Electrical Engineering & Computing Sciences
Computing Security

CSEC 202 Reverse Engineering Fundamentals

Spring 2024| Sections: 600, 601, and 602

MIDTERM

Student Name				**Duration**	**60 minutes**
Student ID		**Section**		**Exam Date**	**16/03/2024**

Questions	CLO	Marks	Score
Q1	CLO 1, 2	5	
Q2	CLO 1, 2	5	
Q3	CLO 1, 2	5	
Q4	CLO 1, 2	5	
Total		**20**	

Instructions

1. Answer all questions.
2. No notes or textbooks are permitted in the examination hall.
3. Use your time carefully. If you feel you are stuck, move on to the next question.
4. Answer questions in the space provided. You may use the back of a page if the space for a question is insufficient.
5. Please hand in the complete questionnaire at the end of the examination.
6. **IMPORTANT:** All cell phones, electronic devices, smartwatches, and earbuds are prohibited. They must be switched off and stored away.
7. Any cheating will result in an F and may lead to disciplinary action.

All the best!

Instructor: Emad AbuKhousa (eakcad@rit.edu)

Q1: Select the correct Answer [10 questions, 5 points]

1) What is the primary purpose of hashing in malware analysis?

A) To compress the malware file size
B) To uniquely identify malware
C) To encrypt sensitive data within the malware
D) To increase the execution speed of malware

2) What is the result of executing mov eax, ebx on a 64-bit CPU?

A) It results in the upper 32 bits of RAX being preserved as they were before the operation.
B) It automatically zeroes the upper 32 bits of RAX.
C) It generates an 8-bit result and leaves the upper 56 bits of RAX intact.
D) It only affects the lower 16 bits of RAX and does not change the upper 48 bits.

3) Why do malware authors use packing and obfuscation techniques?

A) To reduce the file size of the malware
B) To make reverse engineering more difficult
C) To speed up the execution of malware
D) To improve the malware's compatibility with different systems

4) What is the difference between static and dynamic linking in the context of malware?

A) Static linking makes the executable larger, while dynamic linking does not.
B) Dynamic linking encrypts the malware, while static linking does not.
C) Static linking speeds up malware execution, while dynamic linking slows it down.
D) Dynamic linking requires internet access, while static linking does not.

5) In x86 assembly, one important role of the EAX register is:

A) To store the base address of the stack
B) To store the return value of a function
C) To control CPU decisions
D) To point to the current top of the stack

6) RISC architecture differs from CISC architecture by having:

A) Fewer instructions with more cycles per instruction
B) More instructions with simpler operations
C) More complex instructions aimed at high performance
D) No difference, they are fundamentally the same

7) In x86-64 assembly, which register is not used to pass the first six arguments to a function in Linux?

A) RDI B) RSI C) RAX D) RDX

8) Which of the following is NOT a direct application of performing advanced static analysis on malware?

A) Developing new malware
B) Developing signatures for antivirus software
C) Understanding malware to develop new defense techniques
D) Identifying the malware's behavior and weaknesses

9) Which of the following instructions definitely turns ON the zero flag?

A) mov eax, 1
B) xor eax, eax
C) add eax, ebx
D) lea eax, [eax]

10) What will be the hexadecimal values of DX & AX, after the following instructions execute?

```
mov eax, 0x20241603
mov ecx, 0x10000
mul ecx
```

A) DX = 0x0000, AX = 0x1603
B) DX = 0x2024, AX = 0x1603
C) DX = 0x2024, AX = 0x0000
D) DX = 0x1603, AX = 0x0000

Q2: Stack Operation [5 points]

2-1 Stack PUSH- POP [1 point]

Given the initial state of the registers and the sequence of assembly instructions provided below, determine the final values of the registers after executing the entire sequence of instructions:

Initial Registers State	
Name	Value
EAX	0xBEEF
EBX	0xCAFE
ECX	0xFEED
EDX	0xFACE

Instruction Sequence:

1. push eax
2. push ecx
3. push ebx
4. push edx
5. pop eax
6. pop ebx
7. pop ecx
8. pop edx

