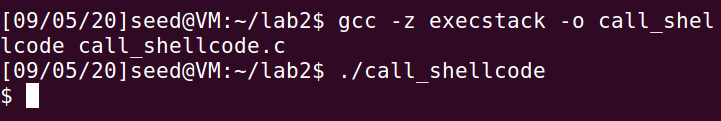
# BUFFEROVERFLOW\_LAB

我们首先将buff\_size值设置为24。编译call\_shellcode.c文件，执行后可见成功进入shell。

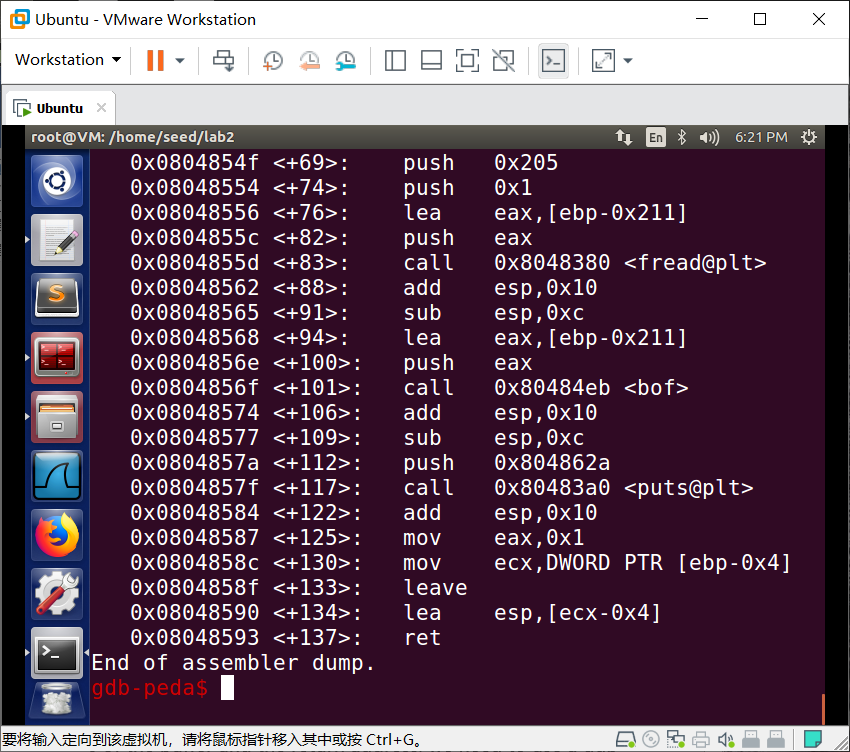


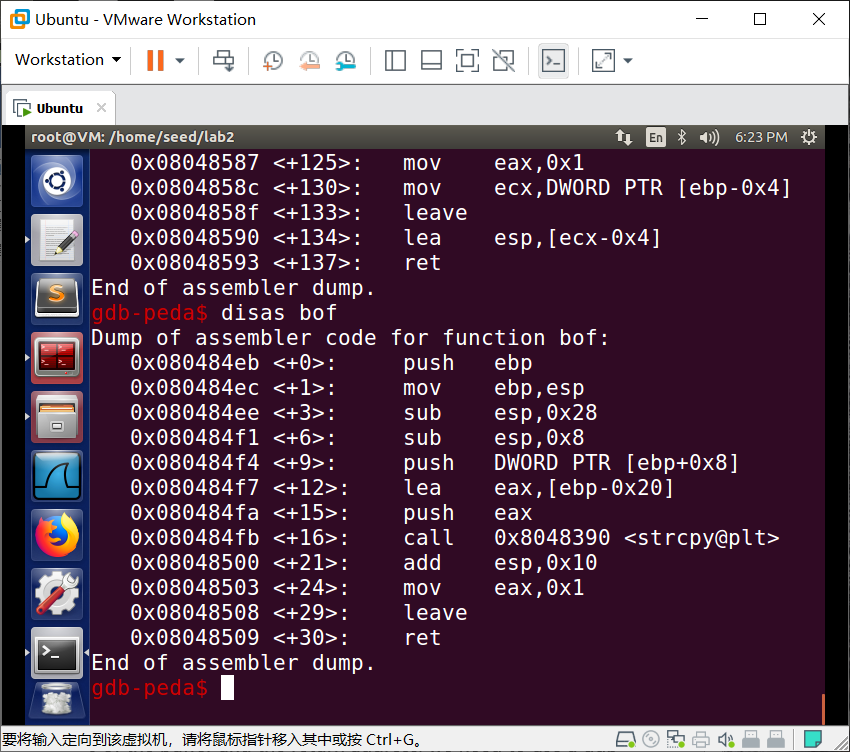
编译stack.c文件，编译时关闭地址随机化和栈保护机制，生成可执行文件stack。



