某个openwrt二次开发的设备lua字节码逆向初探
概述
lua字节码编译方法
luadec工具使用
32bit的luadec环境搭建
某设备lua字节码反编译问题
lua-5.1字节码头部特征
定位问题step by step
不同环境下的luac编译代码
gdb调试
Lua源码修改
修改Integral flag定义
总结
参考文章

# 某个openwrt二次开发的设备lua字节码逆向初探

## 概述

目前越来越多的嵌入式设备会基于openwrt进行二次开发，openwrt中的web管理采用的是luci统一配置接口来实现，其中的各种配置功能都是在lua脚本中实现。lua和python一样也是一种解释型语言，同样也可以编译成字节码来提高运行的效率，或者是处于代码保护的目的进行编译的。

## lua字节码编译方法

通常常见的编译方法是使用luac或者luajit，luac通常来说更加常见，通过拉取lua官方的仓库或者下载tar包，直接编译就行了，lua也支持跨平台，windows、linux、macos上都可以编译和使用。

```
wget http://www.lua.org/ftp/lua-5.1.5.tar.gz
tar xvf lua-5.1.5.tar.gz
cd lua-5.1.5 && make all
```

编译字节码使用luac工具来进行，例如lua源码为：

```
local con = 66;   -- 1.lua
```

使用luac编译：

```
luac -o out.lua 1.lua
```

最终得到的out.lua就是lua字节码的格式，可以看到大多数都是二进制数据，头部有Lua字符串特征。

```
y@y ~/iot/source_code/luadec$ file out.lua
                  ✹ ⋆master
out.lua: Lua bytecode, version 5.1
y@y ~/iot/source_code/luadec$ hexdump -C out.lua
                  ✹ ⋆master
00000000  1b 4c 75 61 51 00 01 04  04 04 08 00 07 00 00 00  |.LuaQ...........|
00000010  40 31 2e 6c 75 61 00 00  00 00 00 00 00 00 00 00  |@1.lua..........|
00000020  00 02 02 02 00 00 00 01  00 00 00 1e 00 80 00 01  |................|
00000030  00 00 00 03 00 00 00 00  00 80 50 40 00 00 00 00  |..........P@....|
00000040  02 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00  |................|
00000050  04 00 00 00 63 6f 6e 00  01 00 00 00 01 00 00 00  |....con.........|
00000060  00 00 00 00                                       |....|
00000064
```

那么对于此类的lua字节码文件，如果需要对其进行反编译的话，通过采用的工具有luadec和unluac，引用@feicong大佬在安全客上的文章介绍：

> **luadec**与**unlua**是最流行的Luac反汇编与反编译工具，前者使用C++语言开发，后者使用Java语言，这两个工具都能很好的还原与解释Luac文件，但考虑到Lua本身采用C语言开发，并且接下来打算编写**010 Editor**编辑器的Luac.bt文件格式模板，**010 Editor**的模板语法类似于C语言，为了在编码时更加顺利，这里分析时主要针对**luadec**。

那么本文介绍的主要是luadec工具。

# luadec工具使用

luadec是github上开源的项目，git clone在ubuntu下直接编译就能够得到完整的环境：

```
git clone https://github.com/viruscamp/luadec
cd luadec
git submodule update --init lua-5.1
cd lua-5.1
make macosx
cd ../luadec
make LUAVER=5.1
```

- 由于不同版本的lua字节码是有差异的，所以如果需要反编译luac字节码文件的话，需要对应好版本，通常采用file命令就能发现字节码是什么版本的。

如下在luadec目录下，生成了luadec的二进制文件。

## 32bit的luadec环境搭建

由于luac字节码是区分32位和64位的，因此luadec根据差异也需要编译成32位或者64位，如果需要编译成32位的环境，需要在lua-5.1/src下的Makefile以及luadec下的Makefile分别加上-m32参数：

```
MYCFLAGS=-m32
MYLDFLAGS=-m32
MYLIBS=
MYOBJS=
```

