

# Quick Reference Guide for Lua

## Lua and the Nmap Scripting Engine

**Lua** is a dynamically interpreted scripting language characterized for being fast, flexible, portable, small in size and yet very powerful. It has been chosen for those very same reasons by a variety of well recognized projects in many industries including information security. The **Nmap Scripting Engine** uses Lua to allow users to easily extend the capabilities of Nmap by writing scripts that have access to the information collected by the tool.

Entire books can be written about Lua and its wonderful flexibility and amazing features. This guide will merely introduce you to the basics of what I think you need to know to start working on your own Lua scripts. If you would like to dig deeper into Lua after reading this chapter, I highly recommend checking out their online documentation at <http://www.lua.org/docs.html> or supporting the project by buying one of their official books at <http://www.lua.org/donations.html#books>.

In this guide we will work with Lua 5.2 as this is the version included in the latest Nmap build at the moment of writing. However, the basic principles and features described here certainly applies to older versions and newer versions since we will not use any deprecated functions.

This guide covers the following concepts of Lua:

- Control statements.
- Data types and common structures.
- String handling (Patterns and captures).

- I/O operations.
- Co-routines.
- Metamethods

## Flow control structures

Some classic control structures are implemented in Lua such as the “if-then” conditional statements, a few different loop types and the break and continue functions. Let’s review those structures briefly.

### Conditional statements: if, then, elseif

The if-then conditional statement evaluates an expression and executes a block of code if true.

```
if status.body then
  --Do something
end
```

Lua also supports an “else-if” conditional statement with the keyword **elseif**:

```
if status.body then
  --Do something
elseif
  --Do something else
end
```

If-then statements must end with the terminator keyword **end**.

### Loops: while

The **while** loop works similarly than in other programming languages:

```
local x = 1
while(x<1337)
  print x
  x = x + 1
end
```

While loops must end with the terminator keyword **end**.

## Loops: repeat

The **repeat** loop runs the body until the set condition becomes true:

```
done = false
repeat
  ... --Do something
until done
```

## Loops: for

There are two loop formats: one for iterating numeric indexes and another one for working with iterators.

```
for x = 1,1337 do
  print(x)
end
```

The step number (which can be negative) can be set by passing a third argument to the loop statement:

```
for x = 1337,1,-1 do
  print(x)
end
```

Output:

```
1337
1336
1335
...
1
```

For loops must end with the terminator keyword <b>end</b> .
---

The iterator function **pairs()** allows iteration through key and values of a given table:

```
t = {}
t["nmap"] = "FTW"
t[1337] = "b33r"
for index, value in pairs(t) do
  print(index, value)
end
```

The snippet above will produce the following output:

3

```
nmap, ftw
1337, b33r
```

The items returned by the iterator **pairs()** is not guaranteed to be in numeric order. Use the function **ipairs()** if you need to return the values ordered by a numeric key:

```
a = {}
a[2] = "FTW"
a[1] = "NMAP "
for i, val in ipairs(a) do
  print(i,val)
end
```

Output:

```
1, NMAP
2, FTW
```

## Data types

Lua has the following basic data types:

- **number**: Stores integer and double float numbers.
- **string**: Sequence of bytes
- **boolean**: false and true values
- **table**: Store associative arrays that can be used to represent multiple data structures.
- **function**: Object of a function.
- **nil**: Indicates the lack of value of a data type or variable.
- **userdata**: Expose the values of C objects (or other non-Lua objects).
- **thread**: Independent thread of execution.

## String handling

Lua's **string library** supports a lot of handy string operations. Strings will obviously be used frequently when writing NSE scripts since they represent a byte sequences. Let's review the most common functions for string handling.

## Character classes

Character classes are special operators used in patterns. We will need them when matching or subtracting substrings so keep them in mind while we review patterns and string operations.

.	All characters
%a	Letters
%c	Control characters
%d	Digits
%l	Lower case letters
%p	Punctuation characters
%s	Space characters
%u	Upper case letters
%w	Alphanumeric characters
%x	Hexadecimal digits
%z	Null (0x90)

## Magic characters

The following characters have special functions within patterns:

()	Define captures.
.	Any character.
%	Escape character for magic characters and non-alphanumeric characters.
+	Repetition operator.
-	Repetition operator.
*	Repetition operator.
?	Repetition operator.
[	Define sets.

^	Represent the complement of the set.
\$	Represent the end of a string.

