HSVM (Fiber VM Execution Method)

By: NotALeafyWannaBe

How Do You Inject/Hook Into Fiber? (Part 1)

There are a few methods to injecting/hooking onto fiber. One of the best ones working so far is the ***C# NamedPipe Hook Method***, this method abuses Fiber's named pipe that it uses to interact with its C++ dll's and its C# dll's and fiber itself. The NamedPipe allows dll's to hook C++/C# functions to fiber's C# functions, and fiber's C# functions to other C# application/dll's etc.

How Do You Inject/Hook Into Fiber? (Part 2)

We can use this NamedPipe Method to hook namespaces, and classes to it, although fiber only looks for a namespace called IKernel, and specific classes within the IKernel namespace followed by the specific functions within the classes.

This isn't a problem as we can make a IKernel namespace, followed by the specific classes and the functions within the classes within the IKernel namespace. These classes include **OpCodes, LuaBaseFunctions and LuaFunctions**.

How Do You Inject/Hook Into Fiber? (Part 3)

To hook our IKernel namespace and it's contents to fiber, we can use the **.FiberHookScript** file extension method that is hooked to fiber from one of fibers C++ dll's for fibers render engine, and send our C# application/dll file location to fiber and hook our IKernel namespace and its contents to fibers IKernel namespace and its contents.

Using the LuaBaseFunctions class we can actually hook fiber functions to ours. Which allows us to hook lua functions like lua_pcall, lua_pushstring etc, from fiber to our C# application/dll. **(This NamedPipe hook method requires all namespace, class, and function names to be the exact same as the ones your trying to hook in fiber.)**

What Is A HSVM?

[Note: This is explaining a HSVM for Fiber Execution Methods This does not apply to all HSVM's.]

A **HSVM (Hooked State Virtual Machine)** is when you hook the IKernel namespace and it's classes followed by the **OpCodes** class. The OpCodes class is the class that contains all of Fibers VM instructions like GetGlobal, Jmp etc.

After hooking the OpCodes class we can hook our own custom lua instruction functions to fiber, then fiber will use the hooked instruction functions instead of its current ones. Thus making a HSVM.

What are the Downsides to a HSVM?

The Downsides to a **HSVM** include the following.

1. Some lua functions, can't be hooked from the **LuaBaseFunctions** class, which means our VM can't run it (this will happen for some functions and is just the nature of a HSVM for fiber execution), this means you have to run that code on fibers VM which requires doing lots of work and bypassing lots of checks.

2. This is what we try to avoid the most, for the reason being that we want to run as many functions and as much code as we can on our own state and our own VM instead of fibers VM, but this will always happen with a **HSVM**, and there isn't anything we can do about it.

How Is The Lua Code Sent To Fiber?

We can hook fibers NamedPipe and run lua code by using the **.FiberLuaScript** file extension method that is hooked to fiber from one of its C++ dll's to allow the dll to run lua code on fiber for fiber's render engine.

Problems & Issues (Part 1)

- Lua functions can't be created and pushed normally, as it would create incompatibilities between Fiber's code and our code.
  - *Solution*: Use the IKernel.LuaBaseFunctions class and hook fiber's lua functions to ours.

- Fiber crashes if table is not restored, Example:

```
public static object Jmp(lua Jmp args here)
{
    Fiber's lua Jmp code here //<- code
    return 0; //crash happens here because by inserting the code above, we moved a table to insert -
    //that code and now after the code we inserted gets ran everything after it is not there -
    //because we moved the table that stores that code somewhere else to insert our code, which -
    //causes a crash in fiber.
}
```

  - *Solution*: Use the address from fiber that is the pointer to the table we moved to insert the code and return that address which will restore the table to the correct place as if we never inserted code, Example: the Jmp opcode uses the luaM_realloc address for its table, so all we have to do is return that address in the Jmp opcode by doing ***return Fiber's luaM_realloc address here;*** and that will restore the table and fiber will not crash.

Problems & Issues (Part 2)

- When using the IKernel.LuaBaseFunctions class to hook fiber's lua functions to ours, fiber does not hook the function because the table in fiber's function and the table of our function we are trying to hook are completely different, causing fiber to have a ***Invalid Cast Exception***.
  - *Solution*: Use the address from fiber that is the pointer to the table of the function we are trying to hook to our function and return that address in our function by doing ***return Fiber's lua_FunctionNameHere address here;*** in our function.

- Fiber does not run our opcode correctly if base_ is trying to be set to another value, Example: ***base_ = L.base_;***, only our base_ value gets set to L.base_, and not Fiber's causing fiber to not run our opcode correctly.
  - *Solution*: Use a VMStack method to push L.base_ value to fiber after we set our base_ value to L.base_ value, this can be done by doing ***stack.HSVM.push(L.base_);*** in our opcode.

- Some opcodes will error if there is a lua script error Example: ***local a = nil .. nil print(a)***, that script will cause our Concat opcode to stop running because you can't concat a nil with another nil, and this can and may happen to some other opcodes as well.
  - *Solution*: Use a ***try catch*** code block on the opcode, or to be safe use the ***try catch*** code block on all of our opcodes, so if a error happens in one of our opcodes it gets ignored and the execution of our opcode can continue.

Why don’t HSVM’s memory leak?

HSVM’s can and may memory leak if there **not** coded correctly, but when **HSVM’s** are coded correctly then 99.9% - 100% of the time there is no memory leaking, some functions in fiber like for example ”opening a new gui” will **USE** memory **but it wont leak**.

One of the biggest reasons that a **HSVM** does not memory leak when coded correctly is because your running almost everything (besides some functions) on your VM, so you have control of what's being executed on your VM and **HOW** to handle what's being executed on your VM, this gives you the opportunity to make sure that scripts that are being ran on your VM do not memory leak by calling things like “GC.Collect();”, “lua_gc” etc.