luaexts Documentation

Release 1.0.0

Kurt Hutchinson

May 12, 2016

CHAPTER 1

Overview

Luaexts is an extension for Windbg (and the other dbgeng-based debuggers) that provides access to the dbgeng COM API from Lua. It also implements a lightweight framework for writing extensions in Lua.

Project source:	https://bitbucket.org/kbhutchinson/luaexts/
Project documentation:	https://luaexts.readthedocs.io/

1.1 Getting started

- Download luaexts-x86.zip and/or luaexts-x64.zip, depending on your debugger's bitness, and unpack into your .extpath.
- Within Windbg, run the following command: `.load luaexts`
- To verify that the extension loaded correctly, run the following command:

```
!lua say('success')
```

The growing list of supported dbgeng API functions can be accessed through the global table *dbgeng*. The table is split up into sub-tables matching the organization of the dbgeng API itself. So the functions from the IDebugSymbols interface (as well as IDebugSymbols2, etc) can be found in the *dbgeng.symbols* table.

As an example, here's how to call the IDebugSymbols::GetSymbolTypeId function, use the return values to call the IDebugSymbols::GetTypeName function, and then print the type name of a variable named *foo* that exists in the current scope:

```
!lua local module, typeid = dbgeng.symbols.get_symbol_type_id( 'foo' ) say( dbgeng.symbols.get_type_n
```

or without the temporary variables:

```
!lua say( dbgeng.symbols.get_type_name( dbgeng.symbols.get_symbol_type_id( 'foo' ) ) )
```

Small side note: A global function called `say()` is available that takes the place of Lua's normal `print()` function: it prints whatever is given to it and appends a newline. The `print()` function has been overridden to output without a newline.

1.1.1 Building

The dbgeng headers and libs, required to build luaexts, are not included in this repository because I'm unsure of their redistribution license. So to build luaexts, you'll need to do the following:

- Clone this repository.

- Retrieve the dbgeng headers and libs.
  - Once you have Windbg installed, the headers can be found in `<install-dir>\Debuggers\inc` and the libs in `<install-dir>\Debuggers\lib`. Copy the headers to `luaexts\inc` and the libs to the appropriate platform directory under `luaexts\lib`.
- Build `luaexts.sln` with Visual Studio.
- Binaries will be placed in directories under `luaexts\bin`.

Remember when building locally that, in addition to **luaexts.dll**, you also need the contents of `luaexts\src\luaexts\lua` copied to your Windbg extension path, next to the DLL or in a `luaexts` directory next to the DLL.

1.1.2 General usage

Using the facilities provided by luaexts is usually done at the debugger's command prompt by entering extension commands, or by writing Lua scripts and running them. This section will discuss the commands that drive these interactions.

The `!lua` command

The primary interface point for using luaexts is the `!lua` command, which consumes the rest of the command line as a chunk of Lua code, and simply runs it. Quotation marks surrounding the Lua code are not required, as the dbgeng command parser will automatically consume all text until the next semicolon, which is the debugger's command-separation marker. If a semicolon is needed within the Lua code itself, then enclosing quotation marks *are* required, to prevent the dbgeng command parser from interpreting the chunk as multiple dbgeng commands. Fortunately, semicolons as statement-termination markers are optional in Lua syntax, so they are needed very rarely.

Each `!lua` command runs as a separate Lua chunk. This means that variables declared as `local` will be in scope only during the execution of one command. Variables assigned to, but not declared as `local`, will become global variables, available across all subsequent chunks run by luaexts. This can be useful, but care must be taken, because the memory being used by data structures that are pointed to by global variables cannot be freed by the Lua garbage collector as long as they are alive. Assigning `nil` to a global variable will remove the reference to the data the variable was pointing to, at which point the data will become a candidate for garbage collection.

Lua's `print()` function, which normally outputs to stdout, has been replaced by a function that directs output to the debugger command window. Another slight change is that usually Lua's `print()` will append a newline, but the `print()` provided by luaexts does not append a newline, in order to provide more control over exact text output. Instead a global `say()` function exists that automatically appends a newline.

As an example, here's how to call the IDebugSymbols::GetSymbolTypeId function, use the return values to call the IDebugSymbols::GetTypeName function, and then print the type name of a variable named *foo* that exists in the current scope:

```
!lua local module, typeid = dbgeng.symbols.get_symbol_type_id( 'foo' ) say( dbgeng.symbols.get_type_n
```

`dofile()` for running scripts

Although the `!lua` command provides the full expressiveness of Lua, it can be tedious to compose complicated code chunks at the debugger command line. We can take advantage of built-in Lua functionality to ease the pain. Lua provides a `dofile()` function that takes a file path, loads the file as a Lua code chunk, and runs it. Like the scoping of `!lua` chunks, variables declared as `local` within the file will be scoped to the file's chunk, while variables assigned to but not declared `local` will become global and available to any subsequent chunks run by luaexts.

