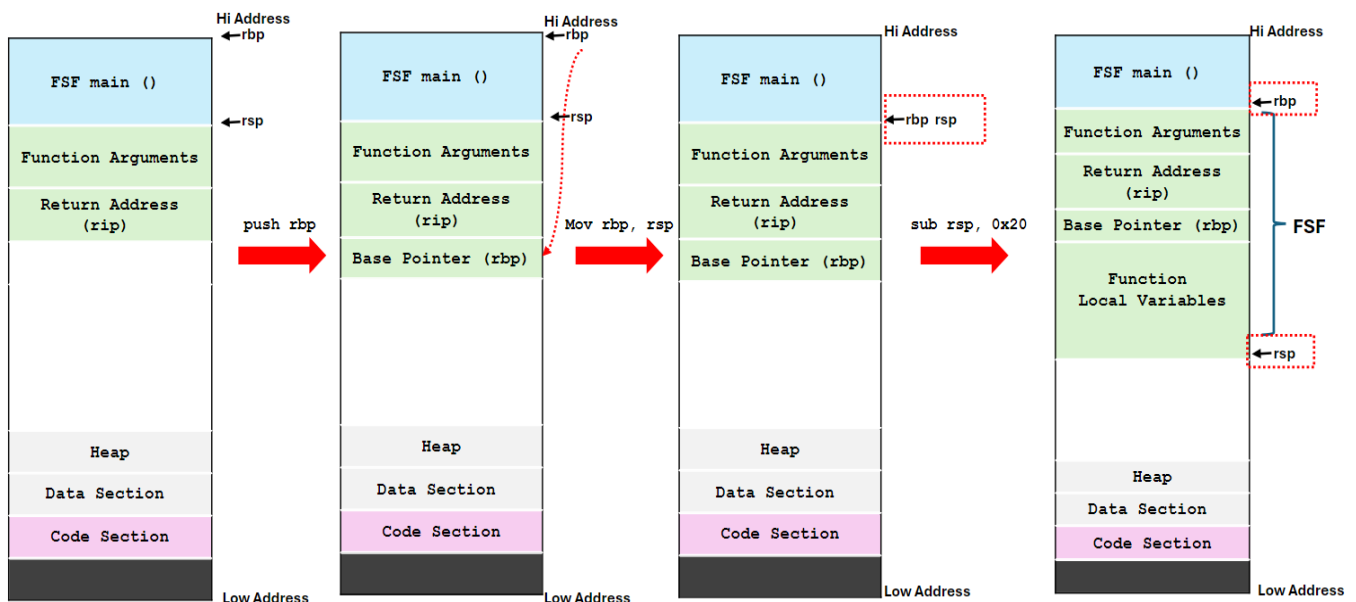
```

; 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

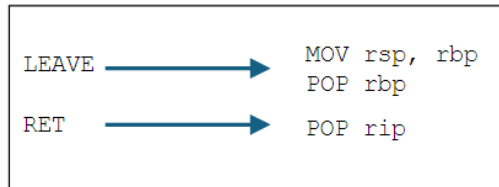
```

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

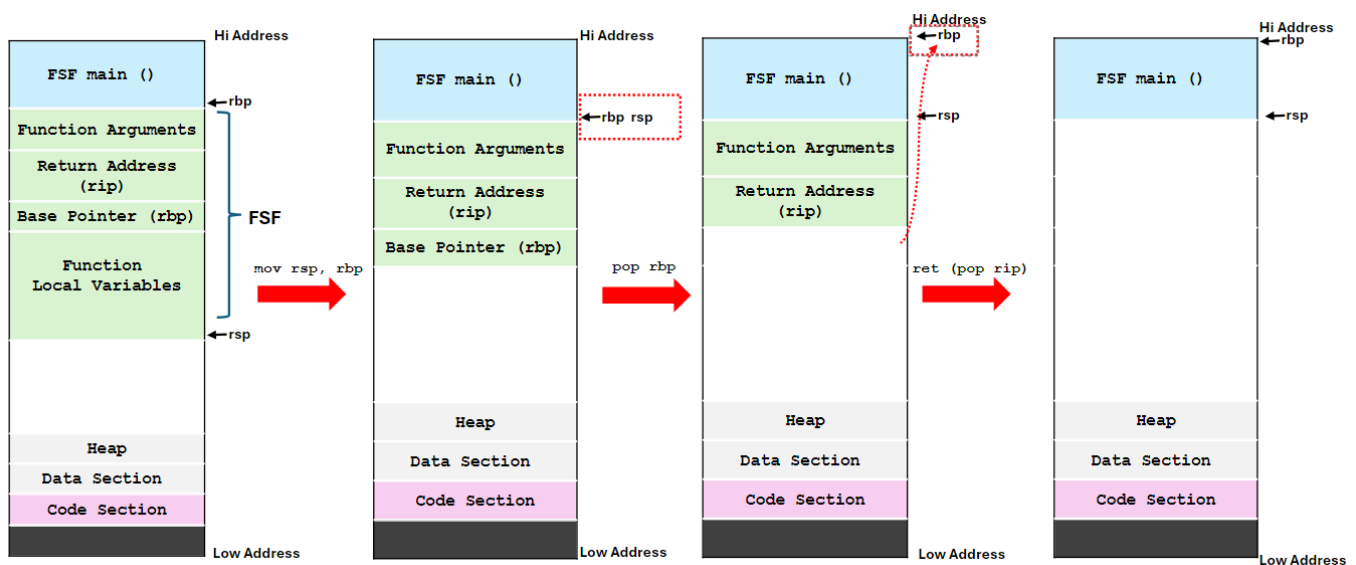


- **Procedure Epilog:** When callee is done with its execution, it first cleans up the FSF and then calls the return statement to transfer control to its caller by performing a procedure epilog:

