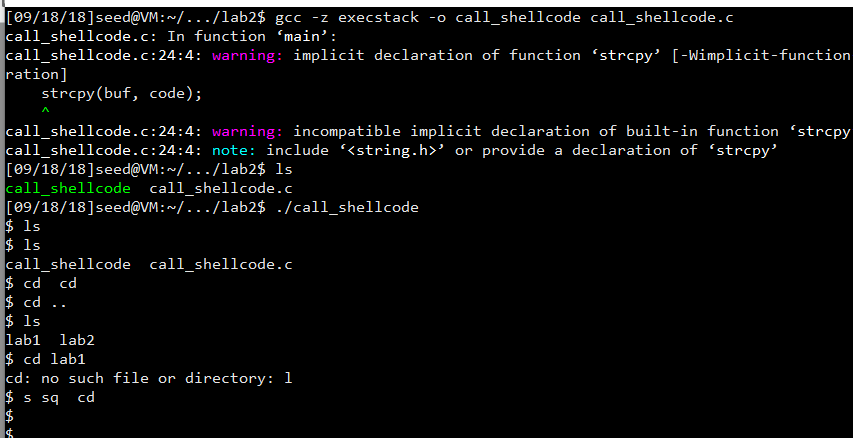
# Buffer Overflow

## Task 1: Running Shellcode

gcc -z execstack -o call\_shellcode call\_shellcode.c

./call\_shellcode



Obervation:

After compiling and running the program, in the terminal it shows the bash program. Everything shows just like bash terminal, except that the backspace is not showing correctly, but working correctly.

Explanation:

Although in the program there is only a string copy, it uses stack overflow to jump to the bash code. The original return address of the function is not valid anymore.

## The Vulnerable Program

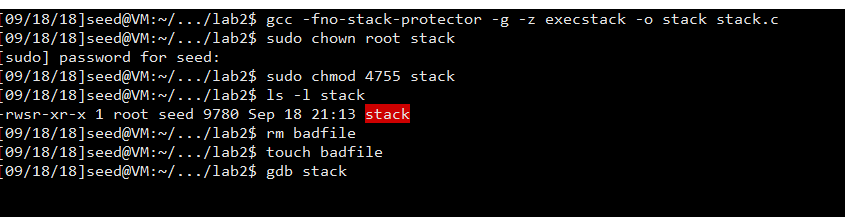
gcc -fno-stack-protector -g -z execstack -o stack stack.c

sudo chown root stack

sudo chmod 4755 stack

Touch badfile

gdb stack



(in gdb)

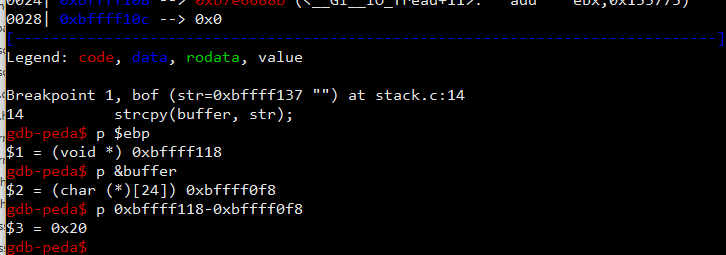
B bof

Run

P $ebp （= 0xbffff118）

P &buffer （= 0xbffff0f8）

P 0xbffff118 - 0xbffff0f8=0x20



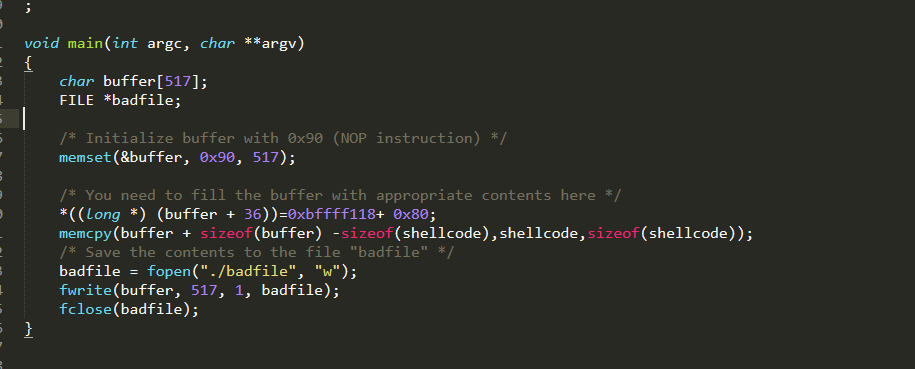
The value of the frame pointer is 0xbffff118, so the return address should be 0xbffff118+4 and the first address we can jump is 0xbffff118+8. And the distance between ebp and the buffer’s starting address is 0x20+4=32+4=36

