CVE-2020-24371 Analysis

Author : Minseok Kang

1. Overview

Crash type : heap-use-after-free (read)

Version : v5.4.0 (git commit hash : c33b1728aeb7dfeec4013562660e07d32697aa6b)

2. PoC code

```
local old = {10}
collectgarbage()

local co = coroutine.create(
  function ()
    local x = nil
    local f = function ()
                return x
              end
    x = coroutine.yield(f)
    coroutine.yield()
  end
)

local _, f = coroutine.resume(co)
collectgarbage("step")
old[1] = {"hello"}
coroutine.resume(co, {})
co = nil

collectgarbage("step")
collectgarbage("step")
assert(old[1][1] == "hello")
```

3. Root Cause Analysis

Following log is observed by executing PoC code in Lua compiled with address sanitizer applied.

```
user@ubuntu20:~/cve-2020-24371$ ./lua/lua poc.lua
=================================================================
==4676==ERROR: AddressSanitizer: heap-use-after-free on address 0x60600000110c at pc 0x555bb9efb220
READ of size 4 at 0x60600000110c thread T0
    #0 0x555bb9efb21f in luaV_execute /home/user/cve-2020-15889/lua/lvm.c:1245
    #1 0x555bb9ec6944 in luaD_call /home/user/cve-2020-15889/lua/ldo.c:504
    #2 0x555bb9ec6b5e in luaD_callnoyield /home/user/cve-2020-15889/lua/ldo.c:526
    #3 0x555bb9eb9cd9 in f_call /home/user/cve-2020-15889/lua/lapi.c:997
    #4 0x555bb9ec36bf in luaD_rawrunprotected /home/user/cve-2020-15889/lua/ldo.c:148
    #5 0x555bb9ec83db in luaD_pcall /home/user/cve-2020-15889/lua/ldo.c:749
    #6 0x555bb9eb9f81 in lua_pcallk /home/user/cve-2020-15889/lua/lapi.c:1023
    #7 0x555bb9eb061f in docall /home/user/cve-2020-15889/lua/lua.c:139
    #8 0x555bb9eb0b0f in handle_script /home/user/cve-2020-15889/lua/lua.c:228
    #9 0x555bb9eb2203 in pmain /home/user/cve-2020-15889/lua/lua.c:603
    #10 0x555bb9ec6407 in luaD_call /home/user/cve-2020-15889/lua/ldo.c:482
```

Objects declared in Lua are manually collected by explicitly calling "collectarbage()" function. In PoC code during garbage collection, Lua marks objects in an erroneous way. Focusing on the age and color of objects in the code, root cause of the crash can be figured out. The objects below are the main elements that need attention.

1) mainthread (lua_State)

2) local old = {10} (table)

```
local old = {10}
```

3) local co = coroutine.create() (thread)

```
local co = coroutine.create()
```

4) function() ... (LClosure object in coroutine "co")

```
function ()
    local x = nil
    local f = function ()
                return x
              end
    x = coroutine.yield(f)
    coroutine.yield()
  end
```

5) local f = function() … (LClosure declared inside “3)”)

```
local f = function()
           return x
          end
```

6) local x (Upvalue of “f”)

```
local x = nil
```

7) {“hello”} (table that is assigned later to local variable “old”)

```
old[1] = {"hello"}
```

8) {} (table that is assigned later to upvalue “x”)

```
coroutine.resume(co, {})
```

- Code flow

This part briefly explains changes in the object’s characteristics(age, color value specifically) after executing certain lines of the PoC code in order.

1) collectgarbage() (line 2)

At first, Lua runs garbage collection in generational mode. As there is only mainthread and “old” table, Their age becomes old.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color			-	-	-	-	-	-
age	old	old	-	-	-	-	-	-

2) local co = coroutine.create() (line 4)

When the coroutine is created through coroutine.create(), "co" and "function ()" object is allocated. Every object created after a recent garbage collection has white color and its age is set as new.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color					-	-	-	-
age	old	old	new	new	-	-	-	-

3) local _, f = coroutine.resume(co) (line 15)

After executing the coroutine "co", elements inside the coroutine are allocated. "co" stops executing by coroutine.yield(f) in line 10. Arguments to coroutine.yield is passed to "f" in line 15 which is same value of "f" inside "co".

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color							-	-
age	old	old	new	new	new	new	-	-

4) collectgarbage("step") (line 16)

This line does the young collection, which does garbage collection only for objects recently created(new objects in this case). As all objects are reachable, their age is set to the next value, "survival".

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color							-	-
age	old	old	survival	survival	survival	survival	-	-

5) old[1] = {“hello”} (line 17)

New table “{“hello”} element is assigned to “old”. In this case, black object points to white object which breaks invariant of garbage collection algorithm in Lua. To fix this, Lua calls LuaC_barrierback_function(lgc.c:214) to mark “old” as gray and set its age as touched1. As “old” becomes gray, it goes to grayagain list in global State.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								-
age	old	touched1	survival	survival	survival	survival	new	-

6) coroutine.resume(co, {}) (line 18)

“co” is resumed by line 18. Table “{}” is also allocated and it is assigned to “x” in line 10.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								
age	old	touched1	survival	survival	survival	survival	new	new

7) co = nil (line 19)

“co” is assigned with nil value. This makes “co” no longer reachable from the code, collected by the garbage collector later on.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								
age	old	touched1	survival	survival	survival	survival	new	new

8) collectgarbage(“step”) (line 21)

This part collects unreachable objects in the code. Collector runs in generational mode. Collection can be separated in to (1)mark phase, and (2)sweep phase.

(1) After mark phase

After the mark phase, “f” is marked as black since it can be reached both from mainthread and “co”. As “co” was set to nil value in “7)”, related objects still remain as white.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								
age	old	touched1	survival	survival	survival	survival	new	new

(2) During sweep phase

As “co” is white before sweep phase, it is freed during the process. Subsequently, Lua closes upvalues related to “co”, which are “x” and its content “{ }” in this case.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								
age	old	touched1	survival	survival	old1	old1	survival	survival

In luaF_close(lfunc.c:239) function, “x” is marked as black. However, its content “{ }” is white. In order to restore invariant, Lua calls luaC_barrier to mark “{ }” gray.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color								
age	old	touched1	survival	survival	old1	old1	survival	old0

(3) After sweep phase

After sweep phase, "co" and "function()" are freed.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color			-	-				
age	old	touched1	(freed)	(freed)	old1	old1	survival	old0

9) collectgarbage("step") (line 22)

After mark phase, although "{"hello"}" is obviously reachable from the mainthread, it still remain as white.

	mainthread	old	co	function()	f	x	{"hello"}	{ }
color			-	-				
age	old	touched1	(freed)	(freed)	old1	old1	survival	old0

This eventually causes "{"hello"}" to be freed after sweep phase, causing use after free crash when it is accessed. Root cause of the crash is originated from barrier function. Although barrier function is to restore invariance of the garbage collection algorithm, it may cause an object to change its color during sweep phase. This leaded some black objects to be in g->gray list, malfunctioning during collection.

4. Patch

Additional verification logics were appended to luaC_barrier_ function(lgc.c:196), remarkupvals function(lgc.c:341), sweepgen function(lgc.c:1013), and atomic2en(lgc.c:1184) Specifically, in luaC_barrier_ function, statement that checks garbage collector mode was added. Detailed patch content can be found from following reference github link.

https://github.com/lua/lua/commit/a6da1472c0c5e05ff249325f979531ad51533110

5. Reference

https://github.com/lua/lua/commit/a6da1472c0c5e05ff249325f979531ad51533110

https://www.lua.org/bugs.html#5.4.0-10