Electrical Engineering & Computing Sciences
Computing Security

CSEC 202 Reverse Engineering Fundamentals

Spring 2024| Sections: 600, 601, and 602

Final Exam

Student Name				**Duration**	**2 hours**
Student ID		**Section**		**Exam Date**	**12/05/2024**

Questions	CLO	Marks	Score
Q1	CLO 1, 2	5	
Q2	CLO 1, 2	5	
Q3	CLO 1, 2	5	
Q4	CLO 1, 2	6	
Q5	CLO 3	3	
Total		**24**	

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 | emadak@gatech.edu)

Q1 [5 points]: Select the correct Answer

1. **In the context of advanced dynamic analysis, what is the primary reason for disabling Address Space Layout Randomization (ASLR)?**
   A. To simplify the recovery of encrypted files
   B. To ensure consistent memory addresses across sessions for predictable analysis
   C. To increase the processing speed of the analysis tools
   D. To allow the malware to spread to adjacent memory segments

2. **What is the purpose of using a debugger in malware analysis?**
   A. To edit the malware code
   B. To observe the behavior of malware and its interaction with the system during execution
   C. To increase the efficiency of malware
   D. To network with other malware analysts

3. **Which of the following best describes the function of the test eax, eax command in an assembly program?**
   A. It multiplies the value in EAX by itself
   B. It sets flags based on the logical AND operation of EAX with itself
   C. It divides EAX by itself
   D. It subtracts EAX from itself

4. **If an assembly code segment is intended to loop until a counter is zero, which instruction is primarily used to check the loop condition?**

   A. mov
   B. sub
   C. jmp
   D. jnz

5. **In reverse engineering a binary using IDA, locating the string "\cmd.exe" typically involves understanding which section of the binary?**

   A. .text
   B. .data
   C. .bss
   D. .heap

6. **Which of the following is a technique used by malware to evade basic dynamic analysis?**

   A. Using complex algorithms
   B. Encrypting its payload
   C. Identifying analysis environments
   D. Overwriting the hard drive

7. Which of the following is not a component of Host Integrity Monitoring?

A. Process Monitoring
B. Registry Monitoring
C. Password Integrity Checking
D. Network Traffic Monitoring/Analysis

8. Which tool is used for visualizing Process Monitor logs?

A. Cuckoo Sandbox
B. Process Hacker
C. ProcDOT
D. Wireshark

9. Analyze the assembly code snippet below from the BlueCAT malware, which includes a call to the Windows API function Sleep. Determine how long the malware will sleep based on the given code and data:

```
.text:00402A2C    mov     eax, 0xE1000
.text:00402A31    mov     ecx, 0x100
.text:00402A36    div     ecx
.text:00402A38    push    eax        ; dwMilliseconds
.text:00402A39    call    ds:Sleep
.text:00402A3F    jmp     short loc_402A08
```

A. 9216 milliseconds
B. 3600 milliseconds
C. 256 milliseconds
D. The BlueCAT doesn't sleep

10. What does the command "x/20x $esp" in GDB do?

A. Executes the program 20 times.
B. Exits the debugger.
C. Examines 20 words from the stack pointer address.
D. Multiplies the stack pointer by 20

Q2 [5 points]: Assembly and X86 Architecture

2.1 [1.5 point] Match the x86 architecture registers with their primary uses:

Registers:	Uses:
[1] EAX	A. Points to the last item on a stack.
[2] ECX	B. Used to reference arguments and local variables.
[3] EBP	C. Used by memory transfer instructions.
[4] ESP	D. Used for addition, multiplication, and return values.
[5] ESI/EDI	E. Points to the next instruction to be executed.
[6] EIP	F. Used as a counter.

Provide the correct letter matching the primary use of each register from the options given.

Answers:

[1] ______

[2] ______

[3] ______

[4] ______

[5] ______

[6] ______

2.2 [1 point] Analyze the representation of values within the EAX register as shown in the accompanying diagram. Given that the EAX register holds the value 1100 1010 0010 0110 1110 1110 0001 0110, determine the hexadecimal values for EAX, AX, AH, and AL. Start your answers with "0x" to indicate hexadecimal notation.

