© The CodeBreakers-Journal, Vol. 1, No. 1 (2004)
http://www.CodeBreakers-Journal.com

Adding functions to any program using a DLL

Dracon

Abstract

Some time ago I have worked on Douby's "ReverseMe 1" and added missing functions to the program. After doing this I looked for a way to make this process easier. It's very uncomfortable to code with HIEW, especially if you must change the finished code. Suddenly I got an idea: why not writing a dll? This allows you to use the language you are most familiar with (ASM, C, C++, Delphi, ...), only few asm-code is needed and it's not difficult.

Keywords: *Reverse Code Engineering; Adding Functionality; DLL*

I. Introduction

Target: douby's ReverseMe1
Essay Level: Intermediate
Tools used:

- HIEW
- IDA
- SICE
- a compiler that can create DLLs (VC++, MASM, Delphi, ...)

This is my second essay about ”adding functions”, I hope you enjoy it as much as the first one. Please visit my Homepage.

II. How to write a DLL

Writing a DLL is simple. First of all you will need a ”DEF” file which specifies the functions or variables to export. This is a simple textfile, here is an example:

```
LIBRARY tstdll
DESCRIPTION "Only a test-dll."
EXPORTS TestProc1
```

”LIBRARY” is the name of your dll, ”DESCRIPTION” the description (optional) and after ”EXPORTS” you can add all the functions you want to export.

Functions can be exported by name (this is the default) or by ordinal, a number greater than 0. Using this approach will make calling easier because you don't need the whole name but only a number. Another problem with names is that the compiler will modify them: TestProc1 with one DWORD as parameter would become ”`_TestProc1`” with C-linkage and ”`?TestProc1@@YAXPAUHWND__@@@Z`” if your C++ compiler mangles the name. This is not only long but also hard to remember, so better avoid it. You can also specify the NONAME parameter and no function name will appear in your dll. That's what I prefer so your EXPORTS-line should look like this:

```
EXPORTS
    TestProc1    @1    NONAME
    TestProc2    @2    NONAME
"@1" means that the function should be exported as ordinal 1.
```

Now you must tell the Linker that it should create a dll. It's easy if you use VC++: choose ”New...” and then ”Win32 Dynamic-Link Library” in the Project-Tab. Create a blank file, add the lines above to create a definition file and save it as ”`my_dll.def`” (”`my_dll`” is the name of your dll of course). For MASM you must use the following commands (put them in a batch-file):

```
\masm32\bin\ml /c /coff tstdll.asm
\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL /DEF:tstdll.def tstdll.obj
```

Important are the ”/DLL” switch and the name of your DEF-file.

DLLs don't have a ”WinMain” function but ”DllMain”. Read a good book (Petzold) or look at my code for more, it's not difficult. The main purpose is to allocate/free additional memory and/or do some initialization.

III. Calling Dll-functions

Now that you know how to create a Dll we want to call it from our target program. First you must load the library into memory (LoadLibraryA). The dll must be in the same directory as the other program or somewhere in the path, else Windows won't find it. The next step is to get the address of the function you wish to call (GetProcAddress) and finally you can call it.

It's possible to use different calling-conventions. `__pascal`, `__fortran`, and `__syscall` are no longer supported in VC++ 6, so the valid ones are now:

1) 1. STDCALL: used in almost all Win32 Api-function. The parameters are pushed from right to left (reverse order!) and the called function is responsible for adjusting the stack. C++ functions in VC++ additionally store the "this" pointer in ECX (THISCALL).

   Short intermezzo: The stack is the memory that is used for local variables and for function-parameters, the pointer is stored in ESP. Every "push" and "pop" will modify it and it's necessary to keep it consistent. The stack grows from the highest memory address to the lowest, so if you "add" something to it the pointer will actually set back and more stack-memory is available.
2) C: the parameters are pushed from right to left and the caller is responsible for adjusting the stack. This is the default for C-functions in VC++. The advantage is that you can use variable arguments e.g. sprintf, but it also results in slightly bigger code.
3) FASTCALL: First the registers ECX and EDX are used, then the stack (Watcom has used more registers...). The called function must adjust the stack.

Which one should we choose? I suggest to you use "STDCALL" (`__stdcall` in VC++) because you don't have to mess around with the stack. Define your function like this:

```
void __stdcall TestProc1(DWORD dwTest, double dTest);
```

If you use the default (`__cdecl`) you must adjust the stack. That's not difficult: sum up the size of every parameter and do a "add esp, (sum)" after the call. In the example above it's DWORD (4) + double (8) = 0Ch so you must write "add esp, 0Ch".

After all this technical information let's return to our code. The last step is to unload the dll. When a program exits all DLLs (if not needed by other programs) will be removed so it's not necessary to free it in the middle of your program, especially if you want to use "static" variables. On the other hand, don't leave marks everywhere you have been. :-) Use "FreeLibrary" to unmap your DLL from memory.

```
push offset "mydll.dll"     ; name of your dll
call LoadLibraryA
mov  esi, eax               ; store handle in esi
push 1                      ; ordinal 1: first function
push eax
call GetProcAddress
push parameter1             ; function expects 1 parameter
call eax
; add  esp, 4               ; adjust the stack if necessary
push esi
call FreeLibrary
...                         ; do some other stuff and return control
```

This is the basic code. Somehow we need the addresses of the 3 Api-functions. Use the program "OpGen" by NeuRaL NoiSE to get them, it can save a list with all imported functions. Sometimes "LoadLibraryA" or "FreeLibrary" aren't imported. In that case you must do the following:

```
push offset "kernel32.dll"     ; the function is in "kernel32.dll"
call GetModuleHandleA          ; Kernel32.dll is already loaded, we only need
                               ; the current handle
push offset "LoadLibraryA"
push eax                       ; this is the handle for kernel32.dll
call GetProcAddress
mov  ebx, eax                  ; store it somewhere
```

”GetModuleHandleA” and ”GetProcAddress” are always imported, I haven’t seen a program without them yet. Now you can use ”call ebx” to call ”LoadLibraryA”.

