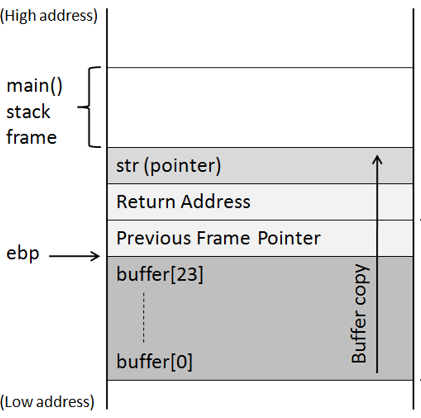
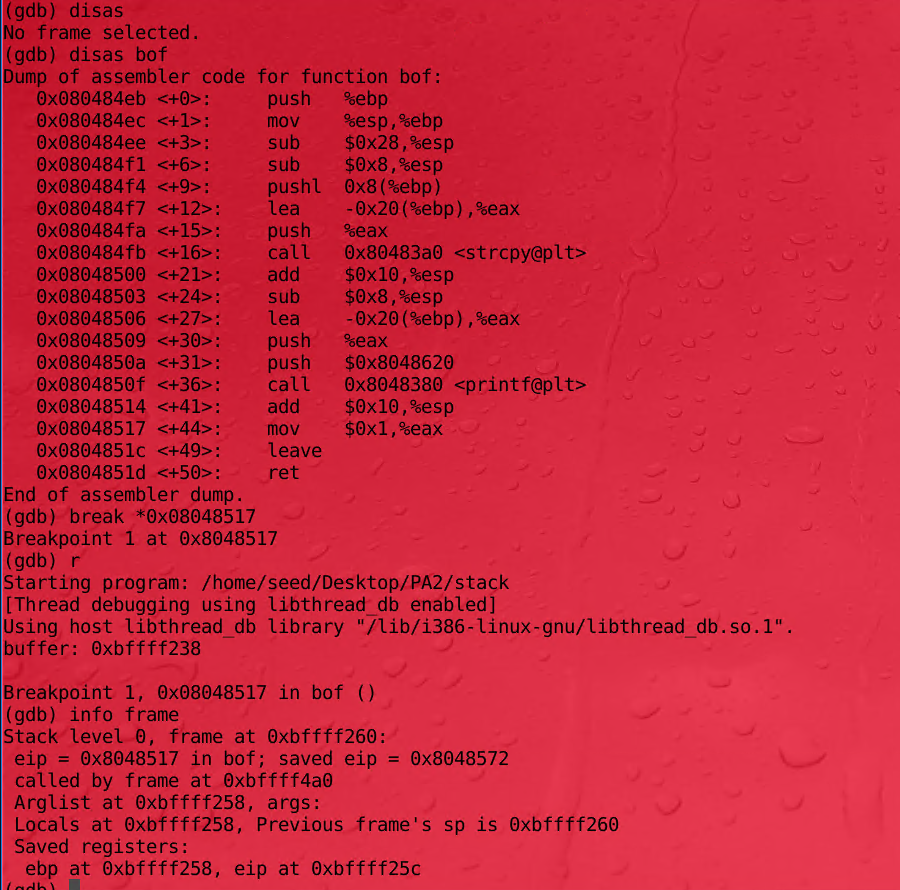
**Homework 2**

