BYTECODE CONVERSION

The idea behind RC7 and other, unheard (private) exploits

Written by Louka

PRESENTED BY

RAIN

Scripting and reversing group.

Credits

- **Brandon/Chirality**: For inventing the first modern script execution exploit that preceded the most recent exploits, such as Intriga and Elysian. He's also the reason why I got into exploits.
- **Jordan/KingJordan**: For inventing the concept of bytecode conversion and using it to create the exploit known as RC7, which is one of the most popular exploits.
- **Rain**: Rain is the spiritual successor to FaZe, and (personally speaking) one of the best reversing groups out here. Previous FaZe members still interested in Roblox exploiting found themselves in Rain shortly after its introduction to the Roblox exploit scene. I need to credit Rain for trying (and succeeding at that!) to improve the exploit section. They're doing a good job.
- **Louka**: For creating and designing this document. That's me! :P
- **ConvexHero**: For not doing anything. Well, he did implement callcheck, but that's it. He's doing relatively nothing that prevents exploits from happening.

What is bytecode?

Bytecode is basically the compiled form of source code. When you create a C++ application in Visual Studio and build it, the compiler (MSVC) simply takes your source code then translates it into a set of simple instructions called assembly, which is then optimized (for speed and/or size) and put into an executable file. **The executable code contains bytecode** *(along with resources, constant data and etc.)*

This does not only apply to C++ applications. In fact, almost every language in the world is compiled to another language easier to understand by your computer. Languages that are considered “interpreted” are usually compiled to a form of bytecode, which is then read by the interpreter to execute what the code does.

Bytecode, unlike executable binaries, are very portable. Yeah, executable binaries ARE portable (it’s in their name: PE, *portable* executable), but bytecode is usually smaller and therefore more portable than entire files.

Today, we’re going to talk about Lua bytecode.

A short overview on Lua bytecode (pt. 1)

So, we know that bytecode is mainly constituted of instructions. Let's take a look at Lua instructions, shall we?

- Each instruction has four parameters: the **opcode**, the **A** register, the **B** register and the **C** register. Each are one byte long, which forms a single signed integer. An instruction that uses all three registers is of type **iABC**.
- Certain instructions merges the **B** register with the **C** register to form a short (an integer with two bytes), which can store bigger values. This creates the **Bx** *(B extended)* register. An instruction that uses the A and Bx registers is of type iABx. (***i***nstruction, ***A***, ***Bx***)
- Certain instructions have a signed Bx register (it can contain negative values). They are of the iAsBx type. (***i***nstruction, ***A***, ***s***igned ***Bx***)
- And there are instructions that simply uses the A register. They are of the iA type. (***i***nstruction, ***A***). This is rare, though.

A short overview on Lua bytecode (pt. 2)

In the previous slide, I've mentioned opcodes. Opcodes is what tells the interpreter what the instruction does. If the instruction's opcode is OP_ADD (iABC), then it tells the interpreter to put in the A register the sum of the values located at B and C.

Lua has exactly 37 instructions, which is small if you compare Lua's instruction set to, for example, x86's instruction set (hint: on average, ~2000 instructions. CPUs are different so they all have different instruction sets, so the number of instructions is different depending on the CPU brand and the architecture.)

Due to the opcode being stored as a single byte, there could be a maximum of 255 instructions if Lua's development team wanted to add more instructions. If they reach that number, then they would need to switch to 64bit integers (8 bytes). This also means they would be able to add in more registers, which could mean a more performant VM, but this is just a dream that'll never happen.

How could we use bytecode for script execution?

First, let’s consider those facts:

- Lua bytecode is compiled Lua scripts.
- When you run a Lua script, Lua automatically compiles it to an intermediary format resembling bytecode before feeding it to the virtual machine.
- This “intermediary format” (which is a mere Proto struct) can be serialized then dumped as bytecode, which can be saved as a file.
- Then, if you wanted to run the bytecode, you just have to feed it to luaU_undump, which will convert the bytecode into Lua’s intermediary format (a Proto).
- Once you obtained the Proto, you can insert it into a Lua function (LClosure) then schedule it for execution.
- If a developer removes Lua’s compiler, then Lua could only run bytecode.

Therefore, in theory, we could simply dump bytecode from an external Lua instance then feed it to another Lua instance so that other Lua instance can run a script for us without having to go through the compiler. This is in fact pretty useful, as Roblox removed the compiler, preventing us from compiling scripts on the client.

So, how could we use it for Roblox script execution?

