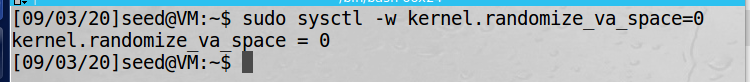
Buffer Overflow Vulnerability Lab

实验报告

57117105 杨哲君

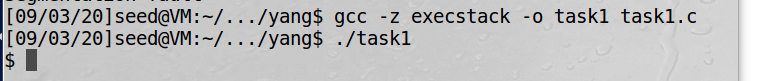
**2.1 Turning Off Countermeasures**

Address Space Randomization.

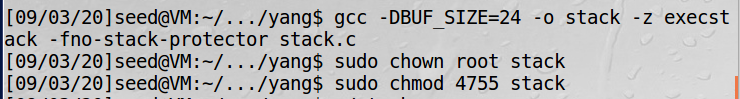


**2.2 Task1 : Running Shellcode**

该程序演示如何通过执行存储在缓冲区中的shell代码来启动shell，编译并运行该程序。发现确实启动了新的shell。

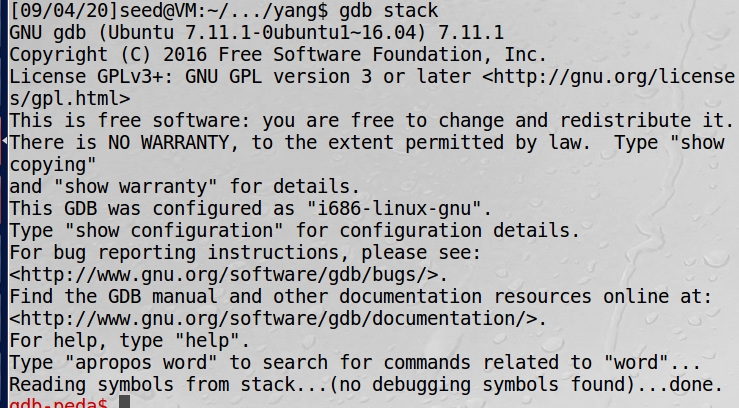


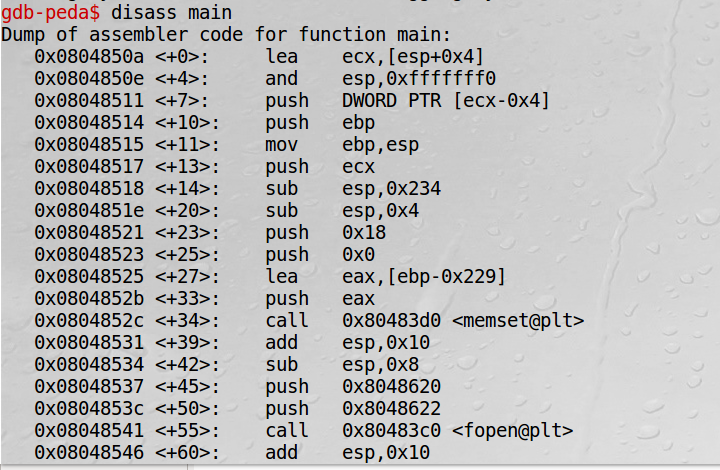
**2.3 The Vulnerable Program**



**2.4 Task2 : Exploiting the Vulnerability**

用gdb命令对stack进行调试

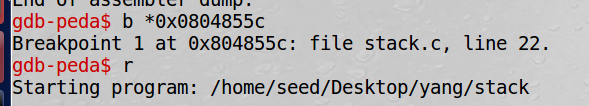


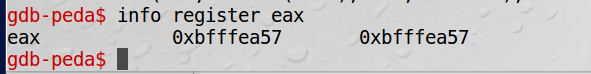


观察程序可以发现，fread函数传入的第一个参数就是str，而在调用fread函数指令上方的0x0804855c处push eax的指令就是代表传入了str，因为传入的第一个参数是被最后一个压入栈的。

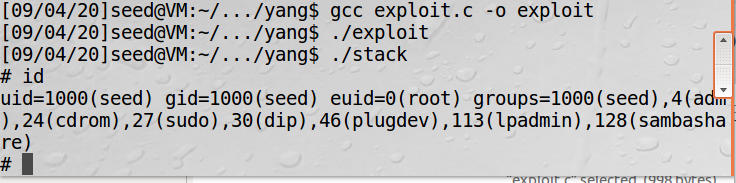
在0x0804855c处设置断点。读取到str的地址是0xbfffea57。加上偏移量100，加上0xbfffeabb。