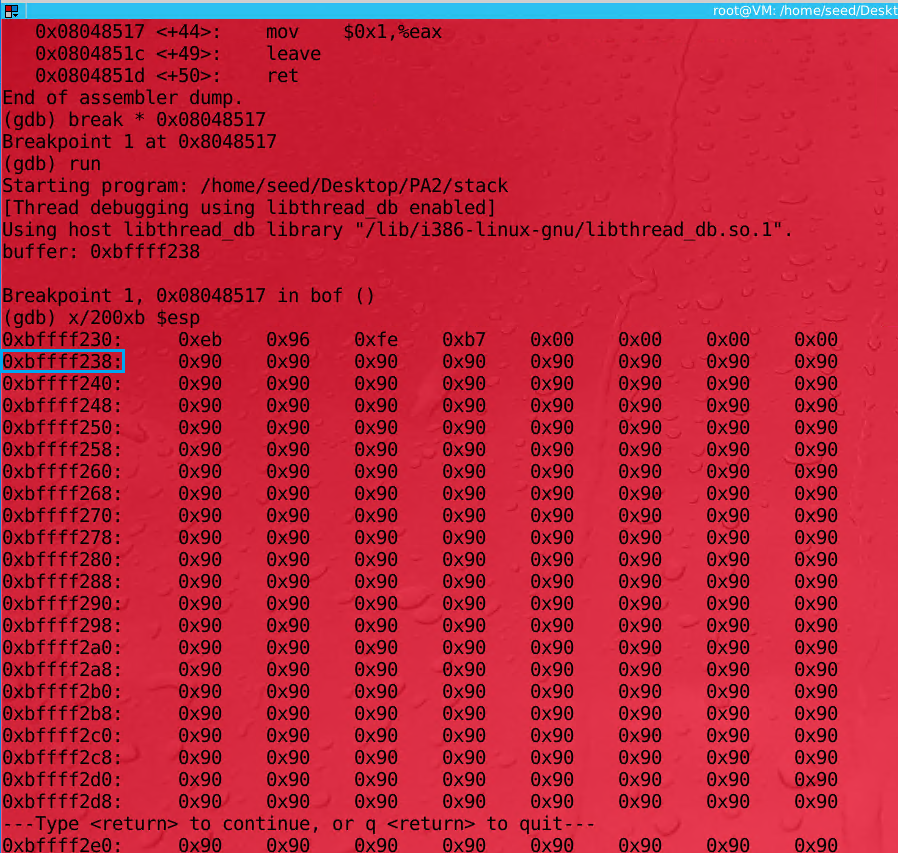
**Task 1:**

The most important part of this task is to find the stack pointer address (esp), frame pointer address (ebp) and the actual size of the buffer. From that we can calculated the offset.



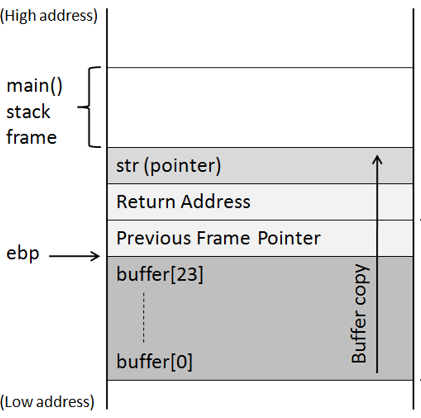
After turn off the ASLR, linking the zsh to bin\sh and compling the program without Stack guard, I ran the stack file in gdb to observe its registers:

It can be seen that the address of the frame pointer (ebp) is 0xbffff258



To find the stack pointer address I ran the command x/200xb $esp to see the memory frame. It can be observed that the NOP operations (from the badfile generated from the original exploit.c) are filled from 0x0bffff238. That’s where the stack pointer is. Now we have both stack pointer and stack frame addresses, subtract these two we will get x20 which is 32 in decimal. That means 32 bytes are reserved for the buffer instead of 24 as the buffer’s size.

