Lua Wrappers

A writing by iChunk

Presented by **iChunk**

Uncover the secrets, they're waiting for you..

Credits

iChunk: Coming up with the idea to enhance the Lua Wrappers as we know them. This meant rewriting the Lua 5.1 source code and writing an entire "system" for handling wrapping. iChunk is responsible for writing this entire documentation.

Shade: The creator of "PerfectWrapper" which changed the wrapping game. It introduced brand new ways to handle the wrapping of functions and many other types. His work will not be mentioned and he did not contribute to this documentation but he is worth being mentioned as a developer.

Before you begin reading..

The Lua version used in the documentation is 5.1 and can be downloaded from Lua's website.
It is completely open sourced and free to use, it is definitely required when writing a Lua wrapper.
A source code for a working Lua wrapper will not be provided, but pieces of code will be.
Using these short snippets you could pull a working wrapper together if you have the will to do it.

The Idea

The idea behind wrapping can sound quite complicated, but I can promise you it is not.
The idea is to achieve unrestricted lua execution on almost *any* lua powered binary without serialization or deserialization being used. This would give us the advantage of being immune to bytecode format modification and proto modification. We would really be immune to *anything* as long as we had the LuaC functions required.
But was this possible at all? Could you really make something like this?

Solution

To achieve execution without serialization or deserialization the community came up with a great idea. Why not just convert globals? Isn't that all we would need? That would partly be true, since the Lua 5.1 source code would compile and perform all mathematical operations for us (and everything else). It wouldn't support special operators if the Lua binary being exploited had ones (i.e Roblox) but it would still function as promised.
So that's basically what wrapping is all about, converting globals and establishing communication between the binary and your exploit.

Advantages

1. Being able to control how each global is converted and how it is handled. This means that you do manage everything going through the __index and __newindex fields giving you more control of the execution. It could be used for very interesting features too!

2. Being immune to the binaries' codebase changes when it comes to bytecode. Since wrappers don't rely on any form of serializing, bytecode structure modification isn't something to worry about.

3. Being immune to binaries' codebase changes when it comes to proto modification. We no longer rely on their structure for protos.

Disadvantages

1. The exploit relies on metatables when it comes to userdatas. A small change to metatables could be the end of having proper userdata wrapping. There is always a way around this, but it might not be as easy.

2. The function handling is a huge risk to take. It is close to the only way to get function calls working but it could be detected in no time. Many games such as Roblox have done it already by checking if the closure has been pushed without permission. There are ways around this such as hex patching and hooking, but I would suggest keeping it clean.

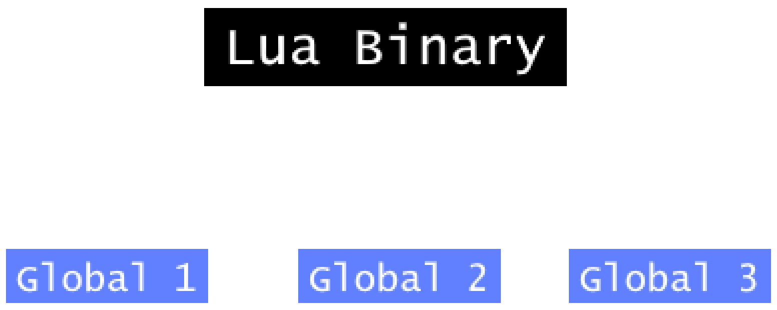
3. If the Lua binary you're exploiting is in use of custom operators such as: +=, -=, *=, /= they are not going to be supported and will cause errors upon execution. You could implement these yourself with the correct knowledge.

4. While using a wrapper function pointers are your best friends. You need to call more than 15 LuaC functions to have a functioning wrapper. If the binary has a return check or anything being able to detect that you're calling their functions without permission you're screwed. This kind of detection is also bypassable but not as easy as some other ones.

What is Lua Wrapping?

Lua wrapping could be described as the art of stealing since that's exactly what you're doing. We're stealing all globals from a Lua binary and placing them in our own globalsindex. This would make us able to use all their globals. But it isn't that easy, each global type has to be handled differently. So we're just executing normal Lua scripts using Lua 5.1 source code but establishing communication between globals.

