1. **Single-select questions (10 points):**
2. **During a web application security review, Alice discovered that one of her organization’s applications is vulnerable to SQL injection attacks. Where would be the best place for Alice to address the root cause issue?**

d. Application code

1. **Which of the following script is vulnerable to SQL injection attack?**

a. var userName;

userName = Request.form(“userName”);

var SQL = “select \*from User where userName = ‘”+userName+”’”;

1. **Any user-controlled parameter that gets processed by the application includes vulnerabilities like \_\_\_\_\_\_\_**

d. All of the mentioned

1. **Which of the following statement is true?**

c. The primary form of SQL injection consists of indirect insertion of code

1. **In cross-site scripting where does the malicious script execute?**

c. In the user’s browser

**2. (10 points) What are SQL Injection Attacks?**

**a. List down the steps involved to carry out such attacks.**

Find out how to run the program  
Find a vulnerable section of that program  
Inject malicious code at the proper location and hope it works  
If step 3 did not work as intended repeat until works or alternatively find a different section of the program that has a different vulnerability

**b. The following figure shows a fragment of code that implements the login functionality for a database application. The code dynamically builds an SQL query and submits it to a database.**

**i. Suppose a user submits login, password, and pin as doe, secret, and 123. Show the SQL query that is generated.**

The SQL query generated would look like this:

SELECT accounts FROM users WHERE login=doe AND pass=secret AND pin=123

**ii. Instead, the user submits for the login field the following: “or 1 = 1 - -” What is the effect?**

This is an example of an SQL injection attack, wih this the attacker can gain root access. In tis case if the attacker types in the wrong login then he would still gain access because the 1= 1 parameter is always true.

1. **(10 points) Describe how an attacker injects arbitrary code into running, vulnerable software and how they cause it to run?**

An attacker can inject code into a vulnerable system by simply using an unsecure login method( as above) or any other way he can make the vulnerable program execute while reading his input. Since most modern systems take user input in order to create files, run and delete programs there are plenty of opportunities for a malicious entity to exploit vulnerable software.

**4. (10 points) What types of databases are more vulnerable to SQL injections? What can you do to harden them to this type of attack?**

One of the most vulnerable types of databases to SQL injection attacks are those that have no input validation of any kind.

Most obvious way to prevent these attacks is to add input validation to the system. Unfortunately, that is easier said than done. One way to limit this exploitation is to set bounds on user inputs, in an above example we had an attacker attempt an SQL injection attack by typing in a random login followed by 1=1”, if the bounds for login were set at say 5 chars this would not allow this to happen, but at the same time allowing users over 60 million letter and number combinations for their login.

**5. (10 points) Give 5 functions that are susceptible to buffer overflow in C language. Do some research and explain why.**

The 5 functions to avoid while coding in C is is printf(), sprint(), strcpy(), gets() and malloc(). All of these functions can and are being used safely without causing buffer overflow, but when certain conditions are met, they can cause it.

malloc() can cause a buffer overflow when memory is allocated, but a char pointer gets assigned a value that is not inside the allocated space.

gets() can cause an overflow if the data it reads is too large to fit into an allocated buffer

strcpy() can cause an overflow due to the same issue, if not enough space is allocated inside the area the string is being copied to it will cause an overflow.

sprintf() can cause an overflow for a very similar problem, but this time it is mainly due to user error if the programmer sets the output cap smaller than is being outputted.

printf() can cause an overflow if a string is not properly terminated, this may make the printf() statement print out bytes until it hits a zero byte, unfortunately this can also mean that it may go on indefinitely if there are none in the output being printed.