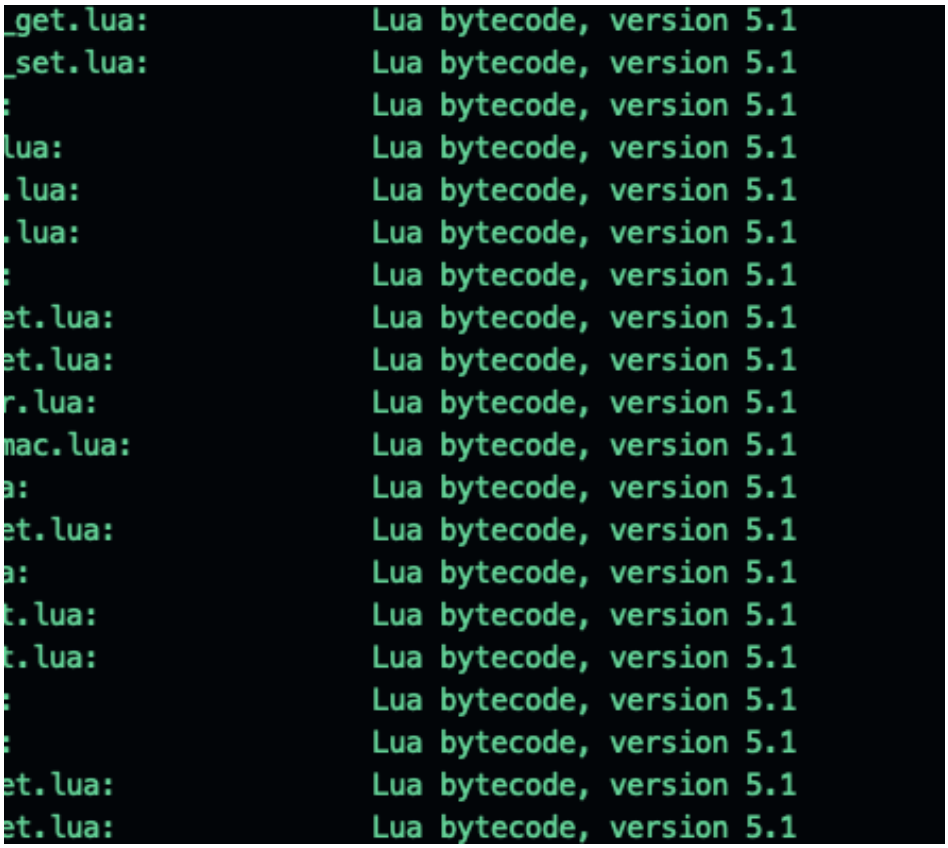
同时需要安装gcc和g++ 32位库的支持（针对ubuntu 64bit的环境）：

```
sudo apt-get install gcc-multilib g++-multilib
sudo apt install libreadline-dev:i386
```

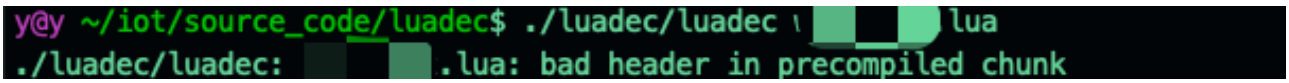
那么最终得到的就是32bit的luadec。上文中out.lua文件就是基于32bit生成的。

# 某设备lua字节码反编译问题

在漏洞挖掘中遇到了某个路由器设备，发现是基于openwrt开发的，同时lua代码很多都被编译成字节码了：




采用的是lua5.1版本编译的，那么使用上文的方法就可以搭建出对应版本的luadec环境，此时如果直接对luac字节码反编译的话，会提示`bad header in precompiled chunk`错误：



这里的意思很明显是lua头部字段出现了问题，那么就可以大概推断设备上的lua字节码的解释器或者编译器被修改过，导致出现这类的问题。

## lua-5.1字节码头部特征

在网上找了关于5.1版本的luac字节码头部信息的字段定义，总共12个字节：

A binary chunk consist of two parts: a header block and a top-level function. The header portion contains 12 elements:

**Header block of a Lua 5 binary chunk**
Default values shown are for a 32-bit little-endian platform with IEEE 754 doubles as the number format. The header size is always 12 bytes.