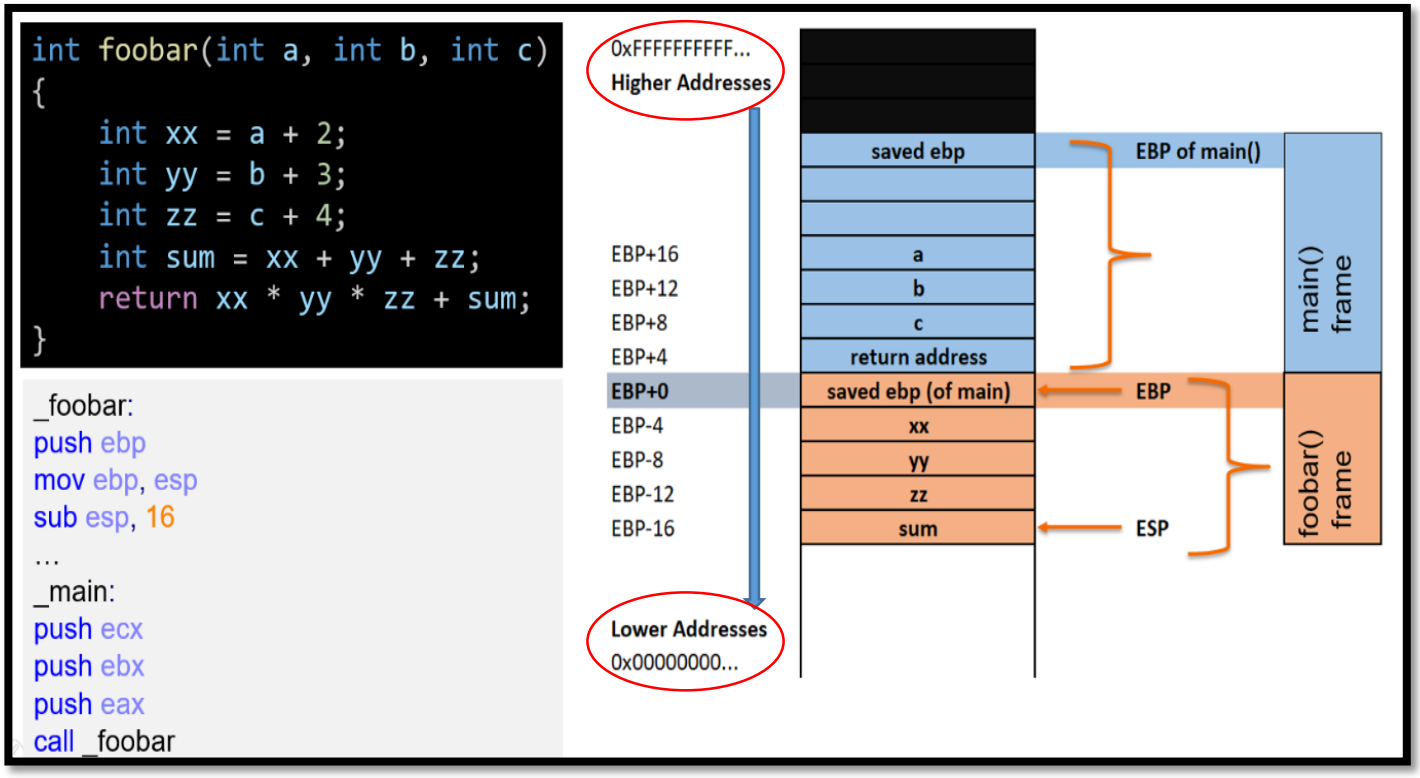
Your Answer	
Name	Value
EAX	
EBX	
ECX	
EDX	

2-2 Stack and function calls [1 point]

Consider the following function signature and its corresponding stack frame just after the function foobar has been called with arguments:

int foobar(int a, int b, int c)

The stack frame layout is depicted as follows, with the stack growing from higher to lower memory addresses:



a. Identify the Mistake: Based on the calling convention for x86 architecture, identify the error in the stack frame layout above in relation to the order of the function arguments on the stack.

.....

.....

b. Correct the Stack Frame: Provide the corrected order of the function arguments on the stack as they should be just after the call to foobar.

.....

.....

2-3 Machine State [3 points]:

This question is based on a hypothetical machine state for an Intel x86-32 architecture system. The state of the machine is described by the contents of its registers and stack memory at a specific point in time. Your task is to understand this state, perform certain operations based on it, and answer related questions.

Registers	
Name	Value
ESP	**0x100**
EAX	0x100
EDX	0x2
EBP	0x10C
ECX	0x0

Stack Memory	
Address	Value
0x100	**0xABCD**
0x104	0xBEEB
0x108	0x3333
0x10C	0x0808
0x110	0xBEEF

a) Given that EAX holds the value 0x100, what would be the result of executing LEA EBX, [EAX+EDX*2] versus MOV EBX, [EAX+EDX*2], considering EDX holds the value 0x2?

Instruction	**LEA EBX, [EAX+EDX*2]**	**MOV EBX, [EAX+EDX*2]**
EBX	.....	.....

b) Given the current register and stack state, analyze the following assembly instructions and determine which flag(s) would be set after their execution:

```
pop ecx
cmp ecx, ABCE
```

i. What is the resulting value in ECX after the first instruction (**pop ecx**)? **ECX =**

ii. Based on the result of the comparison (cmp ecx, ABCE), which flag(s) will be set and why? Consider the Zero Flag (ZF), Sign Flag (SF), Overflow Flag (OF), and Carry Flag (CF) in your explanation.

.....

.....

.....

.....