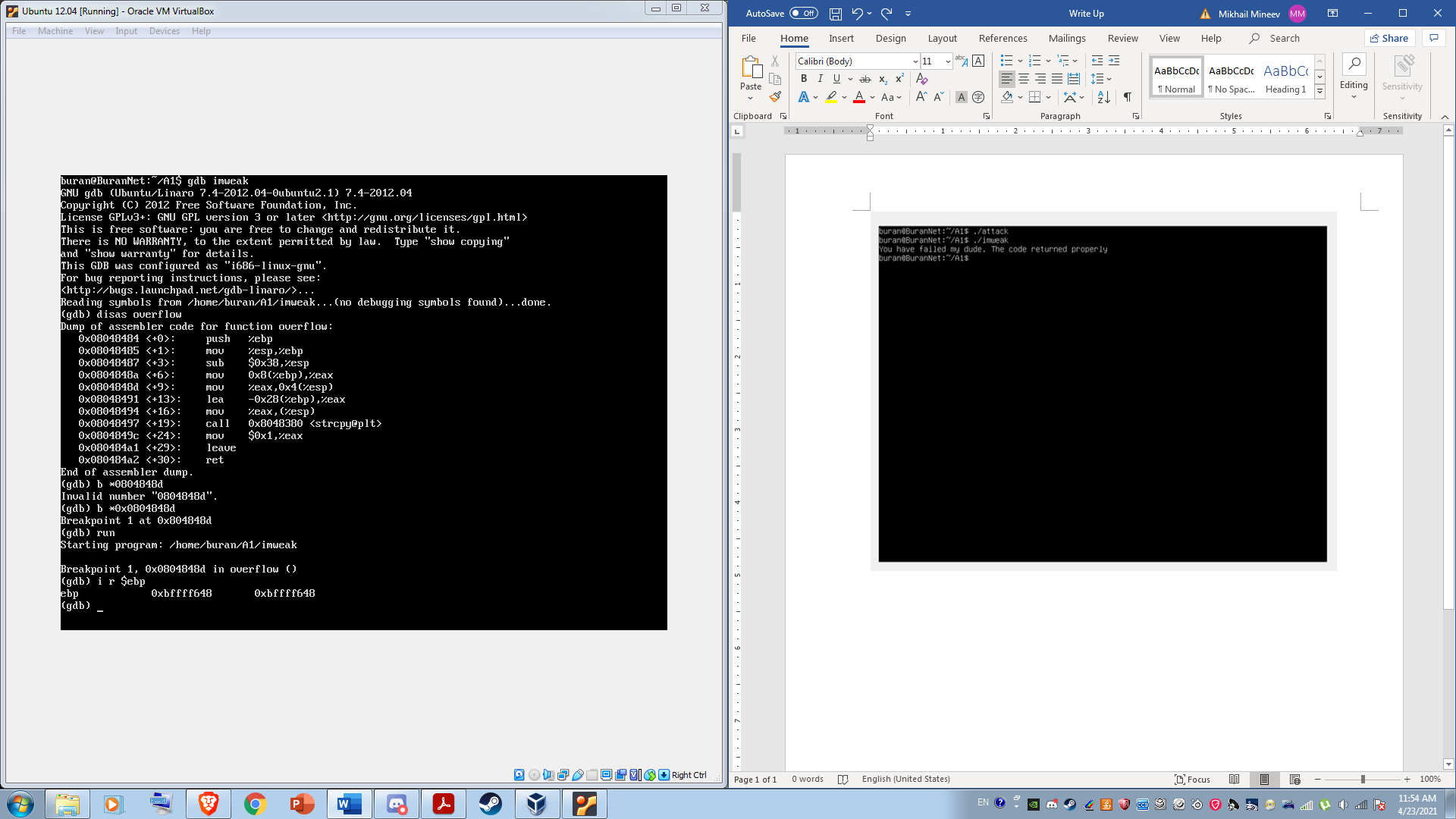
**6. Stack buffer overflow lab  
 Task 1:**

First up before even starting the lab I followed the instructions and made sure to run

**sudo sysctl -w kernel.randomize\_va\_space=0**

Unfortunately, I did not realize that virtual box would reset this value to default after restarting so it did cost me some time figuring that out. But otherwise this task is relatively simple to complete, the hardest part was figuring out the exact address to pass the shell to.