| | |
|---|---|
| **4 bytes** | Header signature: ESC, “Lua” or 0x1B4C7561<br>• Binary chunk is recognized by checking for this signature |
| **1 byte** | Version number, 0x51 (81 decimal) for Lua 5.1<br>• High hex digit is major version number<br>• Low hex digit is minor version number |
| **1 byte** | Format version, 0=official version |
| **1 byte** | Endianness flag (default 1)<br>• 0=big endian, 1=little endian |
| **1 byte** | Size of int (in bytes) (default 4) |
| **1 byte** | Size of size_t (in bytes) (default 4) |
| **1 byte** | Size of Instruction (in bytes) (default 4) |
| **1 byte** | Size of lua_Number (in bytes) (default 8) |
| **1 byte** | Integral flag (default 0)<br>• 0=floating-point, 1=integral number type |

On an x86 platform, the default header bytes will be (in hex):
`1B4C7561 51000104 04040800`

对于使用luac官方编译的out.lua字节码文件，定义如下：

```
y@y ~/iot/source_code/luadec$ hexdump -C out.lua
                  * *master
00000000  1b 4c 75 61 51 00 01 04  04 04 08 00 07 00 00 00  |.LuaQ...........|

1b 4c 75 61      <-- 头部sign，固定为这个值
51               <-- 表示luac字节码的版本为5.1
00               <-- 表示为官方的版本
01               <-- 大小端表示，1为小端
04               <-- int类型的长度，为4
04               <-- size_t类型的长度，为4
04               <-- 指令类型的长度，为4
08               <-- lua_numbers类型的长度，为8
00               <-- 0表示浮点类型数字，1表示整数类型数字
```

通过比较使用luac官方编译出来的和设备上的lua字节码文件进行比较，发现头部有一个字节改动过，即Integral flag，正常这个值为0或者1，但是这里却是4，所以很明显，这个luac字节码文件是用改动过的lua编译器编译出来的：


由于这个值默认情况下为0，我们可以手动将4改成0，然后使用luadec进行反编译查看结果。最终发现，有些lua字节码文件是可以被反编译出来的（针对一两个字节数少的字节码文件），而有些是提示了bad constant in precompiled chunk错误。那么可以猜测constant数据类型被修改过，而其他类型没有被修改。

```
y@y ~/iot/source_code/luadec_src$ ./luadec/luadec/luadec transmit_set.lua
./luadec/luadec/luadec: transmit_set.lua: bad constant in precompiled chunk
```

# 定位问题step by step

对于遇到的此类问题，通常的解决方法是先编译一下最简单的lua文件，通过修改lua脚本源码配合gdb调试来定位问题。

## 不同环境下的luac编译代码

首先我们定义一个简单的语句：

```
local con = 1;
```

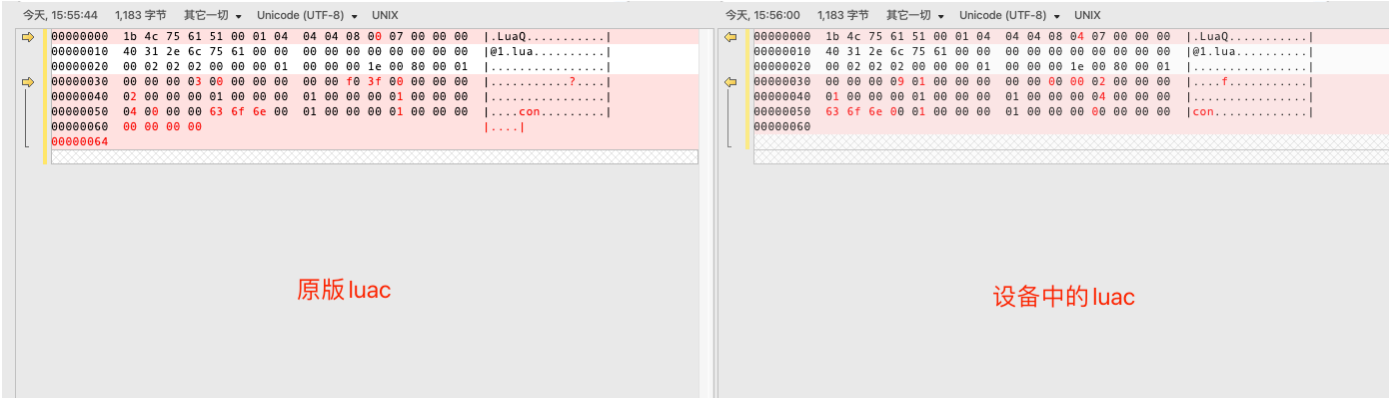
在官方的环境下编译，得到的字节码如下：

```
00000000  1b 4c 75 61 51 00 01 04  04 04 08 00 07 00 00 00  |.LuaQ...........|
00000010  40 31 2e 6c 75 61 00 00  00 00 00 00 00 00 00 00  |@1.lua..........|
00000020  00 02 02 02 00 00 00 01  00 00 00 1e 00 80 00 01  |................|
00000030  00 00 00 03 00 00 00 00  00 00 f0 3f 00 00 00 00  |...........?....|
00000040  02 00 00 00 01 00 00 00  01 00 00 00 01 00 00 00  |................|
00000050  04 00 00 00 63 6f 6e 00  01 00 00 00 01 00 00 00  |....con.........|
00000060  00 00 00 00                                       |....|
00000064
```

