Objective & Goals

- Achieve script execution (Lua) through a custom virtual machine that operates on foreign lua states.
- Call the least API functions from Roblox in order to prevent detection.
- Copy the functions from rbx's Lua C API to our own DLL.
- Do the above without bypassing any checks.

However, due to our virtual machine being a separate entity from roblox's Lua codebase, some incompatibilities seems to occur. For example, Lua functions compiled on our side cannot be pushed then called ((function() end)()) in a convenient way as it would raise problems with the thread scheduler, events and such. In the cLVM, we resorted to using C functions as wrappers for Lua functions.

Problems & Issues

- Lua functions cannot be created and pushed normally, as it would create incompatibilities between Roblox's code and ours.
  - Solution: Use a C wrapper (CClosure with an upvalue pointing to your LClosure). C functions can be easily pushed using **r_lua_pushcclosure** and will not bring any incompatiblities.
- Functions like **lua_resume** and **lua_yield** will not work normally, as the VM is the one handling how yield works. Since we execute our code on the cLVM, yields act differently.
  - Solution: Use a "resume" C wrapper that just restarts the execution of your own VM. Push the wrapper when your VM receives a yield signal (function returns **LUA_YIELD**) then return.
- Implementing error handling without touching roblox's state is a pain in the arse (even if we create a thread using **r_lua_newthread**).
  - Solution: Set your error handler as an upvalue then set `RL->errfunc` to your upvalue index.

Problems & Issues (Part 2)

- Calling normal functions defined from LocalScripts/ModuleScripts does not work as the functions' instructions are not supported by our VM.
  - Solution: Fall back to Roblox's VM (**r_luaV_execute**) if the cLVM detects that the function is not one of ours. The check can be performed in many ways.
- Certain checks (such as the one located in `r_luaD_precall`) need to be circumvented, and copying isn't a possible solution here.
  - Solution: Modify the je instruction that precedes the return check to a jmp (essentially jumping over the check entirely) then restore it after the function has been called to prevent the memory checker from seeing our modifications.

Comparison between cLVM/(Seven, Elysian)

Comparison between Seven/Elysian and Last Occult

Seven/Elysian:

- Functions are ran on Roblox's lua environment (function conversion needed due to changes between our and their env)
- Many security features such as the memory integrity checker (code change detection) and the return checker needs to be circumvented to avoid detection
- A lot of elements (memory offsets, functions, keys, etc) need to be updated individually each time Roblox pushes an update, which takes a lot of time and work
- Quite easy to patch (changing how the VM works can break both exploits very easily)
- Mostly developed by a single individual with little to no organization. Development rate is slow.

Last Occult:

- Functions are ran on the Occult virtual machine (no conversion needed)
- Little to no security features need to be circumvented as we almost never touch Roblox's memory
- Less elements need to be updated than conventional exploits as we rarely use Roblox's own code
- Very hard to patch (Roblox will need to resort to physical detection)
- Codeveloped with competent developers and organized. Working as a team is always good.

Miracles that occurred during the cLVM’s development

- YieldFunctions worked without us having to implement any specific support for them. Continuing execution after a YieldFunction runs, however, requires proper **lua_resume** handling (via the C wrapping method). Roblox’s API
- functions such as **require**, **LoadLibrary** and the likes worked straight out of the box. Those functions are known to crash in a lot of exploits. Any of the
- new security measures Roblox decided to implement since the cLVM’s inception did not affect development and/or the exploit at all, unlike other exploits, effectively demonstrating the VM’s resilience against patches.
- Development was usually bugfree (other than functions and for loops). Implementing most of the LASM instruction set in cLVM was a breeze. The
- The cLVM is faster than Roblox’s own VM. Isn’t that nice?

The inner workings of the cLVM (represented as pseudocode)

```
const Instruction* i = function->code;    // retrieve the instructions from the func
for (;;)
i++;                                      // increase the PC register
switch (getopcode(*i))                    // do action based on instruction
      case OP_MOVE:
           -> do roblox's OP_MOVE
      case OP_LOADK:
           -> do roblox's OP_LOADK
      case OP_LOADNIL
           -> do roblox's OP_LOADNIL
      …                                   // repeat for every instruction
```

For every instruction there is, we execute Roblox's code part for every instruction there is (except **OP_CALL**, **OP_TAILCALL**, **OP_RETURN** and **OP_CLOSURE**). This makes the cLVM do what Roblox's VM intends to do but instead retrieves the opcode and register values (A, B/Bx/sBx, C) from our own instructions. This prevents us from spending 300 hours on researching Roblox's modifications to their LASM implementation (cough cough Seven cough cough Elysian cough cough RC7 cough cough).

