PROTO CONVERSION

The idea behind Seven, Intriga and Elysian

Written by Louka

PRESENTED BY

ROBLOX INTELLIGENCE SERVICE
Classified. Division of Rain.

RAIN
Scripting and reversing group.

Credits (pt. 1)

- **Brandon/Chirality**: For inventing the concept of proto conversion and developing Seven, 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 and consecutively, why I'm writing this very document.
- **Austin**: For developing the second modern script execution exploit, Elysian. Also for helping me with a lot of stuff since I joined the exploit section.
- **Alureon**: For helping me and some other developers with script execution exploits. What he did might seem bad, but in the end, it was rewarding for some.
- **FaZe**: FaZe is what gave birth to a new generation of exploits, only to be terminated when Jordan introduced RC7 and ended that era of prosperity. However, FaZe's destruction lead to the creation of…
- **Rain**: Rain is the spiritual successor to FaZe. 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.

Credits (pt. 2)

- **Louka:** For writing this document and classifying a lot of information concerning exploits and for trying to improve the exploit section (hey, I'm just following Rain's philosophy here!)
- **Chrome**: For helping certain Rain members with stuff despite being a relatively new guy to Roblox exploits. He's living proof that you can actually learn how to develop exploits without copy/pasting someone else's source. *(also for helping austin out with csgo hax but whatever :u)*

- **The Community**: We are developing exploits for them. Without you, we would be nothing, and we would be doing nothing interesting of our lives. And to be honest, our bank accounts would be empty too, so we also have to thank you for that.

  Don't go away!!

The idea

This is the idea in its most basic form:

1. We want unsigned script execution the client.
2. The client has no code that allows unsigned script execution, considering that the compiler was removed entirely from the code.
3. Therefore, we must come with our own way to do script execution.

To achieve that, we must first understand how you get code running in Lua. Roblox uses Lua for all of its in game scripts (LocalScripts, ModuleScripts and CoreScripts), so we obviously need to comprehend how we could possibly get Lua scripts running in Lua. Take note that Roblox heavily modified their Lua version (more protection has been added to make reverse engineering a pain more than anything).

Once we understand how we can get code running in Lua, it’s just a matter of finding how we could get that process to run without having the Lua compiler.

How Roblox runs LocalScripts

This is what Roblox does (client/server) to get LocalScripts running on the client:

1. Upon connecting to the Roblox server, the server sends the bytecode of every existing LocalScript in the game to the client that's connecting for further usage, along with additional information concerning security and encryption. The data received also contains the bytecode's hash, which is important.
2. The client then stores the received bytecode in a safe place and leaves it serialized and compressed.
3. When a LocalScript demands execution, the client will retrieve the bytecode from its emplacement, decompress it, deserialize it then feed it to LuaVM::Load, which is Roblox's routine for loading bytecode retrieved from the server.
    a. Roblox's deserializer will convert bytecode into a **Proto structure**, which we will see later. This should ring a bell.
4. LuaVM::Load will then insert the resulting Proto into a Lua function (internally called as "LClosure"), which is then pushed onto the stack, reading for execution. To execute the function, you can simply use lua_call/lua_pcall, or lua_resume if you're running said function inside of a coroutine.

Lua functions and what they are made of

- Each Lua function has a structure called a Proto. The proto contains a whole range of information concerning the function, such as the constants (and how many there is), the number of upvalues the function uses, how much stack size is necessary for the function to operate and most importantly, **the function's code**.
- A proto can only be ran if it's inserted into an LClosure then fed to luaD_precall. This means that if you want to run a proto and its code, you'll have to create an LClosure then set the LClosure's proto member to, well, the proto you want to execute.
- Serialized lua bytecode is basically a proto represented as pure binary data. Lua's undump function (luaU_undump) does the boring task of converting bytecode into the resulting proto, which then can be put inside of an LClosure then fed to luaD_precall… basically, same thing as the second point.
- In short: if you want to run any lua code, you must create a function (proto) first. Then you can feed that function to the lua API in order to execute it. However, Roblox does not have the Lua compiler for lua scripts, so what do we do instead?

