unicorn模拟执行学习

前言

`unicorn` 是一个模拟执行软件，用于模拟执行各种平台的二进制文件，前几天在 `twitter` 上看到一篇文章，这里做个记录。

正文

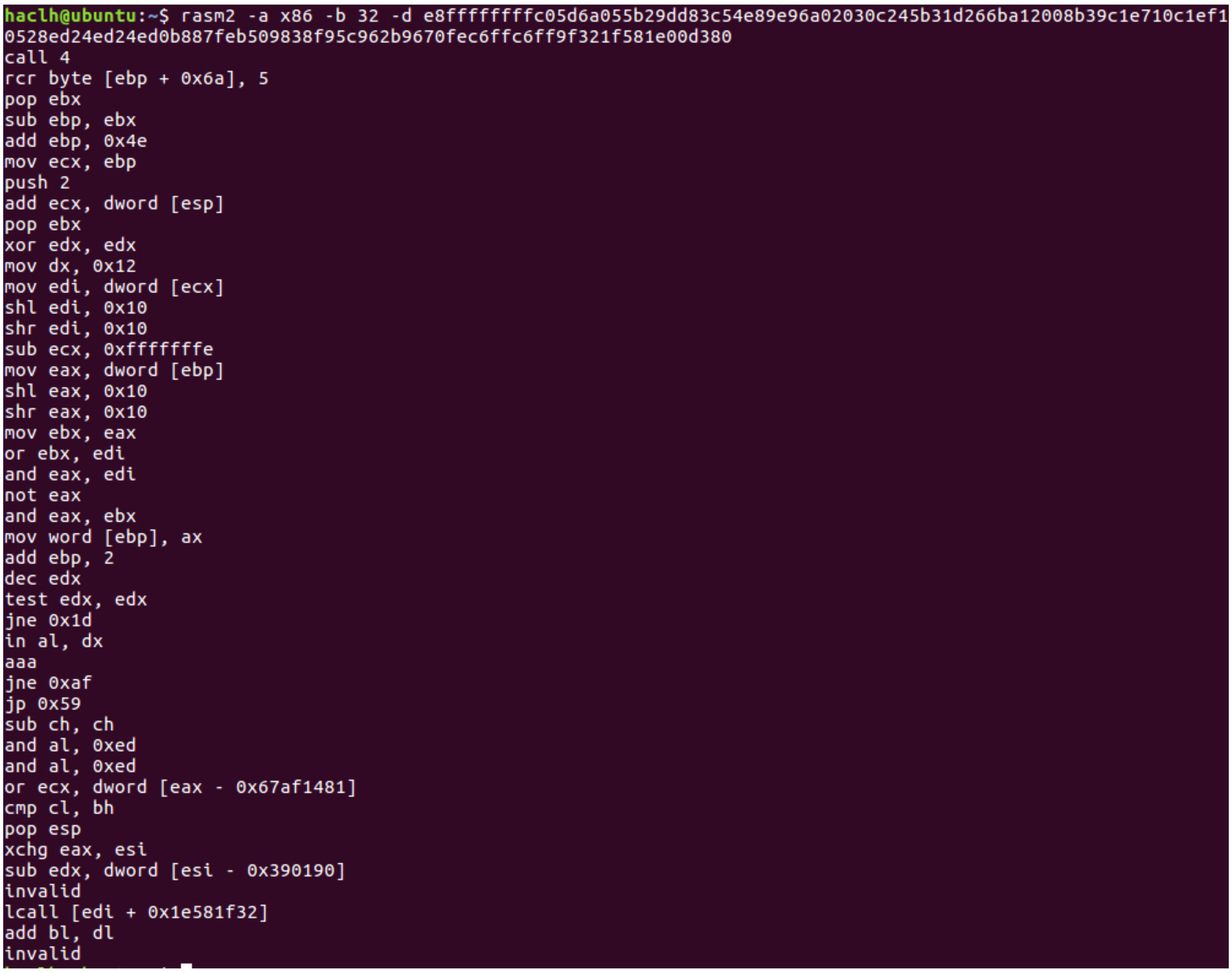
记录系统调用

首先是一个简单的示例

```
e8ffffffffc05d6a055b29dd83c54e89e96a02030c245b31d266ba12008b39c1e710c1ef1081e9feffffff8b4500c1e010c1e81089c309fb21f8f7d021d86689450083c5024a85d20f85cfffffffec37755d7a0528ed24ed2
```