Converting *("Wrapping")* Globals



As you can notice in the image above there are three globals, no type specified. They are all in the ***globalsindex*** of the Lua binaries' lua state. If we would want to steal their "Global 2" this is how we would do it, ***using their Lua functions***.

```
namespace L
{
	uintptr_t extState = 0xdeadbeef; // Lua state of the binary.
	namespace W // Lua Wrapper
	{
		void Convert(uintptr_t RL, lua_State* L, int Index); // Convert
global from Binary to luaState
		void Convert(lua_State* L, uintptr_t RL, int Index); // Convert
global from luaState to Binary
	}
}
const char* Member = "Global 2";
L::GetField(L::extState, LUA_GLOBALSINDEX, Member); // Push global to stack
(binary)
L::W::Convert(L::extState, L::luaState, -1); // Convert from binary to our
```

```
luastate
lua_setfield(L::luaState, LUA_GLOBALSINDEX, Member); // Cache in our own
globalsindex
L::Pop(L::extState, 1); // Free stack space by removing global (binary)
```

We're first using getfield on globalsindex to push the global onto the stack. Of course we're doing this using their functions and luaState since there is no other easy way to access their globals. Now with our global on the stack we're going to wrap it (convert from binary global to our luaState). This would place it on our side of the stack where we could access it. With the global on our side of the stack we're using setfield on globalsindex to cache it as our own global. The last step is optional but everyone does it since it's a good practice. Just pop the global from the stack using Pop. This would free up some space on stack and memory.

Binary To Lua

BINARY_TNIL

In this case the item on the given index is nil, it doesn't hold a value. To convert this into our own stack we only pushnil on the luaState.

```
// Case: BINARY_TNIL
lua_pushnil(L);
```

BINARY_TSTRING

In this case the item on the given index is a string. Using a tolstring pointer from the binary we will retrieve this value and push it onto our own stack using pushstring.

```
// Case: BINARY_TSTRING
const char* String = L::ToLString(R, Index, NULL);
lua_pushstring(L, String);
```

BINARY_TNUMBER

In this case the item on the given index is a number. Using a tonumber pointer from the binary we will retrieve this value and push it onto our own stack using pushnumber.

```
// Case: BINARY_TNUMBER
int Number = L::ToNumber(R, Index);
lua_pushnumber(L, Number);
```

BINARY_TBOOLEAN

In this case the item on the given index is a boolean. Using a toboolean pointer from the binary we will retrieve this value and push it onto our own stack using pushboolean. Please note that toboolean returns the boolean as an integer (1: true, 0: false) and you would push this number, but I'd rather do it correctly and convert it using a ternary.

```
// Case: BINARY_TBOOLEAN
int Boolean = L::ToBoolean(R, Index);
lua_pushboolean(L, (Boolean == 1) ? true : false);
```

Binary To Lua (page 2)

BINARY_TTABLE

In this case the item on the given index is a table element. To convert this into our own stack we need to do some work, including transferring some table "settings" such as *readonly* and *metatables*. This is optional since most wrappers do not do this, but I recommend you to.

```
uintptr_t Table = L::Index2Adr(R, Index);
lua_createtable(L, 0, 0);
StkId o = index2adr(L, -1);

// Convert the readonly value, you need to rewrite your Lua source for
this. Should be really simple!
hvalue(o)->readonly = *(int*)(Table + L::O::ReadOnly);
```

Before implementing the __newindex handler I'd swap metatables. You could do this however you would like, I would do it by creating a new kind of table wrapping that excludes converting tables in detail and would just convert the elements of the table. ***You do not need to transfer metatables, but it is useful in some cases.***

Now that you've done this you would need to create metatable handlers, I am going to provide an example for *__newindex* handler. It would register whenever you modify a table.

```
static int __newindex(lua_State* LS)
{
    StkId self = index2adr(LS, 1);
    if(hvalue(self)->readonly != 0)
    {
        return luaL_error(LS, "Attempted to modify a readonly table.");
    }
    /* Your newindex code here */
}
```

This would be all you would need for the readonly handler, except for converting it back to the binary later on. I didn't provide the actual newindex magic for modifying the table since you could do it using settable.

To apply this version of *__newindex* many would use setmetatable, but let's not overwrite the other metatables ***(skip if you didn't transfer metatables)***. I'd instead hook our handler onto their handler, this would make two handlers responsible. Not the greatest idea but hey, you came here looking to learn!

NOT FINISHED