## Patterns

**Patterns** are used to match strings and they are very powerful, think about them as simplified regular expressions in Lua. Character classes and captures are used in combination with patterns to allow programmers to perform advanced matching and string substitution and extraction.

For example, the character class that represents a null byte (0x90) is `%z`, to remove all null bytes in a buffer we might do something like:

```
...
buffer = io.read()
buffer = string.gsub(buffer, "%z", "") --This will remove all null
bytes from the buffer
...
```

Let's say we would like to match a string containing a version number that has the following format:

```
version 1.21
```

A matching pattern could be:

```
version%s%d%p%d%d
```

And the pattern above will match strings like:

```
version 1.21
version 8,32
version 4!20
```

We can create sets of characters by using square brackets. A set will match any of the characters enclosed in the brackets:

```
print(string.match("Nmap", "[mn]ap"))
map
print(string.match("Hakin9 sucks!", "Hackin[g9]"))
Hakin9
> print(string.match("Error code:52c", "%d%d[0-9,abc]"))
52c
```

Patterns are nothing more than strings in Lua internally so the same rules apply to them.

### Captures

Captures are delimited by parenthesis and they are used to extract information from a matched pattern. The following example is a snippet from the script http-majordomo2-dir-traversal. It uses a capture to store the content of a remote file obtained via a security vulnerability if a match is found:

```
...
local _, _, rfile_content = string.find(response.body,
'<pre>(.*?)<!%-%- Majordomo help_foot format file %-%->')
...
```

### Repetition operators

The following repetition operators affect the previous character or character set in different ways depending on the operator. This function allow us to match strings with unknown lengths.

?	Optional.
*	Zero or more times, as many times as possible.
+	At least one time, as many times as possible.
-	Zero or more times, a few times as possible.

Examples:

```
> print(string.match("52c111d111", "[0-9,abc]+"))
52c111
> print(string.match("xx", "[0-9,abc]?xx"))
xx
> print(string.match("1xx", "[0-9,abc]?xx"))
1xx
> print(string.match("dxx", "[0-9,abc]?xx"))
xx
```

## Concatenation

To **concatenate** strings use the operator “..”:

```
local c = "Hey "  
local b = c.."nmaper!"  
print(b)
```

Output:

```
Hey nmaper!
```

String to number (and viceversa) conversion is done automatically by Lua.

## Finding substrings

There will be a lot of occasions when you will need to know if certain string is a substring of another string object, for example to match the response of a network request. We can do this with Lua in a few different ways with some help of the following functions:

```
string.find(s, pattern [, init [, plain]])  
string.match(s, pat)  
string.gmatch(s, pat)
```

The function **string.find** returns the position of the beginning and end of the string occurrence or nil if not found. It should be used when we need to find a string and the position offsets are needed:

```
> print(string.find("hello", "ello"))  
2 5
```

On the other hand, if you don't need the position indexes, you could use the function **string.match**:

```
If string.match(resp.body, "root:") then  
... --Do something here  
end
```

**string.find** and **string.match** only work with the first occurrence of the string. If there are multiple occurrences, you must use **string.gmatch** (The ‘g’ stands for global) to get an iterator of the objects found:

```
> for i in string.gmatch("a1b2c3d4e5f6", "%d") do print(i) end
```

Output:

```
1
```

```
2
3
4
5
6
```

## String repetition

To concatenate “n” times the string “s” with Lua we have the function **string.rep**:

```
string.rep(string, number)
```

Example:

```
> print(string.rep("a", 13))
aaaaaaaaaaaaa
```

## String length

To determine the length of a string use the function **string.len**:

```
string.len(string)
```

Example:

```
> print(string.rep("A", 10))
AAAAAAAAAA
```

## Formatting strings

We can create strings with a given format and variables. This saves up time and produce better code (easier to read) than using multiple concatenation operators.

```
string.format(string, arg1, arg2, ...)
```

Example:

```
--Here both strings are equal but the second one is much easier to
read
local string1 = "hey "..var1..":"
local string2 = string.format("hey %:", var1)
```

## Splitting and joining strings

Although there is no built-in function for splitting and joining strings, the NSE library **stdnse** can take care of that:

```
stdnse.strjoin(delimiter, list)
stdnse.strsplit(pattern, text)
```

Example:

```
local stdnse = require "stdnse"
...
local csv_str = "a@test.com,b@foo.com,c@nmap.org"
local csv_to_emails = stdnse.strsplit(",", emails)
for email in pairs(csv_to_emails) do
    print(email)
end
```

Output:

```
a@test.com
b@foo.com
c@nmap.org
```

## Common data structures

In Lua you will use the data type “**table**” to implement all your data structures. This data type has great features such as being able to store functions and being dynamically allocated, among many others. Hopefully after reviewing some common data structures you will find yourself loving their flexibility.

