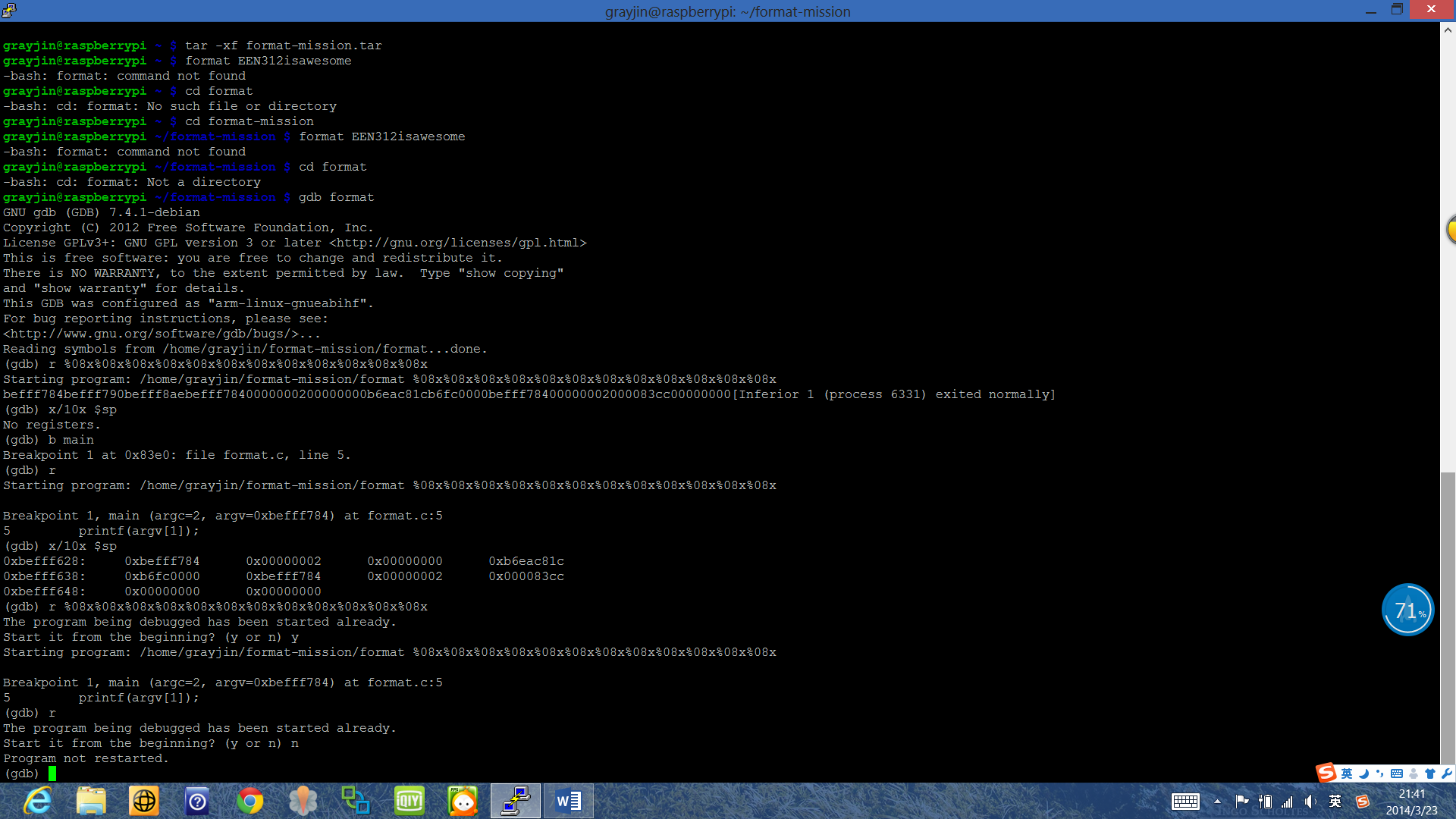
Lab 3 – Sabotaging the Stack

Nathan Paternoster

Mission 1

**Purpose –** To understand buffer overflows.



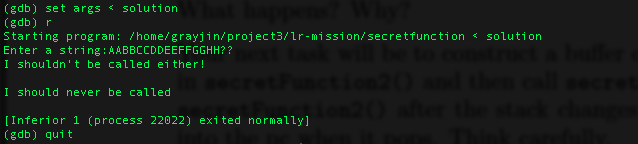
**Description**

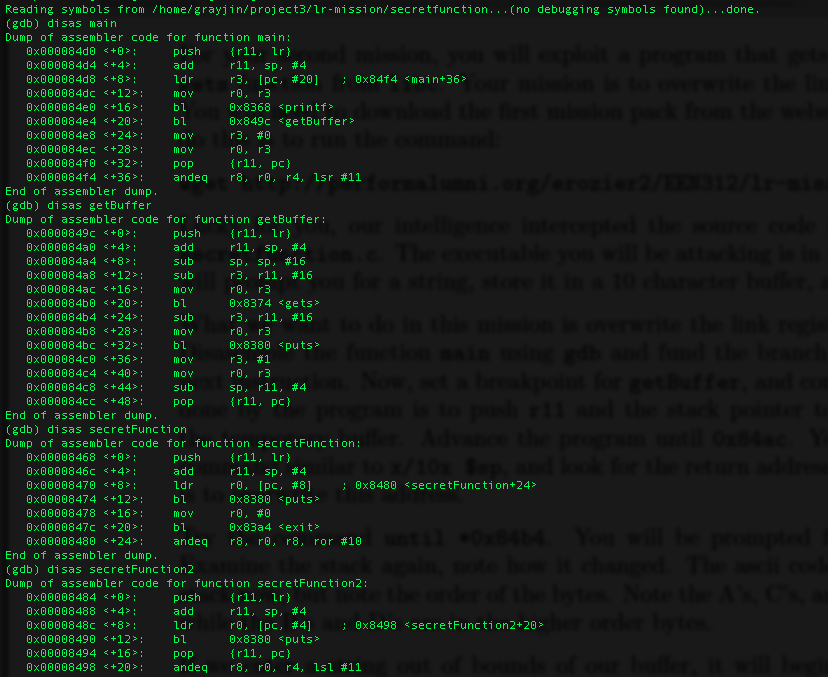
When the program is run normally it outputs the string that the user enters. However when we input “%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x” the output instead changed to strange values. These values were actually taken off the stack. The first three values were memory addresses and the next 9 were taken directly from the stack. We discover that using %08x we can access bytes from the stack and use them as inputs. About mid-way down the screenshot (the longest line) we can see the new output matches almost exactly the values on the stack (after the line x/10x $sp)