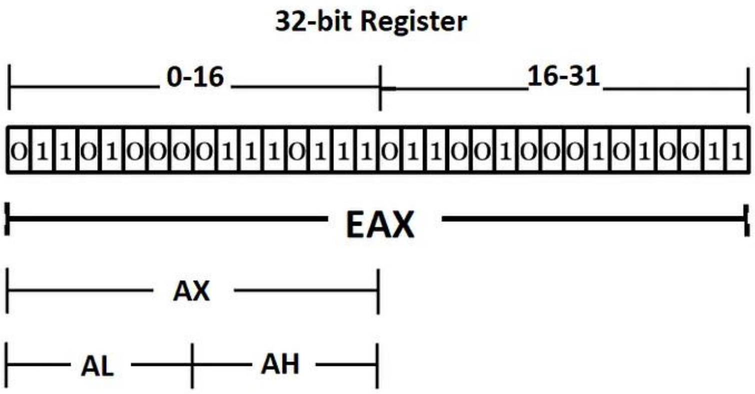
Provide your answers in the following format:

[1] EAX = 0x……………………………

[2] AX = 0x……………………………

[3] AH = 0x……………………………

[4] AL = 0x……………………………



2.3 [1 point] Analyze the CPU flags after executing the following assembly instructions. Indicate which flags are set as a result of the final instruction. [1 point]

```
mov eax,20h
mov ebx,40h
cmp eax,ebx
test eax,ebx
```

□ Carry
□ Overflow
□ Sign
□ Zero
□ Parity
□ Auxiliary
□ Trap
□ Direction

2.4 [1.5 points] The partial contents of a stack are shown below before the execution of the program segment listed. Determine the contents of the stack after the program has been executed and indicate the new top of the stack

	Stack Before		
	Low Byte	High Byte	
ESP →	E4	11	**Low Addr**
	7E	0	
			High Addr

Stack After		
Low Byte	High Byte	
		Low Addr
		High Addtr

Assembly Instructions:

```
POP BX
MOV AH, BH
ADD AH, BL
MOV BH, AH
PUSH BX
```

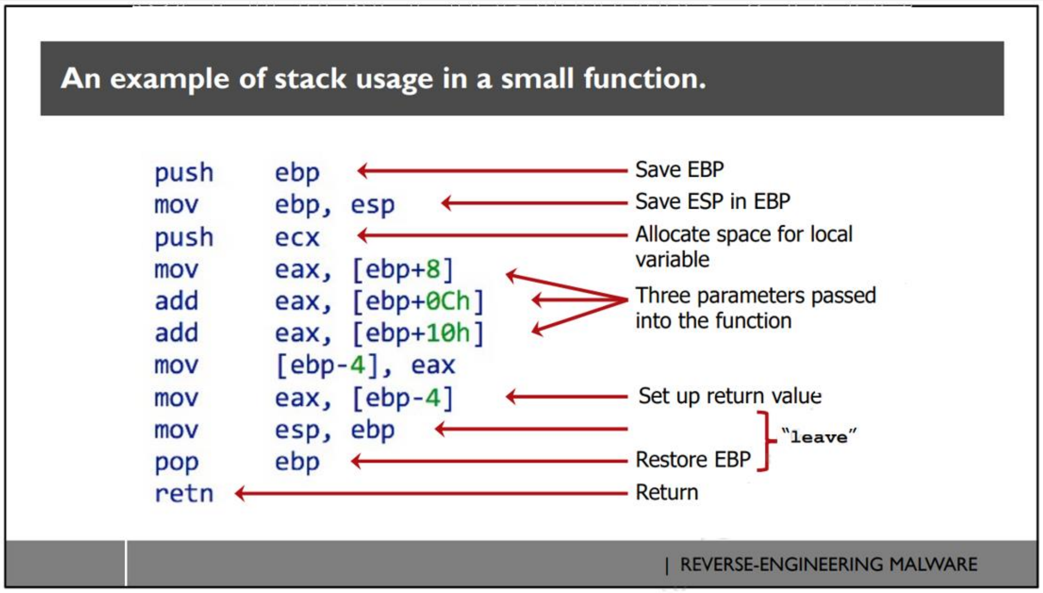
Tasks:

A. Fill in the provided "Stack After" template to illustrate the stack contents after the execution of the program. **[1 point]**

B. Indicate the position of ESP after the program segment has been executed. (Hint: Use an arrow to point to the correct row of the "Stack After" diagram)**[0.5 point]**