使用设备上的luac进行编译，得到的字节码如下：

```
00000000  1b 4c 75 61 51 00 01 04  04 04 08 04 07 00 00 00  |.LuaQ...........|
00000010  40 31 2e 6c 75 61 00 00  00 00 00 00 00 00 00 00  |@1.lua..........|
00000020  00 02 02 02 00 00 00 01  00 00 00 1e 00 80 00 01  |................|
00000030  00 00 00 09 01 00 00 00  00 00 00 00 02 00 00 00  |....f...........|
00000040  01 00 00 00 01 00 00 00  01 00 00 00 04 00 00 00  |................|
00000050  63 6f 6e 00 01 00 00 00  01 00 00 00 00 00 00 00  |con.............|
00000060
```

将两个字节码在beyond compare中进行比较，发现除了头部字段存在差异之外，后面的字段也存在差异：



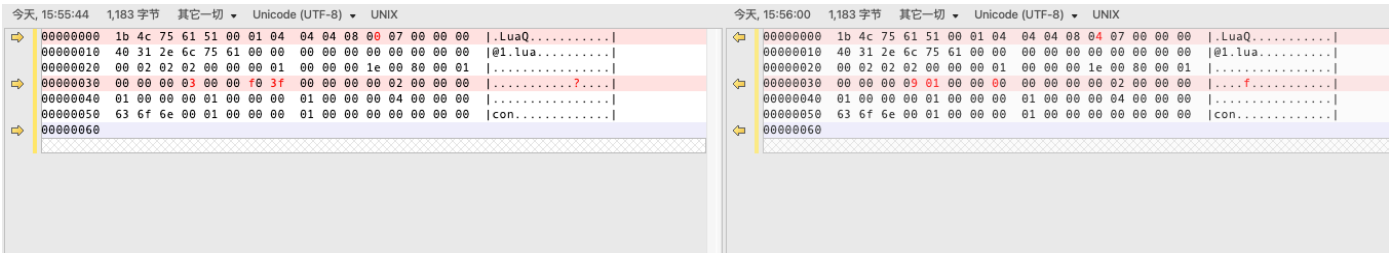

发现设备内的luac编译得到的字节码少了4个字节，同时主要的差异是从0x34开始位置后的8个字节，原版的luac为：

```
03 00 00 00 00  00 00 f0 3f
```

设备中的luac为：

```
09 01 00 00 00
```

经过仔细对比会发现，将原本03后的四个字节去掉，后面的字节进行补齐，会发现后面的字节几乎都是一样的。那么这个03和09很可能是某个操作码。



# gdb调试

为了定位具体的问题，我们可以将设备中的luac字节码文件使用原版lua编译出来的luadec进行gdb调试，定位问题所在。

首先我们需要将字节码中的Integral flag由原来的4改成0（避免bad header in precompiled chunk错误），执行提示bad constant：

```
y@y ~/iot/source_code/luadec_src/luadec$ ./luadec/luadec device.lua
./luadec/luadec: device.lua: bad constant in precompiled chunk
```