这是一段 `x86_32` 的 `shellcode`，可以用 `radare2` 反汇编它

```
rasm2 -a x86 -b 32 -d e8ffffffffc05d6a055b29dd83c54e89e96a02030c245b31d266ba12008b39c1e710c1ef1081e9feffffff8b4500c1e010c1e81089c309fb21f8f7d021d86689450083c5024a85d20f85cffffff
```



```
haclh@ubuntu:~$ rasm2 -a x86 -b 32 -d e8ffffffffc05d6a055b29dd83c54e89e96a02030c245b31d266ba12008b39c1e710c1ef1
0528ed24ed24ed0b887feb509838f95c962b9670fec6ffc6ff9f321f581e00d380
call 4
rcr byte [ebp + 0x6a], 5
pop ebx
sub ebp, ebx
add ebp, 0x4e
mov ecx, ebp
push 2
add ecx, dword [esp]
pop ebx
xor edx, edx
mov dx, 0x12
mov edi, dword [ecx]
shl edi, 0x10
shr edi, 0x10
sub ecx, 0xfffffffe
mov eax, dword [ebp]
shl eax, 0x10
shr eax, 0x10
mov ebx, eax
or ebx, edi
and eax, edi
not eax
and eax, ebx
mov word [ebp], ax
add ebp, 2
dec edx
test edx, edx
jne 0x1d
in al, dx
aaa
jne 0xaf
jp 0x59
sub ch, ch
and al, 0xed
and al, 0xed
or ecx, dword [eax - 0x67af1481]
cmp cl, bh
pop esp
xchg eax, esi
sub edx, dword [esi - 0x390190]
invalid
lcall [edi + 0x1e581f32]
add bl, dl
invalid
```

这里的目标是记录他的系统调用，在 `32` 中使用 `int 80` 来执行系统调用，所以我们在 执行 `int 80` 前 记录它的 寄存器信息，就可以记录系统调用了。

```
from unicorn import *
from unicorn.x86_const import *



shellcode = "e8ffffffffc05d6a055b29dd83c54e89e96a02030c245b31d266ba12008b39c1e710c1ef1081e9feffffff8b4500c1e010c1e81089c309fb21f8f7d021d86689450083c5024a85d20f85cfffffffec37755d

BASE = 0x400000
STACK_ADDR = 0x0
STACK_SIZE = 1024*1024

mu = Uc (UC_ARCH_X86, UC_MODE_32)
```

```
mu.mem_map(BASE, 1024*1024)
mu.mem_map(STACK_ADDR, STACK_SIZE)


mu.mem_write(BASE, shellcode)
mu.reg_write(UC_X86_REG_ESP, STACK_ADDR + STACK_SIZE/2)

def syscall_num_to_name(num):
    syscalls = {1: "sys_exit", 15: "sys_chmod"}
    return syscalls[num]

def hook_code(mu, address, size, user_data):
    #print('>>> Tracing instruction at 0x%x, instruction size = 0x%x' %(address, size))

    machine_code = mu.mem_read(address, size)
    if machine_code == "\xcd\x80":

        r_eax = mu.reg_read(UC_X86_REG_EAX)
        r_ebx = mu.reg_read(UC_X86_REG_EBX)
        r_ecx = mu.reg_read(UC_X86_REG_ECX)
        r_edx = mu.reg_read(UC_X86_REG_EDX)
        syscall_name = syscall_num_to_name(r_eax)

        print "--------------"
        print "We intercepted system call: "+syscall_name

        if syscall_name == "sys_chmod":
            s = mu.mem_read(r_ebx, 20).split("\x00")[0]
            print "arg0 = 0x%x -> %s" % (r_ebx, s)
            print "arg1 = " + oct(r_ecx)
        elif syscall_name == "sys_exit":
            print "arg0 = " + hex(r_ebx)
            exit()

        mu.reg_write(UC_X86_REG_EIP, address + size)

mu.hook_add(UC_HOOK_CODE, hook_code)

mu.emu_start(BASE, BASE-1)
```