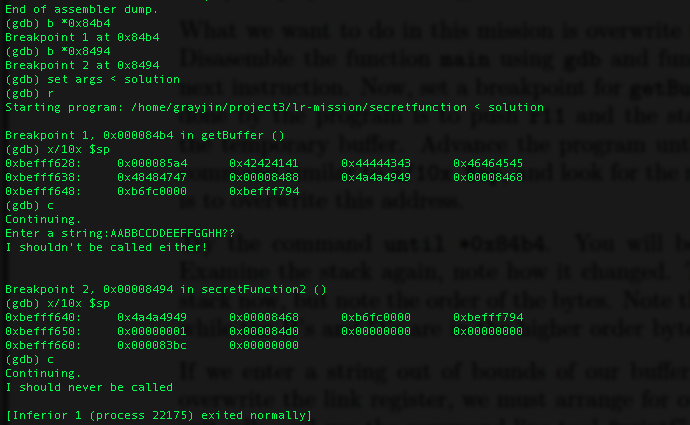
Mission 2

**Purpose –** To implement buffer overflow by calling two “uncallable” functions – secretFunction and secretFunction2.

**Solution input** = “AABBCCDDEEFFGGHH\x88\x84\x00\x00IIJJ\x68\x84\x00\x00”

secretFunction2 output followed by secretFunction output





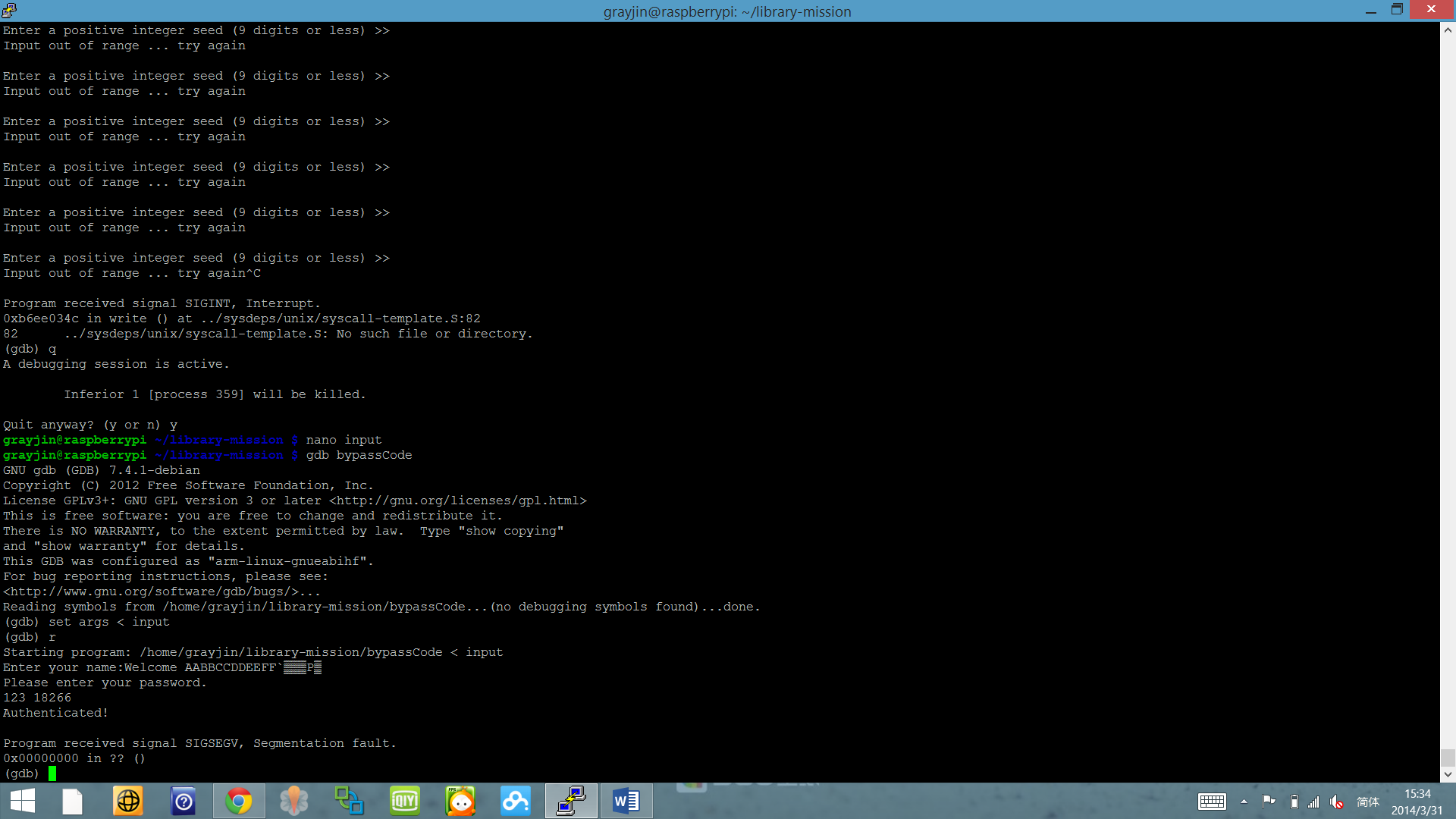
**Description**

In getBuffer our string input is put in in the “gets” function. The stack looks like what it is displayed as at Breakpoint 1. When getBuffer finishes it reads in our value “0x00008488” into the pc and branches there. That address is the second line of secretFunction2. In secretFunction2 the stack now looks like what it is displayed as at Breakpoint 2. When secretFunction2 finishes our dummy value (the first byte on the stack) is placed into r11 and then “0x00008468” is placed into the pc. The program then branches to that address which is the first line of the secretFunction. The program runs through secretFunction and then exits normally.