在源码中搜索字符串，定位到了LoadConstants函数，代码使用switch case判断了t的值，判断了是否为LUA_TBOOLEAN、LUA_TNUMBER、LUA_TSTRING等类型，如果不存在的话，就返回了这个错误。

```
static void LoadConstants(LoadState* S, Proto* f)
{
 int i,n;
 n=LoadInt(S);
 f->k=luaM_newvector(S->L,n,TValue);
 f->sizek=n;
```

```
for (i=0; i<n; i++) setnilvalue(&f->k[i]);
for (i=0; i<n; i++)
{
 TValue* o=&f->k[i];
 int t=LoadChar(S);
 switch (t)
 {
  case LUA_TNIL:
   setnilvalue(o);
 break;
  case LUA_TBOOLEAN:
   setbvalue(o,LoadChar(S)!=0);
 break;
  case LUA_TNUMBER:
 setnvalue(o,LoadNumber(S));
 break;
  case LUA_TSTRING:
 setsvalue2n(S->L,o,LoadString(S));
 break;
  default:
 error(S,"bad constant");
 break;
 }
}
```

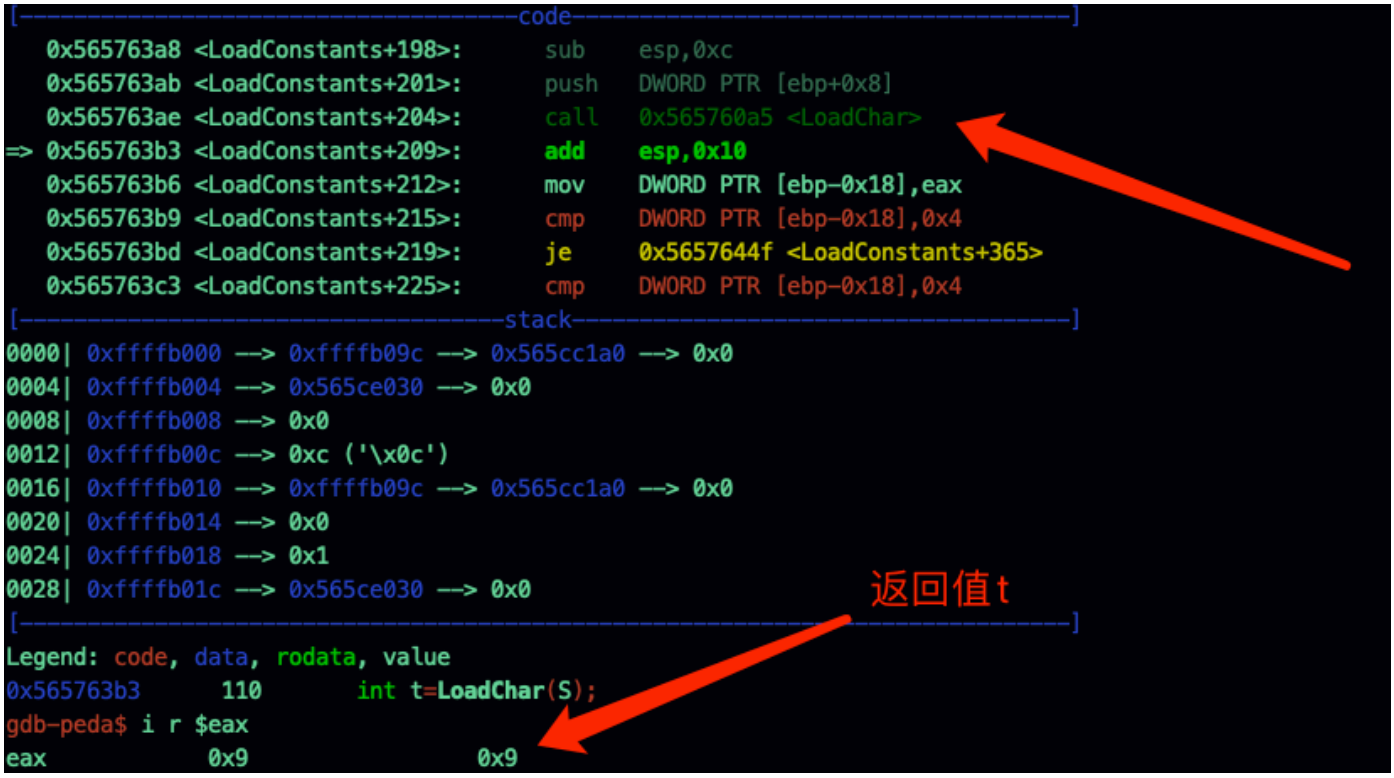
在lua源码中找到这些常量的定义，发现其中数字类型的值为3，刚好在上方我们比较过的原版luac和设备中的luac第一个差异的值就是03，所以可以猜测这个字段用来定义数字类型：

```
68
69    /*
70    ** basic types
71    */
72    #define LUA_TNONE    (-1)
73
74    #define LUA_TNIL    0
75    #define LUA_TBOOLEAN    1
76    #define LUA_TLIGHTUSERDATA  2
77    #define LUA_TNUMBER    3
78    #define LUA_TSTRING    4
79    #define LUA_TTABLE    5
80    #define LUA_TFUNCTION    6
81    #define LUA_TUSERDATA    7
82    #define LUA_TTHREAD    8
83
84
```