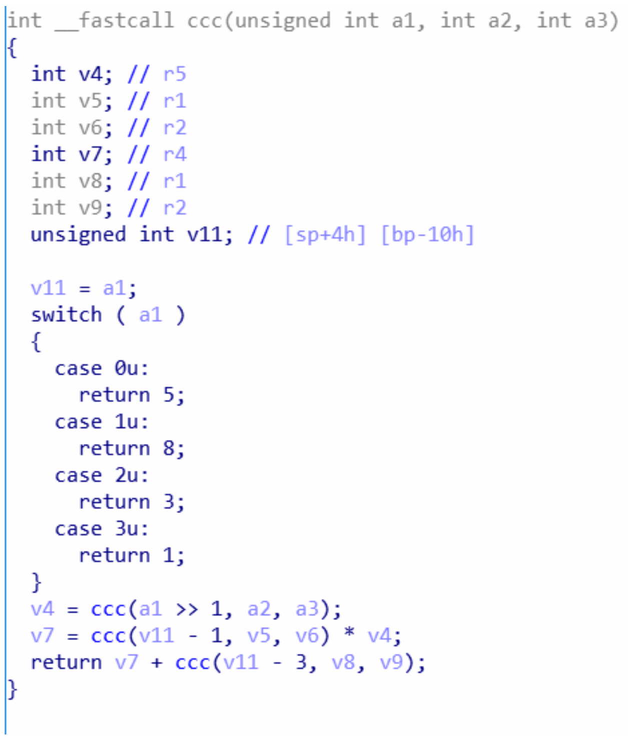
关键就是使用 `mu.hook_add`，使得在 `unicorn` 执行一条指令之前会先执行 `hook_code` 并且传入了与程序运行状态相关的参数，便于我们对程序状态进行操纵。在这里就是获取了寄存器的值，然后根据系统调用号解析参数。

ARM代码模拟执行

测试程序位于

```
http://t.cn/RQ6viS6
```

其实就是执行一个递归函数，最后打印返回值



```
int __fastcall ccc(unsigned int a1, int a2, int a3)
{
  int v4; // r5
  int v5; // r1
  int v6; // r2
  int v7; // r4
  int v8; // r1
  int v9; // r2
  unsigned int v11; // [sp+4h] [bp-10h]

  v11 = a1;
  switch ( a1 )
  {
    case 0u:
      return 5;
    case 1u:
      return 8;
    case 2u:
      return 3;
    case 3u:
      return 1;
  }
  v4 = ccc(a1 >> 1, a2, a3);
  v7 = ccc(v11 - 1, v5, v6) * v4;
  return v7 + ccc(v11 - 3, v8, v9);
}
```

我们的目标是加速程序的执行，可以加速的原理在于，这里是递归调用，对于的参数，返回值确定，所以我们就可以对已经执行过的参数，直接设置返回值，进而加速程序的运行。

```
from unicorn import *
from unicorn.arm_const import *
import struct

def read(name):
    with open(name) as f:
        return f.read()

def u32(data):
    return struct.unpack("I", data)[0]

def p32(num):
    return struct.pack("I", num)


mu = Uc(UC_ARCH_ARM, UC_MODE_LITTLE_ENDIAN)


BASE = 0x10000
STACK_ADDR =
STACK_SIZE = 1024*10240x300000

mu.mem_map(BASE, 1024*1024)
mu.mem_map(STACK_ADDR, STACK_SIZE)


mu.mem_write(BASE, read("./task4"))
mu.reg_write(UC_ARM_REG_SP, STACK_ADDR + STACK_SIZE/2)

instructions_skip_list = []

CCC_ENTRY = 0x000104D0
CCC_END = 0x00010580

stack = []                                          # Stack for storing the arguments
d = {}                                              # Dictionary that holds return values for given function arguments

def hook_code(mu, address, size, user_data):
    #print('>>> Tracing instruction at 0x%x, instruction size = 0x%x' %(address, size))
    if address == CCC_ENTRY:                        # Are we at the beginning of ccc function?
        arg0 = mu.reg_read(UC_ARM_REG_R0)           # Read the first argument. it is passed by R0
        if arg0 in d:                               # Check whether return value for this function is already saved.
            ret = d[arg0]
            mu.reg_write(UC_ARM_REG_R0, ret)        # Set return value in R0
            mu.reg_write(UC_ARM_REG_PC, 0x105BC)    # Set PC to point at "BX LR" instruction. We want to return from fibonacci function

        else:
            stack.append(arg0)                      # If return value is not saved for this argument, add it to stack.

    elif address == CCC_END:
        arg0 = stack.pop()                          # We know arguments when exiting the function

        ret = mu.reg_read(UC_ARM_REG_R0)            # Read the return value (R0)
        d[arg0] = ret                               # Remember the return value for this argument

mu.hook_add(UC_HOOK_CODE, hook_code)
mu.emu_start(0x00010584, 0x000105A8)
return_value = mu.reg_read(UC_ARM_REG_R1)           # We end the emulation at printf("%d\n", ccc(x)).
print "The return value is %d" % return_value
```

关键点，用一个数组存储了 `参数：返回值` 对， 从而规避一些冗余的运算。

参考

http://eternal.red/2018/unicorn-engine-tutorial/

来源： https://www.cnblogs.com/hac425/p/9416898.html