Q3 [5 points] Training Center and Dr.Evil Malware Bomb

Q3.1 [2 points]: You paid approximately $8,000 to study the "Reverse-Engineering Malware" course at one of the world's most prestigious cybersecurity training centers. During a session, the instructor presented a slide with the disassembled Myfunction() code. Examine the assembly code and its annotations on the slide. Complete the following tasks:



An example of stack usage in a small function.

```
push    ebp                 <- Save EBP
mov     ebp, esp            <- Save ESP in EBP
push    ecx                 <- Allocate space for local variable
mov     eax, [ebp+8]        <- Three parameters passed into the function
add     eax, [ebp+0Ch]
add     eax, [ebp+10h]
mov     [ebp-4], eax
mov     eax, [ebp-4]        <- Set up return value
mov     esp, ebp            <-                  } "leave"
pop     ebp                 <- Restore EBP      }
retn                        <- Return
```

| REVERSE-ENGINEERING MALWARE

1. **Identify the Mistake:** There appears to be a mistake in one of the comments during the function prologue. Identify which comment is incorrect and explain why. **[0.5 point].**

2. What is the instruction that accurately matches the comment identified as incorrect in part 1 above? **[0.5 point].**

3. Based on the assembly instructions provided, write the equivalent C code of **MyFunction()** that performs the same operations as described in the assembly **[1 points].**

Your C pseudocode:

Q3.2 [3 points] Dr.Evil is back with a 64-bits E-Bomb: You are a cybersecurity analyst working to defuse a potentially dangerous e-bomb malware. Part of your task involves understanding and neutralizing specific phases of the malware's operation. The provided assembly code represents one such critical phase where the malware checks certain conditions to decide whether to detonate.

Your job is to reverse engineer the provided assembly segment into a C function named **phase_1**. This function should integrate with existing C code to simulate the original assembly logic safely. If the conditions in the provided assembly are not met, the bomb will "explode," which should be simulated by calling a function named explode_bomb().

Assembly code of phase_1 as shown when you debug the program:

```
pwndbg> disass phase_1
Dump of assembler code for function phase_1:

0x400efc <+0>:    push   rbp
0x400efd <+1>:    push   rbx
0x400efe <+2>:    sub    rsp,0x28
0x400f02 <+6>:    mov    rsi,rsp
0x400f05 <+9>:    call   0x40145c <read_six_numbers>
0x400f0a <+14>:   cmp    DWORD PTR [rsp],0x1
0x400f0e <+18>:   je     0x400f30 <phase_1+52>
0x400f10 <+20>:   call   0x40143a <explode_bomb>
0x400f15 <+25>:   jmp    0x400f30 <phase_1+52>
0x400f17 <+27>:   mov    eax,DWORD PTR [rbx-0x4]
0x400f1a <+30>:   add    eax,eax
0x400f1c <+32>:   cmp    DWORD PTR [rbx],eax
0x400f1e <+34>:   je     0x400f25 < phase_1+41>
0x400f20 <+36>:   call   0x40143a <explode_bomb>
0x400f25 <+41>:   add    rbx,0x4
0x400f29 <+45>:   cmp    rbx,rbp
0x400f2c <+48>:   jne    0x400f17 <phase_1+27>
0x400f2e <+50>:   jmp    0x400f3c <phase_1+64>
0x400f30 <+52>:   lea    rbx,[rsp+0x4]
0x400f35 <+57>:   lea    rbp,[rsp+0x18]
0x400f3a <+62>:   jmp    0x400f17 <phase_1+27>
0x400f3c <+64>:   add    rsp,0x28
0x400f40 <+68>:   pop    rbx
0x400f41 <+69>:   pop    rbp
0x400f42 <+70>:   ret
```

Your C code:

```
void explode_bomb() {
  printf("Bomb exploded!\n"); exit(1);}
void read_six_numbers(int *nums) {
    for (int i = 0; i < 6; i++) {
        scanf("%d", &nums[i]);}
void phase_1() {
     // Your implementation goes here
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
   ..........................................................................
    .......................................................................}
int main() {
phase_1(); return 0;
}
```

Q4 [6 points]: Assembly Reverse Engineering

Note: You may need to scroll to view the full blocks of code.