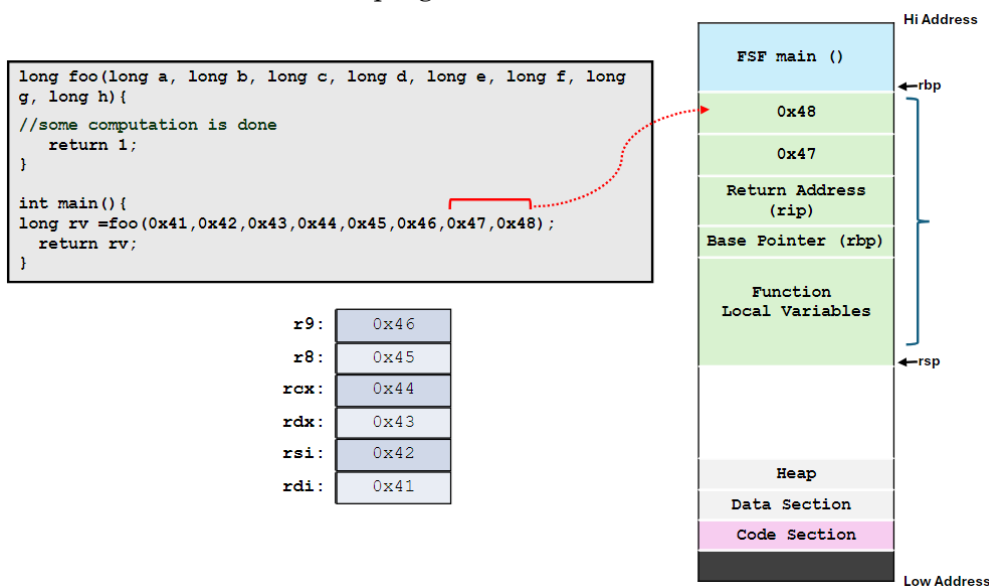
- Restore the old stack pointer (`mov rsp, rbp`).
- Restore the old base pointer (`pop rbp`).
- Return to the caller using `ret`.



- 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:



What is Reverse Engineering?

Reverse engineering in cybersecurity refers to the process of analyzing and deconstructing software, hardware, or systems to understand how they work. It often involves examining the inner workings of a program or device to extract valuable information, identify vulnerabilities, or determine its functionality. In order to perform reverse engineering, you need to have following skills:

- Proficiency in assembly language (e.g., x86/x64, ARM).
- Knowledge of binary file formats (e.g., ELF, PE).
- Familiarity with debugging tools and techniques.
- Understanding of operating systems, memory management, and processor architecture.
- Programming skills (e.g., C, C++, Python).

Key Purposes of Reverse Engineering

- **Malware Analysis:**
 - Understand the behavior of malware (e.g., viruses, ransomware, worms).
 - Identify how the malware spreads, its payload, and any vulnerabilities it exploits.
 - Develop antivirus signatures or patches to mitigate the threat.
- **Vulnerability Research:**
 - Analyze software or systems to uncover security flaws.
 - Help in the creation of exploits or, conversely, in fixing vulnerabilities through patches.
- **Software Cracking or Protection:**
 - Understand how software license checks or protections (e.g., DRM) are implemented.
 - Develop mechanisms to strengthen software protection or, in some cases, bypass restrictions (though this can be illegal depending on the context).
- **Digital Forensics:**
 - Analyze compromised systems to determine the root cause of an incident.
 - Extract data or artifacts from damaged or encrypted systems.

How Reverse Engineering Works

- **Static Analysis:** It involves examining the software or system without running it, with the focus on assembly code, binary structure, strings, imports/exports. The tools used are Radare2, IDA Pro, Ghidra, Hopper, strings.
- **Dynamic Analysis:** It involves analyzing the software or system while it runs, with the focus on its runtime behavior, memory usage and system interactions. The tools used are Radare2, IDA Pro, Ghidra, GNU gdb, OllyDbg, x64dbg, valgrind, strace, ltrace.
- **Protocol Analysis:** It involves intercepting and analyzing communication between devices or softwares. The tools used are Wireshark, Burp Suite, Fiddler.
- **H/W Reverse Engineering:** It involves analyzing physical devices or firmware. The tools used are Oscilloscopes, logic analyzers, or JTAG debuggers.

A Hello to Reversing Binaries

Example 1: Hello World with puts () Function

Create an executable of this simple C program:

```
$ gcc p1.c -o p1
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of main function shown below:

```
$ objdump -M intel -D ./p1
```

```
$ objdump -M intel -D ./p1 | grep -A 9 "<main>:"
```

```
00000000000001139 <main>:
    1139:      55                push    rbp
    113a:      48 89 e5          mov     rbp, rsp
    113d:      48 8d 05 c0 0e 00 00 lea     rax, [rip+0xec0]
    1144:      48 89 c7          mov     rdi, rax
    1147:      e8 e4 fe ff ff    call    1030 <puts@plt>
    114c:      90                nop
    114d:      5d                pop     rbp
    114e:      c3                ret
```

```
//3.2/cprogs/p1.c
#include <stdio.h>

void main(void) {
    puts("Hello World!");
}
```

Description:

- The first two lines of assembly represent the function **prologue**. The `push rbp` stores the `rbp` register of caller on stack, so we can use `rbp` for our purpose. The `mov rbp, rsp` copies the value of `rsp`, which is pointing to the stack frame of `main`, to `rbp`. Now both `rsp` and `rbp` are pointing to the same location in memory.
- The third line: `lea rax, [rip+0xec0]` is the load effective address instruction that is calculating an address by adding the contents of `rip` with a constant. This is actually the address of the string passed to the `puts()` function. The `rip` register contains an address, and `0xec0` is the offset from this address to where the string starts. The string is stored in the `.rodata` section, which is part of the binary file and is loaded in memory with the program. It usually contains program constants.
- Let us examine the `.rodata` section.

```
$ objdump -d -s -j .rodata ./p1
```

Disassembly of section `.rodata`:

```
00000000000002000 <_IO_stdin_used>:
    2000:      01 00 02 00 48 65 6c 6c 6f 20 57 6f 72 6c 64 21      ....Hello World!
```

- From above output, you can note that the `.rodata` section starts at address `0x2000` while the string starts four bytes farther, i.e., `0x2004`.
- At this moment the contents of `rip` register are `0x1144` and when you add `0xec0` in it you get `0x2004`, which is the address where the string starts. Thus, the `lea rax, [rip+0xec0]` will place `0x2004` inside the `rax` register, which is the starting address of the string.
- After this, the `mov rdi, rax` instruction will copy the address of the string inside the `rdi` register, which is the only and 1st argument to the `puts` function. ☺
- The fifth instruction calls the `puts()` function. This function is implemented in the GNU C library and its address is stored in the `.plt` section, hence the `puts@plt`. The `.plt` section is used to locate library functions whose addresses are not known at link time. According to the man pages, the `puts` function takes one parameter, a string. Given that we know that `rdi` contains the address of the string to be printed, we can safely assume that `puts` will get its parameter value from `rdi`.
- The last two instructions are the function **epilog**, which restores the previously saved `rbp` value into `rbp`. Finally, we have the `ret` instruction ☺

Example 2: Hello World with printf() Function

Create an executable of this simple C program:

```
$ gcc p2.c -o p2
```

Let us view the disassembly of this binary file using say objdump command. Right now, we are interested in the disassembly of main function shown below:

```
$ objdump -M intel -D ./p2 | grep -A 14 "<main>:"
```

```
00000000000001139 <main>:
    1139:      55                push    rbp
    113a:     48 89 e5          mov     rbp, rsp
    113d:     48 83 ec 10       sub     rsp, 0x10
    1141:     c7 45 fc 0a 00 00 00 mov     DWORD PTR [rbp-0x4], 0xa
    1148:     8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
    114b:     89 c6            mov     esi, eax
    114d:     48 8d 05 b0 0e 00 00 lea     rax, [rip+0xeb0]
    1154:     48 89 c7          mov     rdi, rax
    1157:     b8 00 00 00 00     mov     eax, 0x0
    115c:     e8 cf fe ff ff     call    1030 <printf@plt>
    1161:     b8 00 00 00 00     mov     eax, 0x0
    1166:     c9              leave   %eax
    1167:     c3              ret
```

```
//3.2/cprogs/p2.c
#include <stdio.h>

int main(void){
    int x = 10;
    printf("x=%d\n", x);
    return 0;
}
```

Description:

- Most of the code generated by the compiler is similar to the one as in the previous example, so we will discuss the lines which are new to us.
- The 113d: sub rsp, 0x10 instruction is part of function prolog, which is actually creating space for the local integer variable named **x**. Although it needs 4 bytes but additional space is allocated for alignment purposes.
- The 1141: mov DWORD PTR [rbp-0x4], 0xa instruction is copying the decimal value 10 (0xa) on the stack for variable **x**. Since the variable has type integer, so it is stored in 4 bytes and its value starts 4 bytes below the rbp.
- Now before calling the printf function, we need to place its two arguments inside the rdi and rsi registers respectively, starting from right to left.
 - The 1148: mov eax, DWORD PTR [rbp-0x4] is copying the value of variable x from stack into the eax register. Since this will be second argument to the printf() function, so in the next instruction, it is copied to rsi as shown 114b: mov esi, eax
 - The 114d: lea rax, [rip+0xeb0] is loading the address of the format string inside rax register. Since this will be first argument to the printf() function, in the next instruction it is copied to rdi as shown 1154: mov rdi, rax
 - Next 1157: mov eax, 0x0 the compiler places a zero in the eax register and then call to printf is made. The System-V AMD64 ABI specifies that before a function from the standard C library is called, the value in the rax register must specify the number of floating-point arguments passed to the function in XMM registers. So rax register is explicitly set to 0, indicating that no floating-point arguments are passed to printf.
- Next 1161: mov eax, 0x0 the compiler places a zero in the eax (or rax) register as the return value from the main() function.
- Finally, we have the function epilog containing the leave and the ret instructions, that will roll back the FSF.

Example 3: Hello World with Arrays

Create an executable of this simple C program:

```
$ gcc p3.c -o p3
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

```
$objdump -M intel -D ./p3 | grep -A 17 "<main>:"
```

```
00000000000001139 <main>:
1139:      55                push    rbp
113a:      48 89 e5          mov     rbp, rsp
113d:      48 83 ec 10       sub     rsp, 0x10
1141:      c7 45 f0 01 00 00 00 mov     DWORD PTR [rbp-0x10], 0x1
1148:      c7 45 f4 02 00 00 00 mov     DWORD PTR [rbp-0xc], 0x2
114f:      c7 45 f8 03 00 00 00 mov     DWORD PTR [rbp-0x8], 0x3
1156:      c7 45 fc 04 00 00 00 mov     DWORD PTR [rbp-0x4], 0x4
115d:      8b 45 f4          mov     eax, DWORD PTR [rbp-0xc]
1160:      89 c6            mov     esi, eax
1162:      48 8d 05 9b 0e 00 00 lea     rax, [rip+0xe9b]
1169:      48 89 c7          mov     rdi, rax
116c:      b8 00 00 00 00    mov     eax, 0x0
1171:      e8 ba fe ff ff    call    1030 <printf@plt>
1176:      b8 00 00 00 00    mov     eax, 0x0
117b:      c9              leave   %eax
117c:      c3              ret
```

```
//3.2/cprogs/p3.c
#include <stdio.h>

int main(){
    int x[4] = {1,2,3,4};
    printf("x[1] = %d\n",x[1]);
    return 0;
}
```

Description:

- The 1141: `mov DWORD PTR [rbp-0x10], 0x1` instruction is copying the decimal value 1 (0x1) on the stack as the first integer value inside the array `x`. Since the variable has type integer, so it is stored in 4 bytes and its value starts 0x10 bytes below the `rbp`.
- Similarly, the 1148: `mov DWORD PTR [rbp-0xc], 0x2` instruction is copying the decimal value 2 (0x2) on the stack as the second integer value inside the array `x`. Note that the value is stored starting at address 0xc bytes below the `rbp`. In the same fashion, all the four values. Of the array are stored on the stack.
- Now before calling the `printf` function, we need to place its two arguments inside the `rdi` and `rsi` registers respectively, starting from right to left.
 - Following two instructions places the 2nd argument inside `esi`, which is `x[1]` i.e., 2:


```
mov eax, DWORD PTR [rbp-0xc]
mov esi, eax
```
 - Next two instruction places the 1st argument inside `esi`, which is address of format string:


```
lea rax, [rip-0xe9b]
mov rdi, rax
```
 - Next 116c: `mov eax, 0x0` the compiler places a zero in the `eax` register to specify that XMM registers are not involved in arguments passing.
- Remaining code is already discussed in previous examples.

Example 4: Hello World with if---else

Create an executable of this simple C program:

```
$ gcc p4.c -o p4
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

```
//3.2/cprogs/p4.c
#include <stdio.h>
int main(){
    int x = 10;
    int y = 5;
    if(x<=100){
        y = y - 3;
        printf("Less than\n");
    }
    else{
        y = y + 3;
        printf("Greater than\n");
    }
    return 0;
}
```

```
$objdump -M intel -D ./p4 | grep -A 20 "<main>:"
```

```
00000000000001139 <main>:
1139:      55                push    rbp
113a:      48 89 e5          mov     rbp, rsp
113d:      48 83 ec 10       sub     rsp, 0x10
1141:      c7 45 fc 0a 00 00 00 mov     DWORD PTR [rbp-0x4], 0xa
1148:      c7 45 f8 05 00 00 00 mov     DWORD PTR [rbp-0x8], 0x5
114f:      83 7d fc 64       cmp     DWORD PTR [rbp-0x4], 0x64
1153:      7f 15            jg      116a <main+0x31>
1155:      83 6d f8 03       sub     DWORD PTR [rbp-0x8], 0x3
1159:      48 8d 05 a4 0e 00 00 lea     rax, [rip+0xea4]
1160:      48 89 c7          mov     rdi, rax
1163:      e8 c8 fe ff ff    call    1030 <puts@plt>
1168:      eb 13            jmp     117d <main+0x44>
116a:      83 45 f8 03       add     DWORD PTR [rbp-0x8], 0x3
116e:      48 8d 05 99 0e 00 00 lea     rax, [rip+0xe99]
1175:      48 89 c7          mov     rdi, rax
1178:      e8 b3 fe ff ff    call    1030 <puts@plt>
117d:      b8 00 00 00 00    mov     eax, 0x0
1182:      c9              leave
1183:      c3              ret
```

To Do:

Understand the disassembly, and write down description of above compiler generated assembly code.
Happy Learning ☺

Example 5: Hello World with Loops

Create an executable of this simple C program:

```
$ gcc p5.c -o p5
```

Let us view the disassembly of this binary file using say `objdump` command. Right now, we are interested in the disassembly of `main` function shown below:

```
//3.2/cprogs/p5.c
#include <stdio.h>
int main(){
    int i = 0;
    int limit = 5;
    for(i = 0;i<limit; i++){
        printf("%d ",i);
    }
    printf("\n");
    return 0;
}
```

```
$ objdump -M intel -D ./p5 | grep -A 22 "<main>:"
```

```
00000000000001149 <main>:
1149:      55                push    rbp
114a:      48 89 e5          mov     rbp, rsp
114d:      48 83 ec 10       sub     rsp, 0x10
1151:      c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
1158:      c7 45 f8 05 00 00 00 mov     DWORD PTR [rbp-0x8], 0x5
115f:      c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
1166:      eb 1d            jmp     1185 <main+0x3c>
1168:      8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
116b:      89 c6            mov     esi, eax
116d:      48 8d 05 90 0e 00 00 lea     rax, [rip+0xe90]
1174:      48 89 c7          mov     rdi, rax
1177:      b8 00 00 00 00    mov     eax, 0x0
117c:      e8 bf fe ff ff    call    1040 <printf@plt>
1181:      83 45 fc 01       add     DWORD PTR [rbp-0x4], 0x1
1185:      8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
1188:      3b 45 f8          cmp     eax, DWORD PTR [rbp-0x8]
118b:      7c db            jl      1168 <main+0x1f>
118d:      bf 0a 00 00 00    mov     edi, 0xa
1192:      e8 99 fe ff ff    call    1030 <putchar@plt>
1197:      b8 00 00 00 00    mov     eax, 0x0
119c:      c9              leave
119d:      c3              ret
```

To Do:

Understand the disassembly, and write down description of above compiler generated assembly code.
Happy Learning ☺

Example 6: Hello World with Function Call

Create an executable of this simple C program:

```
$ gcc p6.c -o p6
```

Let us view the disassembly of this binary file using say objdump command. Right now, we are interested in the disassembly of main function shown below:

```
//3.2/cprogs/p6.c
#include <stdio.h>
int foo(int,int);
int main(){
    int val1 = 10;
    int val2 = 20;
    int sum = foo(val1,val2);
    printf("sum is: %d\n",sum);
    return 0;
}
int foo(int a, int b){
    int out = a + b;
    return out;
}
```

```
$ objdump -M intel -D ./p6 | grep -A 33 "<main>:"
```

```
00000000000001149 <main>:
1149:      55                push    rbp
114a:      48 89 e5          mov     rbp, rsp
114d:      48 83 ec 10       sub     rsp, 0x10
1151:      c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
1158:      c7 45 f8 05 00 00 00 mov     DWORD PTR [rbp-0x8], 0x5
115f:      c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
1166:      eb 1d            jmp     1185 <main+0x3c>
1168:      8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
116b:      89 c6            mov     esi, eax
116d:      48 8d 05 90 0e 00 00 lea     rax, [rip+0xe90]
1174:      48 89 c7          mov     rdi, rax
1177:      b8 00 00 00 00    mov     eax, 0x0
117c:      e8 bf fe ff ff    call    1040 <printf@plt>
1181:      83 45 fc 01       add     DWORD PTR [rbp-0x4], 0x1
1185:      8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
1188:      3b 45 f8          cmp     eax, DWORD PTR [rbp-0x8]
118b:      7c db            jl      1168 <main+0x1f>
118d:      bf 0a 00 00 00    mov     edi, 0xa
1192:      e8 99 fe ff ff    call    1030 <putchar@plt>
1197:      b8 00 00 00 00    mov     eax, 0x0
119c:      c9              leave   %eax
119d:      c3              ret
```

