MemLua Introduction

Table of Contents:

I. Initializing

MemLua was designed to reduce the number of headaches when writing a DLL exploit in C++. By replacing every tedious coding task with functional that's so utterly simple, you can have more than just an exploit running; you can have multiple exploits running, and you can run them from the Web, or from a Lua file on your system.

This means with MemLua you can modify your own exploit in your own text editor, without even having to recompile the project in Visual Studio.

To begin using MemLua in C++, include this at the top of your project:

```
#include "MemLua/MemLua.h"
```

Now, in your main function(whatever primary function you're working with in your DLL), add these lines:

```
MemLua::init(); -- initialize memlua!
MemLua::load(R"(
	print("MemLua loaded! Hi guys!");
)");
```

That seems pretty easy, and self-explanatory (really what I was aiming for).
Maybe you want to load it from a file:

```
MemLua::init();
MemLua::loadfile("C:/Users/.../Desktop/memlua_script1.lua");
```

Or from the internet:

```
MemLua::init();
MemLua::loadhttp("http://www.yourwebsite.com/examples/yourfile.lua");
```

So there you go, now we've covered how to load it up.
With MemLua you can essentially make plugins, and communicate with

your DLL C-side (for more info see **VI. C side**)
Now let's see what we can actually do

II. Basics

With MemLua, text translation/converting is the standard that it's built on.

To convert a string representing an address to an address, we use toAddress. e.g.

```
local address = toAddress("0x1248BBC0"); -- `address` stores the number value, 0x1248BBC0 now
```

To convert a string representing a byte value to a byte, we use toByte.

```
local byte = toByte("0xC0"); -- `byte` stores the number value, 0xC0 now
```

To convert a byte or an address to its string representation, we use toString.

```
local address = 0x05E09008;
local byte = 0x88; -- this is really just 0x00000088 in lua.
print(toString(address, 8)); -- prints '05E09008' to the console
print(toString(byte, 2)); -- prints '88' to the console
```

toString(address) will default to toString(address, 8)

MemLua automatically configures these important globals once initialized:
base, basesize, vmp0, vmp1

So, if you're intending to make an aslr/rebase function to let you use addresses in IDA Pro(based to 0x400000), you would do something like:

```
function aslr(address)
	return (address + base) - 0x400000;
end
```

That's easy enough

Now, there are an abundance of functions for manipulating and controlling the game's memory. Let's go over the basics -- reading and writing.
For simplicity I left the majority of functions open in the global scope (this will probably change later on).

Here are their descriptions and usages:

readByte(address)
Returns the byte value that it reads at address, as a lua number.

readShort(address)
Returns the short/word value(2 bytes) that it reads at address, as a lua number.

readInt(address)
Returns the int/dword value(4 bytes) that it reads at address, as a lua number.

readFloat(address)
Returns the floating point value(4 bytes) that it reads at address, as a lua number.

readQword(address)
Returns the qword-sized value(8 bytes) that it reads at address, as a lua number.

readString(address, [optional]: length)
Returns the full string that it reads at address, as a lua string. Or, it force reads the specified length.

readBytes(address, length)
Reads the specified number of bytes starting at address, and returns it as a lua table.

writeByte(address, value)
Writes the byte value to address

writeShort(address, value)
Writes the short/word value to address

writeInt(address, value)
Writes the int/dword value to address

writeFloat(address, value)
Writes the floating point value to address

writeQword(address, value)
Writes the qword-sized value to address

writeString(address, value)

Writes the entire string at address

writeBytes(address, value)
Writes the table of bytes (all of them) at address

Memory pages are very easy to manage; I intended to mimic the Win API as closely as possible, in Lua.
Here are some examples:

allocPageMemory([optional]: size)
Returns the address of a blank, newly allocated memory page that has execute/read/write priviledges. (if no size is provided, it defaults to 4096 bytes)

freeMemoryPage(address)
Deallocates and frees the whole memory page that address is a part of.

setMemoryPage(address, size, protection)
Sets the protection of the memory page at address. (you can use most of the Win API macros in MemLua, such as PAGE_EXECUTE_READWRITE, PAGE_READONLY, etc.)

queryMemoryRegion(address)
Returns a lua table/replica of a MEMORY_BASIC_INFORMATION for the entire memory page that address is a part of.
Example:

```
local mbi = queryMemoryRegion(0xD0GF00D);
print("Base address of region: " .. toString(mbi.BaseAddress) .. ".
Region size: " .. toString(mbi.RegionSize) .. ".");
```