{Optional: Provide your answer with reasoning for partial credit even if the final conclusion about the flags might not be accurate.}

C)

i. If you execute **mov eax, [ebp-8]**, what value is loaded into EAX? **EAX =**

ii. Similarly, what value is loaded into EDX with **mov edx, [ebp+4]?** **EDX =**

Q3 Basic of Assembly [5 Points]:

3-1 Code Tracing [2 points]

Trace the following code. What is the value of eax in <u>decimal</u> at the end?

Initial values: **edi** = 100 (in **<u>decimal</u>**), **esp** = 0x7fff0abc (in hexadecimal)

```
0040540 <leaf>:
        lea edi, [edi+15]
        ret

0040545 <top>:
        sub edi, 5
        call 0040540 <leaf>
        add edi, edi
        ret

0040550 <main>:
; ...
        call 0040545 <top>
        mov eax, edi
; ...
```

eax =

{*Optional: Please show your work for the possibility of earning partial credit*}

3-2 Reading and Displaying User Input in Assembly [3 points]

Develop an assembly program using the x86-32 architecture that engages with the user by requesting them to input an integer number. Upon receiving the input, the program is designed to display the entered number back to the user.

You have the freedom to use native **NASM** assembly instructions **OR** call C library functions like **printf** and **scanf** for input and output operations.

Requirements:

- Input Prompt: The program should display a clear prompt to the user, asking them to enter a number. For instance, "**Please enter an Integer number:** ".
- Reading Input: Capture the user's input effectively. You may choose to use scanf if opting for C library functions or appropriate NASM instructions for direct system calls.
- Output: Display the entered number with a message, such as "**You entered: X**", where X is the number input by the user.
- Exit Gracefully: Ensure your program terminates properly after displaying the entered number.

Your Assembly code:	
1/2	2/2

Q4-Advanced Static Analysis [5 points]:

4-1 Reverse Engineering Assembly Code to High-Level Language (C Pseudocode) [2 points]

Carefully analyze the provided assembly code snippets. Your goal is to understand the functionality of each instruction and then accurately translate the assembly code into its equivalent high-level language representation, specifically in the form of C pseudocode. The assembly code performs a series of arithmetic operations and concludes with a system call. Pay special attention to how data is manipulated and stored.

```
section .text
global _start

_start:
   mov dword [ebp-4], 5
   mov dword [ebp-8], 10
   mov dword [ebp-12], 15

   mov eax, [ebp-4]
   add eax, [ebp-8]
   add eax, [ebp-12]
   mov [ebp-16], eax

   mov eax, [ebp-4]
   sub eax, [ebp-8]
   sub eax, [ebp-12]
   mov [ebp-20], eax

   mov eax, [ebp-4]
   imul eax, [ebp-8]
   imul eax, [ebp-12]
   mov [ebp-24], eax

   mov eax, 1
   xor ebx, ebx
   int 0x80
```

Your C pseudocode:

4-2: Reverse Engineering Assembly to C [3 points]:

You are provided with disassembled code from an executable. Your task is to reverse engineer this assembly code back into its equivalent C code. The goal is to create a fully functioning C program that mirrors the functionality of the given assembly instructions.

			Your C code:
8049166 <main>:			
1	8049166:	push ebp	
2	8049167:	mov ebp, esp	
3	804916a:	sub esp, 0x8	
4	8049178:	mov DWORD PTR [ebp-0x8], 0x0	
5	804917f:	mov DWORD PTR [ebp-0xc], 0x1	
6	804918d:	jmp **80491a4**	
7	**804918f:**	push DWORD PTR [ebp-0xc]	
8	8049192:	push DWORD PTR [ebp-0x8]	
9	8049195:	call **80491c6 <myfunc>**	
10	804919a:	add esp, 0x8	
11	804919d:	mov DWORD PTR [ebp-0x8], eax	
12	80491a0:	add DWORD PTR [ebp-0xc], 0x1	
13	**80491a4:**	**cmp DWORD PTR [ebp-0xc], 0x5**	
14	80491a8:	jle **804918f**	
15	80491aa:	push DWORD PTR [ebp-0x8]	
16	80491ad:	lea eax, **format** ; "The result: %d\n"	
17	80491b3:	push eax	
18	80491b4:	call 8049040 <printf@plt>	
19	80491b9:	add esp, 0x8	
20	80491bc:	mov eax, 0x0	
21	80491c1:	mov ebx, DWORD PTR [ebp-0x4]	
22	80491c4:	leave	
23	80491c5:	ret	
080491c6 <myfunc>:			
24	80491c6:	push ebp	
25	80491c7:	mov ebp, esp	
26	80491d3:	mov edx, DWORD PTR [ebp+0x8]	
27	80491d6:	mov eax, DWORD PTR [ebp+0xc]	
28	80491d9:	add eax, edx	
29	80491db:	pop ebp	
30	80491dc:	ret	