```
00000000000001181 <foo>:
1181:      55                push    rbp
1182:      48 89 e5          mov     rbp, rsp
1185:      89 7d ec          mov     DWORD PTR [rbp-0x14], edi
1188:      89 75 e8          mov     DWORD PTR [rbp-0x18], esi
118b:      8b 55 ec          mov     edx, DWORD PTR [rbp-0x14]
118e:      8b 45 e8          mov     eax, DWORD PTR [rbp-0x18]
1191:      01 d0            add     eax, edx
1193:      89 45 fc          mov     DWORD PTR [rbp-0x4], eax
1196:      8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
1199:      5d              pop     rbp
119a:      c3              ret
```

To Do:

- The following is the objdump of the executable from the C source file 3.2/cprogs/pass1.c
- Study it carefully (perform static analysis), and give as much information about the working of this binary as possible. Happy Learning ☺

0000000000001149 <func>:

1149: 55	push	rbp
114a: 48 89 e5	mov	rbp, rsp
114d: 48 83 ec 10	sub	rsp, 0x10
1151: 89 7d fc	mov	DWORD PTR [rbp-0x4], edi
1154: 81 7d fc 2b 02 00 00	cmp	DWORD PTR [rbp-0x4], 0x22b
115b: 75 11	jne	116e <func+0x25>
115d: 48 8d 05 a0 0e 00 00	lea	rax, [rip+0xea0]
1164: 48 89 c7	mov	rdi, rax
1167: e8 c4 fe ff ff	call	1030 <puts@plt>
116c: eb 0f	jmp	117d <func+0x34>
116e: 48 8d 05 9f 0e 00 00	lea	rax, [rip+0xe9f]
1175: 48 89 c7	mov	rdi, rax
1178: e8 b3 fe ff ff	call	1030 <puts@plt>
117d: 90	nop	
117e: c9	leave	
117f: c3	ret	

0000000000001180 <main>:

1180: 55	push	rbp
1181: 48 89 e5	mov	rbp, rsp
1184: 48 83 ec 20	sub	rsp, 0x20
1188: 89 7d ec	mov	DWORD PTR [rbp-0x14], edi
118b: 48 89 75 e0	mov	QWORD PTR [rbp-0x20], rsi
118f: 48 8b 45 e0	mov	rax, QWORD PTR [rbp-0x20]
1193: 48 83 c0 08	add	rax, 0x8
1197: 48 8b 00	mov	rax, QWORD PTR [rax]
119a: 48 89 c7	mov	rdi, rax
119d: e8 9e fe ff ff	call	1040 <atoi@plt>
11a2: 89 45 fc	mov	DWORD PTR [rbp-0x4], eax
11a5: 8b 45 fc	mov	eax, DWORD PTR [rbp-0x4]
11a8: 89 c7	mov	edi, eax
11aa: e8 9a ff ff ff	call	1149 <func>
11af: b8 00 00 00 00	mov	eax, 0x0
11b4: c9	leave	
11b5: c3	ret	

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, radare2, IDA Pro, Ghidra, OllyDbg, X64dbg, Immunity Debugger, strace, ltrace, 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, alpha, arm, mips, powerpc, sparc and many others. GDB works for many programming languages including Assembly, C/C++, Objective C, OpenCL, Go, Modula-2, Fortran, Pascal and Ada. To install gdb on your Kali Linux machine:

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
<pre>\$ nasm -g -felf64 prog1.nasm \$ gcc -ggdb -c prog1.c</pre>	In order to load and properly analyze a program in gdb you need to compile it with <code>-g</code> or <code>-ggdb</code> option, to instruct the compiler to keep debugging symbols, source file names and line numbers in the object files
<pre>\$ gdb (gdb) file myexe OR \$ gdb myexe (gdb)</pre>	There are two ways to load a binary inside gdb by either running <code>gdb</code> command and then specifying the binary name with the <code>file</code> command. Or by specifying the binary name as an argument to <code>gdb</code> .
<pre>(gdb) quit</pre>	Exits the current session of gdb.
<pre>(gdb) help (gdb) help <classname> (gdb) help <command></pre>	The <code>help</code> command of gdb is used to display the listing of twelve different classes in which gdb commands are categorized. You can also specify the <code>classname</code> (breakpoints, running, stack, ...) or the command to get help about it.
<pre>(gdb) run [arg1 arg2 ...] OR (gdb) set args arg1 arg2 ... (gdb) run</pre>	Once the program is loaded and gdb is running, you can pass command line arguments to the binary using the <code>run</code> command of gdb. Or can use the <code>set</code> command instead and later use the <code>run</code> command.
<pre>(gdb) attach <PID></pre>	If you want to debug a process that is already running, you can attach GDB to it using its process ID (PID).
<pre>(gdb) info sources/functions/variables/locals (gdb) info registers/all (gdb) info sharedlibrary (gdb) info address <function name></pre>	Once a program <i>loaded</i> inside gdb, you can use the <code>info</code> 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.
<pre>(gdb) list [1,12] (gdb) list <filename>:<line#> (gdb) list <filename>:<function name></pre>	The <code>list</code> command of gdb is used to display the source code (provided if the source file is there in the pwd)
<pre>(gdb) disassemble (gdb) disassemble <function name> (gdb) set disassembly-flavor intel</pre>	Disassembles the current function or code segment. By default, gdb disassembles in AT&T format, to change the format to intel, use the <code>set disassembly-flavor</code> command.
<pre>(gdb) break <filename>:<line#> (gdb) break <filename>:<function name> (gdb) break <filename>:*0x2xffff0500</pre>	Breakpoint is the LOC in your program where you want to stop the execution. You can set as many breakpoints as you feel like using the <code>break</code> command of gdb by mentioning the <code>line#</code> , <code>function name</code> , or by virtual address
<pre>(gdb) info break (gdb) disable <breakpoint#> (gdb) enable <breakpoint#> (gdb) delete <breakpoint#> (gdb) clear <breakpoint#></pre>	To get the information about the existing breakpoints already set in your program, you can use the <code>info</code> command. Moreover, you can disable/enable/delete/clear breakpoints.

<pre>(gdb) watch <variable name> (gdb) info watch (gdb) disable <watchpoint#> (gdb) enable <watchpoint#> (gdb) delete <watchpoint#> (gdb) clear <watchpoint#></pre>	<p>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.</p>
<pre>(gdb) continue / c / ci (gdb) next / n / ni (gdb) step / s / si (gdb) finish</pre>	<p>Once a breakpoint is hit, you can do the following:</p> <ul style="list-style-type: none"> ○ c: Continue till the next breakpoint or end of program. ○ n: Execute and move to next instruction, but don't dive into functions. ○ s: Execute and move to next instruction, by diving into functions. ○ finish: Continue until the current function returns.
<pre>(gdb) print /format-char <var-name></pre>	<p>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</p> <ul style="list-style-type: none"> ○ /d is for signed decimal ○ /u is for unsigned decimal ○ /x for printing as hex ○ /o for printing as octal ○ /t for printing as binary ○ /f for floating point number ○ /s for C-string ○ /a for address <p>Note: Unlike print the display command is used to display the value of variable, each time the program stops.</p>
<pre>(gdb) set variable <var-name> = <value></pre>	<p>The set command is used to modify the value of a variable.</p>
<pre>(gdb) x/12cb <address> (gdb) x/12db &var1 (gdb) x/4xb *0x601000 (gdb) x/32b \$rsp</pre>	<p>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:</p> <ul style="list-style-type: none"> ○ Count field, which is a number in decimal. ○ Format field, which is a single letter with 'd' for decimal, 'x' for hex, 't' for binary and 'c' for ASCII. ○ Size field, which is single letter with 'b' for byte, 'h' for 16-bit word, and 'w' for 32-bit word.
<pre>(gdb) backtrace</pre>	<p>The backtrace command or its alias b displays the call trace of a program.</p>
<pre>(gdb) ! Clear</pre>	<p>To run the OS shell commands inside gdb, you can precede the command with a ! symbol.</p>

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

Hands On Practice of GDB

Let us write down a multi-file C source code on our Kali Linux machine using some text editor like vim, nano, or gedit

```
//3.2/cprogs/gdb/driver.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include "mymath.h"

int main(int argc, char* argv[]) {
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    int ans1 = myadd(num1, num2);
    int ans2 = mysub(num1, num2);
    printf("%d + %d = %d \n", num1, num2, ans1);
    printf("%d - %d = %d \n", num1, num2, ans2);
    return 0;
}
```

```
//3.2/cprogs/gdb/mymath.h
```

```
int myadd(int, int);
int mysub(int, int);
```

```
//3.2/cprogs/gdb/mysub.c
```

```
int mysub(int a, int b) {
    int diff = a - b;
    return diff;
}
```

```
//3.2/cprogs/gdb/myadd.c
```

```
int myadd(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Compile the source files using gcc,

```
$ gcc -ggdb *.c -o myexe
```