Developing a script file can then be easily done by opening the file in your favorite text editor, making changes, and then running `!lua dofile('path\to\script.lua')` from the debugger command line to test each change. (Protip: In Windbg, the previous command can be retrieved from the command history by pressing the Up arrow key when keyboard focus is in the command line input field.)

CHAPTER 2

API

The luaexts API is divided into two categories: 1) the exposed dbgeng COM API; and 2) the additional API that luaexts provides, sometimes as higher level constructs on top of the dbgeng API like *cppobj*, and sometimes as Lua-extension scaffolding like register_extension().

2.1 luaexts

2.1.1 Global functions

Most of the API provided by luaexts is scoped within an appropriately named global table. However, there are a number of global functions that are accessible without prefixing.

print (*anything*) → nil
Takes any number of arguments of any type, converts them to strings, and outputs those strings to the debugger's output window with no other formatting.

say (*anything*) → nil
Exactly like *print()*, but appends a newline to whatever is output.

dml_print (*anything*) → nil
Similar to *print()*, but outputs using debugger markup language (DML).

2.1.2 cppobj

cppobj provides a high level interface for working with C++ objects. The goal is to allow intuitive manipulation of those objects through familiar syntax.

For example, here's how to wrap an object named *foo* from the current scope, and print the value of its *bar* data member:

```
local foo = cppobj.new( 'foo' )
print( foo.bar )
```

Lua's metatable feature is used to overload the indexing operation performed by `foo.bar`, in order to return a new cppobj for the *bar* data member, which is then stringified. Stringification is overloaded to return a dbgeng "view" of an object, i.e., the same string that the dbgeng COM API would return when asked to return an object as text. A native Lua value can be retrieved for fundamental types by using *cppobj.value()*.

Due to this overloading of the indexing operation, and the fact that instance methods are also implemented in Lua by overloading indexing, most of the functions that operate on cppobj instances are instead implemented as class functions that simply take the cppobj as a paramater.

Class Functions

Class functions are accessible through the global *cppobj* table. Many of them operate on a cppobj, and return information about the wrapped C++ object.

`cppobj.`**`new`**`(`*expr*`)` → cppobj
Creates a new cppobj, representing a C++ symbol.

Parameters **`expr`** (*`string`*) – C++ expression that evaluates to a symbol

Returns New cppobj representing the given symbol

`cppobj.`**`is_cppobj`**`(`*value*`)` → boolean
Checks whether the given Lua value is a cppobj.

Parameters **`value`** – Lua value to check

Returns True if *value* is a cppobj, otherwise false

`cppobj.`**`value`**`(`*obj*`)` → Lua value
For fundamental types (*`cppobj.kind()`* returns "base", "enum", or "pointer"), returns an object's value as an appropriately typed Lua value.

For array types, (*`cppobj.kind()`* returns "array), returns the memory offset of the array.

For other types, returns `nil`.

Parameters **`obj`** (`cppobj`) – Object whose value to return

`cppobj.`**`kind`**`(`*obj*`)` → string
Returns the "kind" of a C++ object. The kind will be one of the following values:

- `base`: Fundamental type like `bool`, `char`, `int`, `float`, etc
- `enum`: Enum value
- `pointer`
- `array`
- `class`: Class instance
- `function`
- `unknown`: Some other, unrecognized kind

Parameters **`obj`** (`cppobj`) – Object whose kind to return

`cppobj.`**`type`**`(`*obj*`)` → string
Returns an object's type.

Parameters **`obj`** (`cppobj`) – Object whose type to return

`cppobj.`**`name`**`(`*obj*`)` → string
Returns an object's name, usually the variable or field name.

Parameters **`obj`** (`cppobj`) – Object whose name to return

`cppobj.`**`long_name`**`(`*obj*`)` → string
Returns an object's "long" name, which is a fully qualified expression that is valid for the current scope.

Parameters **`obj`** (`cppobj`) – Object whose long name to return

`cppobj.`**`offset`**`(`*obj*`)` → integer
Returns an object's memory offset in the debugging target's memory.

Parameters **obj** (cppobj) – Object whose offset to return

cppobj.**size**(*obj*) → integer
Returns an object's size.

Parameters **obj** (cppobj) – Object whose size to return

cppobj.**type_id**(*obj*) → integer
Returns an object's type id. This id is used in various dbgeng COM API functions.

