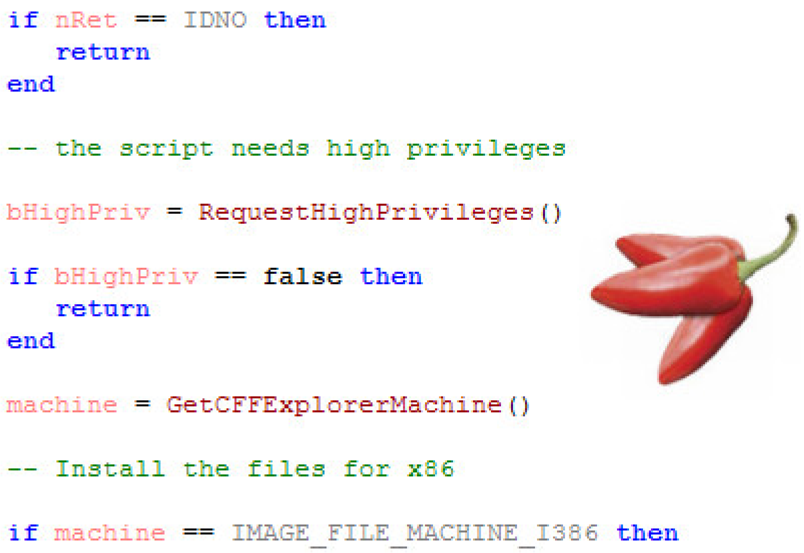
CFF Explorer's Scripting Language V1



```
if nRet == IDNO then
    return
end

-- the script needs high privileges

bHighPriv = RequestHighPrivileges()

if bHighPriv == false then
    return
end

machine = GetCFFExplorerMachine()

-- Install the files for x86

if machine == IMAGE_FILE_MACHINE_I386 then
```

Introduction

The first version of this scripting language was introduced in the CFF Explorer VII. The initial idea was a very simple one: offering a basic command line support to modify resources in a Portable Executable, just like Resource Hacker. The command line / script support was requested by a community of people who needed it for their software to change some Windows resources. After bouncing a little bit the idea in my head it evolved into a bigger project: providing a scripting language to do most PE tasks.

The first two possibilities I came up with were Python and Lua. Initially, I thought Python would've been a little too much for a PE editor; it seemed to me that it would've been like opening a door with a panzer. However, I might have been wrong. In fact, a lot of tools like IDA are opening up to Python and my choice might have been a misjudgement. Anyway, the Lua language is not very different in its syntax from Python. So, at least, it'll be easy for Python programmers to use it.

For this is the first version of this scripting language, I have no idea if people will like it. Even if it'll turn to be useless, I think it's an interesting feature, and, even though time consuming and sometimes not so easy, it was fun adding it to the CFF Explorer.

The Lua Language: Brief Guide

Lua is a very easy language to learn. This paragraph is absolutely not an extensive guide about Lua, rather a brief description of its main features. If you want to learn more about the language, you can visit Lua's documentation page. Keep in mind that there are a few differences between the standard Lua and the one I implemented in the CFF Explorer, so check out the paragraph about the differences.

I think the reader will eventually be surprised by Lua, which, despite of being a small language, is very powerful. It takes quite a bit of time to discover all its features and possibilities. This paragraph should give developers a basic idea of how to write simple scripts in Lua, but for more advanced topics consult the online documentation or one of the several books (yes there are quite a few) written about the Lua language.

First of all, this language is case sensitive. Thus, you can't call the function "msgbox", when, in fact, it ought to be written "MsgBox".

The version of Lua implemented in the CFF Explorer supports 64bit numbers. This factor was very important regarding the language choice. A language without 64bit number support would've been unfit for the tasks it's supposed to accomplish.

Here's a little 64bit-numbers test script:

```lua
var = 0xFFFFFFFFFFFFFFFF - 0xFFFFFFFF00000000

if var == 0x00000000FFFFFFFF then
    MsgBox("OK 1")
end

var = 0xF000000000000000
var2 = 0xF100000000000000

if var2 > var then
    MsgBox("OK 2")
end
```

As you can see, the hex numbers syntax is exactly the same as in C-like languages. From the script above you can also notice that Lua's variables are not declared with a fixed type, but just like in Python can assume any possible type:

```lua
var = "string"
var = 4
var = true
```

This is called dynamic typing. One thing I like about Lua is that it doesn't rely on indentation like Python: all code blocks are terminated by the word "end".

Lua supports arrays (or tables as they call them). A table can contain any type of data:

```lua
var = { "hello", 3, true }
```

Tables are made of the data itself and the key to access this data. In the example above no key is specified. Thus, a default keys is assigned to all members: a number. To access one of the members of the table above just write var[n]. In Lua tables are 1-based, although in the CFF Explorer implementation of Lua this isn't the case. This is one of the things listed in the next paragraph explaining the differences about standard Lua and the one implemented in the CFF Explorer.

Keys can be either number or strings:

```lua
var = { ["param1"] = true, ["param2"] = false, ["param3"] = true, ["param4"] =
true }
```

To change the first item of this table I write:

```
var["param1"] = false
```

However, I don't think that keys are very important for the CFF Explorer scripting, since most of the tasks I can think of involve byte arrays. Another little thing about tables: to get the size of a table just put the # (size) operator before the table's name:

```
MsgBox(#var)
```

This will print the number of rows in the table var.

To comment something in Lua use this syntax:

```
-- This is a single-line comment
```

Lua also offers multi-line comments:

```
--[[
    This is
    a multi-line
    comment
]]
```

Concetenating strings is very easy:

```
MsgBox("This is " .. "a string!")
```

You can also concatenate a string to a number in the same way:

```
MsgBox("The number of items is: " .. n)
```

Here is a list of the Lua language key words:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

And here is a code sample that shows some common code blocks:

```
var = 1000
var2 = true

-- if

if var == 1000 then
   -- code
end

-- if / elseif / else

if var == 1000 then
   -- code
elseif var == 1001 or var == 1002
   -- code
elseif var == 1004 and var2 == true
   -- code
```

```
else
   -- code
end

-- while

local n = 0

while n < var do
   -- code
   n = n + 1
end

-- repeat / until

n = 0

repeat
   if var2 == false then
      break
   end

   n = n + 1
until n < var

-- for (exp3 default)

for n = 0, var do
   -- code
end

-- for (exp3 explicit)

for n = 0, var, 2 do
   -- code
end

-- for (array iteration)

gen = { 10, 20, 30, 40 }

for i = 0, #gen - 1 do

   MsgBox("Key: " .. i .. " Value: " .. gen[i])

end
```

I think the only block which needs a bit of explanation is the for loop. The syntax of the for statement is: for exp1, exp2, exp3 do. Every value of exp1 must reach exp2 using exp3 as the step to increment exp1. exp3 is optional, if no step is specified, the default step value is 1.

Also, in Lua it doesn't make any difference if you write:

```
for n = 0, #gen - 1 do

   MsgBox("Key: " .. n .. " Value: " .. gen[n])

end
```

Or:

```
for n = 0, #gen - 1 do MsgBox("Key: " .. n .. " Value: " .. gen[n]) end
```

It's just a coding style issue.

In the above example you've seen making use of the key word "local". You can use "local" to declare variables which are going to be valid only the scope of the current function / block.

```
do

   local var = 0

   -- code

end
```

The variable var does not exist outside this do-block. By the way, do-blocks can be used as in the example above to have a better overview of local variables.