Load the binary inside gdb a, pass the command line arguments, and run the binary:

```
$ gdb -q ./myexe
```

```
Reading symbols from ./myexe ...
```

```
(gdb) set args 27 18
```

```
(gdb) run
```

```
Starting program: myexe 27 18
```

```
[Thread debugging using libthread_db enabled]
```

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so"
```

```
27 + 18 = 45
```

```
27 - 18 = 9
```

```
[Inferior 1 (process 281093) exited normally]
```

```
(gdb) quit
```

To Do:

Practice all the commands of gdb mentioned in the previous table to practically understand all the concepts discussed in this handout 😊

GDB with PEDA & GEF

In the last handout we have practically used `gdb`, which 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
```

Hands On Practice of GDB with GEF Plugin

Let us write down the following C source code on our Kali Linux machine using some text editor like vim, nano, or gedit. 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 shell.

```
//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;
}
```

To Do:

Compile the source 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

Here is a brief description of each:

The screenshot displays the GEF debugger interface with the following panels:

- Registers Panel (1):** Shows the state of CPU registers. For example, `$rax` is `0x000055555555190` pointing to `<main+0000> endbr64`, and `$rsp` is `0x00007fffffdfa8` pointing to `<__libc_start_main+00f3> mov edi, eax`.
- Stack Panel (2):** Displays the call stack. The top of the stack shows `0x00007fffffdfa8` with instruction `+0x0000: 0x00007fffffdfa8 → <__libc_start_main+00f3> mov edi, eax`.
- Code Panel (3):** Shows assembly code. The current instruction is `0x55555555190 <f1+0047> mov eax, 0x1`, which is highlighted in green.
- Threads & Trace Panels (4):** The **Threads** panel shows `[#0] Id 1, Name: "myexe", stopped 0x55555555190 in main (), reason: BREAKPOINT`. The **Trace** panel shows `[#0] 0x55555555190 → main(argc=0x5555, argv=0x7ffff7fb72e8 <__exit_funcs_lock>)`.

- 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`.
- 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.
- 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.
- Source Panel:** This panel displays the corresponding high level language code, with the current LOC highlighted. This way you can correlate the high-level code with its corresponding assembly.
- Threads & Trace Panels:** This provides information about the threads in a multithreaded program, including their states and stack traces.

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.
```

Procedure Prolog

call to f1()

Procedure Epilog

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 `-fcf-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:


```

$rbx : 0x00005555555210 → <_libc_csu_init+0000> endbr64
$rcx : 0x00005555555210 → <_libc_csu_init+0000> endbr64
$rdx : 0x00007fffffffe098 → 0x00007fffffffe3c7 → "SHELL=/bin/bash"
$rsp : 0x00007fffffffd98 → 0x00007ffff7dea083 → <_libc_start_main+00f3> mov edi, eax
$rbp : 0x0
$rsi : 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling"
$rdi : 0x1
$rip : 0x00005555555190 → <main+0000> endbr64
$cr8 : 0x0
$cr9 : 0x00007ffff7fe0d60 → <_dl_fini+0000> endbr64
$cr10 : 0x0
$cr11 : 0x00007ffff7f758f0 → 0x000080003400468
$cr12 : 0x00005555555040 → <_start+0000> endbr64
$cr13 : 0x00007fffffffe080 → 0x0000000000000001
$cr14 : 0x0
$cr15 : 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

```

rip containing the address of current instruction

```

0x00007fffffffd98: 0x0000: 0x00007ffff7dea083 → <_libc_start_main+00f3> mov edi, eax ← $rsp
0x00007ffffffdfa0: 0x0008: 0x00007ffff7ffc620 → 0x00050fa000000000
0x00007ffffffdfa8: 0x0010: 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling"
0x00007ffffffdfb0: 0x0018: 0x0000000010000000
0x00007ffffffdfb8: 0x0020: 0x00005555555190 → <main+0000> endbr64
0x00007ffffffdfc0: 0x0028: 0x00005555555210 → <_libc_csu_init+0000> endbr64
0x00007ffffffdfc8: 0x0030: 0x0000000000000001
0x00007ffffffdfd0: 0x0038: 0x00005555555040 → <_start+0000> endbr64

```

stack growing towards lower addresses

```

0x55555555189 <f1+0047> mov eax, 0x1
0x5555555518e <f1+004c> leave
0x5555555518f <f1+004d> ret
→ 0x55555555190 <main+0000> endbr64
0x55555555194 <main+0004> push rbp
0x55555555195 <main+0005> mov rbp, rsp
0x55555555198 <main+0008> sub rsp, 0x30
0x5555555519c <main+000c> mov DWORD PTR [rbp-0x24], edi
0x5555555519f <main+000f> mov QWORD PTR [rbp-0x30], rsi

```

```

20
21 return 1;
22
23 }
24
→ 25 // argc=0x5555, argv=0x00007fffffffd98 → [...] → 0x0000000000000000
26 int main(int argc, char *argv[]){
27     unsigned long main_var1 = 0x1122334455667788;
28
29     unsigned long main_var2 = 0x99aabbccddeeff00;
30

```

- 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 : 0x00005555555210 → <_libc_csu_init+0000> endbr64
$rcx : 0x00005555555210 → <_libc_csu_init+0000> endbr64
$rdx : 0x00007fffffffe098 → 0x00007fffffffe3c7 → "SHELL=/bin/bash"
$rsp : 0x00007fffffffd98 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling"
$rbp : 0x00007fffffffd98 → 0x0000000000000000
$rsi : 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling"
$rdi : 0x1
$rip : 0x000055555551b1 → <main+0021> movabs rax, 0x99aabbccddeeff00
$cr8 : 0x0
$cr9 : 0x00007ffff7fe0d60 → <_dl_fini+0000> endbr64
$cr10 : 0x0
$cr11 : 0x00007ffff7f758f0 → 0x000080003400468
$cr12 : 0x00005555555040 → <_start+0000> endbr64
$cr13 : 0x00007fffffffe080 → 0x0000000000000001
$cr14 : 0x0
$cr15 : 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

0x00007fffffffd98: 0x0000: 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling" ← $rsp
0x00007fffffffd98: 0x0008: 0x0000000055555210
0x00007fffffffd98: 0x0010: 0x0000000000000000
0x00007fffffffd98: 0x0018: 0x1122334455667788 → variable created on the
0x00007fffffffd98: 0x0020: 0x0000000000000000
0x00007fffffffd98: 0x0028: 0x0000000000000000
0x00007fffffffd98: 0x0030: 0x0000000000000000 ← $rbp
0x00007fffffffd98: 0x0038: 0x00007ffff7dea083 → <_libc_start_main+00f3> mov edi, eax

0x5555555519f <main+000f> mov QWORD PTR [rbp-0x30], rsi
0x555555551a3 <main+0013> movabs rax, 0x1122334455667788
0x555555551ad <main+001d> mov QWORD PTR [rbp-0x18], rax
→ 0x555555551b1 <main+0021> movabs rax, 0x99aabbccddeeff00
0x555555551bb <main+002b> mov QWORD PTR [rbp-0x10], rax
0x555555551bf <main+002f> lea rax, [rip+0xe3e] # 0x555555556004
0x555555551c6 <main+0036> mov QWORD PTR [rbp-0x8], rax
0x555555551ca <main+003a> push 0xffffffff88888888
0x555555551cf <main+003f> mov DWORD PTR [rsp+0x4], 0x0

24
25 int main(int argc, char *argv[]){
26
27     unsigned long main_var1 = 0x1122334455667788;
28
→ 29     // 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.