Presenting: Proto Conversion (pt. 1) (the good)

- What we could do to get functions to execute on the Roblox client is to create a function through a normal Lua instance, then convert it to Roblox's format so Roblox can execute it.
- On paper, it sounds like a really clever idea: after all, the only thing Roblox did is change a few things to make it harder for us to run lua code, right?
- Sadly, that's untrue. While Roblox's Lua virtual machine is relatively the same *(except for instructions and how they get the registers from them)*, they added a metric ton of security to make sure that any reverse engineer cannot do anything to get unsigned lua functions running on their vm.
- However, this doesn't mean it's impossible. By bypassing Roblox's checks and doing some (stupid) things to get your code working, you can successfully attain stable script execution on the client. Of course, the "stability" of your exploit depends on how good your code is. Bad code = Bad stability.
- Proof that it's possible can be found everywhere: Seven, Intriga and Elysian are all exploits using proto conversion for script execution.

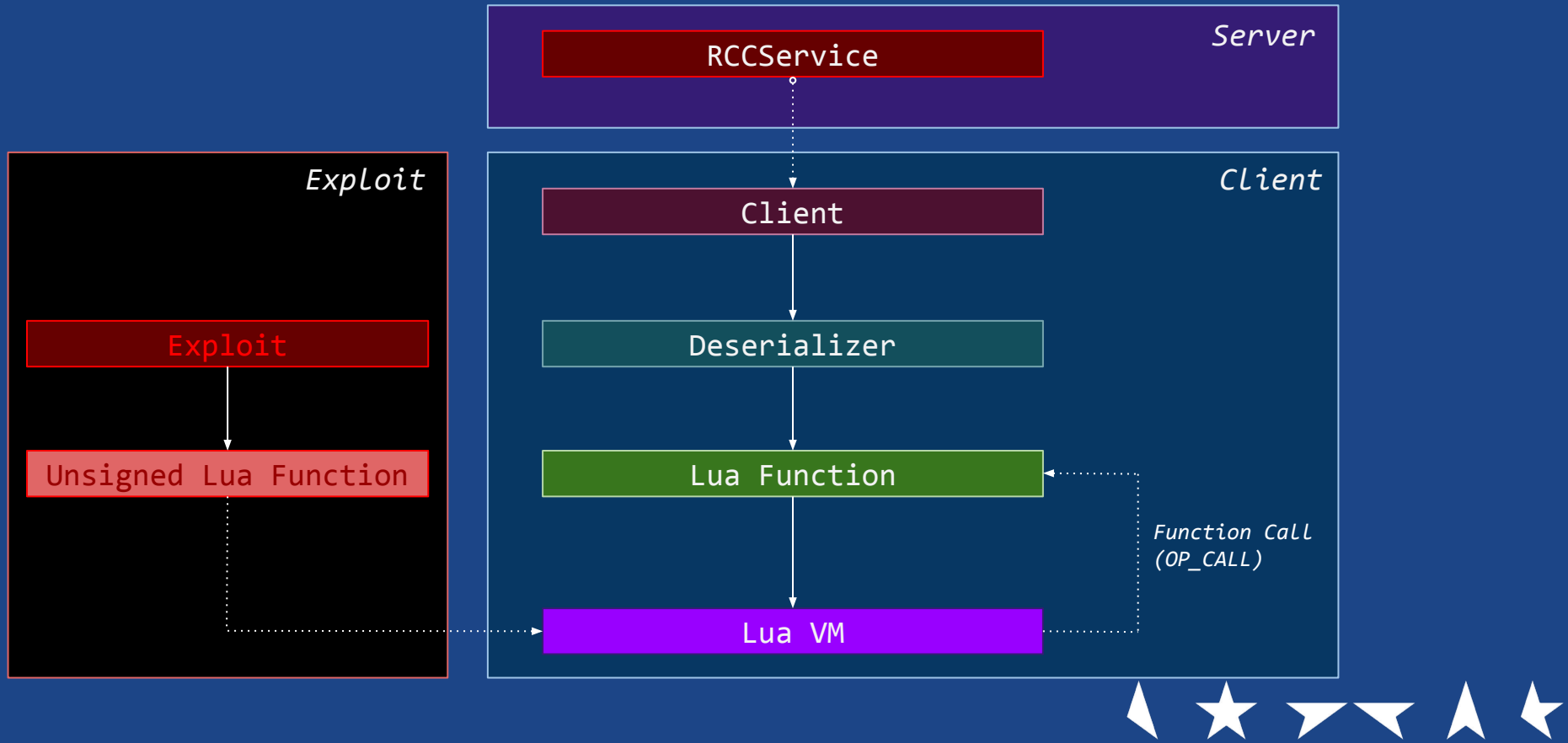
Presenting: Proto Conversion (pt. 2) (the bad)