Let's see how to use a function:

```
function add1(num)
   return num + 1
end

function add2(num1, num2)
   return num1 + num2, num2 + 1
end

local var = 1

-- prints "1"
MsgBox(var)

local var2 = add1(var)

-- prints "1 and 2"
MsgBox(var .. " and " .. var2)

var, var2 = add2(var, var2)

-- prints "3"
MsgBox(var)
```

As I said before, you will be impressed by the power of Lua. As shown in the example above, you can return more than just one value in a function's return. And not only that: Lua supports a variable number of arguments in its functions. Here's an example:

```
function multibox(...)
   for i = 1, #arg - 1 do
      MsgBox(tostring(arg[i]))
   end
end


multibox("ciao", "ecco")
```

After reading this short paragraph, I believe, every developer has entered the Lua logic and is able to write some simple scripts.

CFF Explorer's Modified Lua

As mentioned in the previous paragraph, the Lua implemented in the CFF Explorer doesn't respect the standard implementation. I modified / added quite a few things. Most of the changes were, in my opinion, necessary in order to make the language easier for C/C++ developers and for the task it is supposed to fulfil. I'm going to summarize the main differences between the standard and this implementation of Lua in very brief chapters.

Bit Operators

CFF Explorer's Lua has all C/C++ bit operators:

- & (and)
- | (or)
- ^ (xor)
- << (shift left)
- >> (shift right)
- ~ (not)

Notice that the ^ operator in standard Lua stands for power and is followed by the exponent. Since I introduced C/C++ bit operators, I thought it was more important to have a normal xor, than to preverve Lua's standard power operator. If you want to use the power operator, write ^^ instead of ^.

Other Operators

A very small but useful addition was the operator !=. Most developers are used to it and, so, one can now write:

```
if var != 10 then
```

0-Based Arrays (Or Tables)

Standard Lua uses 1-based arrays (or tables, as they are called in Lua). Meaning that if you have an array (with no specfied keys) like:

```
array = { "hi", "how", "are", "you?" }
```

Its first element is array[1].

I changed this behavior. In fact, being a C++ developer, I think arrays ought to be 0-based. So, in the CFF Explorer Lua implementation the first member of the array is array[0]. Here's an array iteration already shown in the Lua Guide paragraph:

```
-- for (array iteration)

t = { 10, 20, 30, 40 }

for n = 0, #t - 1 do

   MsgBox("Key: " .. n .. " Value: " .. t[n])

end
```

0-based tables are one of the reasons (not the only one) I didn't enable most of the standard Lua functions (although some basic functions are available), since most functions exported by these libraries don't work correctly with 0-based tables.

I think 1-based tables are Lua's biggest flaw. A lot of people are complaining about them, but changing the inner behavior of Lua to let it treat 0 just as any other index is not so easy. This was really the most difficult choice I had to face when implementing Lua: leave 1-based tables or convert them? Making them 0-based brought a performance and a compatiblity loss that I accepted (not without doubts, though), since most developers would've gone made the other way (having also in mind the kind of developers who would use this scripting language). Again, I think I made the right call.

C# Strings

In addition to regular strings like:

```
"C:\\path\\...\\myfile"
```

I introduced C# strings:

```
@"C:\path\...\myfile"
```

I think this is quite useful and comes handy if one wants to execute a script function from command-line.

Imagine this scenario: I want to execute from command-line a function, say UpdatedChecksum, on a file. If I have the file name in a string, then, in order to pass it to the script function, I would have to duplicate all back-slashes in the string. This, of course, isn't necessary if I use a C# string:

```
wsprintf(CmdLine, _T("-CFFSCRIPT=UpdateChecksum(@\"%s\")"), FileName);
```

Definitions

- All Portable Executable related definitions in WinNT.h are provided.
- Most definitions of CorHdr.h are provided.
- Some Win32 definitions connected with the Win32 APIs wrapped by the scripting language are provided.

For instance, one could write:

```
MsgBox("Hello world!", "Caption", MB_ICONINFORMATION | MB_YESNO)
```

or:

```
if var == IMAGE_DOS_SIGNATURE then
```

The available definitions are thousands.

One other thing you'll frequently see in CFF Explorer scripts is the use of the type "null" instead of Lua's standard "nil". Lua's "nil" is still available, I added "null" because "nil" reminded me too much of Pascal/Delphi. It's simply a style choice.

How It Works

There are several ways to execute a CFF Explorer script:

- Double-click on it.
- Drag it with the mouse on the GUI.
- Load it from Open File or from the dedicated menu.
- Specify the script path command-line argument. E.g.: CFF Explorer.exe myscript.cff.
- Execute script functions directly through command-line. E.g.: CFF Explorer.exe -CFFSCRIPT=RealignPE("hello.exe", 0x200)

CFF Explorer's scripts have the extension "*.cff".

You can write a script in any of these encodings: ascii, unicode, UTF-8. The CFF Explorer recognizes the encoding with the same procedure used by the Windows Notepad. Unicode and UTF-8 files have a two-byte signature at the beginning (you can't see it directly in the editor). Once the CFF Explorer has determined the encoding, it'll handle it the right way. Lua doesn't natively support unicode strings. Thus, you have to understand that at runtime, the script will be working with UTF-8 strings. To sum up, the point is that you can use chinese or russian characters, as long as you save the script as unicode or UTF-8 text file (just like the notepad does).

To get the current script version from within a script you can rely on the definition "SCRIPT_VERSION":

```
-- shows the current script version
MsgBox(SCRIPT_VERSION)
```

All handles opened by a script are automatically closed after its execution. Nonetheless, closing handles during the execution is just a better coding style and is very advisable if the script opens many handles (which might be memory consuming). To close a handle (of any kind) use CloseHandle.

Functions can have a variable number of arguments. Moreover, some arguments can be of multiple type. When a function is declared like this:

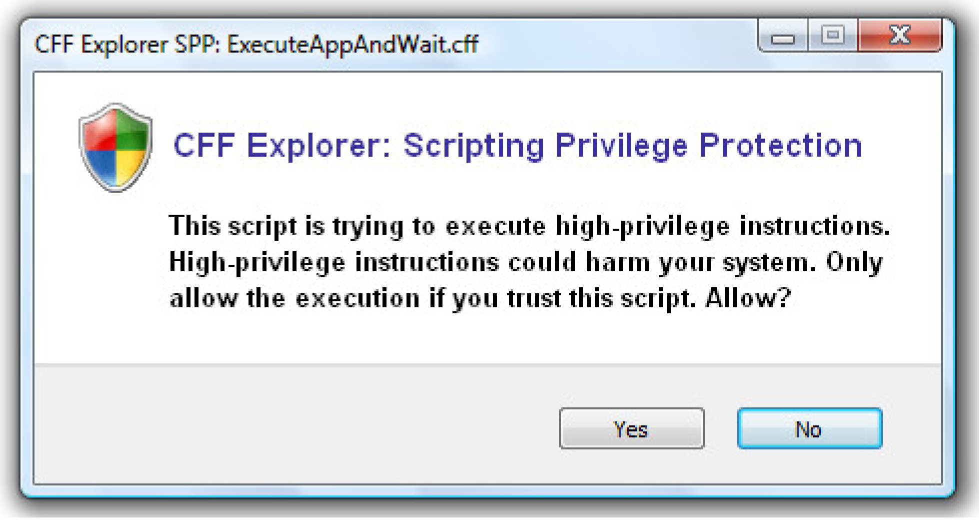
```
IsPE64(FileName/Handle)
```

It means that you can either pass a string (containing the file name) or a handle (of a previously opened file). If the functions modifies the file and you called the function through file name, then it'll apply the changes directly on the file. Viceversa, if you called the function through file handle, changes will be applied in memory only and to save them to disk, you'll have to call either SaveFile or SaveFileAs. Using handles is advised when doing more than just one operation on a file or when you want to save the file with a different name.