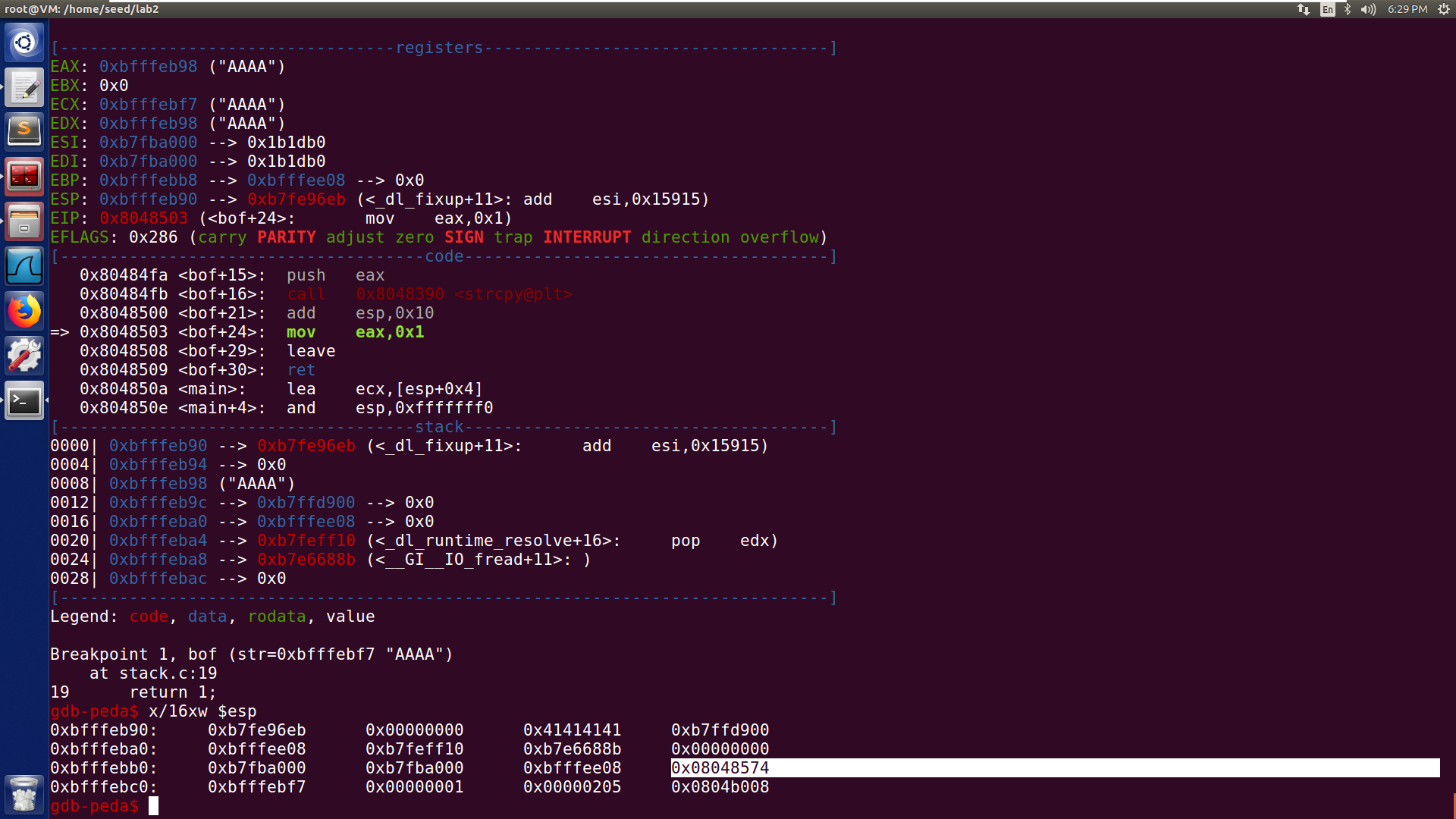
使用gdb调试stack，反汇编mian函数，可见调用bof函数后的返回地址是0x08048574。