Consider the assembly representations of two functions, **func1** and **func2,** which are integral parts of a fully functioning program. These functions have been compiled using the **-m32** flag on a 64-bit machine, resulting in specific stack alignments visible in the debugger. Examine these assembly codes and use them to answer the following questions:

```
pwndbg> disass func1
Dump of assembler code for function func1:

   0x565561c4 <+0>:     push   ebp
   0x565561c5 <+1>:     mov    ebp,esp
   0x565561c7 <+3>:     push   ebx
   0x565561c8 <+4>:     sub    esp,0x14
   0x565561cb <+7>:     call   0x56556090 <__x86.get_pc_thunk.bx>
   0x565561d0 <+12>:    add    ebx,0x2e24
   0x565561d6 <+18>:    mov    DWORD PTR [ebp-0xc],0x4
   0x565561dd <+25>:    mov    DWORD PTR [ebp-0x10],0x0
   0x565561e4 <+32>:    jmp    0x5655621c <func1+88>
   0x565561e6 <+34>:    mov    eax,DWORD PTR [ebp-0x10]
   0x565561e9 <+37>:    add    eax,0x1
   0x565561ec <+40>:    lea    edx,[eax*4+0x0]
   0x565561f3 <+47>:    mov    eax,DWORD PTR [ebp+0x8]
   0x565561f6 <+50>:    add    eax,edx
   0x565561f8 <+52>:    mov    edx,DWORD PTR [eax]
   0x565561fa <+54>:    mov    eax,DWORD PTR [ebp-0x10]
   0x565561fd <+57>:    lea    ecx,[eax*4+0x0]
   0x56556204 <+64>:    mov    eax,DWORD PTR [ebp+0x8]
   0x56556207 <+67>:    add    eax,ecx
   0x56556209 <+69>:    mov    eax,DWORD PTR [eax]
   0x5655620b <+71>:    push   edx
   0x5655620c <+72>:    push   eax
   0x5655620d <+73>:    call   0x5655618d <func2>
   0x56556212 <+78>:    add    esp,0x8
   0x56556215 <+81>:    add    DWORD PTR [ebp-0xc],eax
   0x56556218 <+84>:    add    DWORD PTR [ebp-0x10],0x2
   0x5655621c <+88>:    mov    eax,DWORD PTR [ebp+0xc]
   0x5655621f <+91>:    sub    eax,0x1
   0x56556222 <+94>:    cmp    DWORD PTR [ebp-0x10],eax
   0x56556225 <+97>:    jl     0x565561e6 <func1+34>
   0x56556227 <+99>:    sub    esp,0x8
   0x5655622a <+102>:   push   DWORD PTR [ebp-0xc]
   0x5655622d <+105>:   lea    eax,[ebx-0x1fec]
   0x56556233 <+111>:   push   eax
   0x56556234 <+112>:   call   0x56556040 <printf@plt>
   0x56556239 <+117>:   add    esp,0x10
   0x5655623c <+120>:   nop
   0x5655623d <+121>:   mov    ebx,DWORD PTR [ebp-0x4]
   0x56556240 <+124>:   leave
   0x56556241 <+125>:   ret

pwndbg> break func2
pwndbg> continue
pwndbg> disass func2
Dump of assembler code for function func2:

   0x5655618d <+0>:     push   ebp
   0x5655618e <+1>:     mov    ebp,esp
   0x56556190 <+3>:     sub    esp,0x10
   0x56556193 <+6>:     call   0x565562ad <__x86.get_pc_thunk.ax>
   0x56556198 <+11>:    add    eax,0x2e5c
=> 0x5655619d <+16>:    mov    edx,DWORD PTR [ebp+0x8]
   0x565561a0 <+19>:    mov    eax,edx
   0x565561a2 <+21>:    add    eax,eax
   0x565561a4 <+23>:    add    eax,edx
   0x565561a6 <+25>:    add    eax,eax
   0x565561a8 <+27>:    lea    edx,[eax+0x3]
   0x565561ab <+30>:    nop
   0x565561ad <+32>:    nop
   0x565561b0 <+35>:    sar    eax,0x2
   0x565561b3 <+38>:    mov    DWORD PTR [ebp-0x4],eax
   0x565561b6 <+41>:    mov    eax,DWORD PTR [ebp+0xc]
   0x565561b9 <+44>:    mov    edx,DWORD PTR [ebp-0x4]
   0x565561bc <+47>:    mov    ecx,eax
   0x565561be <+49>:    shl    edx,cl
   0x565561c0 <+51>:    mov    eax,edx
   0x565561c2 <+53>:    leave
   0x565561c3 <+54>:    ret
```