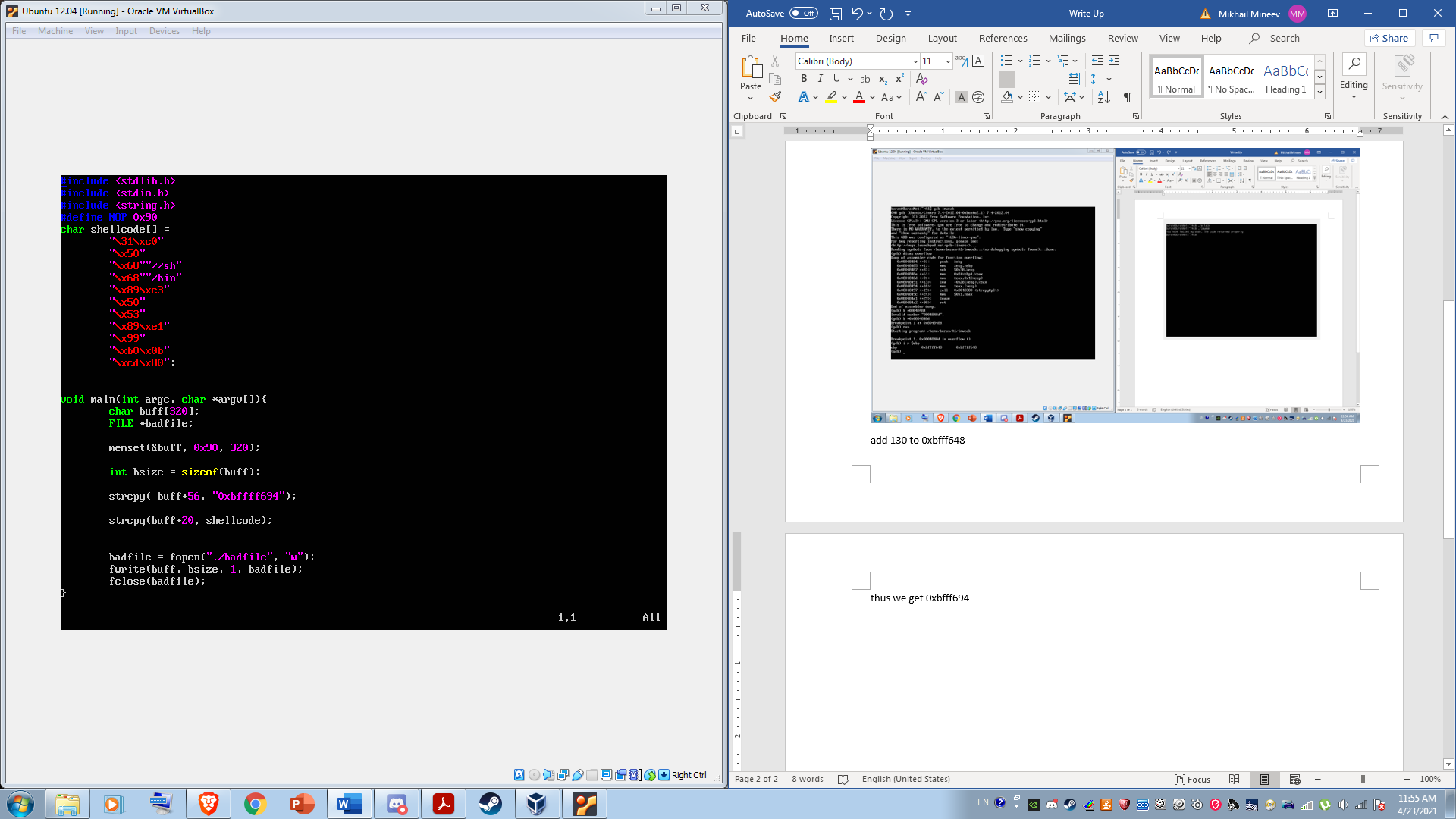
First up I had to figure out the proper address for the beginning of the buffer pointer. To do this I had to run GDB and create a breakpoint somewhere after we have moved the stack pointer, esp, to the base pointer, ebp. Meaning that we are looking at the bytes after the fresh stack was initialized. After we find this break point, we need to run the program until the break point and then look up where $ebp was loaded to.