Security Issues

The CFF Explorer scripting language offers many APIs to easily manipulate Portable Executables. In addition to that, it offers functions to walk through the file system, functions to execute external application, to delete files etc. Thus, running a script could potentially be dangerous just like running an executable and could, in theory, behave just like a virus. To avoid problems for the user, I implemented some sort of UAC (don't worry it's not annoying like the real one), which I called Scripting Privilege Protection (SPP). Certain functions in order to be performed ask the user for consent. Once the user has given his consent, the script will be high-privileged and can execute any function. If the users doesn't give his consent, the function fails. A script might check if it's high-privileged through the function HasHighPrivileges. Moreover, a script can explicitly ask the user to give high privileges to it through the function RequestHighPrivileges.

Now, let us look at a real world example. Let's assume a script just called the function MoveFile. This is the dialog which the Scripting Privilege Protection will prompt to the user:

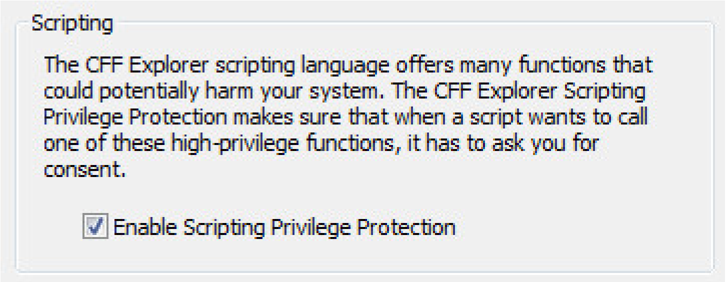

CFF Explorer SPP: ExecuteAppAndWait.cff

CFF Explorer: Scripting Privilege Protection

This script is trying to execute high-privilege instructions. High-privilege instructions could harm your system. Only allow the execution if you trust this script. Allow?

You can know if a function needs high privileges or not by looking it up in the [Functions Reference](#).

If you think that the Scripting Privilege Protection is just annoying, you can disable it from the CFF Explorer preferences; just uncheck this box:


Scripting

The CFF Explorer scripting language offers many functions that could potentially harm your system. The CFF Explorer Scripting Privilege Protection makes sure that when a script wants to call one of these high-privilege functions, it has to ask you for consent.

☑ Enable Scripting Privilege Protection

Code Samples

This paragraph goes down to the practice. What follows are just a few examples of what the CFF Explorer scripting language is able to do. Listing every possible application would be impossible for me, there are just too many.

PE Resources

I start with this paragraph, because if people will use the CFF Explorer scripting feature, most of them will do it to edit PE resources, I guess. Scripting functions to handle resources have a few advantages over Win32 APIs. In fact, Windows APIs do not handle special resources in a specif way. Adding a bitmap (or another special resource) to the resources rawly won't work: special resources are to be handled in a specific way before storing them in the Resource Directory of a PE. And that's what the CFF Explorer scripting APIs do. Also, Windows APIs for resource handling require the language coordinates for the resource. If the language is not the same as the one of the resource in the file, Windows APIs fails. These APIs, viceversa, can (as this is optional) be used without language coordinates. This comes handy if, for instance, one needs to delete a resource in a PE, but doens't know the resource's language. In that case the DeleteResource API deletes the first resource with the given name / id.

In the code example which follows a manifest resource is transferred from one file to another.

```
-- get the manifest resource of a file
-- and the puts it into another one

filename = GetOpenFile("Get Manifest",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

if filename == null then
   return
end

if SaveResource(filename, "res.manifest", RT_MANIFEST, 1) == false then
   DeleteFile("res.manifest")
   return
end

filename = GetOpenFile("Set Manifest",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

if filename == null then
   DeleteFile("res.manifest")
   return
end

AddResource(filename, "res.manifest", RT_MANIFEST, 1)

DeleteFile("res.manifest")
```

Easy PE Editing

The CFF Scripting language can be used to accomplish easy PE editing task such as the one which follows. This very little script patches x86 executables in order to let them have 4GB (instead of only 2) of virtual memory on x64 platforms by setting a flag in the File Header structure.

```
-- this script patches a PE32 in order let it handle 4GB
-- address space on x64 systems

filename = GetOpenFile("Open...",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

if filename == null then
   return
end

pehandle = OpenFile(filename)

if pehandle == null then
   return
end

fileheader = GetOffset(pehandle, PE_FileHeader)

-- check if it's a valid PE
if fileheader == null then
   -- CloseHandle is really not necessary,
   -- since the file will be automatically closed
   return
end

-- check if it's a PE64
-- if so, there's no need to patch
```

```
if IsPE64(pehandle) == true then
   MsgBox("This is a PE64. No need to patch.", "4GB Patch", MB_ICONEXCLAMATION)
   return
end

-- set large address space awareness flag

Characteristics = ReadWord(pehandle, fileheader + 18)

Characteristics = Characteristics | IMAGE_FILE_LARGE_ADDRESS_AWARE

WriteWord(pehandle, fileheader + 18, Characteristics)

-- save the file

nRet = MsgBox("Overwrite the file?", "4GB Patch", MB_ICONQUESTION | MB_YESNO)

if nRet == IDYES then
   if SaveFile(pehandle) == true then
      MsgBox("File successfully saved.", "4GB Patch", MB_ICONINFORMATION)
   else
      MsgBox("Couldn't save file", "4GB Patch", MB_ICONEXCLAMATION)
   end
else
   filename = GetSaveFile("Save As...",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

   if filename == null then
      MsgBox("Couldn't save file", "4GB Patch", MB_ICONEXCLAMATION)
   else
      if SaveFileAs(pehandle, filename) == true then
         MsgBox("File successfully saved.", "4GB Patch", MB_ICONINFORMATION)
      else
         MsgBox("Couldn't save file", "4GB Patch", MB_ICONEXCLAMATION)
      end
   end
end
```

I guess this needs no further explanation.