4.A [2.75] Fill in the C code below so that it aligns with the above x86-32 assembly. Ensure your C code fits into the provided blanks without adding additional lines or circumventing the format (this may require slight adjustments to the syntax or style of your initial decoding guess to ensure it fits properly). All constants of type signed or unsigned int must be written in decimal.

```
#include <stdio.h>

int func2(int x, int y) {

   int z = ____________;

   return ____________ << ____________;

}


void func1(int *arr, int count) {

   int z = ____________;

   for (int i = ____________; i < ____________; i += ____________) {

     ____________ += func2(____________, ____________);

   }

   printf("%d\n", ____________);

}

int main() {
    // Example array and count
    int array[] = {1, 2, 3, 4, 5, 6};
    int count = sizeof(array) / sizeof(array[0]);  // Calculate the
number of elements in the array

    // Call func1 with the array and its size
    func1(array, count);

    return 0;
}
```

B) [1-point] Explain the purpose of the following lines of assembly code in func2?

```
0x5655618d <+0>:      push   ebp
```

....

....

```
0x5655618e <+1>:      mov    ebp,esp
```

....

....

```
0x56556190 <+3>:      sub    esp,0x10
```

....

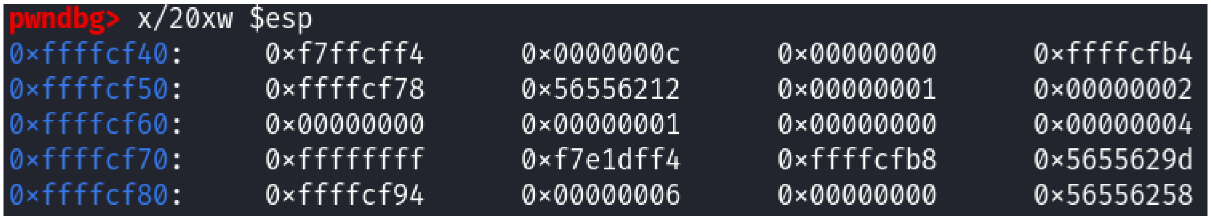
....

C) [1.25 points] Stack Frame Content Analysis:

The following is the stack frame content for func2 at the point where execution breaks at this line:

```
=> 0x5655619d <+16>:    mov    edx,DWORD PTR [ebp+0x8]
```

Here is the output from pwndbg showing the stack values at that moment:



```
pwndbg> x/20xw $esp
0xffffcf40:     0xf7ffcff4      0x0000000c      0x00000000      0xffffcfb4
0xffffcf50:     0xffffcf78      0x56556212      0x00000001      0x00000002
0xffffcf60:     0x00000000      0x00000001      0x00000000      0x00000004
0xffffcf70:     0xffffffff      0xf7e1dff4      0xffffcfb8      0x5655629d
0xffffcf80:     0xffffcf94      0x00000006      0x00000000      0x56556258
```

Using this information, determine the values for the following registers and stack entries:

C1	What is the value of ESP?	
C2	What is the value of EIP?	
C3	What is the return address or the saved EIP?	
C4	What is the value of the current frame EBP?	
C5	What is the value of the previous frame EBP or the pushed EBP of func1?	

Q5 [3 Points]: Login to an ATM Banking System

After your notable success with the GreenCAT and your ingenious handling of the electronic bomb defusing scenario, your expertise is once again in demand. The developer of the **atm.c** program, recognizing your adept problem-solving skills, has urgently requested your assistance with their banking system.

The author of the **atm.c** program has implemented login functionality in their banking system. However, there have been cases where, even without knowing the hard secret password **MyPwd@2024**, users are able to log in. They've narrowed down the issue to a single function, **logMeIn()**, for which they have provided you with the source code. You can run the code on your local computer, but you don't have access to the read_real_password function, so you make that function up yourself for your testing.