```

0x00007ffffffdf60|+0x0000: 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling" ←$rsp
0x00007ffffffdf68|+0x0008: 0x00000000155555210
0x00007ffffffdf70|+0x0010: 0x0000000000000000
0x00007ffffffdf78|+0x0018: 0x1122334455667788
0x00007ffffffdf80|+0x0020: 0x99aabbccddeeff00
0x00007ffffffdf88|+0x0028: 0x0000555555556004 → 0x0000000066697241 ("Arif?")
0x00007ffffffdf90|+0x0030: 0x0000000000000000 ←$rbp
0x00007ffffffdf98|+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 : 0x00007ffffffdf48 → 0x000055555555201 → <main+0071> add rsp, 0x10
$rbp : 0x00007ffffffdf98 → 0x0000000000000000
$rsi : 0x22222222
$rdi : 0x11111111
$rin : 0x0000555555555142 → <f1+0000> endbr64
$r8 : 0x55555555
$r9 : 0x66666666
$r10 : 0x0
$r11 : 0x00007ffff7f58f0 → 0x000080003400468
$r12 : 0x000055555555040 → <_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

0x00007ffffffdf48|+0x0000: 0x000055555555201 → <main+0071> add rsp, 0x10 ←$rsp
0x00007ffffffdf50|+0x0008: 0x0000000077777777 "www"?
0x00007ffffffdf58|+0x0010: 0x0000000088888888
0x00007ffffffdf60|+0x0018: 0x00007fffffffe088 → 0x00007fffffffe3a6 → "/home/user/sec/func/func_calling"
0x00007ffffffdf68|+0x0020: 0x00000000155555210
0x00007ffffffdf70|+0x0028: 0x0000000000000000
0x00007ffffffdf78|+0x0030: 0x1122334455667788
0x00007ffffffdf80|+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:
0x000055555555190 <+0>: endbr64
0x000055555555194 <+4>: push rbp
0x000055555555195 <+5>: mov rbp,rsp
0x000055555555198 <+8>: sub rsp,0x30
0x00005555555519c <+12>: mov DWORD PTR [rbp-0x24],edi
0x00005555555519f <+15>: mov QWORD PTR [rbp-0x30],rsi
0x0000555555551a3 <+19>: movabs rax,0x1122334455667788
0x0000555555551ad <+29>: mov QWORD PTR [rbp-0x18],rax
0x0000555555551b1 <+33>: movabs rax,0x99aabbccddeeff00
0x0000555555551bb <+43>: mov QWORD PTR [rbp-0x10],rax
0x0000555555551bf <+47>: lea rax,[rip+0xe3e] # 0x55201
0x0000555555551c6 <+54>: mov QWORD PTR [rbp-0x8],rax
0x0000555555551ca <+58>: push 0xffffffff88888888
0x0000555555551cf <+63>: mov DWORD PTR [rsp+0x4],0x0
0x0000555555551d7 <+71>: push 0x77777777
0x0000555555551dc <+76>: mov r9d,0x66666666
0x0000555555551e2 <+82>: mov r8d,0x55555555
0x0000555555551e8 <+88>: mov ecx,0x44444444
0x0000555555551ed <+93>: mov edx,0x33333333
0x0000555555551f2 <+98>: mov esi,0x22222222
0x0000555555551f7 <+103>: mov edi,0x11111111
0x0000555555551fb <+108>: call 0x55555555142 <f1>
0x000055555555201 <+113>: add rsp,0x10
0x000055555555209 <+117>: mov QWORD PTR [rbp-0x1c],eax
0x00005555555520d <+125>: leave
0x00005555555520e <+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).

```

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

0x5555555515a <f1+0018> mov QWORD PTR [rbp-0x40], rcx
0x5555555515e <f1+001c> mov QWORD PTR [rbp-0x48], r8
0x55555555162 <f1+0020> mov QWORD PTR [rbp-0x50], r9
→ 0x55555555166 <f1+0024> movabs rax, 0x123456789
0x55555555170 <f1+002e> mov QWORD PTR [rbp-0x10], rax
0x55555555174 <f1+0032> mov QWORD PTR [rbp-0x18], 0xabcdef
0x5555555517c <f1+003a> mov edi, 0x5
0x55555555181 <f1+003f> call 0x55555555129 <f2>
0x55555555186 <f1+0044> mov DWORD PTR [rbp-0x14], eax

10
11 }
12
13 int f1(long a, long b, long c, long d, long e, long f, long g, long h){
14
15     // f1_var1=0x555555554040
→ 15     unsigned long f1_var1 = 0x123456789;
16
17     unsigned long f1_var2 = 0x0abcdef;
18
19     int rv = f2(5);
20
21 }

```

- 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

0x55555555166 <f1+0024> movabs rax, 0x123456789
0x55555555170 <f1+002e> mov QWORD PTR [rbp-0x10], rax
0x55555555174 <f1+0032> mov QWORD PTR [rbp-0x18], 0xabcdef
→ 0x5555555517c <f1+003a> mov edi, 0x5
0x55555555181 <f1+003f> call 0x55555555129 <f2>
0x55555555186 <f1+0044> mov DWORD PTR [rbp-0x14], eax
0x55555555189 <f1+0047> mov eax, 0x1
0x5555555518e <f1+004c> leave
0x5555555518f <f1+004d> ret

14     unsigned long f1_var1 = 0x123456789;
15
16     unsigned long f1_var2 = 0x0abcdef;
17
18     // rv=0x0
→ 19     int rv = f2(5);
20
21     return 1;
22 }
23
24