Traversing through and analyzing functions in memory is key to finding what you need, and it should be made as easy as possible.
Here are some functions to help:

nextPrologue(function)
Returns the very next function in memory, right after function. (can also be used at any point inside function)

nextEpilogue(function)
Returns the very next epilogue of the function (will return its Ret or Retn instruction).

getPrologue(address)
Goes backwards until it reaches the prologue of the function that address is a part of.

getFuncSize(function)
Returns the full size of the function in bytes.

getFuncRet(function)
Returns the local stack size, as indicated by either a Ret or a Retn instruction at the end of a function. (returns the Ret value, or 0 if Retn)

getFuncPointers(function)
Returns a table of all static variables/memory locations used in a function.

getFuncCalls(function)
Returns a table of every function(from every call instruction) used in the function.*

isPrologue(address)
Returns a boolean of whether address is a function prologue or not

isEpilogue(address)
Returns a boolean of whether address is a function epilogue or not

nextCall(address, [optional]: location, [optional]: include_naked_prologue_calls*)
Returns the very next function call found in memory, starting at address. ignores the one it's currently at (if there's a call instruction at address).
if location is set to true, it will return the address of the call instruction rather than the function it is calling.
include_naked_prologue_calls sounds self-explanatory but consider the * footnote

Example:

```
.text:0118BC0B          mov eax, [ecx+esi+60h]
.text:0118BC0F      cmp eax, [ecx+esi+68h]
.text:0118BC13          jb short loc_118BC20
.text:0118BC15          push 1
.text:0118BC17          push esi
.text:0118BC18          call sub_119E680

local address = 0x118BC0B;
print(toString(nextCall(address))); -- prints '119E680'
print(toString(nextCall(address, true))); -- prints '118BC18'
```

prevCall(address, [optional]: location)
Returns the previous function call found in memory, starting at address. ignores the one it's currently at (if there's a call instruction at address).
if location is set to true, it will return the address of the call instruction rather than the function it is calling.

nextFuncRef(address, function, return_prologue)
if return_prologue is set to true, it will return the prologue of the very next function in memory that has a call to 'function', starting at address.
Basically, address is just a marker/starting point, and it goes to the very next XREF of the function that you provide.
if return_prologue is false it will just return the location of the very next call instruction that calls 'function'

prevFuncRef(address, function, return_prologue)
Exactly the same as nextFuncRef, but looks behind address (continually goes backwards until the next XREF of function is found).
Returns the prologue of that function is return_prologue is set to true.

* There is a small issue with MemLua's call-instruction-related functions, but also a compensation for it.
Basically, when you get all the calls in a function, or go to the next call, the current method doesn't rely on disassembly. It may have a chance of picking up whatever looks like a call but isn't.
The only reason it does this however, is because what if the function has a naked prologue?
It would have no way to determine if it is really a function or not.
this WILL get fixed in a later version. for now, I add booleans at the end of call-based functions to ignore naked prologue functions in exchange for accuracy.

MemLua has alot of miscellaneous functions, most of which mimic C++:

malloc(size)
Returns an address for [size] bytes of allocated memory

free(address)
Deallocates/frees memory created with malloc

wait(milliseconds)
Suspends the current lua thread for [milliseconds] and then continues execution

system(cmd)
This calls the system() function with the string `cmd` as the arg. This function is deprecated due to possible abuse in obfuscated codes. It may become limited later on.

MessageBox(msg)
Displays a messagebox with the text 'msg'

BIT_AND(a, b)
Returns a & b

BIT_OR(a, b)
Returns a | b

BITSHIFT_LEFT(a, b)
Returns a << b

BITSHIFT_RIGHT(a, b)
Returns a >> b

And who could forget these:

readFile(filename)
Returns the data in the file as a string/byte-string

saveFile(filename, data)
Saves `data` as either bytes or a string into the file.

readRawHttp(url)
Reads the data at URL into a lua string and returns it.

Now, the next couple functions are extremely useful (to me at least), and this is what I was aiming for with MemLua:

getExport(dllname, functionname)
Returns the DLL export, 'functionname', as found in the DLL.
Example:

```
local freeconsole_address = getExport("KERNEL32.dll",
"FreeConsole");
```

invokeFunc(function, arg1, arg2, arg3, arg4, ...)
invokes a function as either an __stdcall or a __cdecl, with any number of args.
This allows you to easily call **any function in the processes memory, using its address alone**.
The args you pass can be a string or a number. You may have to compensate for pointers or other arg types by allocating that arg with malloc.