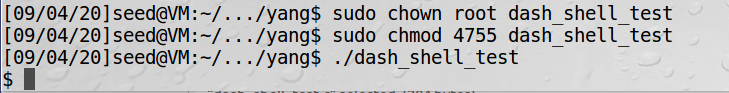
改写exploit代码，将str前段覆盖为大量循环的0xbfffeabb地址，中段覆盖为大量的nop指令，后段覆盖为shellcode，编译运行，



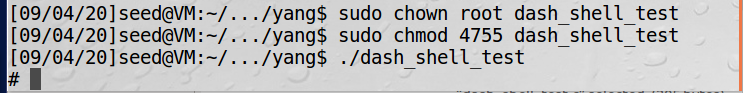
通过攻击，获得了root权限。

**2.5 Task3 Defeating dash’s Countermeasure**

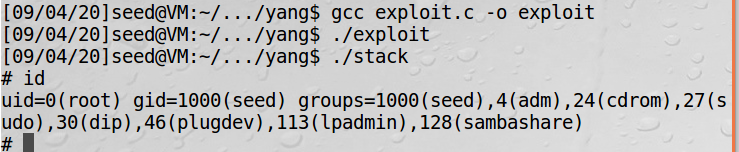
先修改/bin/sh符号链接指向/bin/dash，然后在没有取消注释的情况下运行程序。可见，新开的shell没有获得root权限。



将setuid(0)取消注释后，发现新开的shell拥有了root权限，因为它将程序的uid设为了0。



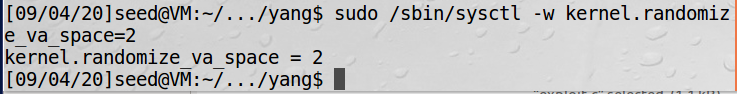
修改shellcode，



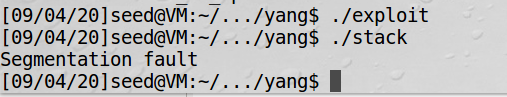
可以看到，shell获得root权限。

**2.6 Task 4: Defeating Address Randomization**

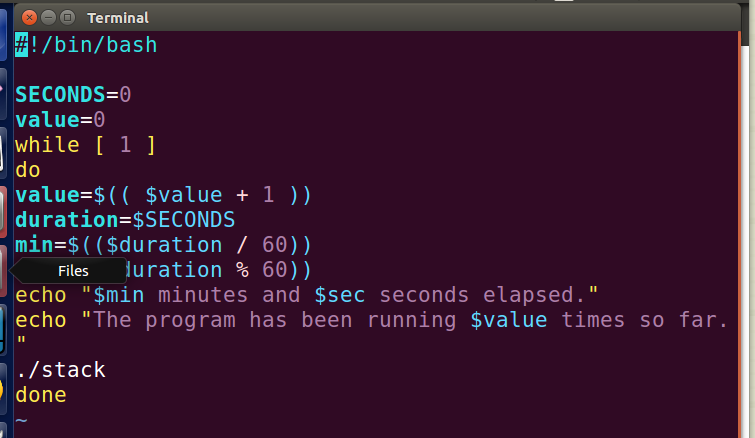
打开地址随机化：



重复运行一下task2，发现出现了分段错误。

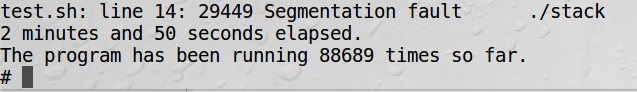


将指导手册中的循环执行的shell脚本编写进test.sh，然后执行



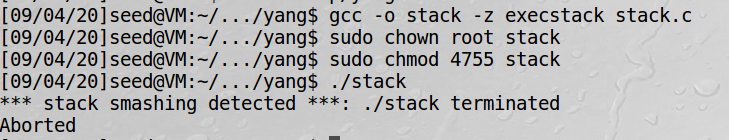


经过88689次的循环执行，终于找到了shellcode的地址。



**2.7 Task 5: Turn on the StackGuard Protection**

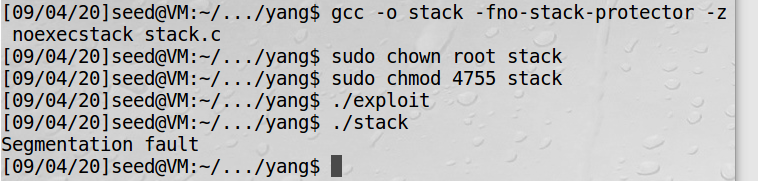
打开栈保护机制，编译程序时应该不使用-fno-stack-protector选项