如果是这样的话，调试设备中的luac字节码文件时，那么LoadChar(S);获取到的t的值就应该是9，于是使用gdb进行调试，断点下在LoadConstants函数处：

```
gdb ./luadec/luadec -ex 'set args device.lua' -ex 'b LoadConstants'
```

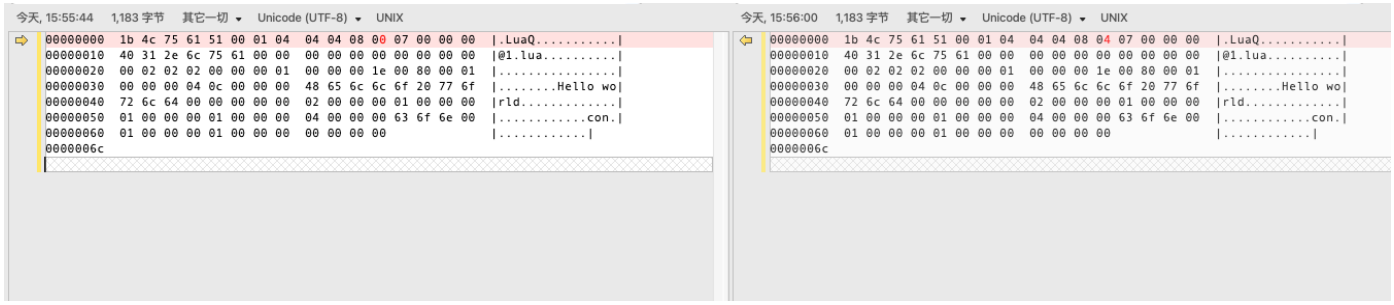
调用完LoadChar函数之后，查看返回值，发现正好是9，最终就会走到bad constant这个分支，于是可以知道这个字节码定义肯定是被修改过的。




我们在知道关于LUA_TNUMBER的值定义被修改过之后，就需要确定一下其他数据类型是否也被修改过。同样也是选择一个最简单的开始分析：

```
local con = "Hello world";
```

对比的情况如下，发现只有Integral flag存在差异：



对比了其他数据类型的发现也是只有Integral flag有变化。所以这里我的研究的重点就是对于LUA_TNUMBER类型数据的逆向和调试。

首先我们还是先拿原版的luac代码，定义了一个con变量为1的情况来调试，看看在LoadConstants函数中进入了LUA_TNUMBER分支之后做了什么。查看源码发现在得到了t的值为3之后，执行了：

```
setnvalue(o,LoadNumber(S));
```