Advanced PE Editing

Even advanced PE editing can be easily accomplished using the CFF Explorer scripting language. In the next sample, we'll see how to add code to the last section of a PE, redirect the entry point to our code, jump from our code to the original entry point and, finally, update the PE's checksum value and remove the strong name signature in case it's a .NET assembly. I don't want to encourage someone to infect a PE through scripting. I just want to show how easily a more advaced task like this is performed with very little of code. Also, the logging functions will be used in this script, so we will be informed about every step of our code.

```
-- this script adds code to the last section of a PE,
-- changes the entry point to the added code
-- and rebuilds the PE

-- this functions checks if a flag is set

function IsFlag(value, flag)
   if (value & flag) == flag then
      return true
   end
   return false
end
```

```
-- prints string to the current log and goes to new line
function AddToLog(str)
   -- we can do this because hLog is a global variable
   if hLog then
      LogPrint(hLog, str .. "\n")
   end
end

-- show log

function ShowLog()

   -- Open log?

   if hLog != null then

      CloseLog(hLog)

      nRet = MsgBox("Open log file?", "Advanced PE Editing", MB_ICONQUESTION |
MB_YESNO)

      if nRet == IDYES then
         ExecuteAppAndWait(@"C:\Windows\System32\notepad.exe",
GetShortPathName(logname))
      end

      DeleteFile(logname)
   end
end


-- --------------------------------------------------
-- the main code starts here
-- --------------------------------------------------

filename = GetOpenFile("Open...",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

if filename == null then
   return
end

pehandle = OpenFile(filename)

if pehandle == null then
   return
end

optheader = GetOffset(pehandle, PE_OptionalHeader)

-- check if it's a valid PE
if optheader == null then
   -- CloseHandle is really not necessary,
   -- since the file will be automatically closed
   return
end

-- open log

logname = GetCurrentDirectory() .. @"\peadv.txt"


hLog = CreateLog(logname)
```

```
-- read original entry point

OEP = ReadDword(pehandle, optheader + 0x10)

AddToLog("Original Entry Point: " .. string.format("%08X", OEP))

-- get number of sections

nSections = GetNumberOfSections(pehandle)

if nSections == null or nSection == 0 then
   ShowLog()
   return
end

AddToLog("Number of sections: " .. nSections)

-- get section headers

sectheaders = GetOffset(pehandle, PE_SectionHeaders)

-- get the offset of the last section

lastsection = ((nSections - 1) * IMAGE_SIZEOF_SECTION_HEADER) + sectheaders

-- read the last section's characteristics

Characteristics = ReadDword(pehandle, lastsection + 36)

AddToLog("Last section original characteristics: " .. string.format("%08X",
Characteristics))

-- check the last section characteristics
-- make sure the section is readable & executable

if IsFlag(Characteristics, IMAGE_SCN_MEM_READ) == false or
   IsFlag(Characteristics, IMAGE_SCN_MEM_EXECUTE) == false then
   Characteristics = Characteristics | IMAGE_SCN_MEM_READ
   Characteristics = Characteristics | IMAGE_SCN_MEM_EXECUTE
   WriteDword(pehandle, lastsection + 36, Characteristics)
   AddToLog("The last section is now readable and executable")
end

-- here is the new code to add
-- in our case just a jump (we'll add the address later on)

newepcode = { 0xE9, 0x00, 0x00, 0x00, 0x00 }

codeoffset = AddDataToLastSection(pehandle, newepcode)

if codeoffset == null then
   AddToLog("Error: Couldn't add code to the last section")
   return
end

-- get the new entry point RVA

NewEP = OffsetToRva(pehandle, codeoffset)

-- calculate the relative address to the OEP
-- (5 stands for the jump size)
```

```
reladdr = (OEP - NewEP) - 5

WriteDword(pehandle, codeoffset + 1, reladdr)

AddToLog("Code successfully added to the last section")

-- set new entry point

if NewEP != null then
   WriteDword(pehandle, optheader + 0x10, NewEP)
   AddToLog("New entry point: " .. string.format("%08X", NewEP))
else
   AddToLog("Couldn't set new entry point")
   ShowLog()
   return
end

-- removes the Strong Name Signature if the PE is a .NET assembly

if IsDotNET(pehandle) == true then
   if RemoveStrongNameSignature(pehandle) == true then
      AddToLog("Strong Name Signature removed from the assembly")
   end
end

-- updates the checksum in case the PE is a driver

if UpdateChecksum(pehandle) == true then
   AddToLog("PE checksum successfully updated")
end

-- save the file

nRet = MsgBox("Overwrite the file?", "Advanced PE Editing", MB_ICONQUESTION |
MB_YESNO)

if nRet == IDYES then
   if SaveFile(pehandle) == true then
      AddToLog("File successfully saved")
   else
      AddToLog("Couldn't save file")
   end
else
   filename = GetSaveFile("Save As...",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

   if filename == null then
      AddToLog("Couldn't save file")
   else
      if SaveFileAs(pehandle, filename) == true then
         AddToLog("File successfully saved")
      else
         AddToLog("Couldn't save file")
      end
   end
end

CloseHandle(pehandle)

-- Show log

ShowLog()
```

I don't want to imply that the CFF Explorer scripting is almighty, but, as you can see, what I did through this script would take a little bit more in C++. Also, what I did here might not be very useful, I just wanted to show the scripting's potential. I wrote this script in 20 minutes, not more.

Generic Patches

This is one of the most "on the fly" applications for the CFF Explorer scripting. The function SearchBytes searches for an array of bytes where wildcards are allowed. This is perfect for code patches.

```
-- generic patch

filename = GetOpenFile()

if filename == null then
   return
end

-- "ND" elements are wildcards

signature = {
   0x74, 0x07, 0x56, 0xE8, ND, ND, ND, ND, 0x59, 0x8B, 0xC6, 0x5E, 0xC2, 0x04,
0x00, 0x8B,
   0x44, 0x24, 0x04, 0x83, 0xC1, 0x09, 0x51, 0x83, 0xC0, 0x09, 0x50, 0xE8
}

offset = SearchBytes(filename, 0, signature)

if offset != null then
   InvertJump(filename, offset)
   MsgBox("File patched!", "Generic Patch", MB_ICONINFORMATION)
end
```

As you can see, we wrote a generic patch in about 10 lines of code. I think this is quite a comfort for most experts who need to write a patch on the fly.

We can even write a patch that patches all the occurrences of a given byte signature:

```
-- generic patch 2

filename = GetOpenFile()

if filename == null then
   return
end

-- "ND" elements are wildcards

signature = {
   0x74, 0x07, 0x56, 0xE8, ND, ND, ND, ND, 0x59, 0x8B, 0xC6, 0x5E, 0xC2, 0x04,
0x00, 0x8B,
   0x44, 0x24, 0x04, 0x83, 0xC1, 0x09, 0x51, 0x83, 0xC0, 0x09, 0x50, 0xE8
}

filehandle = OpenFile(filename)

if filehandle == null then
   return
end
```

```
offset = SearchBytes(filehandle, 0, signature)

nPatches = 0

while offset != null do
   InvertJump(filehandle, offset)
   nPatches = nPatches + 1
   offset = SearchBytes(filehandle, offset + 1, signature)
end

if SaveFile(filehandle) and nPatches > 0 then
   MsgBox("File patched!", "Generic Patch", MB_ICONINFORMATION)
end
```

To retrieve a Lua array from a given file use the CFF Explorer: open the Hex Editor and then click on Copy Lua Array. To patch you can either use the Data Access functios or the Assembler functions.

PE Reports

Reports of PE information can be easily generated through scripting. In the example which follows I'll show how to create a report of a PE Import Directory.

```
-- this script generates a report
-- of a PE' Import Directory

-- this functions checks if a flag is set
function IsFlag(value, flag)
   if (value & flag) == flag then
      return true
   end
   return false
end

-- prints string to the current report and goes to new line
function AddToReport(str)
   -- we can do this because hReport is a global variable
   LogPrint(hReport, str .. "\n")
end

-- --------------------------------------------------
-- the main code starts here
-- --------------------------------------------------

filename = GetOpenFile("Open...",  "All\n*.*\nexe\n*.exe\ndll\n*.dll\n")

if filename == null then
   return
end

pehandle = OpenFile(filename)

if pehandle == null then
   return
end

-- get Import Directory offset if any

itoffset = GetOffset(pehandle, PE_ImportDirectory)
```

```
-- check if it's a valid PE and has a IT
if itoffset == null then
   -- CloseHandle is really not necessary,
   -- since the file will be automatically closed
   return
end

-- Get report name and create the file

repname = GetSaveFile("Save Report As..",  "Text File\n*.txt\n")

if repname == null then
   return
end

hReport = CreateLog(repname)

if hReport == null then
   return
end

-- sets additional definition

ImportDescriptorSize = 20

-- walk through the Import Directory

nCurrentDescr = 0

FirstThunk = ReadDword(pehandle, itoffset + 16)

AddToReport("Import Directory report for \"" .. filename .. "\"")

while FirstThunk != 0 do

   CurImpDescrName = (nCurrentDescr * ImportDescriptorSize) + itoffset + 12

   ModNameOffset = RvaToOffset(pehandle, ReadDword(pehandle, CurImpDescrName))

   ModName =  ReadString(pehandle, ModNameOffset)

   -- add the mod name to the report

   AddToReport("\n" .. (nCurrentDescr + 1) .. " - " .. ModName .. "\n")

   -- add all the functions to the report

   AddToReport("Ordinal Name")
   AddToReport("------- -------------------------------------")

   OFTs = ReadDword(pehandle, itoffset + (nCurrentDescr * ImportDescriptorSize))

   -- use OFTs or FTs

   Thunks = 0

   if OFTs != 0 then
      Thunks = RvaToOffset(pehandle, OFTs)
   else
      Thunks = RvaToOffset(pehandle, FirstThunk)
   end

   -- list functions
```

```
bPE64 = IsPE64(pehandle)

CurThunkOffset = Thunks

CurThunk = 0

if bPE64 == true then
   CurThunk = ReadQword(pehandle, CurThunkOffset)
else
   CurThunk = ReadDword(pehandle, CurThunkOffset)
end


while CurThunk != null and CurThunk != 0 do

   -- check if it's ordinal only

   bOrdinal = false

   if bPE64 == true then
      bOrdinal = IsFlag(CurThunk, IMAGE_ORDINAL_FLAG64)
   else
      bOrdinal = IsFlag(CurThunk, IMAGE_ORDINAL_FLAG32)
   end

   if bOrdinal == true then

      local Ordinal = ReadWord(pehandle, CurThunkOffset)

      AddToReport("0x" .. string.format("%04X", Ordinal))

   else

      FuncOffset = RvaToOffset(pehandle, (CurThunk & 0xFFFFFFFF))

      local Ordinal = ReadWord(pehandle, FuncOffset)

      FuncName = ReadString(pehandle, FuncOffset + 2)

      AddToReport("0x" .. string.format("%04X", Ordinal) .. " " .. FuncName)
   end

   -- next thunk

   if bPE64 == true then
      CurThunkOffset = CurThunkOffset + 8
      CurThunk = ReadQword(pehandle, CurThunkOffset)
   else
      CurThunkOffset = CurThunkOffset + 4
      CurThunk = ReadDword(pehandle, CurThunkOffset)
   end

end


-- next import descriptor

nCurrentDescr = nCurrentDescr + 1

NextImportDescr = (itoffset + (nCurrentDescr * ImportDescriptorSize)) + 16

FirstThunk = ReadDword(pehandle, NextImportDescr)
```

```
end

-- Open the report?

CloseLog(hReport)

nRet = MsgBox("Open report file?", "IT Report", MB_ICONQUESTION | MB_YESNO)

if nRet == IDYES then
   ExecuteAppAndWait(@"C:\Windows\System32\notepad.exe",
GetShortPathName(repname))
end
```