Parameters **obj** (cppobj) – Object whose type id to return

cppobj.**type_module**(*obj*) → integer
Returns the base memory offset of the module that the object's type is from. This is effectively an id for the module, and is used in various dbgeng COM API functions.

Parameters **obj** (cppobj) – Object whose module to return

cppobj.**address_of**(*obj*) → cppobj
Returns a new cppobj representing a pointer to *obj*.

Parameters **obj** (cppobj) – Object to create a pointer to

cppobj.**dereference**(*obj*) → cppobj
Given a cppobj representing a pointer, returns a new cppobj representing the pointee.

Parameters **obj** (cppobj) – Object to dereference

cppobj.**bases**(*obj*) → array[cppobj]
Returns an array of cppobj instances that are the base class representations of *obj*.

Parameters **obj** (cppobj) – Object whose base class representations to return

cppobj.**members**(*obj*) → array[cppobj]
Returns an array of cppobj instances that are the data members of *obj*.

Parameters **obj** (cppobj) – Object whose data members to return

Overloaded Operators

Overloaded operators comprise the remainder of the interactions with and between cppobj instances.

cppobj.**__index**(*name*) → cppobj
The indexing operator (`.` or `[]`) is overloaded to return a new cppobj representing a data member of the cppobj being indexed.

Assuming *foo* is a formerly created cppobj, the following retrieves its *bar* data member as a cppobj:

```
local bar = foo.bar
-- equivalent:
local bar = foo[ 'bar' ]
```

cppobj.**__eq**(*cppobj*) → boolean
The equality operator (==) is overloaded to test C++ objects for equality, according to the following rules:

- arrays are treated as pointers
- pointers, base, and enum types are compared as numbers
- class types are considered equal if two instances represent the same type at the same memory offset

If the first operand is a cppobj, any other equality result will be false.

cppobj.**__sub**(*cppobj*) → number

cppobj.**__sub**(*number*) → cppobj or number

The subtraction operator (−) is overloaded to perform subtraction with C++ objects, according to the following rules:

- if the 1st operand is a cppobj pointer or array, and the 2nd operand is:
  - cppobj pointer or array of the same type
    - result is the scaled distance between the two memory locations, scaled by the object size
  - cppobj base integral type
    - result is a new pointer, offset by the given scaled distance
  - Lua integer
    - result is a new pointer, offset by the given scaled distance
- if the 1st operand is a cppobj base type or enum, and the 2nd operand is:
  - cppobj base type or enum or Lua number; result is the difference of the two operands
- if the 1st operand is a Lua value, and the 2nd operand is:
  - cppobj base type or enum or Lua number; result is the difference of the two operands
- otherwise, the result is `nil`

2.2 dbgeng

The dbgeng COM API is divided into several categories of COM objects, each relating to a specific set of functionality. The provided Lua functions are intended to be a thin facade over the actual COM functions, and so are patterned very closely after them.

2.2.1 dbgeng.control

The control functions are all accessible from the global *dbgeng.control* table, and represent the functions from the IDebugControl interfaces of the dbgeng COM API.

Functions

dbgeng.control.**add_assembly_options**(*options*) → boolean

Turns on some of the assembly and disassembly options. The given options will be added to the existing options. See AddAssemblyOptions.

Parameters **options** (*bitfield*) – Combination of values from the *asmopt* table

Returns True if successful, otherwise false

Other

dbgeng.control.**asmopt**

Table containing assembly and disassembly options that affect how the debugger engine assembles and disassembles processor instructions for the target. Contains the following values, which are intended to be combined in a bitfield:

- `VERBOSE`: When set, additional information is included in the disassembly

- `NO_CODE_BYTES`: When set, the raw bytes for an instruction are not included in the disassembly
- `IGNORE_OUTPUT_WIDTH`: When set, the debugger ignores the width of the output display when formatting instructions during dissamebly
- `SOURCE_LINE_NUMBER`: When set, each line of dissembly output is prefixed with the line number of the source code provided by symbol information

See DEBUG_ASMOPT_XXX.

2.2.2 dbgeng.symbols

The symbols functions are accessible from the global *dbgeng.symbols* table, and represent the functions from the IDebugSymbols interfaces of the dbgeng COM API.

Functions

Other

2.2.3 dbgeng.symbol-group

The symbol group functions are accessible from the global *dbgeng.symbol_group* table, and represent the functions from the IDebugSymbolGroup interfaces of the dbgeng COM API.

Functions

Other

2.2.4 dbgeng.breakpoint

The breakpoint functions are accessible from the global *dbgeng.breakpoint* table, and represent the functions from the IDebugBreakpoint interfaces of the dbgeng COM API.

Functions

Other

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

c

d

Symbols

A

B

C

D

I

K

L

M

N

O

P

S

T

V