To calculate the offset, I added 32 bytes I just have found with 8 more bytes since we need to get the address right after the return address



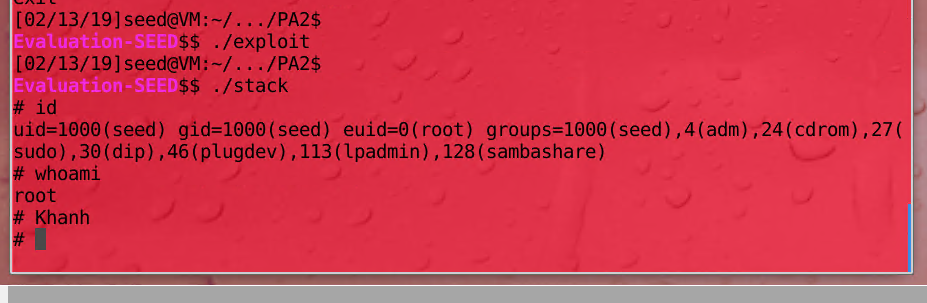
32 bytes

4 + 4 = 8 bytes

So the offset is 40 bytes. To overflow the buffer, firstly I filled the buffer with NOP operations. After that, I got the stack pointer address and added 40 to that address to get the location of the NOP operation right after the return address. Finally, I added the shellcode to the end of the buffer. Refer to the code below for more details:

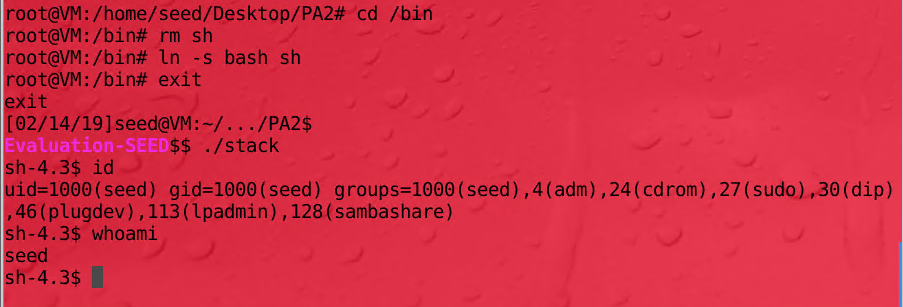


This is the screenshot of the successful buffer overflow attack to get to root from shell:



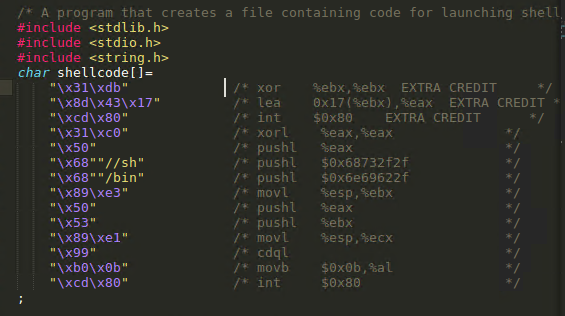
**Task 2:**

I linked the /bin/sh back to the /bin/bash and this is the result I got:

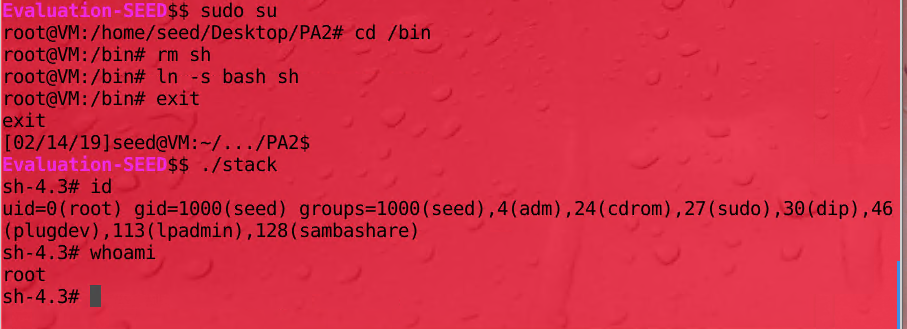


The program pulled a shell but it seems to be the bash shell and does not have root privilege. When bash is called, it automatically drops root privilege to protect the root from being invoked by someone who shouldn’t have access.