In theory, you could basically do this:

1) Compile a Lua function on your own Lua environment.
2) Dump it into bytecode via either lua_dump or string.dump.
3) Save that bytecode somewhere, preferably in memory.
4) Call Roblox's lua_loadbuffer to load in the bytecode, which will eventually invoke luaU_undump to convert it into a proto.
5) You insert the obtained Proto into a Lua function (LClosure)
6) You call the resulting function, effectively running your script.

While the above would work in any other standard Lua environment, sadly, it doesn't work in Roblox for a multitude of reasons, but the most important being the fact that Roblox has heavily modified their instruction set and added some obfuscation to it. This was done in order to prevent exploiters from simply copy and pasting Lua bytecode over to the client and expecting it to run. However, just like every security measure in the world, this does not prevent anybody from running bytecode, it just makes their task harder as they have to convert their bytecode, hence the name of the procedure: **BYTECODE CONVERSION**. You convert bytecode into Roblox's format.

Differences between Lua and Roblox Lua bytecode (pt. 1)

- Roblox Lua bytecode instructions are changed. That is, OP_MOVE in a normal Lua environment is 0, but in Roblox's Lua environment, it's 6. This is very easy to figure out and convert, whatsoever.
- Certain Roblox Lua instructions are obfuscated using a mixture of **modular multiplicative inverses and a lot of bit math**. To be honest, it's a lot of boring stuff and it almost requires someone to possess a computer science degree to understand what Roblox is doing. Fortunately, we have smart people out here (cough Chirality, Alureon, Austin <3) who figured most of it out and were able to reverse Roblox's obfuscation process. This obfuscation is only applied to OP_CALL, OP_TAILCALL, OP_RETURN and OP_CLOSURE. Every other encryption is not affected.
- Every instruction is obfuscated, albeit this form of obfuscation is easier to reverse than CALL/TAILCALL/RETURN/CLOSURE's special encryption, as it's virtually only multiplication. (deobfuscated opcode = obfusop * key). Of course, OBFUSCATING the opcode is the hard part here.
- OP_MOVE now uses the C register (normal Lua only used the A and B registers), turning it into an iABC instruction. The difference is only artificial and doesn't have an actual usage in the virtual machine.

Differences between Lua and Roblox Lua bytecode (pt. 2)

- The bytecode magic header (what is used by the undumper to verify if what you gave it is actually bytecode) has a different value. Of course, if you find this hard to reverse, you probably have to study reverse engineering a lil' bit more :u
- The bytecode is compressed using LZ4, which is an extremely fast compression algorithm. This, again, isn't difficult to reverse either, considering LZ4's source code is available to everyone.
- The registers are reversed. That is, in normal Lua, the registers are ordered like this in the integer: A, B and C. In Roblox Lua, it's C, B and A.

This is all the differences we currently know of. There are some additional obfuscation and protection steps Roblox took but they're only superficial and obvious so it's not worth mentioning here. They would only take tens of minutes to reverse.

I mean, that's how much they took for me. Maybe some people will reverse it slower than I do. Who knows.

How would someone convert Lua BC to Roblox BC?

1. The most important thing is converting instructions. You do it like this for every instruction in the bytecode dump:
   a. You take the opcode and find the corresponding Roblox opcode as they've changed. Then, you set the opcode in your instruction to that one.
   b. You set the C, B and A registers. Don't forget that they're reversed!
   c. If it's OP_MOVE, then don't forget to set the C register. They use it now as an artificial difference.
   d. If it's an instruction with special encryption *(OP_CALL, OP_TAILCALL, OP_RETURN and OP_CLOSURE)*, you apply the special encryption to the instruction.
   e. Finally, you apply general encryption to the instruction then put it into your output bytecode dump.
2. The second most important thing is moving stuff around. Of course, the bytecode dump's structure must be changed in order for Roblox to accept it. In fact, Roblox has completely remade Lua's bytecode format. They didn't just modify it.
3. You also have to fix line number as well. Roblox now encrypts line numbers based on the line number's associated instruction.