反汇编bof函数，其结束地址是0x08048503。

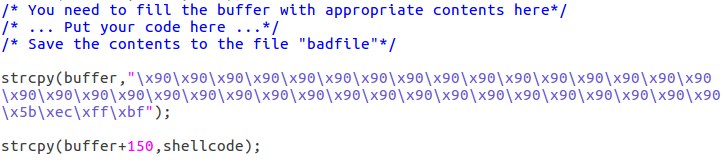




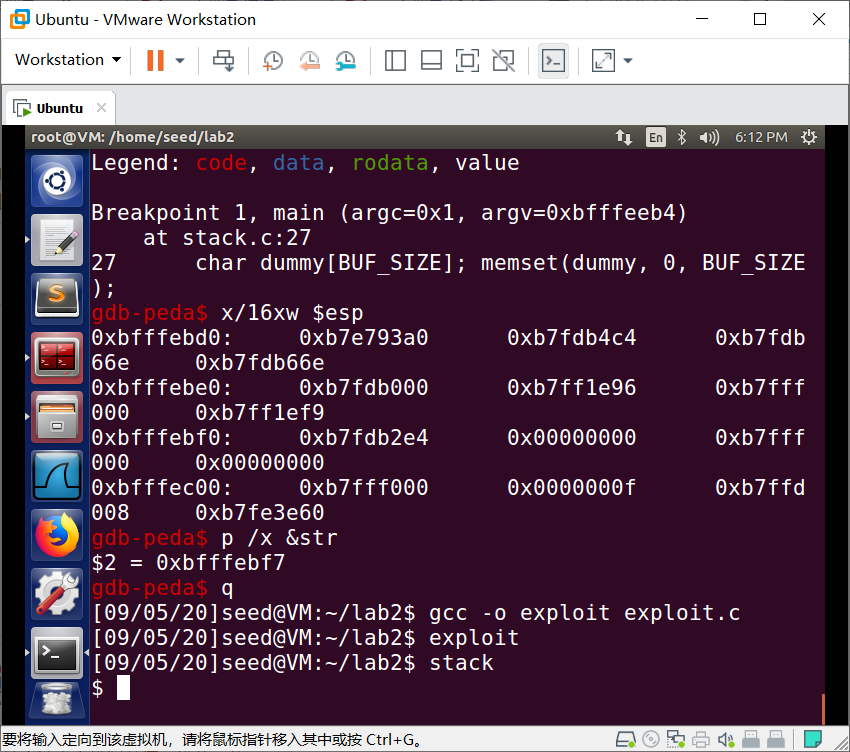
我们首先设置buffer中为“AAAA“，在寄存器中为“41414141“。在bof函数执行结束后的位置设置断点，显示附近的寄存器的值。可见变量的起始位置与返回地址相差36个字节。我们应当在exploit.c中的buffer填充36字节后添加上shellcode的地址。