### Tables

Tables are very convenient and allow us to implement data structures like dictionaries, sets, lists and arrays very efficiently. A table can be initialized empty or with some values:

```
T1={} --empty table
T2={"a", "b", "c"}
```

Integer indexes or **hashkeys** can be used to assign or dereference the values in a table. One important thing to keep in mind is that we can have both types in the same table:

```
t={}
t[1] = "hey "
t["nmap"] = "hi " --This is valid
```

To get the number of elements store in a table you may prepend the operator ‘#’ . :

```
if #users>1 then
    print(string.format("There are %d user(s) online.", #users))
    ... --Do something else
end
```

Keep in mind that the operator # only counts entries with integer indexes and is not deterministic. If you are working with non-linear integer indexes you need to iterate through the table to get the number of items:

```
function tlength(t)
  local count = 0
  for _ in pairs(t) do count = count + 1 end
  return count
end
```

## Arrays

Arrays can be implemented simply by using tables with integer indexes. The table's size do not need to be declared at the beginning and can grow as you need it to:

```
a={}
for i=1,10 do
  a[i] = 0
end
```

Another example:

```
a = {4,5,6}
print(a[1]) --will print 4
print(a[3]) --will print 6
a[5] = 9 --This assignment is valid.
print(a[5]) --This will print 9
```

## Linked lists

Since tables can store references to other tables, we can implement linked lists pretty straightforward by assigning a field as the next-link reference:

```
linked_list = nil
contactA = { name="Paulino Calderon", num=123456789 }
contactB = { name="John Doe", num=11111111 }
contactC = { name="Mr T", num=123 }

linked_list = {data = contactA, ptr = linked_list }
linked_list = {data = contactB, ptr = linked_list }
linked_list = {data = contactC, ptr = linked_list }
```

```

local head = linked_list
while head do
  print(string.format("%s:%s", head.data["name"],
head.data["num"]))
  head = head.ptr
end

```

Output:

```

Mr T:123
John Doe:1111111
Paulino Calderon:123456789

```

## Sets

Sets are commonly used for lookup tables and since we can use hashkeys as indexes in Lua, lookups are executed in constant time and very efficiently.

```

set={}
items = { "2013-02-01", "2013-02-02", "2013-02-03" }
for _, key in pairs(items) do
  set[key]=true
end

```

```

--To look up a key, we simply access the field.
if set["2013-02-01"] then
  print("Record found.")
end

```

## Queues

A FIFO queue can also be implemented in very few lines of source code.

```

--Initializes a new queue
--@return Index table
function queue_new ()
  return {head = 0, tail = -1}
end

--Adds element to the queue
--Inserts are FIFO
--@param queue Queue
--@param value value of new element

```

```

function queue_add (queue, value)
    local last = queue.tail + 1
    queue.tail = last
    queue[last] = value
end

--Removes element from queue
--Deletions are FIFO
--@param queue Queue
--@return True if operation was succesfull
--@return Error string
function queue_remove (queue)
    local first = queue.head
    if first > queue.tail then
        return false, "Queue is empty"
    end
    local value = queue[first]
    queue[first] = nil
    queue.head = first + 1
    return true, value
end

--Returns true if queue is empty
--@param queue Queue
--@return True if given queue is empty
function queue_is_empty(queue)
    if queue.head > queue.tail then
        return true
    end
    return false
end

```

## Custom data structures

Tables can also be used for representing many other custom data structures. Some NSE scripts use tables stored in files as the databases. Tables can also reference other tables or even store functions and this is very handy when modelling data.

In the next sections you will learn how the NSE scripts `http-num` and `http-default-accounts` use tables to easily store fingerprints that can also be loaded into a script without the need of a parser.