This is the output of this script:

```
Import Directory report for "C:\...\CFF Explorer.exe"

1 - KERNEL32.dll

Ordinal Name
------- -------------------------------------
0x02D7 RtlUnwind
0x021A HeapReAlloc
0x02A7 RaiseException
0x00B9 ExitProcess
0x0066 CreateProcessA
etc.

2 - USER32.dll

Ordinal Name
------- -------------------------------------
0x0204 PostQuitMessage
0x028F ShowOwnedPopups
0x027F SetWindowContextHelpId
0x0037 CharUpperW
0x02B4 UnregisterClassW
0x002C CharNextW
etc.

3 - GDI32.dll

Ordinal Name
------- -------------------------------------
0x0160 GetClipBox
0x004B CreateRectRgn
0x0026 CopyMetaFileW
0x0207 SaveDC
0x0200 RestoreDC
0x0216 SetBkMode
0x022B SetMapMode
etc.

4 - comdlg32.dll

Ordinal Name
------- -------------------------------------
0x000C GetSaveFileNameW
0x000A GetOpenFileNameW
0x0008 GetFileTitleW
```

etc.

This is even a complicated report, but the code is quite easy as you can see. Of course, not all reports have to be this complicated.

Extension Setups

One last code sample is about extension setups. CFF Explorer extensions can now be installed in an elegant way through scripting. This is way better than manually placing the files in the CFF Explorer extensions directory. Not only that, but the scripting language is able to install the right files for the current platform by using the function GetCFFExplorerMachine, which returns the machine of the CFF Explorer executable.

Here's the setup of my Resource Tweaker extension for the CFF Explorer:

```
-- This is the setup for Resource Tweaker
-- © 2007 Daniel Pistelli. All rights reserved.

nRet = MsgBox("This is the setup for Resource Tweaker. Resource Tweaker is an
extension for the CFF Explorer which allows to edit resources of non-x86
executables with older applications (e.g. Resource Hacker). Click \"Yes\" to
continue the setup process.", "Resource Tweaker Setup", MB_ICONINFORMATION |
MB_YESNO)

if nRet == IDNO then
   return
end

-- the script needs high privileges to install the files

bHighPriv = RequestHighPrivileges()

if bHighPriv == false then
   return
end

SetupDir = GetCurrentDirectory() .. @"\Resource Tweaker\"
CFFDir = GetCFFExplorerDirectory()
ExtDir = CFFDir .. @"\Extensions\CFF Explorer\Resource Tweaker\"

-- Make sure we have our extension directories

CreateDirectory(CFFDir .. @"\Extensions")
CreateDirectory(CFFDir .. @"\Extensions\CFF Explorer")
CreateDirectory(ExtDir)


-- Now we need to know what kind of files have to be installed given a certain
platform

machine = GetCFFExplorerMachine()

-- Install the files for x86

if machine == IMAGE_FILE_MACHINE_I386 then

   CopyFile(SetupDir .. "Resource Tweaker.dll", ExtDir .. "Resource
Tweaker.dll", true)

-- Install the files for x64
```

```
elseif machine == IMAGE_FILE_MACHINE_AMD64 then

   CopyFile(SetupDir .. "Resource Tweaker_x64.dll", ExtDir .. "Resource
Tweaker_x64.dll", true)

-- Install the files for IA64

elseif machine == IMAGE_FILE_MACHINE_IA64 then

   CopyFile(SetupDir .. "Resource Tweaker_IA64.dll", ExtDir .. "Resource
Tweaker_IA64.dll", true)

end

-- Install all other files that are not platform dependent

CopyFile(SetupDir .. "ResHacker.exe", ExtDir .. "ResHacker.exe", true)

CreateDirectory(ExtDir .. "ResHacker_Docs")
CopyFile(SetupDir .. @"ResHacker_Docs\ReadMe.txt", ExtDir ..
@"ResHacker_Docs\ReadMe.txt", true)
CopyFile(SetupDir .. @"ResHacker_Docs\Version_History.txt",
   ExtDir .. @"ResHacker_Docs\Version_History.txt", true)

-- Inform the user that the setup is now complete

MsgBox("The Resource Tweaker extension for the CFF Explorer was successfully
installed.",
   "Resource Tweaker Setup", MB_ICONINFORMATION)
```

This is quite easy and doesn't need any coment.

Functions Reference

General

Security

HasHighPrivileges
RequestHighPrivileges

Misc

GetCFFExplorerDirectory
GetCFFExplorerMachine
GetCurrentDirectory
OpenWithCFFExplorer

Handles

CloseHandle

Files

CloseFile
GetFileSize
OpenFile
SaveFile
SaveFileAs

Logging

CreateLog
CloseLog
LogPrint

Assembler

Data Access

ReadByte
ReadWord
ReadDword
ReadQword
ReadBytes
ReadString
WriteByte
WriteWord
WriteDword
WriteQword
WriteBytes
WriteString

FillBytes
SearchBytes

	InvertJump MakeJumpUnconditional NopBytes	

Portable Executable

General GetOffset IsDotNET IsPE64 **Addresses** IsRvaValid OffsetToRva RvaToOffset SectionFromRva VaToOffset VaToRva	**Resources** AddResource AddResourceRaw DeleteResource ImportResourceDirectory MAKELANGID SaveResource SaveResourceRaw **Sections** AddDataToLastSection AddSection AddSectionHeader AddSectionWithData DeleteSection DeleteSectionHeader DumpSection GetNumberOfSections	**Rebuilding** AfterDumpHeaderFix BindImports RealignPE RebuildImageSize RebuildPEHeader RemoveDataDirectory RemoveDebugDirectory RemoveRelocSection RemoveStrongNameSignature SetImageBase UpdateChecksum

Win32

Common Dialogs GetDirectory GetOpenFile GetSaveFile InputBox MsgBox	**Searching** InitFindFile FindFile **Applications** ExecuteApp ExecuteAppAndWait	**Files & Directories** CopyFile DeleteFile MoveFile CreateDirectory DeleteDirectory GetLongPathName GetShortPathName

---

HasHighPrivileges

`HasHighPrivileges()`

Retruns true if the script has high privileges, otherwise false.