其中LoadNumber的定义如下，继续调用了LoadVar函数，函数返回的值为x，那么LoadVar函数的作用应该就是从S结构体中的某个字段读取内存存储到x变量中：

```
static lua_Number LoadNumber(LoadState* S)
{
 lua_Number x;
 LoadVar(S,x);
 return x;
}
```

LoadVar为宏定义，最终调用的是LoadBlock：

```
#define LoadMem(S,b,n,size) LoadBlock(S,b,(n)*(size))
#define LoadByte(S)   (lu_byte)LoadChar(S)
#define LoadVar(S,x)    LoadMem(S,&x,1,sizeof(x))
```

LoadBlock又调用了luaZ_read函数：

```
static void LoadBlock(LoadState* S, void* b, size_t size)
{
 size_t r=luaZ_read(S->Z,b,size);
 IF (r!=0, "unexpected end");
}
```

该函数定义在lua源码的lzio.c中，b指针的值由z结构体的p指针中取的，m为n传递过来的：

```
/* --------------------------------------------------------------- read --- */
size_t luaZ_read (ZIO *z, void *b, size_t n) {
  while (n) {
    size_t m;
    if (luaZ_lookahead(z) == EOZ)
      return n;  /* return number of missing bytes */
    m = (n <= z->n) ? n : z->n;  /* min. between n and z->n */
    memcpy(b, z->p, m);
    z->n -= m;
    z->p += m;
    b = (char *)b + m;
    n -= m;
  }
  return 0;
}
```

找到Zio结构体的定义，p为当前buffer的指针：

```
/* --------- Private Part ------------------ */

struct Zio {
  size_t n;     /* bytes still unread */
  const char *p;    /* current position in buffer */
  lua_Reader reader;
  void* data;     /* additional data */
  lua_State *L;     /* Lua state (for reader) */
};
```

在gdb中下断点，查看z->p指针指向的位置：

```
gdb-peda$ p z->p
$7 = 0xffffb340 ""
gdb-peda$ p *z
$8 = {
  n = 0x30,
  p = 0xffffb340 "",
  reader = 0x5657b8ec <getF>,
  data = 0xffffb304,
  L = 0x565cc1a0
}
gdb-peda$ hexdump 0xffffb340
0xffffb340 : 00 00 00 00 00 00 f0 3f 00 00 00 00 02 00 00 00   .......?........
gdb-peda$ hexdump 0xffffb340 20
0xffffb340 : 00 00 00 00 00 00 f0 3f 00 00 00 00 02 00 00 00   .......?........
0xffffb350 : 01 00 00 00                                       ....
gdb-peda$ hexdump 0xffffb340 30
0xffffb340 : 00 00 00 00 00 00 f0 3f 00 00 00 00 02 00 00 00   .......?........
0xffffb350 : 01 00 00 00 01 00 00 00 01 00 00 00 04 00         ..............
```

发现正好是字节码文件中03（LUA_TNUMBER类型）后的内存区域，调用memcpy函数时，第二个指针就是这个位置，第三个字节是m的值。

```
=> 0x5657a600 <luaZ_read+76>:   call   0x565560d0 <memcpy@plt>
   0x5657a605 <luaZ_read+81>:   add    esp,0x10
   0x5657a608 <luaZ_read+84>:   mov    eax,DWORD PTR [ebp+0x8]
   0x5657a60b <luaZ_read+87>:   mov    eax,DWORD PTR [eax]
   0x5657a60d <luaZ_read+89>:   sub    eax,DWORD PTR [ebp-0xc]
Guessed arguments:
arg[0]: 0xffffafe0 --> 0xffffb2a8 --> 0x30 ('0')
arg[1]: 0xffffb340 --> 0x0
arg[2]: 0x8
```

那么就很容易理解luaZ_read函数的作用主要是读取映射到缓冲区中的文件内容，回过头来看m的值刚好是LoadNumber函数里传入的，大小是(1)*sizeof(lua_Number)，由于字节码头部字段定义的sizeof lua_Number的大小就是8，因此memcpy函数复制内存的大小就刚好是8字节。

