Lua as a Configuration And Data Exchange Language

Marc Balmer <mbalmer@NetBSD.org>
The NetBSD Foundation, Micro Systems Marc Balmer
January 4th, 2013

Abstract

When Lua was created, it emerged from a previous language, Sol, which had been created specifically to describe data and to be used as a data entry language[1].

Lua uses a very concise syntax to describe data and, being a library, it is easy to integrate Lua in to existing programs. „*Lua is a scripting language in the original meaning of the expression. A language to control other components, usually written in another language*".[2]

The data description features and the easy integration into existing software make Lua a candidate as a program configuration and data exchange language.

This article is not a complete description of the Lua language, it merely looks at the language from the perspective of using it for configuration files and as a data exchange language. See [1] for the full language reference.

1 A first look at configuration and data files in Lua

Before looking at implementation details and discussing security and other runtime issues, a few examples of configuration and data definition files are shown to give an impression how these files actually look.

In Lua, variables do not have a type and thus don't need to be declared before use. It's actually the value that has a type, so by assigning a number to a variable it becomes a numeric variable, by assigning text to it, the variable becomes a string. Unlike in other languages, functions are first-class data values and as such can be assigned to variables, passed to functions, returned as results etc.

A piece of Lua code is called a "chunk" and a chunk need not to be a complete program. A chunk is always a piece of text that is compiled to bytecode before it gets executed. Lua has one of the fastest compilers.

1.1 A simple configuration file

In the most simple form, variable assignments are used to define configuration parameters. Any text after two dashes (−−) is a comment and is ignored. String parameters have to be enclosed in single or double quotes.

```
-- Define a network interface

device = 're0'
address = '192.168.1.1'
```

The example above creates two global variables, *device*, and, *address* and sets their values. These variables can then be accessed from C using the Lua C API (more on that later).

Global variables in Lua are actually entries in a table named _G, a concept that becomes important when looking at secuity.

Lua allows for multiple assignments, so the above example could also be written as

```
-- Define a network interface

device, address = 're0', '192.168.1.1'
```

[1] See [2]

[2] Ierusalimschy, Roberto, in [4]

1.2 Using the Lua table syntax

Tables are the only data structure in Lua. Every variable, function etc. resides in a table, even global variables are in a special table _G. To create an empty table, assign {} to a variable, then access members using the `var[index]` syntax. You can also write `var.index`, which is a syntactic sugar for `var['index']` (note that the index is always a string when this notation is used).

You can initialize a table when it's created by specifiying a comma-separated list of values, as is used in the next example:

```
-- Define a network interface, using table syntax

interface = {
    device = 're0',
    address = '192.168.1.1'
}
```

If the table items are not named they will get an implicit integer index, starting at 1:

```
interface = {
    're0',
    '192.168.1.1'
}
```

1.3 Dynamic configuration

A configuration file using Lua is actually a Lua program and as such can make use of constructs like if, then, for, etc. to create a more dynamic configuration.

```
-- Define ten network interfaces

interfaces = {} -- Define an empty table

for n = 1, 10 do
    interfaces[n] = {
        device = 'gif' .. n,
        address = '192.168.1.' .. n
    }
end
```

1.4 More complex configuration

When Lua is used for configuration files, it is usually not possible to call library functions, unless they are explicetely allowed.

If, however, calling library functions is allowed (and the host program can exactly define which functions can be called) more complex configuration scenarios are possible.

The following example assumes a table ReceiptPrinter that has an element ['Device'] that defines the tty device used to print text on the receipt printer. A receipt printer simulator, xrpr, exists that opens a pseudo TTY device to receive data and outputs the name of the PTY device on its standard output. A more complex Lua configuration could start said simulator and pass the PTY name to the host program.

```
-- Start a program that uses posix open pt and
-- hands the device name on stdout

function pty(command)
    local pipe = io.popen(command, 'r')
    local pty = pipe:read("*l")
    pipe:close()
    return pty
end

ReceiptPrinter.Device = pty('xrpr -geometry +816+180')
```

Needless to say that such configurations can be problematic and should be used only with great care. But then, `rm`, can also be a dangerous command...

How far one goes with Lua configuration files, i.e. how much freedom of expression is given to the users, is entirely controlled by the host program and a sane approach is to disallow everything by default and specifically enable those features needed (e.g. whether calling libraries is allowed or not, and if it is allowed, which functions are allowed).

1.5 Storing simple data

Storing data is done by writing a Lua file in more or less the same form as the configuration files shown above. As Lua chunks are text strings, writing such a file is very easy and can be done easily in any programming language. Series of values can be stored as a table:

```
values = {
    4711,
    1291,
    42,
    68302,
    64738
}
```

Lua is blindingly fast at parsing such tables with constructors, even for huge amounts of data. It beats the execution speed of similar constructs in Perl or Python by a factor of 6 – 8.

1.6 Storing more complex data

More complex, or rather, more structured data, can be expressed in Lua by nesting tables. As tables are just values like numbers, strings, or, functions, they can themselves be elements of other tables:

```
developers = {
    { login = 'mbalmer', name = 'Marc Balmer' },
    { login = 'lneto', name = 'Lourival Neto' }
}
```

While above example creates a table of developers with an integer index, a string index can be used as well:

```
developers = {
    mbalmer = { shell = '/bin/ksh', name = 'Marc Balmer' },
    lneto = {shell = '/sbin/nologin', name = 'Lourival Neto' }
}
```

Tables can be nested to an arbitrary level.