RequestHighPrivileges

`RequestHighPrivileges()`

Ask consent to the user for high privileges.

Retruns true if successful, otherwise false.

GetCFFExplorerDirectory

`GetCFFExplorerDirectory()`

Returns the directory of the CFF Explorer. E.g. "C:\...\Explorer Suite".

GetCFFExplorerMachine

`GetCFFExplorerMachine()`

Returns the value of FileHeader->Machine, which for the CFF Explorer can either be IMAGE_FILE_MACHINE_I386, IMAGE_FILE_MACHINE_AMD64 or IMAGE_FILE_MACHINE_IA64.

GetCurrentDirectory

`GetCurrentDirectory()`

Returns the current directory. E.g. "C:\...\Files".

If the script was processed through command line, the function returns null.

OpenWithCFFExplorer

`OpenWithCFFExplorer(FileName)`

Doesn't have a return value.

Remarks: this function needs high privileges to be performed.

CloseHandle

`CloseHandle(Handle)`

Closes a handle previously opened. Can be any kind of handle.

Doesn't have a return value.

CloseFile

`CloseFile(FileHandle)`

See CloseHandle.

Doesn't have a return value.

GetFileSize

`GetFileSize(FileName/Handle)`

Returns the size of a given file. If the function fails, it returns null.

OpenFile

`OpenFile(FileName)`

Returns a handle to the opened file. If the function fails, it returns null.

Remarks: the opened file is loaded into memory. This is not a standard OpenFile, but the function providing a handle for all other scripting functions to operate with.

SaveFile

```
SaveFile(FileHandle)
SaveFile(FileHandle, FileName)
```

Saves a file to disk. If only the the file handle is provided, it saves the file with the current name. Otherwise it acts like SaveFileAs.

Returns true if successful, otherwise false.

SaveFileAs

`SaveFileAs(FileHandle, FileName)`

Saves a file to disk with a given name.

Returns true if successful, otherwise false.

CreateLog

```
CreateLog(FileName)
CreateLog(FileName, bAppend)
```

Creates a new log file. If the log file already exists and the bAppend parameter is true, then all messages printed to the log will be added the existing ones.

Example:

```
hLog = CreateLog("log.txt")

if hLog == null then
    return
end

LogPrint(hLog, "hello\world!\nHow are you?")
LogPrint(hLog, "I'm fine, thanks!\n")

CloseLog(hLog)
```

Returns a handle to the created log. If the function fails, it returns null.

CloseLog

```
CloseLog(LogHandle)
```

See CloseHandle.

Doesn't have a return value.

LogPrint

```
LogPrint(LogHandle, String)
```

Prints a string to an already opened log file.

Returns true if successful, otherwise false.

InvertJump

```
InvertJump(FileName/Handle, Offset)
InvertJump(FileName/Handle, Offset, AssemblyType)
```

Inverts a conditional jump/branch. Possible values for AssemblyType are:

```
DISASM_X86_16
DISASM_X86
DISASM_X64
DISASM_NET
```

If the AssemblyType parameter is not specified, the function tries to automatically recognize the file type. The DISASM_NET value stands for .NET assembly.

Returns true if successful, otherwise false.

MakeJumpUnconditional

```
MakeJumpUnconditional(FileName/Handle, Offset)
MakeJumpUnconditional(FileName/Handle, Offset, AssemblyType)
```

Converts a conditional jump/branch to unconditional. Possible values for AssemblyType are:

```
DISASM_X86_16
DISASM_X86
DISASM_X64
DISASM_NET
```

If the AssemblyType parameter is not specified, the function tries to automatically recognize the file type. The DISASM_NET value stands for .NET assembly.

Returns true if successful, otherwise false.

Remarks: if the jump/branch is already conditional, the function fails.

NopBytes

```
NopBytes(FileName/Handle, Offset, Lenght)
NopBytes(FileName/Handle, Offset, Lenght, AssemblyType)
```

Nops a specified amount of bytes (Lenght) at a given offset. Possible values for AssemblyType are:

```
DISASM_X86_16
DISASM_X86
DISASM_X64
DISASM_NET
```

If the AssemblyType parameter is not specified, the function tries to automatically recognize the file type. The DISASM_NET value stands for .NET assembly.

Returns true if successful, otherwise false.

ReadByte

`ReadByte(FileName/Handle, Offset)`

Reads a byte from a given file at a given offset.

Returns the data if successful, otherwise null.

ReadWord

`ReadWord(FileName/Handle, Offset)`

Reads a word from a given file at a given offset.

Returns the data if successful, otherwise null.

ReadDword

`ReadDword(FileName/Handle, Offset)`

Reads a dword from a given file at a given offset.

Returns the data if successful, otherwise null.

ReadQword

`ReadQword(FileName/Handle, Offset)`

Reads a qword from a given file at a given offset.

Returns the data if successful, otherwise null.

ReadBytes

`ReadBytes(FileName/Handle, Offset, Lenght)`

Reads an array of bytes from a given file at a given offset.

Example:

```
data = ReadBytes(filehandle, 0, 50)
-- prints the lenght: 50
MsgBox(#data)
```

Returns the data if successful, otherwise null.

ReadString

```
ReadString(FileName/Handle, Offset)
ReadString(FileName/Handle, Offset, bUnicode)
ReadString(FileName/Handle, Offset, bUnicode, Lenght)
```

Reads a string from a given file at a given offset. bUnicode tells the function if the string to read is encoded as unicode. The default value for this parameter is false. If the parameter Lenght is not specified, the function looks for a null terminator to calculate the string's lenght.

Returns the string if successful, otherwise null.

WriteByte

```
WriteByte(FileName/Handle, Offset, Data)
```

Writes a byte to a given file at a given offset.

Returns true if successful, otherwise false.

WriteWord

```
WriteWord(FileName/Handle, Offset, Data)
```

Writes a word to a given file at a given offset.

Returns true if successful, otherwise false.

WriteDword

```
WriteDword(FileName/Handle, Offset, Data)
```

Writes a dword to a given file at a given offset.

Returns true if successful, otherwise false.

WriteQword

```
WriteQword(FileName/Handle, Offset, Data)
```

Writes a qword to a given file at a given offset.

Returns true if successful, otherwise false.

WriteBytes

```
WriteBytes(FileName/Handle, Offset, Data)
WriteBytes(FileName/Handle, Offset, Data, Lenght)
```

Writes an array of bytes to a given file at a given offset. If you don't specify the size, the function writes the whole array.

Example:

```
data = { 0x00, 0xFF, 0xFE }

WriteBytes(filehandle, 0, data)
```

Returns true if successful, otherwise false.

WriteString

```
WriteString(FileName/Handle, Offset, String)
WriteString(FileName/Handle, Offset, String, bUnicode)
WriteString(FileName/Handle, Offset, String, bUnicode, bTerminator)
```

Writes a string to a given file at a given offset. bUnicode tells the function if the string to read is encoded as unicode. The default value for this parameter is false. bTerminator specifies if the string's null terminator should be written or not. The defualt value for this parameter is true.

Returns true if successful, otherwise false.

FillBytes

```
FillBytes(FileName/Handle, Offset, Lenght, Byte)
```

Fills a specified amount of bytes in a file with a given byte-value. Basically the same as memset.

Returns true if successful, otherwise false.