Mission 3

**Purpose –** To use buffer overflows to bypass a part of code and manipulate return registers to pass a password check.

**Solution input** = “AABBCCDDEEFF\x60\xf6\xff\xbe\x50\x06\xf1\xb6\x01\x00\x00\x00\x4c\x06\xf1\xb6IIJJ\x20\x94\x00\x00\n10”



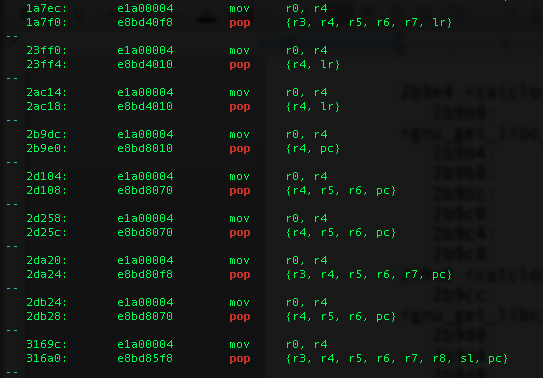
**Description**

In “getBuffer” we overwrote the user input in the gets function. The first thing we overwrote was the address of r11: \x60\xf6\xff\xbe. This is what was originally in r11. We kept it the same to not interfere with the program. Then we wrote the address of the pop instruction in the inet6\_rth\_add function (\x50\x06\xf1\xb6) to jump to when the “gets” function finishes. Next we pushed the value of “1” which is the value that “checkPassword” must receive to authenticate the user. Then we pushed the address of the mov instruction right before the pop instruction (\x4c\x06\xf1\xb6). This way the program will jump back 1 line and write the value of 1 into r0. Then we wrote “IIJJ” nonsense values to place into r4 and finally the address right after the “getBuffer” function finishes so that it will skip everything else in getBuffer. The /n10 will be to input “10” into the check password function to prevent it from going into an infinite loop.

We got the “Authenticated!” message but still ended with a segmentation fault.

Mission 4

**Purpose** **–** To identify “gadgets,” or sets of instructions in library functions that can be used to insert our own code - namely a ‘pop’ and ‘mov’ statement pair.



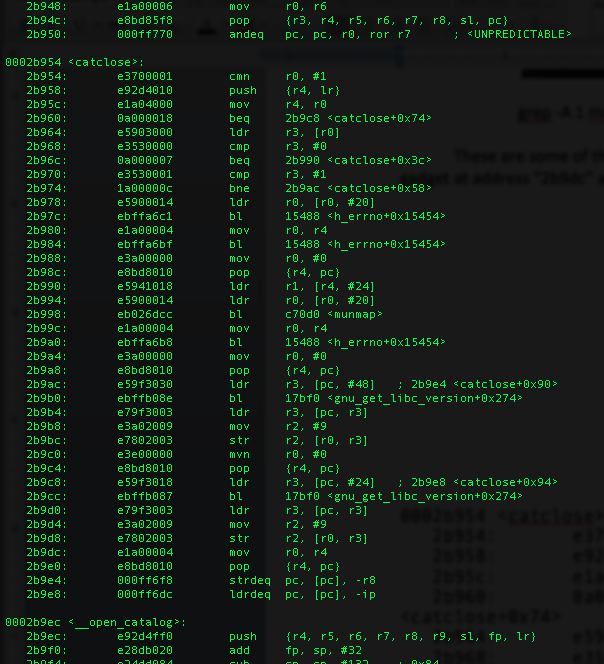
**Solution –** Using the “grep” command we located all of the instances of mov instructions that took “r0, r4” followed by a pop instruction. We chose one of these instances and located that address in memory to be used as our “gadget”:

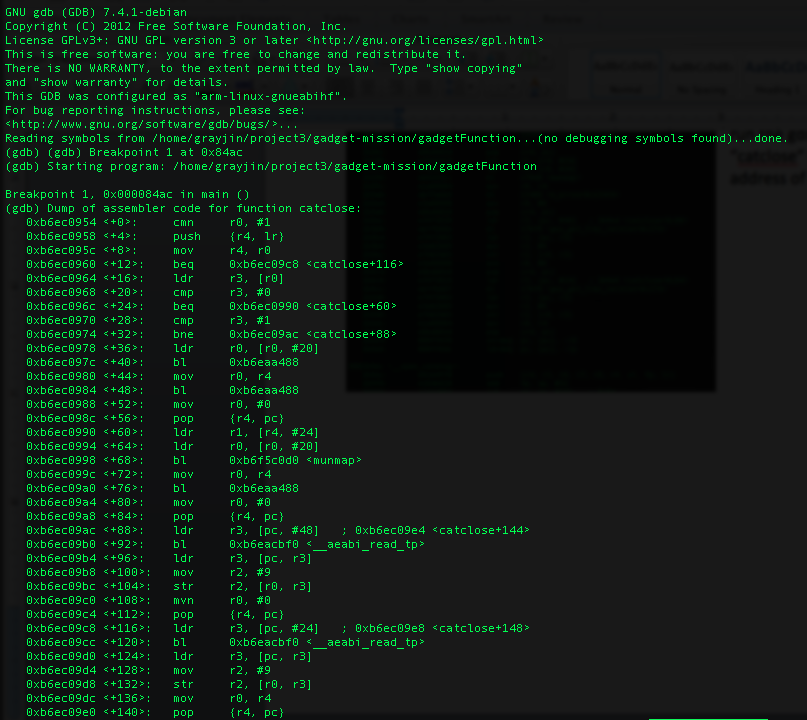
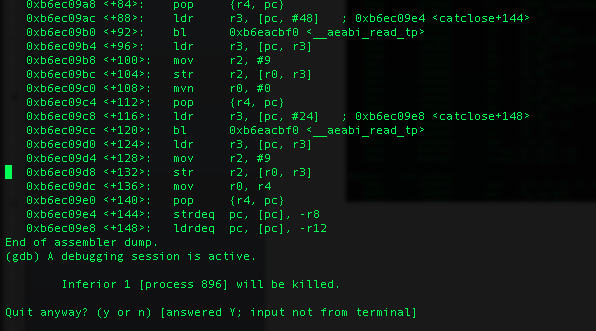
grep -A 1 mov libc.txt | grep -A 1 "r0, r4" | grep -B 1 pop

**Description**

These are some of the results from our search. We chose to implement the gadget at address “2b9dc” and “2b9e0.” Then we used the “less” command to examine libc.txt and find which function these two lines belonged to. We located the address in the libc.txt and determined that it belonged to the “catclose” function. The next thing to do is to go back to gdb and examine the function contents to determine the actual address of our gadget to jump to.

In gdb, we wrote a command file to run the gdb debugger and examine the “catclose” function to determine the actual address of our gadget.