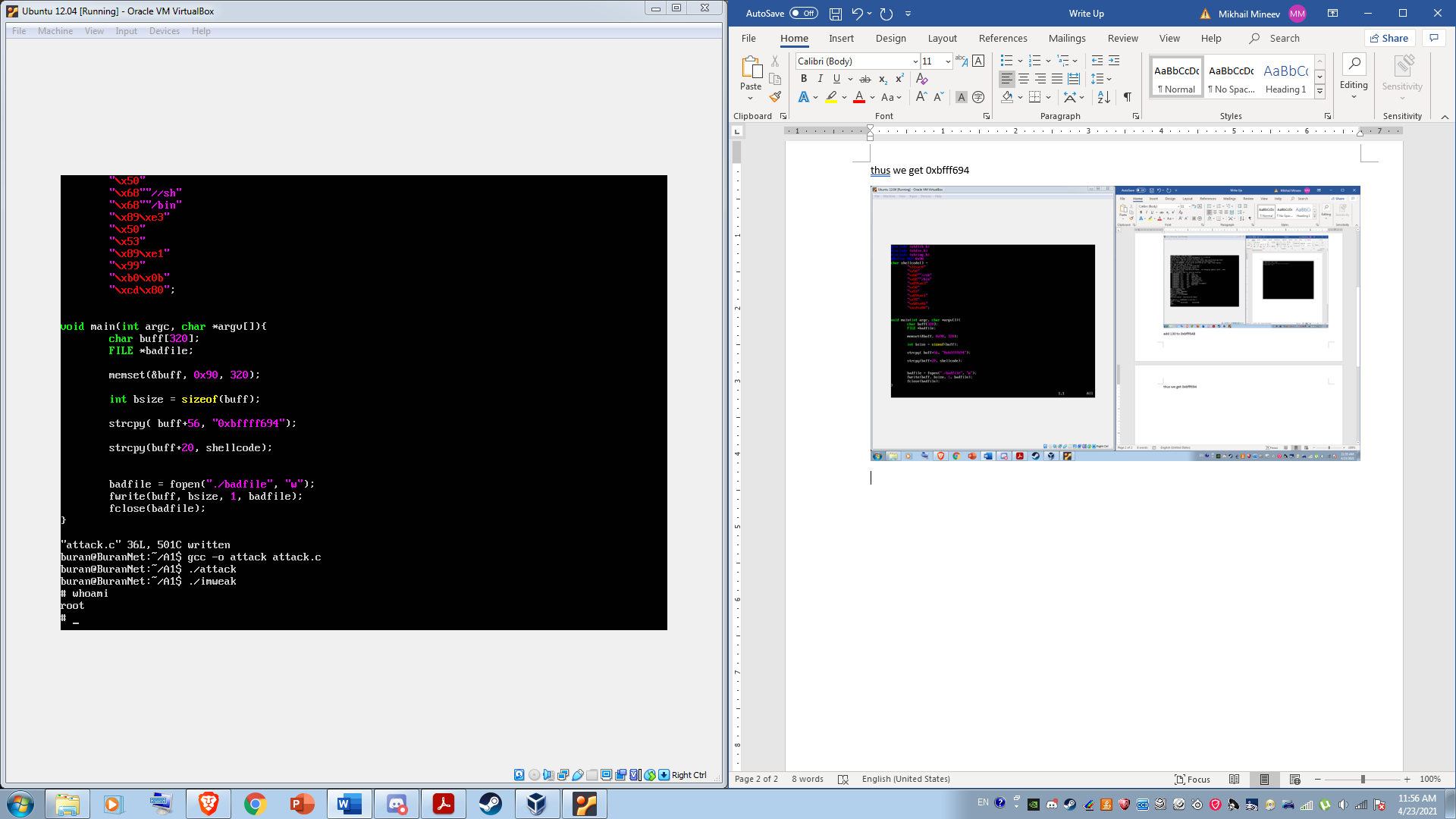
Then I found where our return address placement is and discovered that it was somewhere above buff+44. I did this by looking at the 0x28 command and accounting for 4 bytes of the frame pointer, In my case this ended up working after adding 56 bytes to our stack pointer.

Next I needed to find out where to start on the NOP sled, this was done somewhat arbitrarily with me picking to add 132 bytes to the start of out stack pointer in order to account for unforeseen shifting, and in an attempt to make the program work after the address randomization gets turned on during task 2. Thus, I got:

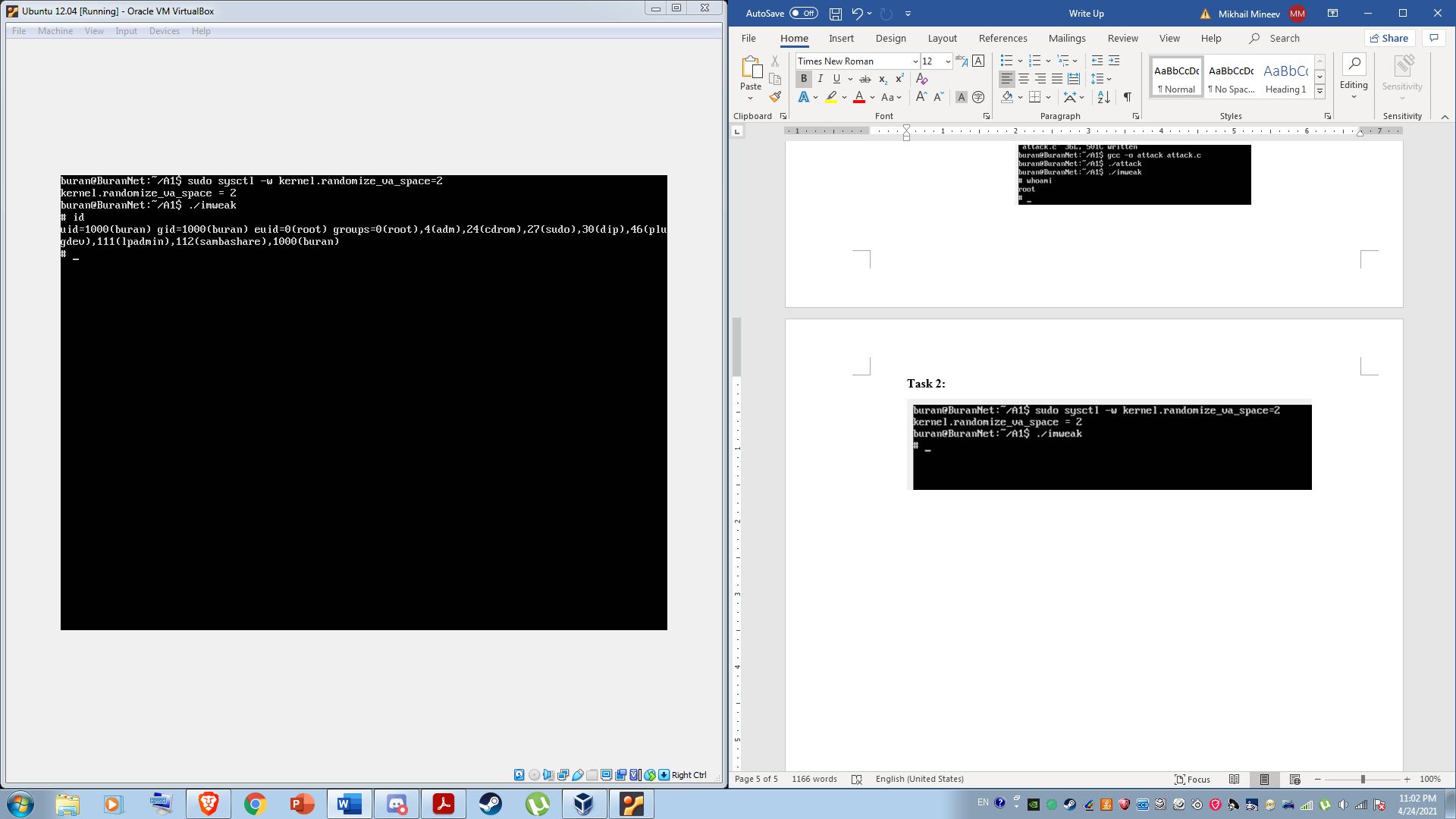
0xbffffb48 + 0x84 = 0xbffffb694



Last but not least we had to choose where to put the shellcode in the buffer to cause it to loop in on itself and thus create the overflow we are exploiting. I have found this to be based on semi guesses and seeing what causes a seg fault and what causes the program to work normally. In our case buff+10 allows it to work as intended (no overflow) and buff + 100 caused a seg fault. But with the buff+20 it functioned just right and let me gain root access.



**Task 2:**



To get the program to work we had to run the program multiple times to get the address we set up initially to line up with the address that gets used by the program in the run. Surprisingly my program worked quite a few times during the 100-time run.   
 I am not entirely certain, but I believe this is due to me putting the shellcode rather close to the front of the buffer allowing for more hits during the randomization of the addresses.  
  
Here is the code I used in the main

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

char buff[320];

FILE \*badfile;

memset(&buff,0x90,320);

int bsize = sizeof(buff);

strcpy(buff+56, “0xbffff694”);

strcpy(buff+20, shellcode);

badfile = fopen(“./badfile”, “w”);

fwrite(buff, bsize,1,badfile);

fclose(badfile);

}