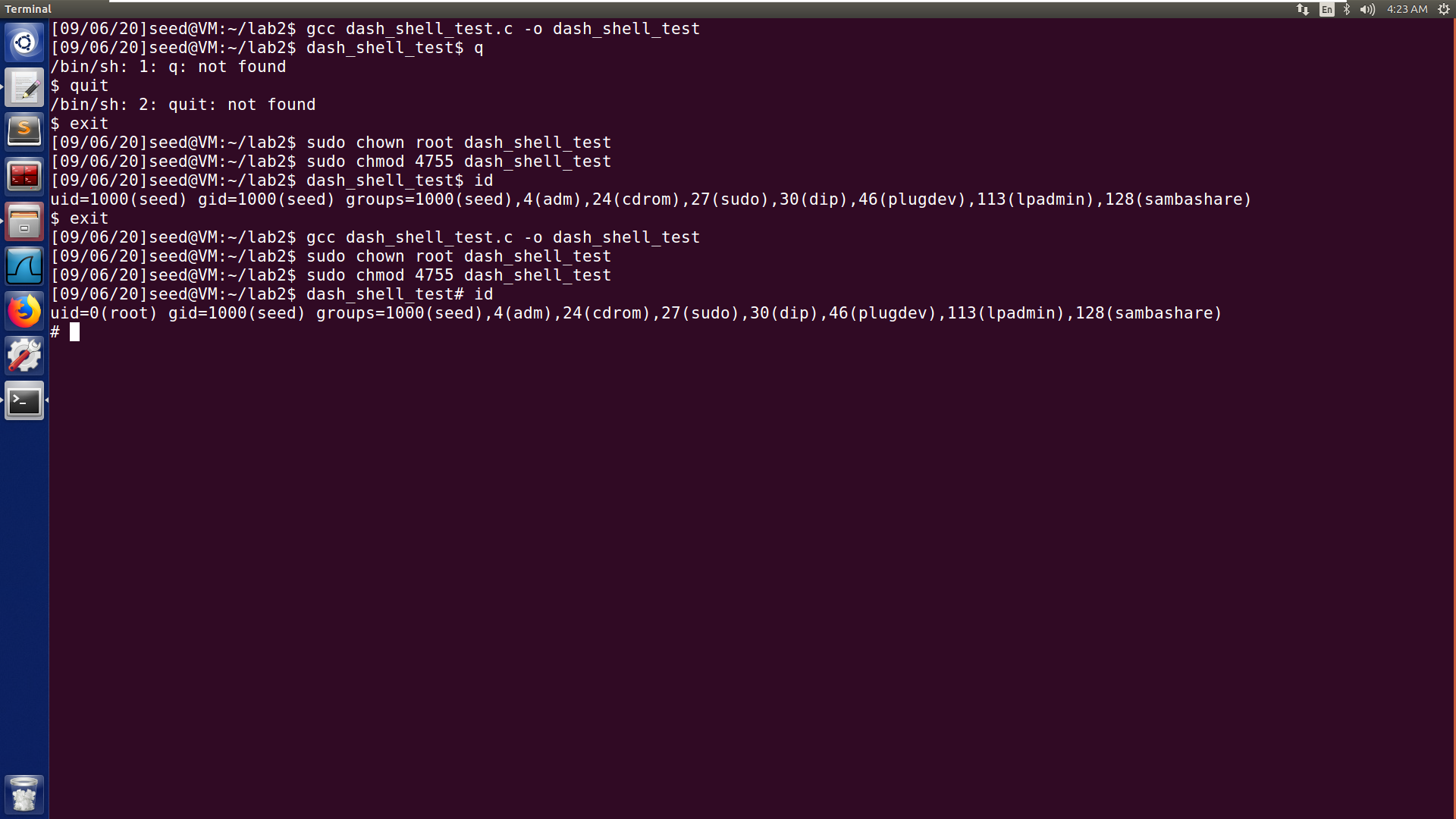
buffer起始地址为0xbfffebf7，将shellcode地址定为0xbfffec5b（增加100）。Shellcode的起始位置是buffer开始150字节后。