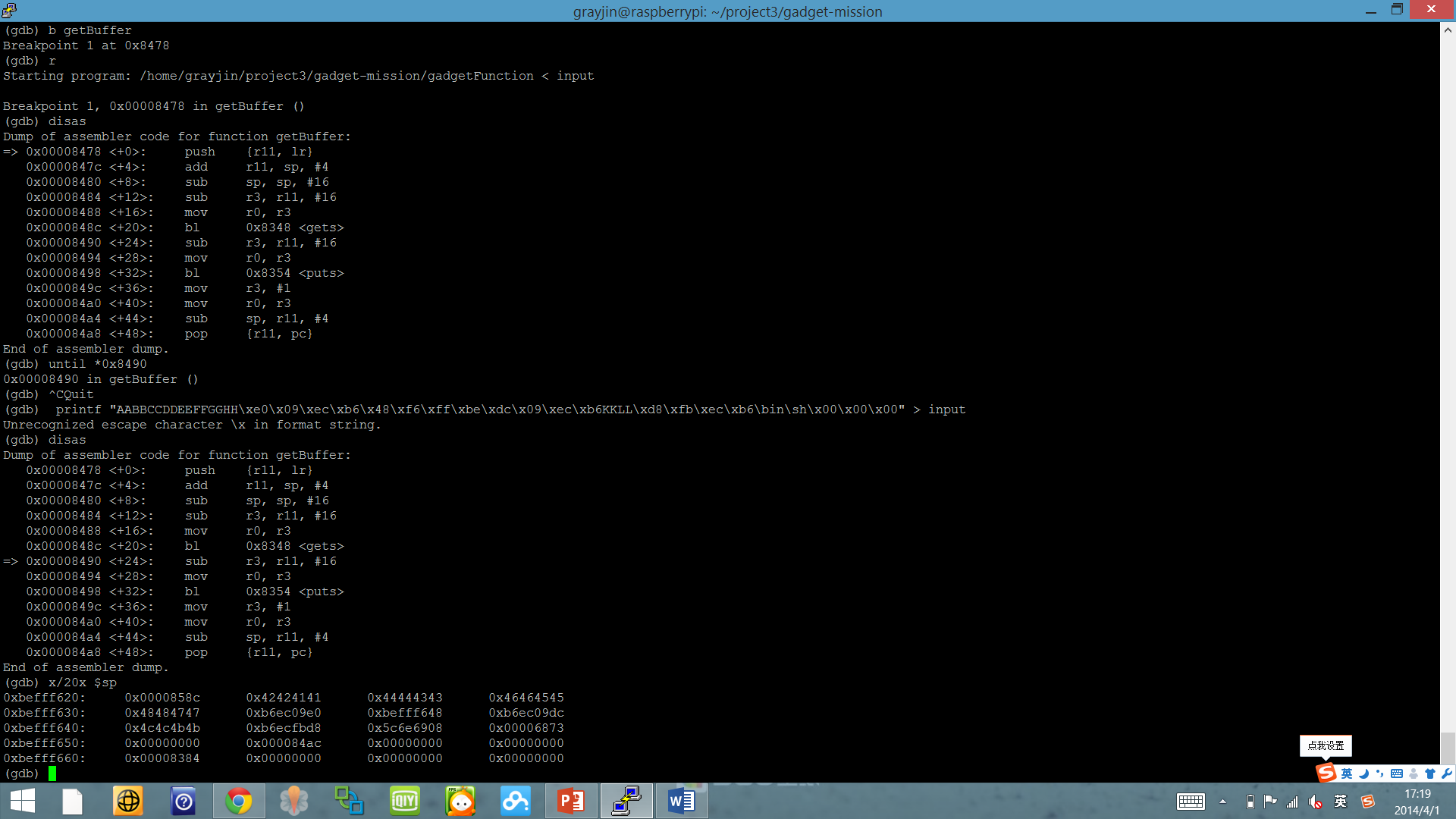


The address of our gadget is 0xb6ec09e0.