But we can even use this to call Win API functions that haven't been binded in MemLua:
Example:

```
local messagebox_address = getExport("USER32.dll", "MessageBoxA");
invokeFunc(messagebox_address, 0, "Hello, world", "Test", 0);
```

Now, what if the function you want to call isn't a __stdcall or __cdecl? No problem. If you want any function to be interpretted as a __cdecl, and completely regardless of its current calling convention (never having to worry that itll change), simply use createRoutine.
Let's pretend the function at base+0x5E09C0 is a __fastcall,
but we KNOW that it takes 2 args.
This is ALL the information that createRoutine needs, in order to perfectly get the calling convention, and create a __cdecl function to replace it:

```
local cdecl_version = createRoutine(base+0x5E09C0, 2);
```

To create and start a new thread, which will execute an assembly code, you can use:
startRoutine(address, [optional]: parameter)

To make an entire copy of a function from the code section in the process, use:
local copy = cloneFunction(address);

III. Detouring

A big part of, and big reason I created MemLua and an x86 disassembler myself, was to reduce the enormous effort that was involved in making Hooks and Trampoline Detours, all just to pull a single value from a register.

There is much more than one way to do this now:

debugRegister(address, string register, offset of register)

if string register is "ebp", and offset of register is 8, then as soon as 'address' is executed, this will return the value at [ebp+8] of the function, as a lua number.
So with a single line, it waits synchronously for the function to be called, and then JUST like that, you snatched the first function arg(ebp+8).
It automatically removes ALL traces of the detour, and execution flow remains the same.
Granted, this function is limited to a single value, per hook.

Thankfully, we have this:

```
createDetour(address, function(data)
	-- you can now read:
	-- data.eax, data.ecx, data.edx, data.ebx, data.ebp, data.esi, data.edi, and much more
	print(toString(readInt(data.eax))); -- display eax's value
	print(toString(readInt(data.ebp+8))); -- display the first arg
```

```
        print(toString(readInt(data.ebp+12)));-- display second arg
        endDetour(); -- provided for this current function
end)
```

Now, as great as this is, it still has another huge limitation...It can't exactly read offsets of the registers per se...except for ebp, since I programmed it to store as many ebp args as possible.
It needs alot more testing and improvement but it's still somewhat elegant.

Now, there's one more method I added for hooking (only a single value), but, it runs asynchronously:

```
local tracer = setTracer(address, string register, offset of register);
tracer.start();
-- put code here that triggers the address to be executed
wait(.1);
local value = tracer.stop(); -- returns the value from [string register + offset of register] (the very first occurence).
```

This is a little more complex, but basically, you place it at an address.
You let it do its thing, or you cause it to get executed naturally.
You then scoop up the value, which will be set to the first occurence it was executed (so, if the function ran 3 times, and that addresses instruction was executed, it returns the register/offset of register's value, from the first time it ran).
something like that.

IV. Breakpoints

My favorite thing about this entire API is this right here.
I wanted to make breakpoints as easy as I made detouring.

So first thing's first, we initialize the VEH:

```
veh.init();
```

We add a prehandler in case there's something we want to ignore (in the case of a game's anti-page-debugger/anti-cheat):

```
veh.preHandler = function(eip)
	-- ignore stuff coming from .vmp or other junk
	if (eip > vmp0 and eip < vmp1) then
		return VEH_BLOCK;
	end
	return VEH_CONTINUE; -- otherwise, continue happily with breakpointing
end
```

Now, we set a breakpoint on an address like so:

```
veh.setBreakpoint(address, size, function(this_id, ctx)
	print(toString(readInt(ctx.Ebp+8))); -- read the first arg in the function using our page exception breakpoint

	veh.endBreakpoint(this_id);
end)
```

The thing that's great about this is address can either be in the processes .text/code section, or it can be anywhere in Virtual memory(where it will pick up anything from .text that tries to access the object in memory).

Now, doing this whole api on my own was pretty stressful. This is not very great at handling threads. This whole API may need some rewrites here and there to better support multi-threading or at least allow hyperthreading in the lua state.