2 How Lua programs are executed

Lua is an interpreted language and Lua programs are first compiled to bytecode which is then executed by a bytecode interpreter. Lua is typeless as variables do not have a type, but values do. The only real data structure in Lua is the table, everything else must be built using tables. There is one special table, called _G, which is the „global" environment. Calling a global function, accessing or creating a global variable actually means accessing an element in the _G table.

```
theAnswer = 42
```

is equivalent to

```
_G['theAnswer'] = 42
```

As functions are first-class data values in Lua, the same applies to functions:

```
function foo ()
    return 'bar'
end
```

is equivalent to

```
_G['foo'] = function return 'bar' end
```

Whether the global environment _G contains any elements or not is at the discretion of the host program, that creates the Lua state. When _G is empty, it does not contain any functions, thus not even the following simple function is possible:

```
print('hello, world!')
```

Since the function `print` evaluates to `_G['print']` and _G is empty. Lua refers to this technique, creating Lua states with empty global states, as sandboxing. Creating sandboxes is not limited to programs using the Lua C API, sandboxes can also be created in Lua. See http://lua-users.org/wiki/SandBoxes for more information about Lua sandboxes.

3 Processing configuration and data files using the Lua C API

The basic operation when using Lua for configuration or data files is as follows:

- Create an empty Lua state
- Define allowed functions in the state (aka define the „sandbox")
- Load and compile the Lua chunk (the configuration or data file)
- Maybe do some security checks on the bytecode, e.g. to disable enless loops or other dangerous constructs
- Execute the compiled Lua chunk, maybe with a limited execution count
- Retrieve the desired values from the Lua state

Lua passes all parameters on a virtual stack, before calling a Lua function, parameters are pushed to the stack, after the function returns, return values are popped from the stack. Functions exist to determine the type of a variable, to access table elements, to call functions etc. The Lua C API is rich and functional, and easy to use.

3.1 Creating an empty Lua state

An initially empty Lua state is created using the lua_newstate() function:

lua_State *L = lua_newstate()

To open a module like e.g. the 'string' module, the luaopen_module() function is used:

luaopen_*module*(L)

3.2 Loading and calling a Lua chunk

There are two functions available for loading and compiling a Lua chunk, one to read the Lua chunk from a file, one to read it from memory.

- int luaL_dofile(lua_State *L, const char *filename)
- int luaL_dostring(lua_State *L, const char *str)

Both functions place the compiled Lua chunk on the Lua virtual stack, so that the function can be called right away using the lua_call() or lua_pcall() function:

- Push parameters on the stack
- Use lua_call(lua_State *L, int index)
- or lua_pcall(lua_State *L, int index)
- Pop return values from the stack

3.3 Accessing variables in the Lua state

Let's assume an element 'Device' is to be retrieved from the global table 'ReceiptPrinter'. The following function could then be used:

```
static char *device;

static char *
cfg_string(lua_State *L, char *table, char *elem, char *dflt)
{
    char *r;

    lua_getglobal(L, table);
    lua_getfield(L, -1, elem);
    if (lua_isnil(L, -1)) {
        lua_pop(L, 2);
        return dflt;
    }
    if (!lua_isstring(L, -1))
        errx(1, "string expected for %s.%s", table, elem);
    r = strdup((char *)lua_tostring(L, -1));
    lua_pop(L, 2);
    return r;
}

device = cgf_string(L, "ReceiptPrinter", "Device", "/dev/null");
```

When the Lua state is no longer used, it can be closed (and all its resources freed) by calling lua_close().

4 Conclusion

Every now and then the question arises which format to use for configuration data or to store data in a file. For NetBSD, candidates are probably problib, sqlite, tcl, plain text files, or, Lua.

Lua is available in the NetBSD base system and is very easy to integrate into existing programs. Lua code is even easier to produce, as Lua chunks are text-only. Processing Lua chunks can be made very secure by using sandboxing techniques.

Lua as a configuration and data exchange language has proven to be a good choice in many commercial and non-commercial software packages. It is certainly an alternative to be considered when choosing a configuration file format, configuration file parser, or data file format.

Lua chunks have even be used as protocol messages for client-server protocols over the network, see [3] for details.

5 References

[1] Ierusalimschy, Roberto; De Figueiredo, Luiz Henrique; Celes, Waldemar: *Lua 5.1 Reference Manual*. Lua.org, Rio de Janeiro, 2006.

[2] Ierusalimschy, Roberto; De Figueiredo, Luiz Henrique; Celes, Waldemar: *The evolution of an extension language: a history of Lua*. SBLP 2001 invited paper. Online at http://www.lua.org/history.html.

[3] Rapin, Patrick: *Lua as a protocol language*. In: Lua programming Gems.

[4] Biancuzzi, Federico; Warden, Shane: *Masterminds of Programming*. O'Reilly: Sebastopol, 2009.