## http-enum database

This a fingerprint structure of the NSE script **http-enum**:

```
{
  category = 'general',
  probes = {
    {
      path = '/archiva/index.action',
      method = 'GET'
    },
    {
      path = '/index.action',
      method = 'GET'
    }
  },
  matches = {
    {
      match = '.*">Apache Archiva (.-</a>',
      output = 'Apache Archiva version \\1'
    },
    {
      match = 'Apache Archiva (%d-%..-)\n',
      output = 'Apache Archiva version \\1'
    },
    {
      match = '<title>Apache Archiva \\',
      output = 'Apache Archiva'
    }
  }
}
});
```

## http-default-accounts

This a fingerprint structure of the NSE script **http-default-accounts**:

```
{
  name = "Apache Tomcat",
  category = "web",
  paths = {
```

```

    {path = "/manager/html/"},
    {path = "/tomcat/manager/html/"},
  },
  login_combos = {
    {username = "tomcat", password = "tomcat"},
    {username = "admin", password = "admin"}
  },
  login_check = function (host, port, path, user, pass)
    return try_http_basic_login(host, port, path, user, pass)
  end
end
}

```

## I/O operations

**File manipulation** in Lua is done either on implicit or explicit file descriptors. We will focus on using explicit file descriptors to perform most of the operations.

If we work with implicit file descriptors by default Lua will use *stdin* and *stdout* respectively. Alternatively we can set the output and input descriptor with **io.output** and **io.input**.

## Modes

r	Read mode.
w	Write mode.
a	Append mode.
r+	Update mode. This mode preserves the existing data
w+	Update mode. This mode deletes any existing data
a+	Append update mode. This mode preserves the existing data and only allows appending at the end of file

## Opening a file

The function **io.open** returns a file descriptor if successful:

```
file = io.open (filename [, mode])
```

When it fails it will return nil and the corresponding error message (Like most Lua functions).

## Reading a file

To read a file using an explicit file descriptor use the function **io.read**:

```
file = io.open(filename)
val = file:io.read("%d")
```

There is a function called **io.lines** that will take a filename as argument and return an iterator to traverse each line of the filename. This function can help us process files in chunks divided by new lines:

```
for line in io.lines(filename) do
  if string.match(line, "<password>(.*?)</password>") then
    ... --Do something here
  end
end
```

## Writing a file

The function **io.write** takes n string arguments and writes them to the corresponding file descriptor:

```
io.write(args,...)
```

Example:

```
local filename
str1 = "hello "
str2 = "nmaper"
file = io.open (filename [, mode])
file:write(str1, str2)
...
```

## Closing a file

After you are done you should close the file to release the file descriptor with the function **io.close**:

```
io.close ([file])
```

## Coroutines

Co-routines allow collaborative multi-tasking and are a very interesting aspect of Lua. Keep in mind that coroutines are not threads. Using coroutines will help you save time when you need different workers using the same context and it also produces code which is easier to read and therefore maintain.

### Creating a coroutine

To create a coroutine use the function **coroutine.create**. This function only creates the coroutine but is not actually executed.

```
local nt = coroutine.create(function()
    print("w00t!")
end)
```

### Executing a coroutine

To execute a coroutine use the function **coroutine.resume**:

```
coroutine.resume(<coroutine>)
```

You can also pass parameters to the coroutine function as additional arguments to the **coroutine.resume** function:

```
local nt = coroutine.create(function(x, y, z)
    print(x,y,z)
end)

coroutine.resume(nt, 1, 2, 3)
```

Output:

```
1,2,3
```

There is a function called **coroutine.wrap** that can replace the need of running **coroutine.create** and **coroutine.resume**. The only difference is that the coroutine must be assigned to a function:

```
local ntwrapped = coroutine.wrap(function()
    print("w00t!")
end)

ntwrapped() --will print w00t!
```

## Determining current coroutine

To obtain the current coroutine running use the function **coroutine.running**:

```
nt =coroutine.create(function()
    print("New CO!")
    print(coroutine.running())
end)

print(coroutine.running())
coroutine.resume(nt)
```

Output:

```
thread: 0x931a008      true
New CO!
thread: 0x931da78      false
```

## Getting the status of a coroutine

To get the current status of a coroutine we can use the function **coroutine.status**. The function can return one of the following values:

running	Coroutine is executing.
dead	Coroutine has finished running.
suspended	Coroutine is waiting to be executed.

