
luacheck Documentation

Release 0.6.0

Peter Melnichenko

July 17, 2015

1	Types of warnings	3
1.1	Global variables	3
1.2	Unused variables	3
1.3	Redefined variables	4
2	Command line interface	5
2.1	Command line options	6
3	Configuration file	9
3.1	Config format	9
3.2	Per-file overrides	9
4	Luacheck module	11
4.1	Options	11
4.2	Report format	11

Contents:

Types of warnings

Luacheck generates warnings of three types:

- warnings related to global variables;
- warnings related to unused local variables and values;
- warnings related to redefined local variables.

1.1 Global variables

A global variable is defined if it is one of the standard globals (set using `--std` option in the command line and `std` option in the config) or custom globals (set using `--globals` option in the command line and `globals` option in the config).

When an undefined global variable is accessed or set, a warning of corresponding subtype is generated.

1.1.1 Implicit definition

If `--allow_defined` option is used, or `allow_defined = true` is in the config, global variables are automatically defined if they are set in one of the checked files. Luacheck will generate a warning if an implicitly defined global variable is never accessed.

1.2 Unused variables

Luacheck generates warnings for all unused local variables except one named `_`. Detection of unused arguments and loop variables can be disabled using `-a` flag in the command line or `unused_args = false` in the config.

1.2.1 Unused values

Luacheck also detects unused values: redundant assignments to variables which are then not used before another assignment. As an example, in the following snippet value assigned to `foo` on line 4 is unused, as it is always overwritten on line 7:

```
1 local foo
2
3 if condition() then
4     foo = expr1()
```

```
5 end
6
7 foo = expr2()
8 return foo
```

1.3 Redefined variables

Luacheck detects declarations of local variables shadowing previous declarations in the same scope, unless the variable is named `_`.

Note that it is **not** necessary to define a new local variable when overwriting an argument:

```
1 local function f(x)
2     local x = x or "default" -- bad
3 end
4
5 local function f(x)
6     x = x or "default" -- good
7 end
```

Command line interface

luacheck program accepts files, directories and [rockspecs](#) as arguments.

- Given a file, luacheck will check it.
- Given `-`, luacheck will check stdin.
- Given a directory, luacheck will check all files with `.lua` extension within it.
- Given a rockspec (a file with `.rockspec` extension), luacheck will check all files with `.lua` extension mentioned in the rockspec in `build.install.lua`, `build.install.bin` and `build.modules tables`.

The output of luacheck consists of separate reports for each checked file and ends with a summary:

```
$ luacheck src
Checking src/bad_code.lua                               Failure

    src/bad_code.lua:3:16: unused variable helper
    src/bad_code.lua:3:23: unused variable length argument
    src/bad_code.lua:7:10: setting non-standard global variable embrace
    src/bad_code.lua:8:10: variable opt was previously defined as an argument on line 7
    src/bad_code.lua:9:11: accessing undefined variable hepler

Checking src/good_code.lua                               OK
Checking src/python_code.lua                             Syntax error
Checking src/unused_code.lua                             Failure

    src/unused_code.lua:3:18: unused argument baz
    src/unused_code.lua:4:8: unused loop variable i
    src/unused_code.lua:5:13: unused variable q
    src/unused_code.lua:7:11: unused loop variable a
    src/unused_code.lua:7:14: unused loop variable b
    src/unused_code.lua:7:17: unused loop variable c
    src/unused_code.lua:13:7: value assigned to variable x is unused
    src/unused_code.lua:14:1: value assigned to variable x is unused
    src/unused_code.lua:22:1: value assigned to variable z is unused

Total: 14 warnings / 1 error in 4 files
```

luacheck exits with 0 if no warnings or errors occurred and with a positive number otherwise.

2.1 Command line options

Short options that do not take an argument can be combined into one, so that `-qqq` is equivalent to `-q -q -q`. For long options, both `--option value` or `--option=value` can be used.

Options taking several arguments can be used several time; `--ignore foo --ignore bar` is equivalent to `--ignore foo bar`.

Note that options that may take several arguments, such as `--globals`, should not be used immediately before positional arguments; given `--globals foo bar file.lua`, `luacheck` will consider all `foo`, `bar` and `file.lua` global and then panic as there are no file names left.

