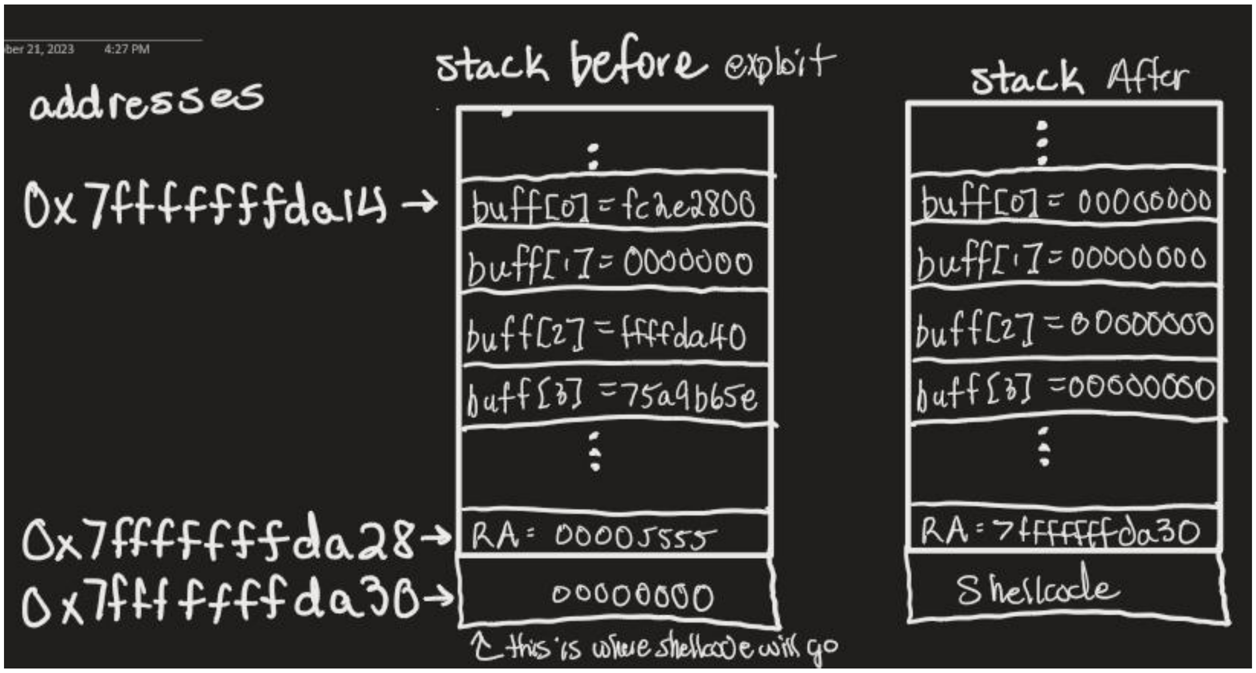
Part 1 Answer was:

```
buf = '\x00' * 20 # just a buffer
print buf + '\x30\xDA\xFF\xFF\xFF\x7F\x00\x00' + shellcode
```

The goal was to find the buffer size and then overflow it so that we rewrite the return address of helper to point to the next available space on the stack, where I will have already placed the shellcode. By inputting a word larger than 10 characters or 10 bytes (the vulnerability), you would be able to write past the designated buffer. I found from gdb when setting breakpoint at line 8 in helper(), that the buffer for buff started at address 0x7fffffffda14 by using the command: 4xg buff. When using the command I got 2 addresses and values associated with them and when inputting words for the program when in gdb, I saw that the words I inputted started to be stored at 0x7fffffffda14 and not the other address that was outputted by that command. The return address of helper() which we know is stored on the stack was found to be 0x7fffffffda28 by using the command "info frames" and reading the address which RIP was located as this would be where the return address of helper() is stored.

We see that the difference in address between the start of the buffer and the RA start is 20 bytes, so the " ' \x00' *20" part of my solution stored in variable buf was to overfill the buffer so that the next character would start to overwrite the return address of helper. The next part of the input which was '\x30\xDA\xFF\xFF\xFF\x7F\x00\x00' is just the address 8 bytes in memory from the location which it is stored. This rewrites the return address of helper so it will not return to main but instead return to that memory location (which is the next location). The last part of my input is the shellcode. Since the return address was changed to point to the location that shellcode is now in, the helper()'s return address just points to the shellcode so it executes the shellcode upon returning.

Basic Diagram: (All numbers in diagram are in hex)


ber 21, 2023 4:27 PM
addresses
0x7fffffffda14 →
0x7fffffffda28 →
0x7fffffffda30 →
stack before exploit
buff[0] = fc2e2800
buff[1] = 0000000
buff[2] = ffffda40
buff[3] = 75a9b65e
RA = 00005555
00000000
this is where shellcode will go
stack After
buff[0] = 00000000
buff[1] = 00000000
buff[2] = 00000000
buff[3] = 00000000
RA = 7fffffffda30
Shellcode