LoadLibraryA will load a DLL into memory and increase its reference counter. Another call wouldn’t load the dll a second time (and waste memory) but only increase the ref-counter. GetModuleHandleA will NOT load the DLL but retrieve the handle of an already loaded DLL, the ref-counter won’t be increased. Finally a call to FreeLibrary will decrease the ref-counter, if it has reached 0 the DLL will be unloaded. On program (process) exit Windows removes all libraries from memory unless they are still used by other programs.

IV. An Example: Adding CRC32-calculation to Douby’s ReverseMe 1

Theory is necessary but a practial example is better. I will use the ReverseMe to add a new function to the program: calculate the CRC32 of the current text. This will give you an idea how small changes will result in a very different number. The only parameter we need is the handle of the main window.

This essay doesn’t show you how to find the place to add code, please read my other big essay how to do it. You will find a jmp-table which takes care of the command-IDs. It’s interesting that one value is not used, this simplifies many things. Start ”BRW” or Symantec’s ”ResEdit” to add a new menu-item and give it the ID 40002. Good, my new code will start at physical offset 0x3B40 (virtual offset 0x403B40). What do we need?

```
Addresses of the Api-Functions:
LoadLibraryA (0x00404028)
GetProcAddress (0x0040402C)
GetModuleHandleA (0x00404044)
```

Hmm, ”FreeLibrary” is not imported. I have shown you above what to do in that case.

```
Offsets for some strings (use a Hex-Editor to enter them):
"patch_dll.dll" (virtual offset 0x403F00, phys. 0x3F00)
"kernel32.dll" (0x403F10)
"FreeLibrary" (0x403F20)
```

Here we go! First we must patch the jmp-table (0x1260), the 2nd entry (0x1264) should point to our new code (0x403B40) so start your Hex-Editor and change: 37 12 40 00 into 40 3B 40 00.

The following code must be entered in HIEW. My FINISH code starts at offset 0x3C00, but you can put it closer to the other code, this will make all jumps ”short”. If you enter jumps in HIEW (jmp, jne, ...) you must always use the physical offset. After pressing F9 (Update) you will see to which location the jump actually goes.

```
; new code which calls a dll-function
; starting at offset 0x3B40
;
; Load Dll into memory
push 403F00                        ; offset "patch_dll.dll"
call d, [00404028]                 ; LoadLibraryA
or   eax, eax                      ; eax != 0 (library loaded?)
jz   FINISH                        ; enter in HIEW: jz 3C00
mov  esi, eax                      ; store handle in esi

; get address of wished function
push 1                             ; ordinal 1 (1st exported function)
push eax                           ; handle to dll
call d, [0040402C]                 ; GetProcAddress
or   eax, eax                      ; function found?
jz   FINISH

; push parameters and call it
push [esp+10h]                     ; window-handle, check my other essay
call eax                           ; call the function

; add  esp, 4                      ; adjust the stack if necessary

; get address of FreeLibrary and call it
; no error-checking, should always work :-)
push 403F10                        ; offset "kernel32.dll"
call d, [00404044]                 ; GetModuleHandleA
push 403F20                        ; offset "FreeLibrary"
push eax                           ; handle kernel32.dll
call d, [0040402C]                 ; GetProcAddress
push esi                           ; handle to "patch_dll.dll"
call eax                           ; FreeLibrary
jmp  FINISH
; Cleanup-code
; Command ID was handled, so we can return
FINISH:
pop edi
pop esi
pop ebx
ret 10h
```

While coding the DLL I have found a bug (or feature) with the `WM_GETTEXT` msg: my buffer has a size of 0x10000 and I used this value to get the text of the edit-control. What happenend? Nothing! After some trying I found out that you can only use WORD-values:

```
GetWindowText(hWnd, pBuffer, 0xFFFF);   // this works
GetWindowText(hWnd, pBuffer, 0x10000);  // this doesn't
```

Download the source of the patch DLL used in this essay.

V. Conclusion and Greetings

Using a DLL is much more comfortable than modifying the target-program directly. You can (1) use any language, (2) don't need to play around with the PE-sections, (3) create big code and your own resources (dialogs, ...) and (4) reuse your functions. The most difficult thing is to find a good place from where you can call your functions. In C or ASM applications this is no problem, but MFC, Delphi, VB, etc. need a lot of reversing.

As an excercise you can add the missing functions (load, save, exit) to the ReverseMe1. Practice is necessary to get used to it, reading is not enough. I think with my first essay and this one you have the required knowledge, and you can always send me a mail if you have problems.

Big shouts go out to:

```
Knotty Dread (he is the man...)
NeuRaL NoiSE (he got skillz...)
Iczelion (check his {Win32ASM} site)
Razzia (he was the first who has added functions to a program)
sˆwitz (cool webmaster and Perl-wizard)
Douby (fellow from DREAD)
MACER and Potsmoke (funny guys from EFNET)
all DREAD members
```

Comments, critics, improvements, questions should go to:

```
andreas(dot)theDragon(at)gmx(dot)net
```