Option	Meaning
<code>-g no-global</code>	Filter out warnings related to global variables.
<code>-r no-redefined</code>	Filter out warnings related to redefined variables.
<code>-u no-unused</code>	Filter out warnings related to unused variables.
<code>-a no-unused-args</code>	Filter out warnings related to unused arguments and loop variables.
<code>-v no-unused-values</code>	Filter out warnings related to unused values.
<code>--std <std></code>	Set standard globals. <code><std></code> must be one of: <ul style="list-style-type: none"> • <code>_G</code> - globals of the Lua interpreter luacheck runs on (default); • <code>lua51</code> - globals of Lua 5.1; • <code>lua52</code> - globals of Lua 5.2; • <code>lua52c</code> - globals of Lua 5.2 compiled with <code>LUA_COMPAT_ALL</code>; • <code>luajit</code> - globals of LuaJIT 2.0; • <code>min</code> - intersection of globals of Lua 5.1, Lua 5.2 and LuaJIT 2.0; • <code>max</code> - union of globals of Lua 5.1, Lua 5.2 and LuaJIT 2.0; • <code>none</code> - no standard globals.
<code>--globals [<global>] ...</code>	Add custom globals on top of standard ones.
<code>--new-globals [<global>] ...</code>	Set custom globals. Removes custom globals added previously.
<code>-c --compat</code>	Equivalent to <code>--std=max</code> .
<code>-d --allow-defined</code>	Allow defining globals by setting them. See <i>Implicit definition</i> .
<code>--no-unused-globals</code>	Filter out warnings related to set but unused global variables.
<code>--ignore <var> [<var>] ...</code>	Filter out warnings related to variables named <code><var></code> .
<code>--only <var> [<var>] ...</code>	Filter out warnings not related to variables named <code><var></code> .
<code>-l <limit> --limit <limit></code>	Exit with 0 if there are <code><limit></code> or less warnings (default: 0).
<code>--config <config></code>	Path to custom configuration file (default: <code>.luacheckrc</code>).
<code>--no-config</code>	Do not look up custom configuration file.
<code>-q --quiet</code>	Suppress report output for files without warnings. <ul style="list-style-type: none"> • <code>-qq</code> - Suppress output of warnings. • <code>-qqq</code> - Only output summary.
<code>--no-color</code>	Do not colorize output.
<code>-h --help</code>	Show help and exit.

Configuration file

By default, `luacheck` tries to load configuration from `.luacheckrc` file in the current directory. Path to config can be set using `--config` option. Config loading can be disabled using `--no-config` flag.

3.1 Config format

Config is simply a Lua script executed by `luacheck`. It may set various options by assigning to globals. See [Options](#).

An example of a config which makes `luacheck` ensure that only globals from the portable intersection of Lua 5.1, Lua 5.2 and LuaJIT 2.0 are used, as well as disables detection of unused arguments:

```
1 std = "min"
2 unused_args = false
```

3.2 Per-file overrides

The environment in which `luacheck` loads the config contains a special global `files`. When checking a file `<path>`, `luacheck` will override options from the main config with entries from `files[<path>]`. For example, the following config re-enables detection of unused arguments only for `myfile.lua`:

```
1 std = "min"
2 unused_args = false
3
4 files["myfile.lua"] = {
5     unused_args = true
6 }
```

Note that `files` table supports autovivification, so that

```
files["myfile.lua"].unused_args = true
```

and

```
files["myfile.lua"] = {
    unused_args = true
}
```

are equivalent.

Luacheck module

`luacheck` module is a single function. Use `local luacheck = require "luacheck"` to import it.

The first argument of the function should be an array. Each element should be either a file name (string) or an open file handle, in which case `luacheck` will read it till EOF and close it.

4.1 Options

The second argument, if present, should be a table of options. Options are interpreted similarly to corresponding command line switches; see [Command line options](#).

Option	Type	Default value
<code>options.global</code>	Boolean	<code>true</code>
<code>options.redefined</code>	Boolean	<code>true</code>
<code>options.unused</code>	Boolean	<code>true</code>
<code>options.unused_args</code>	Boolean	<code>true</code>
<code>options.unused_values</code>	Boolean	<code>true</code>
<code>options.std</code>	String or array of strings	<code>"_G"</code>
<code>options.globals</code>	Array of strings	<code>{}</code>
<code>options.compat</code>	Boolean	<code>false</code>
<code>options.allow_defined</code>	Boolean	<code>false</code>
<code>options.unused_globals</code>	Boolean	<code>true</code>
<code>options.ignore</code>	Array of strings	<code>{}</code>
<code>options.only</code>	Array of strings	(Do not filter)

When checking `n`-th file, `luacheck` will try to combine `options[n]` with general options, similarly to how per file config tables overwrite general config table. See [Configuration file](#).

4.2 Report format

The `luacheck` function returns a report. A report is an array of file reports plus fields `warnings` and `errors` containing total number of warnings and errors, correspondingly.

A file report is an array of warnings. If an error occurred while checking a file, its report will only have `error` field containing `"I/O"` or `"syntax"`.

A warning is a table with fields `type`, `subtype` and `vartype` indicating the type of warning (see [Types of warnings](#)), and fields `line` and `column` pointing to the source of the warning. For warnings related to redefined variables there also are fields `prev_line` and `prev_column` pointing to the previous declaration of the variable.

Luacheck is a tool for linting and static analysis of **Lua** code. It is able to spot usage of undefined global variables, unused local variables and a few other typical problems within Lua programs.

Luacheck provides a command line interface as well as a Lua module which can be used by other programs.

This is the documentation for the 0.6.0 version.