编译exploit.c后执行exploit和stack，可见成功进入shell中。

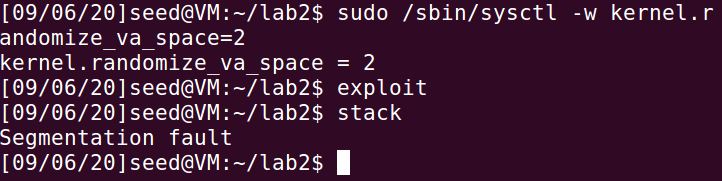


编译并执行dash\_shell\_test.c，上面是注释掉setuid(0)的情况，下面是保留setuid(0)的情况。可见设置setuid(0)可以绕过系统通过检查uid和gid不一致而降权的情况。



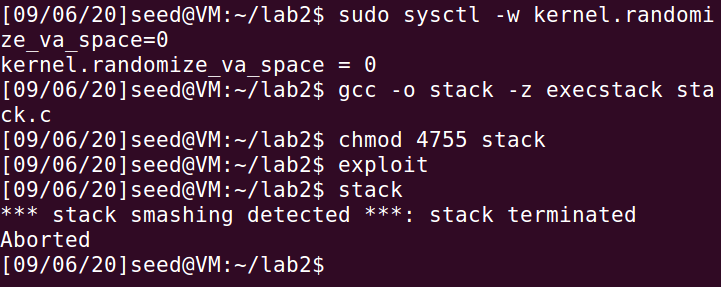
我们在上面所做的shellcode中添加设置setuid(0)的指令，执行后同样获取了root权限。

接下来尝试打开地址随机化，进行同样的攻击，发现攻击无效。

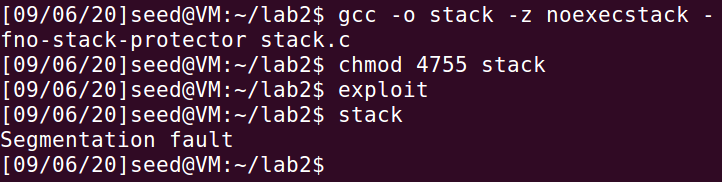


执行指导手册中的循环遍历，较长时间也未见成功，可能是空间较大，随机性较高。理论上遍历整个地址空间总会找到随机化后的地址。

关闭地址随机化，重新编译stack.c，这次没有关闭栈保护机制，重新进行攻击，发现栈溢出已经被检测并阻止了。

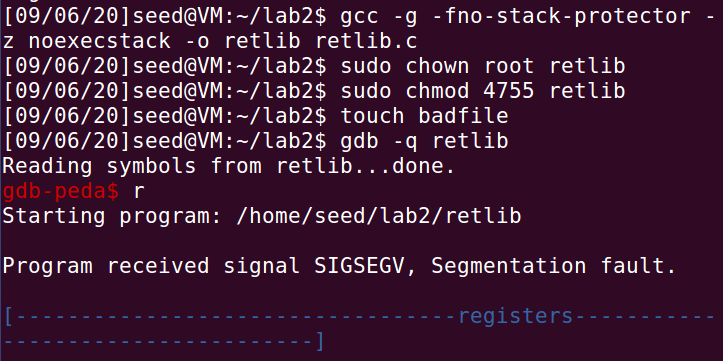


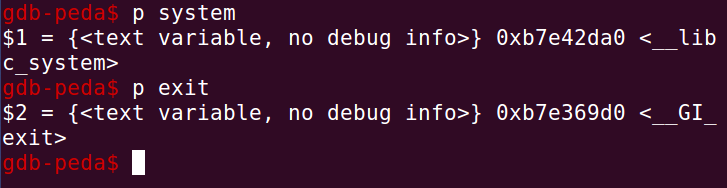
重新编译stack.c，这次关闭栈保护机制，但设置栈不可执行，重新进行攻击，发现攻击同样失败。