SearchBytes

```
SearchBytes(FileName/Handle, Offset, Bytes)
SearchBytes(FileName/Handle, Offset, Bytes, Lenght)
```

Searches a byte array in a given file from a given offset. Wildcards in the byte array are possible through the use of the ND symbol.

Example:

```
-- all the ND elements in the array are wildcards

data = {
   0x54, 0x68, 0x69, 0x73, 0x20, ND, ND, 0x6F, 0x67, 0x72, 0x61, 0x6D, 0x20,
0x63, 0x61, 0x6E,
   0x6E, 0x6F, 0x74, 0x20, 0x62, 0x65, ND, 0x72, 0x75, 0x6E, 0x20, 0x69, 0x6E,
0x20, 0x44, 0x4F,
   0x53, 0x20, 0x6D, 0x6F, ND, 0x65
}

filename = GetOpenFile()

offset = SearchBytes(filename, 0, data)

while offset != null do
   -- valid offset
   MsgBox("Bytes found at offset: " .. offset)
   offset = SearchBytes(filename, offset + 1, data)
end
```

Returns the offset if successful, otherwise null.

GetOffset

```
GetOffset(FileName/Handle, PEField)
```

This function returns an offset of a known PE field. Available fields are:

```
PE_DosHeader
PE_NtHeaders
PE_FileHeader
PE_OptionalHeader
PE_DataDirectories
PE_SectionHeaders
PE_ExportDirectory
PE_ImportDirectory
PE_ResourceDirectory
PE_ExceptionDirectory
PE_SecurityDirectory
PE_RelocationDirectory
PE_DebugDirectory
PE_TLSDirectory
PE_ConfigurationDirectory
PE_BoundImportDirectory
PE_ImportAddressTableDirectory
PE_DelayImportDirectory
PE_DotNETDirectory
```

Returns the PE field offset. If the functions fails, it returns null.

IsDotNET

`IsDotNET(FileName/Handle)`

Returns true if it's a .NET assembly, otherwise false.

IsPE64

`IsPE64(FileName/Handle)`

Returns true if it's a 64bit PE, otherwise false.

IsRvaValid

`IsRvaValid(FileName/Handle, Rva)`

Checks if a relative virtual address is valid in the context of the specified file.

Returns true if valid, otherwise false.

OffsetToRva

`OffsetToRva(FileName/Handle, Offset)`

Converts a file offset to a relative virtual address.

Returns a relative virtual address. The function returns null if it can't convert to Rva.

RvaToOffset

`RvaToOffset(FileName/Handle, Rva)`

Converts a relative virtual address to a file offset.

Returns a file offset. If the Rva is invalid, the function returns null.

SectionFromRva

`SectionFromRva(FileName/Handle, Rva)`

Converts a relative virtual address to a PE section index.

Returns a section index. If unsuccessful, it returns null.

VaToOffset

`VaToOffset(FileName/Handle, Va)`

Converts a virtual address to a file offset.

Returns a file offset. If the virtual address is not valid, the function returns null.

VaToRva

`VaToRva(FileName/Handle, Va)`

Converts a virtual address to a relative virtual address.

Returns a relative virtual address if successful, otherwise null.

AddResource

```
AddResource(FileName/Handle, ResFileName/Handle, ResTypeNameOrId)
AddResource(FileName/Handle, ResFileName/Handle, ResTypeNameOrId, ResNameOrId)
AddResource(FileName/Handle, ResFileName/Handle, ResTypeNameOrId, ResNameOrId,
Language)
```

Adds a resource to a file. Some type of resources are treated in a specific way to be correctly stored into the PE file. If you don't want the resource to be treated in any way, use AddResourceRaw, which puts the raw data of the resource into the PE file regardeless of special resource types.

Example:

```
AddResource(filename, resname, RT_BITMAP, 1, MAKELANGID(LANG_NEUTRAL,
SUBLANG_DEFAULT))
```

Returns true if successful, otherwise false.

Remarks: if the given resource already exists in the PE, it will be replaced.

AddResourceRaw

```
AddResourceRaw(FileName/Handle, ResFileName/Handle, <<
ResTypeNameOrId)
AddResourceRaw(FileName/Handle, ResFileName/Handle, ResTypeNameOrId,
ResNameOrId)
AddResourceRaw(FileName/Handle, ResFileName/Handle, ResTypeNameOrId,
ResNameOrId, Language)
```

Adds the raw data of a resource into a PE file.

Returns true if successful, otherwise false.

Remarks: if the given resource already exists in the PE, it will be replaced.

DeleteResource

```
DeleteResource(FileName/Handle, ResTypeNameOrId, ResNameOrId)
DeleteResource(FileName/Handle, ResTypeNameOrId, ResNameOrId, Language)
```

Deletes a resource of a PE file. If the language is not specified, the function deletes the first resource with the given name / id.

Returns true if successful, otherwise false.

Remarks: the Windows API to do this task requests the language as parameter. If the language is not the same as the one of the resource in the file, the Windows API fails. Thus, this function can be very handy if one is unsure about the resource language.

ImportResourceDirectory

```
ImportResourceDirectory(FileName/Handle, File2Name/Handle)
ImportResourceDirectory(FileName/Handle, File2Name/Handle, bDeleteOld)
```

Imports the Resource Directory from one PE to another. If bDeleteOld parameter is true, the function deletes the old resources.

Returns true if successful, otherwise false.

MAKELANGID

```
MAKELANGID(a, b)
```

Used for resource functions. Creates a language id, just like in Win32.

Example:

```
LangId = MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT)
```

Returns the language ID.

SaveResource

```
SaveResource(FileName/Handle, ResFileName, ResTypeNameOrId, ResNameOrId)
SaveResource(FileName/Handle, ResFileName, ResTypeNameOrId, ResNameOrId, Language)
```

Saves a resource in a PE file to disk. Some type of resources are treated in a specific way to be correctly saved to disk. If you don't want the resource to be treated in any way, use SaveResourceRaw, which saves the raw data of the resource to disk regardeless of special resource types.

Example:

```
SaveResource(filehandle, "hello.ico", RT_ICON)
```

Returns true if successful, otherwise false.

Remarks: the Windows API to do this task requests the language as parameter. If the language is not the same as the one of the resource in the file, the Windows API fails. Thus, this function can be very handy if one is unsure about the resource language.

SaveResourceRaw

```
SaveResourceRaw(FileName/Handle, ResFileName, ResTypeNameOrId, ResNameOrId)
SaveResourceRaw(FileName/Handle, ResFileName, ResTypeNameOrId, ResNameOrId, Language)
```

Saves the raw data of a resource to disk regardeless of special resource types.

Returns true if successful, otherwise false.

Remarks: the Windows API to do this task requests the language as parameter. If the language is not the same as the one of the resource in the file, the Windows API fails. Thus, this function can be very handy if one is unsure about the resource language.

AddDataToLastSection

```
AddDataToLastSection(FileName/Handle, DataFileName/Handle/Bytes)
```

Adds data to the last section of a PE.

Returns the offset of the added data, otherwise null.

AddSection

```
AddSection(FileName/Handle, Size)
AddSection(FileName/Handle, Size, SectionName)
AddSection(FileName/Handle, Size, SectionName, Characteristics)
```

Adds a new section to a PE. The Characteristics are the same as in Win32.

Returns true if successful, otherwise false.

Remarks: the section name cannot contain more than 8 chars. All other characters will be discarded.

AddSectionHeader

```
AddSectionHeader(FileName/Handle)
```

Adds a new section header to a PE.

Returns true if successful, otherwise false.

AddSectionWithData

```
AddSectionWithData(FileName/Handle, SectFileName/Handle/Bytes)
AddSectionWithData(FileName/Handle, SectFileName/Handle/Bytes, SectionName)
AddSectionWithData(FileName/Handle, SectFileName/Handle/Bytes, SectionName,
Characteristics)
```

Adds a new section containing data to a PE. The Characteristics are the same as in Win32. The size of the section will be calculated based on the size of the data to add.