可以看到程序被中止。

**2.8 Task 6: Turn on the Non-executable Stack Protection**

编译的时候选择栈不可执行机制，重复task2，发生了段错误。栈不可执行机制对栈有边界限制，所以运行task2时会发生段错误，有效的防止了攻击者利用缓冲区溢出进行的攻击活动。



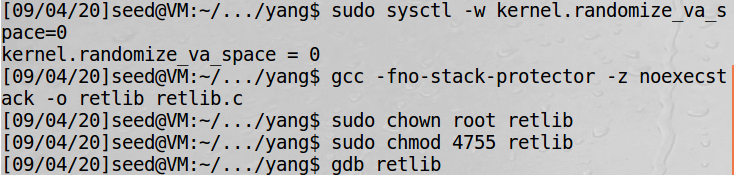
Return-to-libc Attack Lab

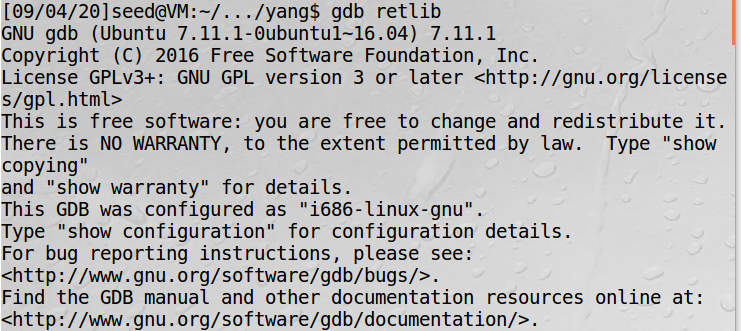
实验报告

57117105 杨哲君

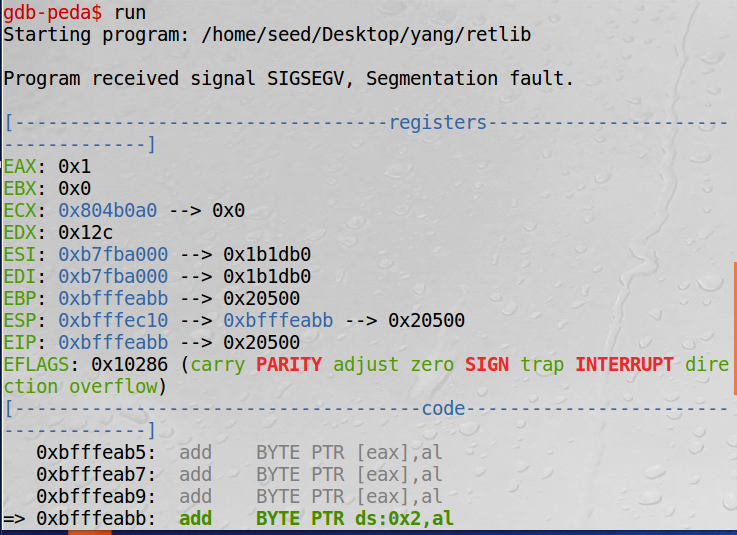
**2.3 Task 1: Finding out the addresses of libc functions**

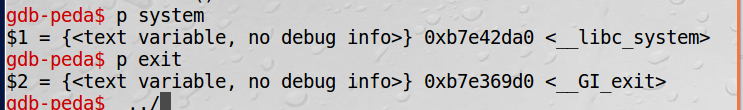
先关闭地址随机化，编译retlib.c，将其设为set-uid程序，使用gdb工具来调试。





按指导手册的命令来调试程序：

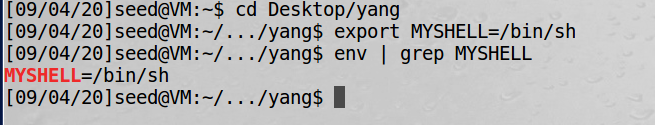




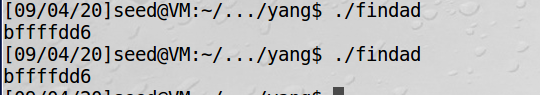
通过指令找到了system函数的地址为0xb7e42da0，exit函数的地址为0xb7e369d0。如果将其设置为非set-uid程序，加载的地址可能不一样，所以要将其设置为set-uid程序，否则地址将会不正确。