因此LoadNumber(S)函数读取到的就是03后8字节的作为LUA_TNUMBER数据类型的具体大小值（至于为什么为0x3ff还不太清楚）。

根据上面我们对比过的字节码文件，原版的luac字节码是多了4个字节，刚好luaZ_read函数读取的是8字节的大小，而且设备中的luac中09字节码后面跟的内容也是LUA_TNUMBER数据类型的具体大小值，因此我们可以猜测是不是修改过的luac调用luaZ_read函数时，第三个参数就是被故意修改成了4字节。

# Lua源码修改

于是我们可以通过修改lua代码来验证我们的猜想。首先在lua-5.1/src目录下的lundump.c文件，找到LoadConstants函数中，case为LUA_TNUMBER的分支，将分支的类型值修改为9：

```
-    case LUA_TNUMBER:
+    case 9:
         setnvalue(o,LoadNumber(S));
```

同时修改LoadNumber函数中x的定义为4字节（字节码头字段定义的sizeof size_t的大小为4字节）：

```
static lua_Number LoadNumber(LoadState* S)
{
- lua_Number x;
+ size_t x;
 LoadVar(S,x);
 return x;
}
```

修改好之后，重新编译lua和luadec，同时还是先手动修改Integral flag的值从4改成0。使用luadec重新反编译，发现完全正常！能够完整还原LUA_TNUMBER类型的数据：

```
y@y ~/iot/source_code/luadec$ ./luadec/luadec device.lua
-- Decompiled using luadec 2.2 rev: 895d923 for Lua 5.1 from https://github.com/viruscamp/luadec
-- Command line: device.lua

-- params : ...
-- function num : 0
local con = 1
```

因此就印证了我们的猜想：设备中的LUA_TNUMBER字节码类型值由3改成了9，同时修改了LoadNumber函数中的x值类型为4字节。尝试了使用luadec反编译设备上文件系统中其他的lua字节码文件，发现都能够正常反编译。

```
y@y ~/iot/source_code/luadec$ ./luadec/luadec transmit_set.lua
-- Decompiled using luadec 2.2 rev: 895d923 for Lua 5.1 from https://github.com/viruscamp/luadec
-- Command line: transmit_set.lua

-- params : ...
-- function num : 0
module("or                    ", package.seeall)
require("                    ")
require("                    ")
require("luci.sys")
require("luci.json")
local uci = (                ci_wrapper).get_uci_cursor)()
local get_token = function(httpurl, isForce)
  -- function num : 0_0
  local command = nil
  if isForce then
    command =                n -a " .. httpurl .. " -f 2>/dev/null"
  else
    command =                -a " .. httpurl .. " 2>/dev/null"
  end
```

## 修改Integral flag定义

为了免于每次都需要对头部的Integral flag定义手动修改为0，在源码中也可以进行修改，同样是在lua-5.1的源码目录下，找到lundump.c文件，修改luaU_header中最后一个字节的定义即可，直接将其返回4：

```
void luaU_header (char* h)
{
 int x=1;
 memcpy(h,LUA_SIGNATURE,sizeof(LUA_SIGNATURE)-1);
 h+=sizeof(LUA_SIGNATURE)-1;
 *h++=(char)LUAC_VERSION;
 *h++=(char)LUAC_FORMAT;
 *h++=(char)*(char*)&x;                          /* endianness */
 *h++=(char)sizeof(int);
 *h++=(char)sizeof(size_t);
 *h++=(char)sizeof(Instruction);
 *h++=(char)sizeof(lua_Number);
- *h++=(char)(((lua_Number)0.5)==0);           /* is lua_Number integral? */
+ *h++=(char)4;
}
```

重新反编译之后，一切正常。

## 总结

越来越多了嵌入式设备中的lua字节码做了代码保护，但是像此类保护，只要研究者对luac字节码的机制比较熟悉的话，其实还是比较容易被破解的。除了本文介绍到的这种情况，还有像其他设备是修改了luac的各种操作码，本文只是基于研究目的来抛砖引玉的，感兴趣的读者可以自行寻找目标和尝试。

## 参考文章

https://www.anquanke.com/post/id/87006

https://blog.csdn.net/u012787710/article/details/53729659