Decompilation and such things

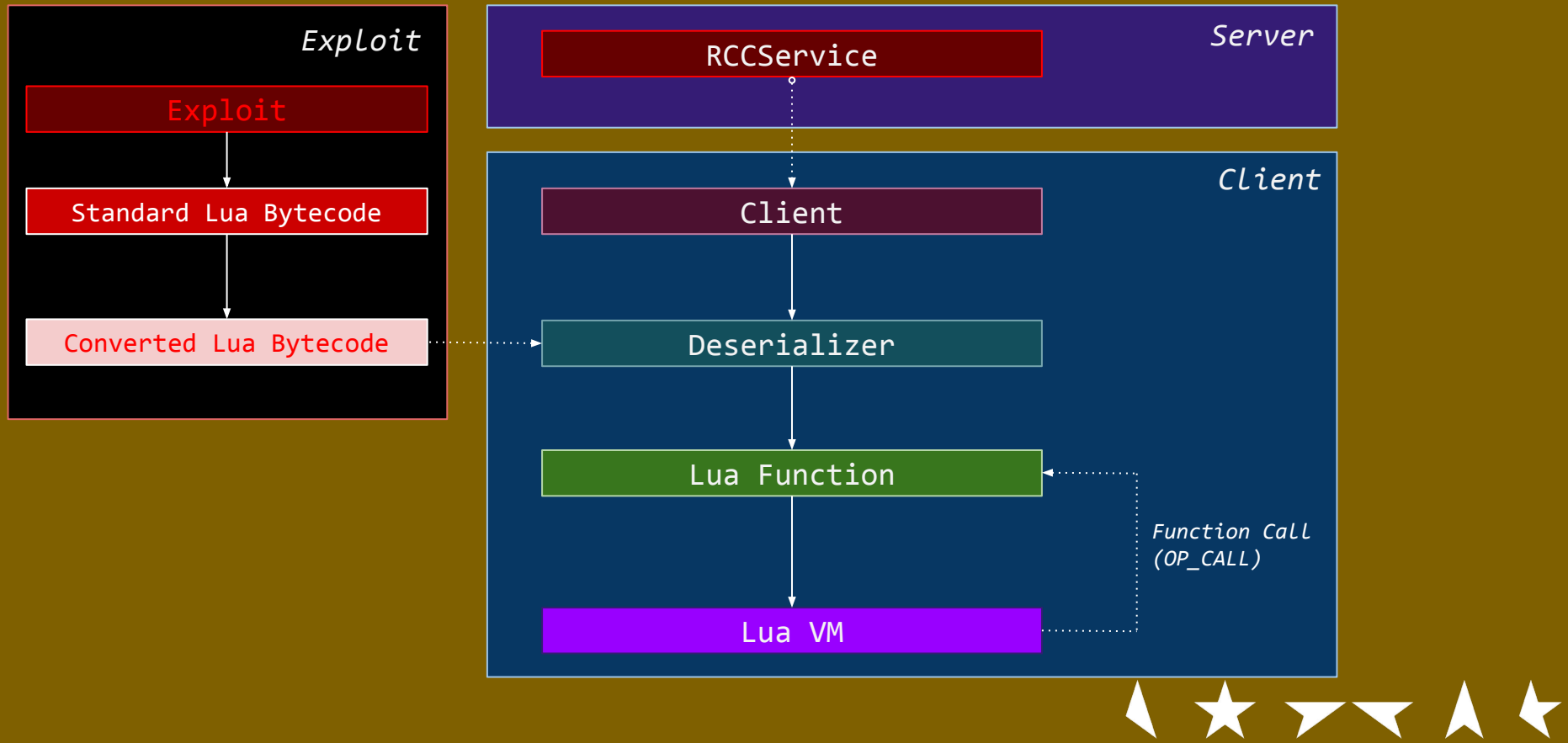
Decompilers in modern exploits (Seven, Elysian, Intriga (if it has one) and etc) works by simply doing the reverse of bytecode conversion: converting Roblox bytecode into Lua bytecode. Then, the Lua bytecode is fed to a program (usually luadec) which creates pseudocode from the bytecode dump you obtained from your conversion.

This process, however, is not without issues: my previous attempts at doing the reverse of bytecode conversion had a lot of issues (usually things with decryption and etc) which generated artifacts during the decompilation process. Relatively simple scripts, however, decompiled with a moderate amount of success.

You could theoretically use proto conversion to achieve decompilation too: you convert a Roblox proto into a normal Lua proto, push it to a normal Lua state then call lua_dump or string.dump on it, which will return a valid bytecode dump you can feed to luadec which will produce a script.

If you wondered how script decompilation worked, now you know.

Bytecode conversion: Diagram



Alternatives to Bytecode Conversion

- **Proto Conversion:** Proto conversion is obtaining the intermediary format used by Lua obtained after compilation then converting it into Roblox's intermediary format. This is very fast and is used by most script execution exploits out here, albeit it's very insecure and can be easily patched. Just like bytecode conversion
- **CLVM** *(Custom Lua Virtual Machine)*: Considered the fastest and most secure yet unstable script execution method there is. Relatively new too, but sadly hard to develop and requires a lot of manhours to get working. This method is simply making a custom Lua virtual machine that uses Roblox's own lua state rather than yours. It works, it's been proven to work with a certain degree of success, but maintaining the VM is quite hard and a lot of errors could occur if you don't know what you're doing. Despite those cons, it's very secure and considered unpatchable by Roblox. See here.

The End

Cya! :P

- Louka