Mission Impossible

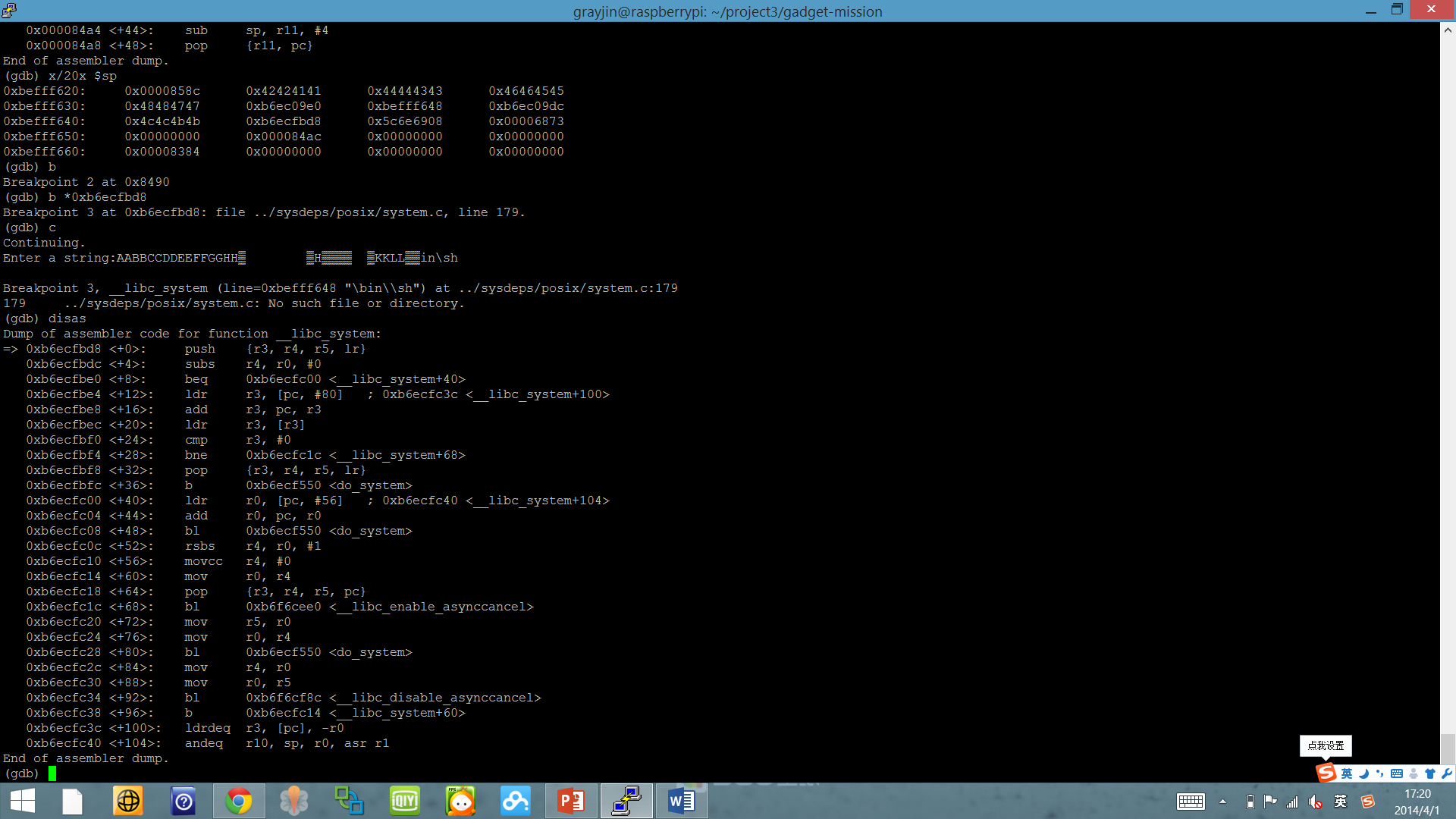
**Purpose –** To use buffer overflows to access a shell utilizing the “system” function with the argument “/bin/sh”.

**Solution** – “AABBCCDDEEFFGGHH\xe0\x09\xec\xb6\x48\xf6\xff\xbe\xdc\x09\xec\xb6IIJJ\xd8\xfb\xec\xb6\bin\sh\x00\x00”



**Description**

We utilized the gadgetFunction mission to buffer overflow and jump into the system function (as shown in the screenshot). To pass “/bin/sh” as an argument we first pushed it onto the stack (using a buffer overflow) then jumped to our previous gadget and placed the memory address of that place on the stack into r0. We then jumped into the system function and passed r0.



The system function is successfully called with the argument “/bin/sh”.