Below, the source code for the **LogMeIn()** function from the **atm.c** program is presented, along with the corresponding disassembled code of the binary you created to analyze the system, enabling a thorough investigation into the suspected security flaw:

login.c

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

void read_real_password(char *buf) {
   strcpy(buf, "MyPwd@2024");
}

bool logMeIn() {
   int allow_login = 0;
   char buffer[12];
   char real_password[12];
   read_real_password(real_password);
   printf("What is your password? ");
   gets(buffer);
   if (strncmp(buffer,real_password, 12) == 0)
           allow_login = 1;
   return allow_login;
}

int main() {
   bool is_logged_in = logMeIn();
   if (is_logged_in) {
      printf("Welcome\n");
   } else {
      printf("Rejected\n");
   }
   return 0;
}
```

logMeIn () assembly

```
<gdb> disass logMeIn

   push   rbp
   mov    rbp,rsp
   sub    rsp,0x20
   mov    DWORD PTR [rbp-0x4],0x0
   lea    rax,[rbp-0x1c]
   mov    rdi,rax
   call   0x1169 <read_real_password>
   lea    rax,[rip+0xe56]
   mov    rdi,rax
   mov    eax,0x0
   call   0x1050 <printf@plt>
   lea    rax,[rbp-0x10]
   mov    rdi,rax
   mov    eax,0x0
   call   0x1060 <gets@plt>
   lea    rcx,[rbp-0x1c]
   lea    rax,[rbp-0x10]
   mov    edx,0xc
   mov    rsi,rcx
   mov    rdi,rax
   call   0x1030 <strncmp@plt>
   test   eax,eax
   jne    0x11ef <login+99>
   mov    DWORD PTR [rbp-0x4],0x1
   cmp    DWORD PTR [rbp-0x4],0x0
   setne  al
   leave
   ret
```

The first action taken in response to this incident involved conducting preliminary tests together with the bank team, which yielded some troubling findings:

```
┌──(kali㉿kali)-[~/Desktop/Final]
└─$ ./login
What is your password? MyPwd@2024
Welcome
┌──(kali㉿kali)-[~/Desktop/Final]
└─$ ./login
What is your password? IamAHacker
Rejected
┌──(kali㉿kali)-[~/Desktop/Final]
└─$ ./login
What is your password? IamNotAHacker
Welcome
```

The login trials revealed a critical security flaw where incorrect passwords like "**IamNotAHacker**" were mistakenly accepted, granting unauthorized access. While valid passwords were correctly authenticated and invalid ones rejected, the system's vulnerability lies in improper handling or validation, necessitating immediate investigation and remediation.

You start investigating with **GDB** debugger, where you uncover the following information about the program state:

```
$ gdb ./login
pwndbg> break 18
Breakpoint 1 at 0x11cc: file login.c, line 18.
pwndbg> run
What is your password? RITDubai
───────────────────────────────[ STACK ]───────────────────────────
00:0000│ rsp 0x7fffffffdd90 ◂— 0x7750794d00000000
01:0008│-018 0x7fffffffdd98 ◂— 0x343230324064 /* 'd@2024' */
02:0010│ rax 0x7fffffffdda0 ◂— 'RITDubai'
03:0018│-008 0x7fffffffdda8 ◂— 0x0
04:0020│ rbp 0x7fffffffddb0 —▸ 0x7fffffffddd0 ◂— 0x1
05:0028│+008 0x7fffffffddb8 —▸ 0x55555555520a (main+18)◂— mov byte ptr [rbp-1], al
06:0030│+010 0x7fffffffddc0 ◂— 0x0
07:0038│+018 0x7fffffffddc8 ◂— 0x0

pwndbg> p &real_password
$1 = (char (*)[12]) 0x7fffffffdd94
pwndbg> p &buffer
$2 = (char (*)[12]) 0x7fffffffdda0
pwndbg> p &allow_login
$3 = (int *) 0x7fffffffddac
pwndbg>  x/s 0x7fffffffdd94
0x7fffffffdd94: "MyPwd@2024"
```

After combining this information with the code, you find the issue! Answer the following questions about this vulnerable program:

5.A [0.5 point] Given the GDB results, is the buffer content capable of overwriting the real_password such that it matches the buffer? In your response, consider how the buffer expands, the arrangement of local variables in the stack memory, and the direction in which the stack grows.

....

....

5.B [0.5 point] Can the buffer content overwrite the allow_login variable? Explain your reasoning.

....

....

5.C [1 point] Write a note to the developer explaining how it is possible to log into the system with the password "`IamNotAHacker` " while attempts with "`IamAHacker`" are unsuccessful. Provide a specific example of an input that could allow unauthorized access and explain why this approach would be successful.

....

....

....

....

....

....

....

....

....

5.D [1 point] Provide the developer with a 1-line code change to logMein() to fix the issue. Explain why your fix resolves the issue.

....

....

....

Tigers, Good Luck!