## Task 2: Exploiting the Vulnerability

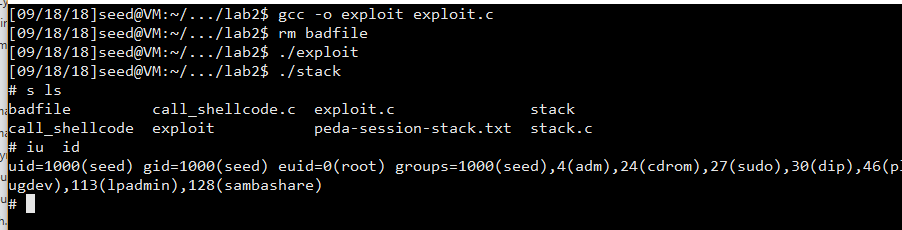
According to the above data, we can finish the explot.c program with the following line:

\*((long \*) (buffer + 36))=0xbffff118+ 0x80;

memcpy(buffer + sizeof(buffer) -sizeof(shellcode), shellcode,sizeof(shellcode));



Compile and run the programs:



Observation:

After creating the badfile, we run the stack program. It shows that we successfully get into the shell. And in the id command, we see that the “euid” is root, which means we have got the root’s bash, and the experiment is successful.

We can get the real user to be root. We can only run the program:

#include <stdio.h>

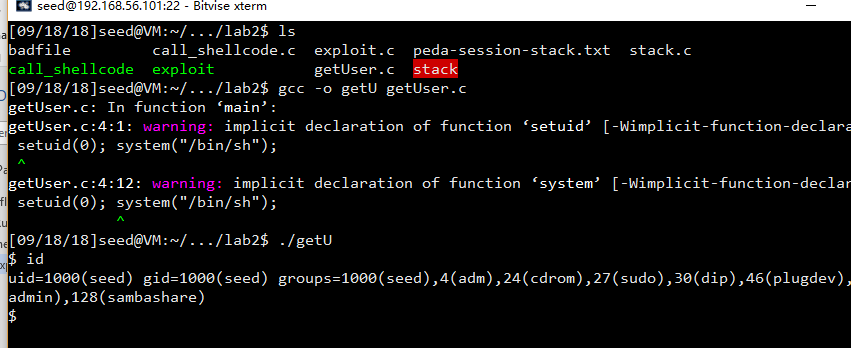
void main()

{

setuid(0); system("/bin/sh");

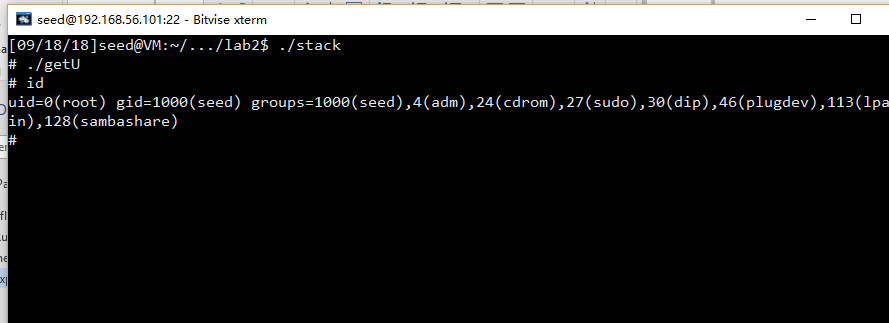
}

To compare, we can run the program under “seed” first:



It shows that the id is seed’s

And In the bash we just get in from stack:

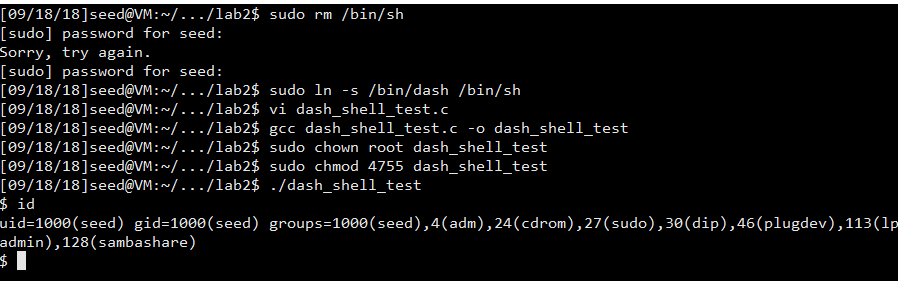


We can see that the real user id has become root. It means that we get the root’s shell successfully.

## Task 3: Defeating dash’s Countermeasure

Modify the file:

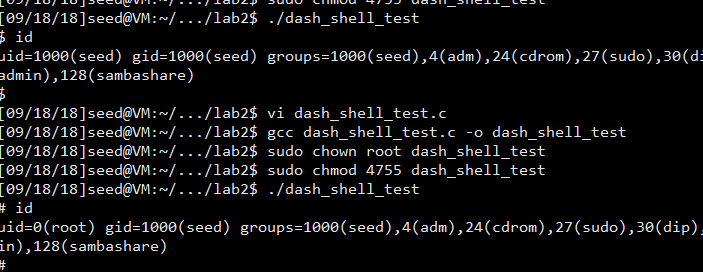
Before uncommenting setuid(0):



Observation:

It shows that the uid is 1000(seed)

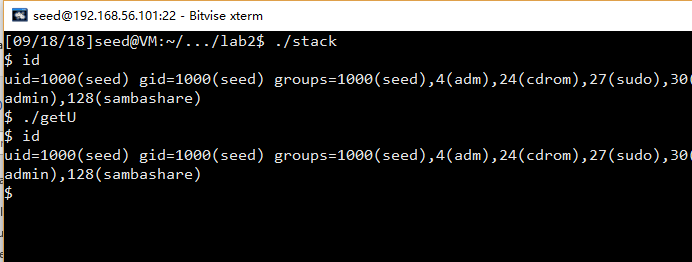
After:



Observation:

After adding setuid(0), the uid become 0(root)

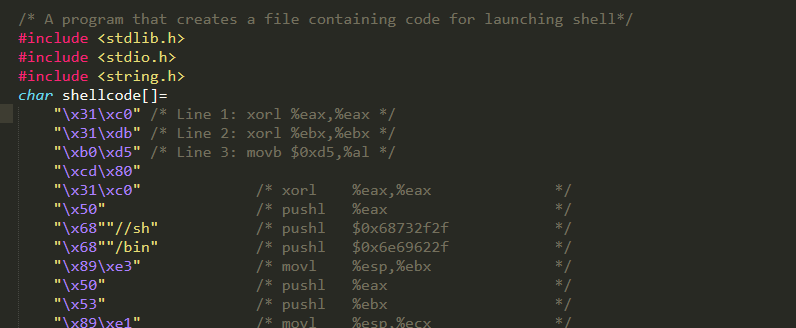
After using dash, we will test task2 again:



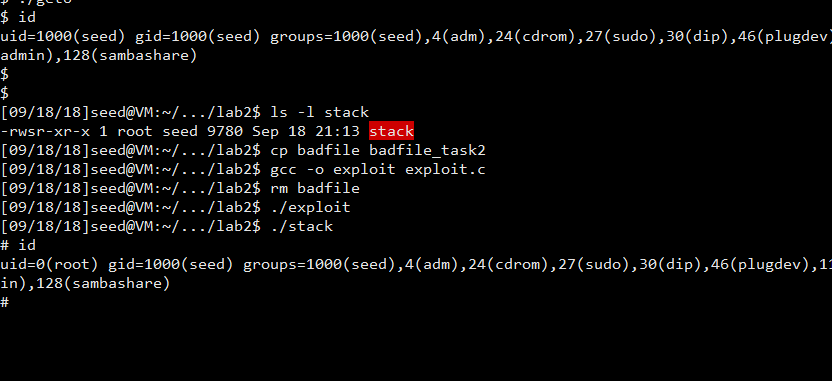
Observation:

It shows that although the program get into the bash successfully, the uid is still seed’s. It shows that the dash countermeasure is working.

To defeat this, we need to modify the exploit.c file and use the new shellcode.



Compile and run:



Observation:

After generating new badfile and run the stack program again, we can see from the shell that the uid is set as root immediately. We defeated the dash countermeasure successfully.

Explanation:

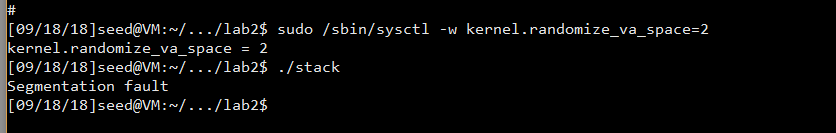
This is because that we add the setuid(0) in the shellcode. And before getting into shell from stack overflow, it executes setuid(0) command first. In the first experiment we can see that setuid(0) + /bin/sh can get into the root’s bash.

## Task 4: Defeating Address Randomization

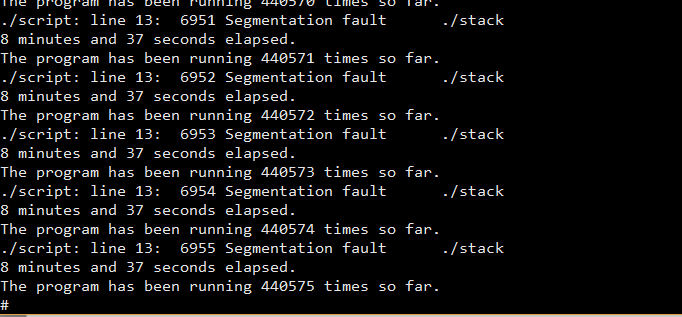
Firstly, we need to change the setting:

sudo /sbin/sysctl -w kernel.randomize\_va\_space=2

After changing the setting, we can see that the attack is not working anymore:



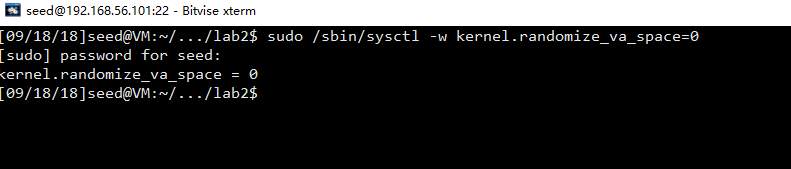
Run the script:



It shows that after 440575 times the program is finished and successfully get into the shell. Because of the randomization, the address in the badfile is not correct at the first time. But after 440575 times trying the incorrect address, the real address happens to be the same as the one in the badfile.

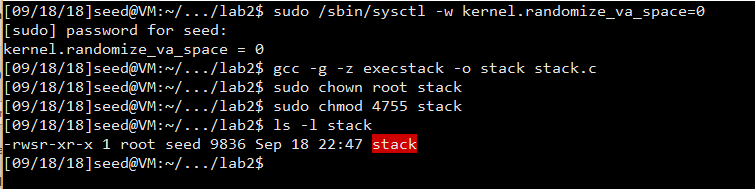
## Task 5: Turn on the StackGuard Protection

Close the Randomization:

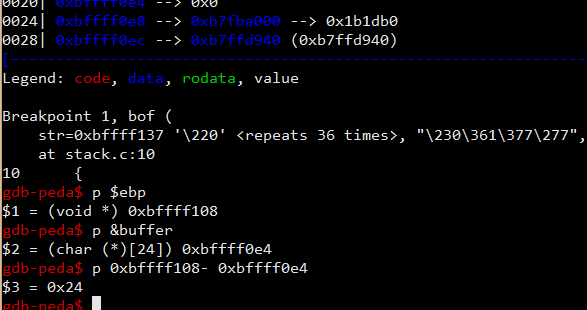


Re-compile the program stack without -fno-stack-protector

gcc -g -z execstack -o stack stack.c

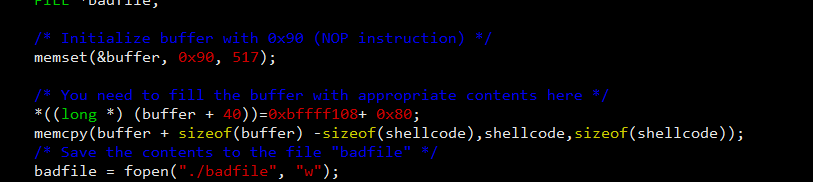


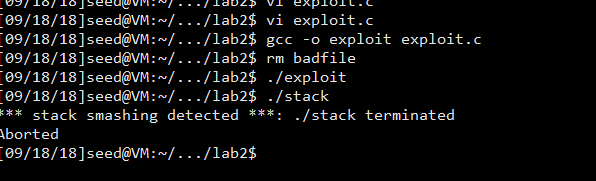
Gdb it again to see if there is any changes:



It shows that the address changes a little. The ebp address is 0xbffff108 and the distance is 36+4=40.

Make changes in the exploit.c file and re-generate the badfile.



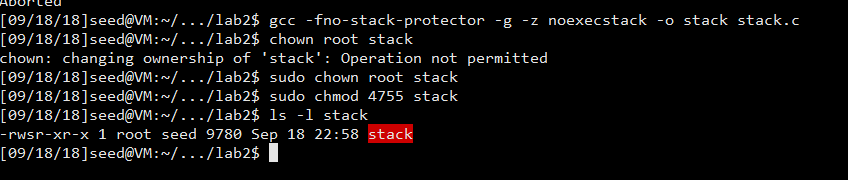


Observation:

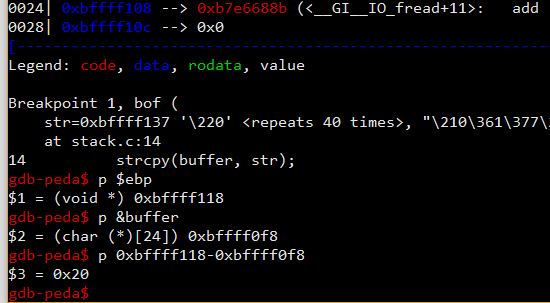
It looks like doing the same operation, the program aborted so that we can not get into the bash. Since we close the randomization, comparing to what we did in the task 3, it proves that StackGuard is working. I also noticed that the ebp address changes only in this way. I think it is the compiler that changes the way ebp and &buffer store.

## Task 6: Turn on the Non-executable Stack Protection

gcc -fno-stack-protector -g -z noexecstack -o stack stack.c



Gdb stack:

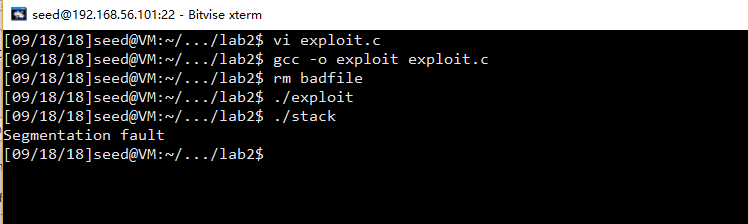


It shows that the address and distance are the same as that in task 2 and 3.

Modify the exploit program back:



Run the program:



Observation:

The program ends with a segmentation fault but not aborted. This shows that the buffer overflow may have occurred but the /bin/sh program is not executed. Comparing to task 3 , we can say that the noexecute flag is making effects. Comparing to the stackGuard, we can draw the conclusion that the two techniques are using different ways to prevent attackers from getting into /bin/sh, but the noexecute flag can not prevent stack-overflow.