**Extra Credit:**

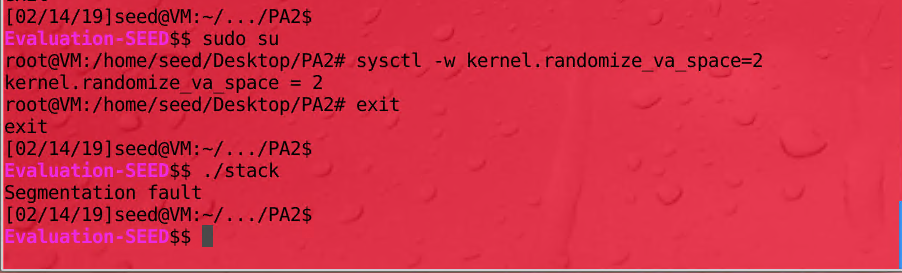
To get around this protection, I added some hex code op top of the shell code given to set setuid=0: 

After recompiling exploit.c and creating a new badfile with new shellcode added, this is the result I got:

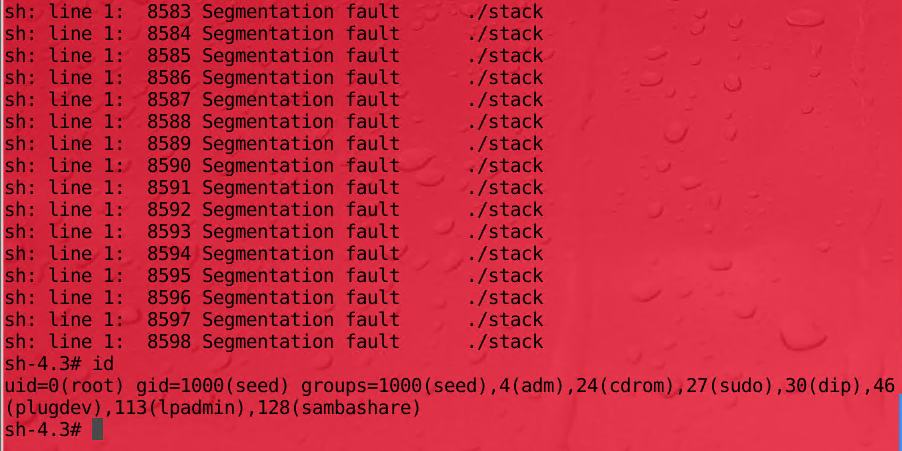


**Task3:**

When the address randomization is on, I got the Segmentation Fault result:



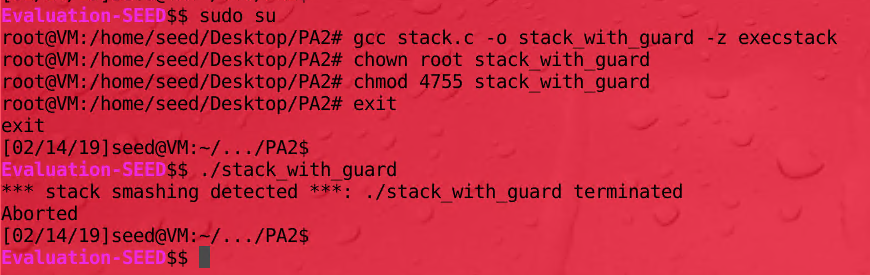
I ran the stack in an infinite loop and kept getting segmentation fault, after a while, I was able to get to the root:



We got the segmentation fault because when the address randomization is on, the stack addresses are no longer in numerical order. They are in random order. That means that the NOP sled now becomes useless and the return address can’t be calculated correctly. We could get to the root after looping the program a while because the correct address in the stack can be randomly “guessed” by the program and after that it will run as it should.

**Task 4:**

After compiling the stack program without -fno-stack-protector flag to enable stack guard, I ran the program (I named it stack\_with\_guard) and got the result:



The program was terminated because the Stack Guard adds a special memory location to the stack frame. When we overflow the buffer, this special address will be overwritten. Therefore the operating system can see the stack is being tampered with an terminated the program.