For now, you can compensate like so:

```
local data = 0;
veh.setBreakpoint(address, size, function(this_id, ctx)
	data = readInt(ctx.Ebp+8)));
	veh.endBreakpoint(this_id);
end)
while data == 0 do wait(1) end

print("Breakpoint Got: " .. toString(data));
```

Yes, pretty bad, but it gets the job done in 8 lines so I'm quite satisfied. Alot of times you won't need more than one of these breakpoints in your code

V. Disassembler

The entirety of MemLua is built around my old disassembler I called "EyeStep", but this time, it's just "disassembler".
Trust me this one is much more refined.

To disassemble an instruction at an address, do this:

```
local instr = disassemble(address);
```

See how easy this is?
Now, instr is a table containing a ton of information about the disassembled instruction. We have:

instr.len	size of instruction (in bytes)
instr.pref	prefix flags
instr.flags	flags, to determine what operands are used, what values they contain, etc.
instr.src	data for source operand
instr.src.r8	the register used in the source operand (if it's 8-bit)
instr.src.r16	the register used in the source operand (if it's 16-bit)
instr.src.r32	the register used in the source operand (if it's 32-bit)
instr.src.imm8	a byte-size offset value used in the source operand
instr.src.imm16	a word/short offset value used in the source operand
instr.src.imm32	a dword/int offset value used in the source operand
instr.src.disp8	a constant byte-size value used in the source operand
instr.src.disp16	a constant word/short value used in the source operand
instr.src.disp32	a constant dword/int value used in the source operand
instr.dest	data for destination operand
instr.dest.r8	the register used in the destination operand

(if it's 8-bit)
instr.dest.r16 the register used in the destination operand
(if it's 16-bit)
instr.dest.r32 the register used in the destination operand
(if it's 32-bit)
instr.dest.imm8 a byte-size offset value used in the destination operand
instr.dest.imm16 a word/short offset value used in the destination operand
instr.dest.imm32 a dword/int offset value used in the destination operand
instr.dest.disp8 a constant byte-size value used in the destination operand
instr.dest.disp16 a constant word/short value used in the destination operand
instr.dest.disp32 a constant dword/int value used in the destination operand

To determine flags of the instruction, we use BIT_AND like so:

```
local i = disassemble(address);
if (BIT_AND(i.flags, FL_DEST_IMM8)) then
	-- . . .
end
```

What this will do, is if the DEST(destination) operand has an imm8 offset value, (for example: mov eax,[edx+08]), it would return true. So, you could do:

```
local i = disassemble(address);
if (BIT_AND(i.flags, FL_DEST_IMM8)) then
	print(toString(i.dest.imm8, 2)); -- Display the byte-size imm8 offset.
```

```
        -- If the instruction at address was: mov eax,[edx+08]
        -- then this would display: "08".
        -- i.dest.imm8 would be the value 0x08.
end
```

On that same note, it's also good to make sure that the instruction uses both a source and destination operand before continuing.
We can change it to the following:

```
  local i = disassemble(address);
-- check if instruction has both a source and destination operand
if (BIT_AND(i.flags, FL_SRC_DEST)) then
        print("instruction has a src and dest");

        if (BIT_AND(i.flags, FL_DEST_IMM8)) then
                print(toString(i.dest.imm8, 2)); -- Display the byte-size imm8
offset
        end

        if (i.dest.r32 == REG32_EDX) then
                print("the destination operand uses the EDX register!!");
        end
end
```

Here are most of the flags that we can use to determine things about the instruction:

```
FL_SRC_DEST
FL_SRC_IMM8
FL_SRC_IMM16
FL_SRC_IMM32
FL_SRC_DISP8
FL_SRC_DISP16
```

FL_SRC_DISP32
FL_DEST_IMM8
FL_DEST_IMM16
FL_DEST_IMM32
FL_DEST_DISP8
FL_DEST_DISP16
FL_DEST_DISP32
REG32_EAX
REG32_ECX
REG32_EDX
REG32_EBX
REG32_ESP
REG32_EBP
REG32_ESI
REG32_EDI

VI. C side

. . . (will add later)

VII. The End

Hope this helps explain as much of MemLua as humanly possible.
It's still very much under development, but it's usable enough to run full exploits from a single lua file on my PC.

The best thing about this, is using Notepad++ you can modify your own DLL. Have no worries about recompiling, having silly visual studio errors(most of which are intellisense being stupid), because Lua is so incredibly simple to use, and fun.

I hope that this invigorates a passion for exploiting once more.