- Proto Conversion, however, is not without risks: since you're relying massively on Roblox's codebase for script execution, them changing one single thing could break your exploit entirely.
- Additionally, there is still the possibility that Roblox'll hire some new security engineers that will prevent script execution through proto conversion for good. In short: proto conversion is **insecure**.
- Proto conversion is also slightly slow, depending on your implementation. If you look at Intriga, you'll probably notice its slowness. Whether proto conversion is what affects Intriga's speed is true or not, we do know that it can be slow.
- Let's not forget that you have to reverse a lot of encryption and obfuscation too. Roblox has a lot of encryption in place to make sure that you can't simply just change an instruction or two and get everything working. They changed registers, xor'd constants and modified the instruction set enum, making conversion a pain in the arse for certain reverse engineers.
- Again, this doesn't mean that proto conversion is a bad idea. Despite being slow, it's the fastest method out here to do script execution (next to CLVMs)

Presenting: Proto Conversion (pt. 3) (the “wat”)

- This might be surprising, but proto conversion is virtually the worse way to do script execution, despite being the (second) fastest and how many people uses it for their exploits, mostly because of how insecure it is and how you must constantly maintain your codebase to get it working.
- This, however, means that Roblox could have a hard time figuring out how the hell your exploit works. I’m serious: a Roblox security engineer spent a month trying to figure out how Seven worked (before he got the source from his creator himself), unable to comprehend what the exploit was doing. Hell, even after receiving the exploit’s code, Roblox is still unable to find a good way to patch those exploits, despite having 10+ (lol) years of experience in computer security and all of that bullshit.
- Yeah, “10+ years of experience in computer security”. That’s what Roblox requires for you to join their team and they somehow still aren’t able to come up with proper security that prevents us from doing anything bad to their client. Roblox, you seriously need to hire better engineers. Like me. Or anybody in Rain.

Presenting: Proto Conversion (pt. 4) (diagram)



Notes

- The deserializer used to convert bytecode into Lua closures is unique to Roblox, as their format has been heavily modified.
- The deserializer also decompresses code -- Roblox uses LZ4 compression in order to make their format hard to reverse AND to make it smaller when transferring the dump from the server to the client (ends up taking less bandwidth).
- The client also verifies the legitimacy of bytecode dumps by checking if the client has the dump's hash stored (as the client receives the hash upon connecting to the server). They use the xxHash algorithm for generating hashes with a seed (42 being the seed. Yeah, it's that easy to figure out.)
- A weird thing is that they encrypt (xor + index in array) the bytecode dump received by the server using the hash of the dump. This was known to be exploitable in the past (cough 1972 USNS Nuclear Plant incident cough), so something hacky could possibly be made with this.
- CoreScript bytecode is static: Servers DOES NOT recompile bytecode for CoreScripts each time a server boots up. Instead, it uses a set of keys already existing on the client for encryption and obfuscation.

Alternatives to Proto Conversion

- **Bytecode Conversion**: This is what RC7 claims to use. It takes Lua bytecode then converts it into Roblox bytecode, which is then fed to the deserializer and/or luaU_undump, exporting a proto in the process. Then, the proto is inserted into a function and eventually pushed onto the stack.
- **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