The inner workings of the cLVM (C wrapper)

```
static int occult_c_wrapper(r_lua_State* rL) {
     TransientWrapperData* twd = r_lua_touserdata(rL, lua_upvalueindex(1));
     if (twd)
          occult_luaV_execute(twd->L, rL, twd->func);
     return LUA_MULTRET;
}
```

We have to thanks Lua to make C functions work like they do. Using a C wrapper, **r_luaD_precall** will adjust the stack for parameters and return values fully automatically, saving us time and making our life easier at handling LClosures. The only tough part of the wrapper is getting the upvalue to register properly without touching the stack at all in **OP_CLOSURE**, but that was done by modifying the **r_lua_pushcclosure** function, and it was relatively easy (with some issues here and there but we fixed ‘em all.) At the moment, the only problem we’re having is the upvalue being garbage collected when the wrapper gets invoked. That’s one of the few obstacles left in the cLVM’s development and we already have an idea or two on how to fix the GC issues.

The inner workings of the cLVM (retcheck bypass)

```
void occult_r_luaV_execute(DWORD rL, int nexeccalls) {
    DWORD robloxVm = OCCULT_OFFSET(OCCULT_IDABASE, OCCULT_RLUAV_EXECUTE);
    auto restore = occult_luaX_backretcheck();
    ((void(__cdecl*)(DWORD rL, DWORD nexeccalls))robloxVm)(rL, nexeccalls);
    restore();
} // example of usage
```

The only check we have to circumvent in cLVM is the return check, which is located in various Lua functions (most notably the C API functions, but also a lot of functions used by the VM -- including the VM itself). To do this without having to bypass Roblox's integrity checker is pretty easy: we simply hook the return check function, store the flags somewhere then call the protected function. After the function has been executed, we just have to restore the flags and the initial bytes of the function (where our hook got inserted), then the magic is done. This works for a vast majority of protected functions, and could be applied to more programs than just Roblox. Ok, maybe not Denuvo or some really OP anticheat but whatever.

Don’t do this if you want to make a script execution exploit

- While the concept looks really great and seems easy in theory, when applied in practice, developing the virtual machine is pretty hard and requires a lot of research.
- It’s very complicated, even if you are a mediumly good programmer. There are definitely easier ways (such as the outdated proto/bytecode conversion method) to perform script execution than using a VM to do your bidding.
- If you were to ask me, I would only recommend you to work on a VM if you’re looking for a new challenge in reverse engineering and you’re already good with Roblox reversing. Other than that, nah, it will take you forever.
- In short: git gud.

Resources

- This neat little PDF explaining how LASM and Lua's VM works. Lua (5.1)'s
- source. This website has a bunch of notes and very useful stuff that might aid you in studying how Lua works. The god damn Lua manual. I found it
- useful to search what function pushes values to the stack and what function pops them from the stack. This explains how malicious Lua bytecode can be
- used to attain shellcode execution -- from Lua itself. Not useful in the development of the cLVM, but certainly interesting and gives insight on how fragile the VM is.

For more Lua questions, use Google. Google is your friend when it comes to questions concerning programming. Or maybe Stack Overflow.

Credits

- Booing: *He's the one who got the VM idea. He developed a prototype this year (or last year?) but it's not completely functional as it lacks some important Lua features (such as upvalues)*
- alureon: *Being a good friend and not bashing on me like Jordan for, well, existing. Very cool guy. Also, he's the second person to (commercially) adopt the VM concept, making Intriga the second VM-based exploit.*
- Autumn: *Helping us with wrappers and C functions in general. Also a very good friend of mine.*
- Austin: *Giving us the C wrapper w/ upvalue(s) idea for Lua functions.*
- Pkamara: *Testing and giving me advice concerning code styles.*
- John/unrealskill: *big milk cucumbers*
- Internal: *soap*
- ConvexHero: *For doing nothing that affected the cLVM (:P)*