# Return-to-libc Attack

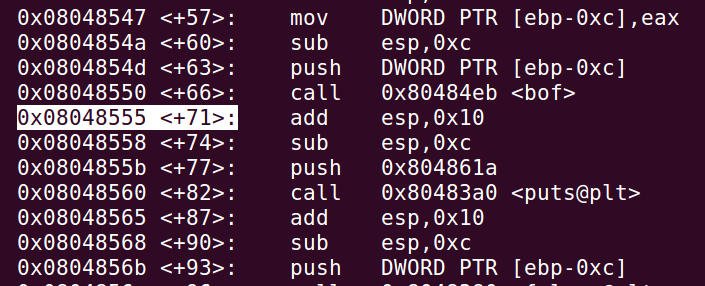
首先编译retlib.c文件，关闭地址随机化和栈保护机制，打开不可执行栈，编译后使用gdb获取system和exit函数的地址。编译实验报告的获取myshell地址的程序来获取myshell地址。

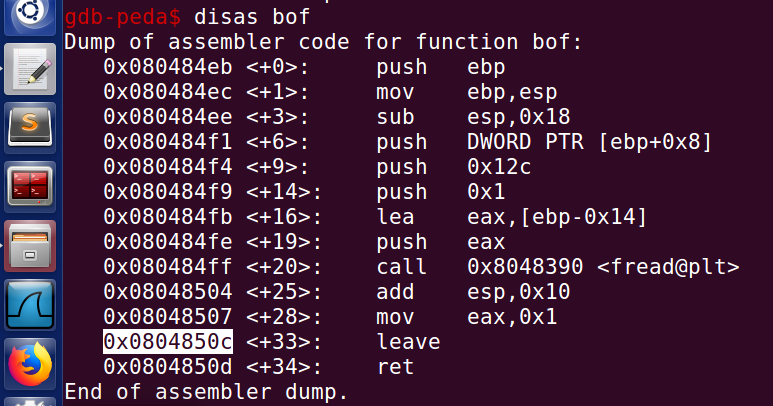


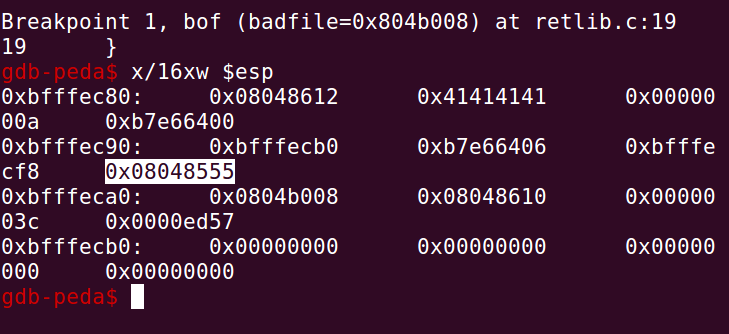




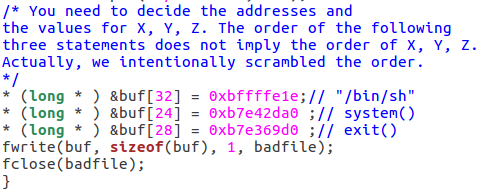
首先设置badfile中为“AAAA“，在内存中为“41414141“。在bof函数执行结束后的位置设置断点，显示附近的寄存器的值。“41414141“与返回地址相差24个字节，基底就设置为24。







将buf的相关位置填上system、exit和/bin/sh的地址，编译后执行。



但是结果并没有成功，相关原因的排查仍在继续。