[#0] Id 1, Name: "func_calling", stopped 0x5555555517c in f1 (), reason: SINGLE STEP
[#0] 0x5555555517c → f1(a=0x11111111, b=0x22222222, c=0x33333333, d=0x44444444, e=0x55555555, f=0x66666666, g=0x77777777)
[#1] 0x55555555201 → main(argc=0x1, argv=0x7fffffffef88)

gef> x/x $rbp-0x10
0x7fffffffdf30: 0x23456789
gef> x/x $rbp-0x8
0x7fffffffdf28: 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: 0x00005555555517e → <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:
0x00005555555513e <+0>:      push    rbp
0x00005555555513f <+1>:      mov     rbp, rsp
0x000055555555142 <+4>:      sub     rsp, 0x50
0x000055555555146 <+8>:      mov     QWORD PTR [rbp-0x28], rdi
0x00005555555514a <+12>:     mov     QWORD PTR [rbp-0x30], rsi
0x00005555555514e <+16>:     mov     QWORD PTR [rbp-0x38], rdx
0x000055555555152 <+20>:     mov     QWORD PTR [rbp-0x40], rcx
0x000055555555156 <+24>:     mov     QWORD PTR [rbp-0x48], r8
0x00005555555515a <+28>:     mov     QWORD PTR [rbp-0x50], r9
0x00005555555515e <+32>:     movabs  rax, 0x123456789
0x000055555555168 <+42>:     mov     QWORD PTR [rbp-0x10], rax
0x00005555555516c <+46>:     mov     QWORD PTR [rbp-0x8], 0xabcdef
0x000055555555174 <+54>:     mov     edi, 0x5
0x000055555555179 <+59>:     call   0x55555555129 <f2>
0x00005555555517e <+64>:     mov     DWORD PTR [rbp-0x14], eax
0x000055555555181 <+67>:     mov     eax, 0x1
0x000055555555186 <+72>:     leave
0x000055555555187 <+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 : 0x0000333333335210 → <__libc_csu_init+0000> endbr64
$rcx : 0x44444444
$rdx : 0x33333333
$rsp : 0x00007fffffffdee0 → 0x00007fffffffdf40 → 0x00007fffffffdf90 → 0x0000000000000000
$rbp : 0x00007fffffffdee0 → 0x00007fffffffdf40 → 0x00007fffffffdf90 → 0x0000000000000000
$rsi : 0x22222222
$rdi : 0x5
$rip : 0x000055555555139 → <f2+0010> mov eax, DWORD PTR [rbp-0x4]
$r8 : 0x55555555
$r9 : 0x66666666
$r10 : 0x0
$r11 : 0x00007ffff7f758f0 → 0x0000800003400468
$r12 : 0x000055555555040 → <_start+0000> endbr64
$r13 : 0x00007fffffe080 → 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: 0x00005555555517e → <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

0x55555555130 <f2+0007>      mov     eax, DWORD PTR [rbp-0x14]
0x55555555133 <f2+000a>      add     eax, 0x1
0x55555555136 <f2+000d>      mov     DWORD PTR [rbp-0x4], eax
→ 0x55555555139 <f2+0010>      mov     eax, DWORD PTR [rbp-0x4]
0x5555555513c <f2+0013>      pop     rbp
0x5555555513d <f2+0014>      ret
0x5555555513e <f1+0000>      push    rbp
0x5555555513f <f1+0001>      mov     rbp, rsp
0x55555555142 <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

```
//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 two 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 → 0x00000000
$esi : 0xf7fb3000 → 0x001ead6c
$edi : 0xf7fb3000 → 0x001ead6c
$eip : 0x565561b0 → <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: 0x0000000e
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

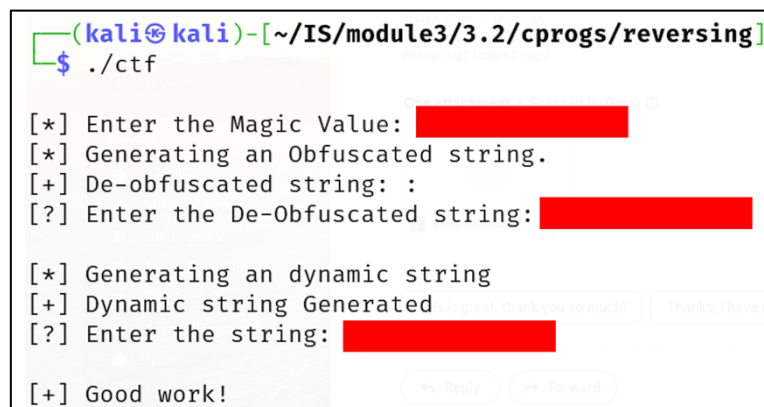
```

- 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:

This is a reverse engineering task, where you are required to download a compiled binary from Google Classroom. This is quite a simple task as the binary is compiled with `-ggdb` option and include all the symbols

You need to load the binary inside `gdb`, to understand the flow by seeing its disassembly. A sample run of the binary is shown in the screenshot, which requires you to give three input values, and on entering all three correctly, it will display the message: “[+] Good work!”



```
(kali㉿kali)-[~/IS/module3/3.2/cprogs/reversing]
$ ./ctf

[*] Enter the Magic Value: 
[*] Generating an Obfuscated string.
[+] De-obfuscated string: :
[?] Enter the De-Obfuscated string: 

[*] Generating an dynamic string
[+] Dynamic string Generated
[?] Enter the string: 

[+] Good work!
```

A brief description of the three tasks is given below:

- **Task 1:** The first part of the program involves determining a hardcoded string in the main function. This is the simplest task.
- **Task 2:** The second task involves determining a string that is XORed with a key. Your goal is to figure out how the program obfuscates and later de-obfuscates this string. **Hint:** Set a breakpoint at the function responsible for handling the obfuscated string. Step through the function, paying close attention to the XOR operation. Identify the key used and reverse the XOR to get the original string. Inspect the function's arguments and local variables to analyze how the obfuscated string is processed.
- **Task 3:** The third task involves determining a dynamically generated string where the case of each character is flipped. **Hint:** Set a breakpoint inside the function that generates the dynamic string. Step through the code to see how the case of each character is flipped (likely using XOR with 0x20). Inspect the program's memory and registers to view the generated string before and after case-flipping.

Note: Dear students, in real life you will seldom get a binary like the one I have shared. Developers use utilities like `strip` and `objcopy` to remove symbols, debugging information and even removing specific sections from executables. If I have done that, this task would have become more challenging.

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 Butt."

```
#include <stdio.h>
#include <stdlib.h>
void f3(){
    return;
}
void f2(){
    f3();
}
void f1(){
    f2();
}
int main(){
    f1();
    return 0;
}
int virus(){
    printf("Let us Hack Planet Earth with Arif Butt.\n");
    exit(0);
}
```

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.