**Task 2: Putting the shell string in the memory**

定义一个新的环境变量MYSHELL，包含“/bin/sh”字符串



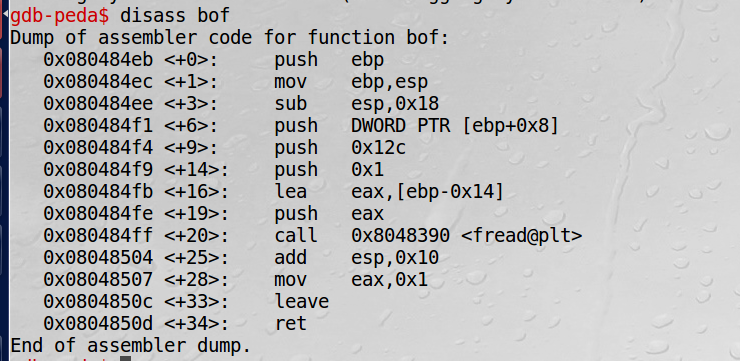
按照指导手册，将打印地址的代码写入findad.c，编译运行。如果关闭了地址随机化，再次运行地址应该是一样的，试验成功。



要注意搜索程序的名称长度应该和retlib.c长度一致，才能搜索到当retlib运行时环境变量的正确地址。

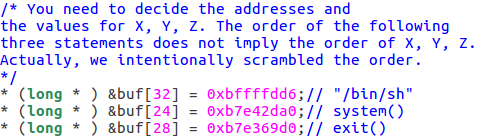
**Task 3: Exploiting the buffer-overflow vulnerability**

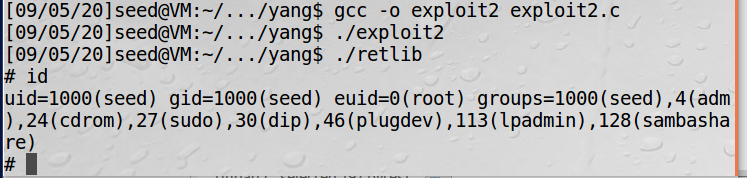
用gdb命令调试bof函数，



Sub esp 0x18表示内存分配给bof函数共24字节的空间，，[ebp-0x14]是buffer的首地址，buffer共有20字节的空间，三个地址一共需要12个字节的空间，为了使system的地址刚好覆盖bof返回地址，应该使前8个字节都填充入nop，从buf[24]开始填充system的地址，剩下两个经过尝试确定是exit地址从buf[28]开始填入，参数/bin/sh从buf[32]开始填入。

将XYZ的值和对应地址写入文件，编译并按照指导手册的命令运行。

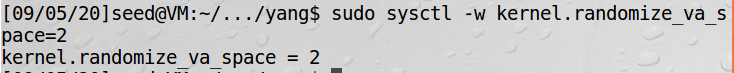




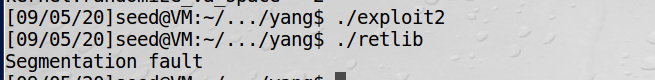
发现获得root权限，说明我们利用缓冲区溢出漏洞获得root权限成功。

**Task 4: Turning on address randomization**

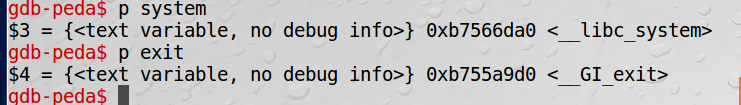
开启随机化：

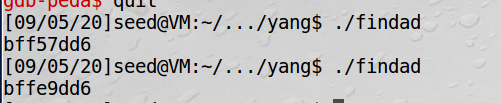


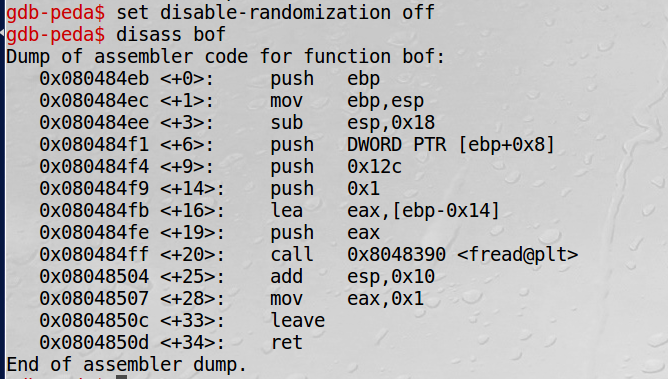
再次运行两个程序，发现出现了段错误。



在六个值中，地址随机化导致了三个地址的值是不正确的，重复运行findad，发现/bin/sh的地址每次都会发生变化。而将gdb中的禁用随机化关掉之后反汇编bof后指令是和之前一样的。







2.7 Task 5: Defeat Shell’s countermeasure

2.8 Task 6: Defeat Shell’s countermeasure without putting zeros in input