Example:

```
AddSectionWithData("my.exe", "datafile")

-- or..

data = { 0xFF, 0xFF, 0xFF }

AddSectionWithData("my.exe", data)
```

Returns true if successful, otherwise false.

Remarks: the section name cannot contain more than 8 chars. All other characters will be discarded.

DeleteSection

```
DeleteSection(FileName/Handle, SectionIndex)
```

Deletes a PE section. This functions deletes the section header and the section's physical space. Thus, it may reduce the file size. If you want to delete the section header only, use DeleteSectionHeader.

Returns true if successful, otherwise false.

DeleteSectionHeader

`DeleteSectionHeader(FileName/Handle, SectionIndex)`

Deletes a PE section header.

Returns true if successful, otherwise false.

DumpSection

`DumpSection(FileName/Handle, SectionIndex, NewFileName)`

Dumps the data of a PE section to disk.

Returns true if successful, otherwise false.

GetNumberOfSections

`GetNumberOfSections(FileName/Handle)`

Gets the number of sections of a PE.

Returns the number of sections. If the function fails, it returns null.

AfterDumpHeaderFix

`AfterDumpHeaderFix(FileName/Handle)`

Repairs a PE after it was dumped from memory.

Returns true if successful, otherwise false.

BindImports

`BindImports(FileName/Handle)`

Binds the Import Table of a PE.

Returns true if successful, otherwise false.

RealignPE

`RealignPE(FileName/Handle, Alignment)`

Realigns a PE to a given value.

Example:

`RealignPE("hello.exe", 0x200)`

Returns true if successful, otherwise false.

RebuildImageSize

`BindImports(FileName/Handle)`

Rebuilds the SizeOfImage field of a PE.

Returns true if successful, otherwise false.

RebuildPEHeader

`RebuildPEHeader(FileName/Handle)`

Rebuilds the header of a PE.

Returns true if successful, otherwise false.

RemoveDataDirectory

`RemoveDataDirectory(FileName/Handle, DataDirectoryEntry)`

Removes a Data Directory from a PE. To remove the Debug Directory, use RemoveDebugDirectory.

Example:

`RemoveDataDirectory(filehandle, IMAGE_DIRECTORY_ENTRY_TLS)`

Returns true if successful, otherwise false.

RemoveDebugDirectory

`RemoveDebugDirectory(FileName/Handle)`

Strips all the debug information contained in a PE.

Returns true if successful, otherwise false.

RemoveRelocSection

`RemoveRelocSection(FileName/Handle)`

Removes the reloc section of a PE.

Returns true if successful, otherwise false.

RemoveRelocSection

`RemoveRelocSection(FileName/Handle)`

Removes the reloc section of a PE.

Returns true if successful, otherwise false.

RemoveStrongNameSignature

`RemoveStrongNameSignature(FileName/Handle)`

Removes the Strong Name Signature, if present, of a .NET assembly.

Returns true if successful, otherwise false.

SetImageBase

`SetImageBase(FileName/Handle, NewImageBase)`

Sets the new ImageBase of a PE. If the PE doesn't have a Relocation Directory, the functions fails.

Returns true if successful, otherwise false.

UpdateChecksum

`UpdateChecksum(FileName/Handle)`

Updates the CheckSum field of a PE.

Returns true if successful, otherwise false.

GetDirectory

```
GetDirectory()
GetDirectory(Title)
GetDirectory(Title, CurrentPath)
GetDirectory(Title, CurrentPath, Flags)
```

Asks the user to choose a directory. Flags are the same as for Win32 Shell API.

Example:

```
DirName = GetDirectory("Title", CurrentPath, BIF_RETURNONLYFSDIRS)
```

Returns the name of the selected directory. If no directory was selected, the function returns null.

GetOpenFile

```
GetOpenFile()
GetOpenFile(Title)
GetOpenFile(Title, Filter)
GetOpenFile(Title, Filter, Flags)
```

Asks the user to choose a file to open. The flags are the same as for the Win32 GetOpenFileName. To separate items in the Filter, use \n istead of \0.

Example:

```
filename = GetOpenFile("Title", "All\n*.*\nexe\n*.exe\ndll\n*.dll\n",
OFN_FILEMUSTEXIST)
```

Returns the name of the selected file/s. If no file was selected, the function returns null.

GetSaveFile

```
GetSaveFile()
GetSaveFile(Title)
GetSaveFile(Title, Filter)
GetSaveFile(Title, Filter, Flags)
```

Asks the user to choose a file to open. The flags are the same as for the Win32 GetSaveFileName. To separate items in the Filter, use \n istead of \0.

Returns the name of the selected file/s. If no file was selected, the function returns null.

InputBox

```
InputBox()
InputBox(Title)
InputBox(Title, Caption)
```

Asks the user to type something. A similar dialog is very useful and is basically the same as in VBS.

Returns the input string. If there is no input string, returns null.

MsgBox

```
MsgBox(Title)
MsgBox(Title, Caption)
MsgBox(Title, Caption, Type)
```

Shows a message box. This function behaves exactly like the Win32 API.

Example:

```
MsgBox(4 + 3)
MsgBox("Hello world!", "Caption", MB_ICONINFORMATION | MB_YESNO)
```

Returns the clicked button of the message box.

InitFindFile

```
InitFindFile(FileName)
```

Initializes a search process. The handle returned by this function can be closed with CloseHandle. However, the handle is automatically closed when FindFile has iterated through all the files and returns null.

Example:

```
hSearchHandle = InitFindFile(path .. "\\*.*")

if hSearchHandle then

    FName = FindFile(hSearchHandle)

    while FName do
        MsgBox(FName)
        FName = FindFile(hSearchHandle)
    end

end
```

Returns a handle to use with FindFile. If the function fails, it returns null.

Remarks: this function needs high privileges to be performed.

FindFile

`FindFile(SearchHandle)`

Used to iterate a search process. For an example of how to search a file, see InitFindFile.

Returns the name of a file. If there are no more files, the function returns null.

ExecuteApp

```
ExecuteApp(FileName)
ExecuteApp(FileName, CmdLine)
```

Starts an external application and returns. If you want to wait for the started application to terminate, use ExecuteAppAndWait instead.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

ExecuteAppAndWait

```
ExecuteAppAndWait(FileName)
ExecuteAppAndWait(FileName, CmdLine)
```

Starts an external application and waits for the application to terminate before returning.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

CopyFile

```
CopyFile(ExistingFileName, NewFileName)
CopyFile(ExistingFileName, NewFileName, bReplaceExisting)
```

Copies a file.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

DeleteFile

`DeleteFile(FileName)`

Deletes a file.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

MoveFile

```
MoveFile(ExistingFileName, NewFileName)
MoveFile(ExistingFileName, NewFileName, bReplaceExisting)
```

Moves a file.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

CreateDirectory

`CreateDirectory(PathName)`

Creates a directory.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

DeleteDirectory

`DeleteDirectory(PathName)`

Deletes a directory.

Returns true if successful, otherwise false.

Remarks: this function needs high privileges to be performed.

GetLongPathName

`GetLongPathName(PathName)`

Returns the long path name if successful, otherwise null.

GetShortPathName

`GetShortPathName(PathName)`

Returns the short path name if successful, otherwise null.

Conclusions

As I already said in this article, I can't predict if this scripting language will be of any interest for the users, but it was fun implementing it and it really makes the CFF Explorer stand apart from other editors, in my opinion. And maybe someone will find it useful after all.

I would have liked to implement many more functions and additions to the Lua syntax, but fact is that I've worked on this feature long enough for its first version: I really felt it was release-ready.

Daniel Pistelli