Example:

```
local nt=coroutine.create(function()
    print(string.format("I'm aliveee! The status of the coroutine
is:%s", coroutine.status(coroutine.running())))
end)
coroutine.resume(nt)
print("Now I'm " ..coroutine.status(nt))
```

Output:

```
I'm aliveee! The status of the coroutine is:running
Now I'm dead
```

## Yielding a coroutine

To put a coroutine in suspended mode with the function **coroutine.yield**:

```
local nt=coroutine.wrap(function(msg)
    print(msg)
```

```

    coroutine.yield()
    print("Resumed!")
    coroutine.yield()
    print("Resumed again")
    coroutine.yield()
    print("Resumed once more")
end)
nt("Hello nmaper!")
nt()
nt()
nt()

```

Output:

```

Hello nmaper!
Resumed!
Resumed again
Resumed once more

```

## Metatables

Metamethods allow us to change the behavior of a table by writing custom functions for operators such as comparing objects, arithmetical operations and more. For example, let's say we would like to overload the add functionality of our table object with a new function that adds up certain field. Normally the addition operation isn't valid on tables but with we can overwrite the metamethod “**\_\_add**” to perform whatever we need.

### Arithmetic methamethods

<code>__add</code>	Addition operator.
<code>__mul</code>	Multiplication operator.
<code>__sub</code>	Subtraction operator.
<code>__div</code>	Division operator.
<code>__unm</code>	Negation operator.
<code>__pow</code>	Exponentiation operator.
<code>__concat</code>	Concatenation operator

## Relational methamethods

<code>__eq</code>	Equality
<code>__lt</code>	Less Than
<code>__le</code>	Less or equal

The function **setmetatable** is used to set the metatable of a table:

```
local vuln1 = {criticality_level = 10, name="vuln #1"}
local vuln2 = {criticality_level = 4, name="vuln #2"}

local mt = {
  __add = function (l1, l2) - override the function "add"
    return { criticality_level = l1.criticality_level +
      l2.criticality_level }
  end
}

setmetatable(vuln1, mt)
setmetatable(vuln2, mt)

local total = vuln1 + vuln2

print(total.criticality_level) --Prints 14 when normally it would
fail before reaching this statement.
```

## Things to remember when working with Lua

### Comments

A comment can be anything in between two hyphens and the next end of line:

```
--This is a comment
```

**Comment blocks** are also supported. They are delimited by the characters "--[[“ and “]]”:

```
--[[
This is a multi-line
comment block.
```

]]

## Dummy assignments

There are occasions where you don't need all the information returned by a function and in Lua you can use dummy assignments to discard a return value. The operator is “\_” (underscore):

```
local _, _, item = string.find(<string>, <pattern with capture>)
```

## Indexes

Indexes start at one, not zero:

```
z={"a","b","c"}
z[1]="b" --This assignment will change the content of the table to
{"b","b","c"}
```

However, you can initialize an “array” at any value:

```
nmap = {}
for x=-1337, 0 do
  nmap[x] = 1
end
```

Keep in mind that all Lua libraries will stick to this convention.

## Semantics

Due to its flexibility you might encounter different **semantics**. In the following example both lines calling the function **gmatch** are perfectly valid and produce the same result:

```
Local str = "nmap"
string.gmatch(str, "%z");
str:gmatch("%z")
```

---

Only functions with up to one parameters can be called using the notation  
obj:func.

---

## Coercion

Lua provides automatic conversion between strings and numbers:

```
surprise = "Pi = "..math.pi
```

```
--The string now contains "Pi = 3.1415926535898" without the need
of casting.
```

## Safe language

Lua is considered a safe language since you can always trace and detect the errors of the program itself and basically you can't cause a memory corruption no matter what you do. Yet you still need to be careful when you introduce your own C modules.

## Booleans

All values except false and nil are treated as true.

```
str = "AAA"
num = -1
zero = 0
--the following statement will evaluate to "true"
if str and num and zero then
... -- This will execute because even 0 evaluates to true
end
```

## Summary

In this guide we reviewed the most important aspects of programming with Lua to develop NSE scripts. Once you have a solid grasp on flow control structures, string handling and data structures, you will see how simple NSE scripts can be.

Lua is a very interesting and powerful language yet the power of the Nmap Scripting Engine also comes from the NSE libraries. I recommend you take a look at some of the official scripts to start getting familiar with NSE libraries and functions such as **stdnse.get\_script\_args()**, which as the name states, serves to load script arguments or the NSE library brute, which provides a framework to easily launch custom brute force attacks.

Like in any other programming language, practice makes the master.