



Loading a DLL from memory

Shub-Nigurrath of ARTeam

Version 1.2 - September 2005

1.	Overview.....	2
2.	Windows executables - the PE format	2
2.1.	DOS header / stub	2
2.2.	PE header	3
2.3.	Section header	4
3.	Loading the library.....	5
3.1.	Allocate memory.....	5
3.2.	Copy sections	6
3.3.	Base relocation.....	6
3.4.	Resolve imports.....	7
3.5.	Protect memory.....	7
3.6.	Notify library.....	8
4.	Exported functions	8
5.	Freeing the library.....	9
6.	MemoryModule	10
6.1.	Downloads	10
6.2.	Known issues	10
6.3.	License	10
7.	Additional Examples.....	11
7.1.	An MFC Client and Dll.....	11
7.2.	MFC Client with the Dll included into its Resources	12
7.2.1	How the program looks from inside OllyDbg	15
8.	References.....	17
9.	Conclusions.....	17
10.	History.....	17
11.	Greetings	17

Keywords

Process, Dlls, Memory Loading

Please note that up to Section 7 of this document what reports is most NOT my original work, the Original Author is Joachim Bauch.

I'm just reporting his work here because it's interesting, the original link is dead now and it's required to understand the additional work I added starting from Section 7.



1. Overview

The default windows API functions to load external libraries into a program (*LoadLibrary*, *LoadLibraryEx*) only work with files on the filesystem. It's therefore impossible to load a DLL from memory. But sometimes, you need exactly this functionality (e.g. you don't want to distribute a lot of files or want to make disassembling harder). Common workarounds for these problems are to write the DLL into a temporary file first and import it from there. When the program terminates, the temporary file gets deleted.

This same approach is incredibly also common to most, even complex, protectors/packers, which before or later create temporary DLLs in the system. Before or later these weakness points are discovered and used to break protections. It's thus better to approach the problem differently.

In this tutorial, I will describe first, how DLL files are structured and will present some code that can be used to load a DLL completely from memory - without storing on the disk first.

2. Windows executables - the PE format

For a complete overview see [1].

Most windows binaries that can contain executable code (.exe, .dll, .sys) share a common file format that consists of the following parts:

DOS header
DOS stub
PE header
Section header
Section 1
Section 2
...
Section n

All structures given below can be found in the header file *winnt.h*.

2.1. DOS header / stub

The DOS header is only used for backwards compatibility. It precedes the DOS stub that normally just displays an error message about the program not being able to be run from DOS mode.

Microsoft defines the DOS header as follows:

```
typedef struct _IMAGE_DOS_HEADER {          // DOS .EXE header
    WORD   e_magic;                          // Magic number
    WORD   e_cblp;                           // Bytes on last page of file
    WORD   e_cp;                             // Pages in file
    WORD   e_crlc;                           // Relocations
    WORD   e_cparhdr;                        // Size of header in paragraphs
    WORD   e_minalloc;                       // Minimum extra paragraphs needed
```



```
WORD    e_maxalloc;           // Maximum extra paragraphs needed
WORD    e_ss;                 // Initial (relative) SS value
WORD    e_sp;                 // Initial SP value
WORD    e_csum;               // Checksum
WORD    e_ip;                 // Initial IP value
WORD    e_cs;                 // Initial (relative) CS value
WORD    e_lfarlc;             // File address of relocation table
WORD    e_ovno;               // Overlay number
WORD    e_res[4];             // Reserved words
WORD    e_oemid;              // OEM identifier (for e_oeminfo)
WORD    e_oeminfo;            // OEM information; e_oemid specific
WORD    e_res2[10];           // Reserved words
LONG    e_lfanew;             // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

2.2. PE header

The PE header contains informations about the different sections inside the executable that are used to store code and data or to define imports from other libraries or exports this libraries provides.

It's defined as follows:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

The *FileHeader* describes the *physical* format of the file, i.e. contents, informations about symbols, etc:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD    TimeDateStamp;
    DWORD    PointerToSymbolTable;
    DWORD    NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

The *OptionalHeader* contains informations about the *logical* format of the library, including required OS version, memory requirements and entry points:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD    SizeOfCode;
    DWORD    SizeOfInitializedData;
    DWORD    SizeOfUninitializedData;
    DWORD    AddressOfEntryPoint;
    DWORD    BaseOfCode;
    DWORD    BaseOfData;

    //
    // NT additional fields.
    //
```



```
DWORD ImageBase;
DWORD SectionAlignment;
DWORD FileAlignment;
WORD MajorOperatingSystemVersion;
WORD MinorOperatingSystemVersion;
WORD MajorImageVersion;
WORD MinorImageVersion;
WORD MajorSubsystemVersion;
WORD MinorSubsystemVersion;
DWORD Win32VersionValue;
DWORD SizeOfImage;
DWORD SizeOfHeaders;
DWORD CheckSum;
WORD Subsystem;
WORD DllCharacteristics;
DWORD SizeOfStackReserve;
DWORD SizeOfStackCommit;
DWORD SizeOfHeapReserve;
DWORD SizeOfHeapCommit;
DWORD LoaderFlags;
DWORD NumberOfRvaAndSizes;
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

The *DataDirectory* contains 16 (*IMAGE_NUMBEROF_DIRECTORY_ENTRIES*) entries defining the logical components of the library:

Index	Description
0	Exported functions
1	Imported functions
2	Resources
3	Exception informations
4	Security informations
5	Base relocation table
6	Debug informations
7	Architecture specific data
8	Global pointer
9	Thread local storage
10	Load configuration
11	Bound imports
12	Import address table
13	Delay load imports
14	COM runtime descriptor

For importing the DLL we only need the entries describing the imports and the base relocation table. In order to provide access to the exported functions, the exports entry is required.

2.3. Section header

The section header is stored after the *OptionalHeader* structure in the PE header. Microsoft provides the macro *IMAGE_FIRST_SECTION* to get the start address based on the PE header.

Actually, the section header is a list of informations about each section in the file:



```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[ IMAGE_SIZEOF_SHORT_NAME ];
    union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize;
    } Misc;
    DWORD    VirtualAddress;
    DWORD    SizeOfRawData;
    DWORD    PointerToRawData;
    DWORD    PointerToRelocations;
    DWORD    PointerToLinenumbers;
    WORD     NumberOfRelocations;
    WORD     NumberOfLinenumbers;
    DWORD    Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

A section can contain code, data, relocation information, resources, export or import definitions, etc.

3. Loading the library

To emulate the PE loader, we must first understand, which steps are necessary to load the file to memory and prepare the structures so they can be called from other programs.

When issuing the API call *LoadLibrary*, Windows basically performs these tasks:

1. Open the given file and check the DOS and PE headers.
2. Try to allocate a memory block of *PEHeader.OptionalHeader.SizeOfImage* bytes at position *PEHeader.OptionalHeader.ImageBase*.
3. Parse section headers and copy sections to their addresses. The destination address for each section, relative to the base of the allocated memory block, is stored in the *VirtualAddress* attribute of the *IMAGE_SECTION_HEADER* structure.
4. If the allocated memory block differs from *ImageBase*, various references in the code and/or data sections must be adjusted. This is called *Base relocation*.
5. The required imports for the library must be resolved by loading the corresponding libraries.
6. The memory regions of the different sections must be protected depending on the section's characteristics. Some sections are marked as *discardable* and therefore can be safely freed at this point. These sections normally contain temporary data that is only needed during the import, like the information for the base relocation.
7. Now the library is loaded completely. It must be notified about this by calling the entry point using the flag *DLL_PROCESS_ATTACH*.

In the following paragraphs, each step is described.

3.1. Allocate memory

All memory required for the library must be reserved / allocated using *VirtualAlloc*, as Windows provides functions to protect these memory blocks. This is required to restrict access to the memory, like blocking write access to the code or constant data.

The *OptionalHeader* structure defines the size of the required memory block for the library. It must be reserved at the address specified by *ImageBase* if possible:



```
memory = VirtualAlloc((LPVOID)(PEHeader->OptionalHeader.ImageBase),  
    PEHeader->OptionalHeader.SizeOfImage,  
    MEM_RESERVE,  
    PAGE_READWRITE);
```

If the reserved memory differs from the address given in *ImageBase*, base relocation as described below must be done.

3.2. Copy sections

Once the memory has been reserved, the file contents can be copied to the system. The section header must get evaluated in order to determine the position in the file and the target area in memory.

Before copying the data, the memory block must get committed:

```
dest = VirtualAlloc(baseAddress + section->VirtualAddress,  
    section->SizeOfRawData,  
    MEM_COMMIT,  
    PAGE_READWRITE);
```

Sections without data in the file (like data sections for the used variables) have a *SizeOfRawData* of 0, so you can use the *SizeOfInitializedData* or *SizeOfUninitializedData* of the *OptionalHeader*. Which one must get chosen depending on the bit flags *IMAGE_SCN_CNT_INITIALIZED_DATA* and *IMAGE_SCN_CNT_UNINITIALIZED_DATA* that may be set in the section's characteristics.

3.3. Base relocation

All memory addresses in the code / data sections of a library are stored relative to the address defined by *ImageBase* in the *OptionalHeader*. If the library can't be imported to this memory address, the references must get adjusted => *relocated*. The file format helps for this by storing informations about all these references in the base relocation table, which can be found in the directory entry 5 of the *DataDirectory* in the *OptionalHeader*.

This table consists of a series of this structure

```
typedef struct _IMAGE_BASE_RELOCATION {  
    DWORD    VirtualAddress;  
    DWORD    SizeOfBlock;  
} IMAGE_BASE_RELOCATION;
```

It contains $(SizeOfBlock - IMAGE_SIZEOF_BASE_RELOCATION) / 2$ entries of 16 bits each. The upper 4 bits define the type of relocation, the lower 12 bits define the offset relative to the *VirtualAddress*.

The only types that seem to be used in DLLs are

IMAGE_REL_BASED_ABSOLUTE

No operation relocation. Used for padding.

IMAGE_REL_BASED_HIGHLOW

Add the delta between the *ImageBase* and the allocated memory block to the 32 bits found at the offset.



3.4. Resolve imports

The directory entry 1 of the *DataDirectory* in the *OptionalHeader* specifies a list of libraries to import symbols from. Each entry in this list is defined as follows:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;           // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;        // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                           // -1 if bound, and real date\time stamp
                                           // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new
                                           // BIND)
                                           // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has
                                           // actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

The *Name* entry describes the offset to the NULL-terminated string of the library name (e.g. *KERNEL32.DLL*). The *OriginalFirstThunk* entry points to a list of references to the function names to import from the external library. *FirstThunk* points to a list of addresses that gets filled with pointers to the imported symbols.

When we resolve the imports, we walk both lists in parallel, import the function defined by the name in the first list and store the pointer to the symbol in the second list:

```
nameRef = (DWORD *)(baseAddress + importDesc->OriginalFirstThunk);
symbolRef = (DWORD *)(baseAddress + importDesc->FirstThunk);
for (; *nameRef; nameRef++, symbolRef++)
{
    PIMAGE_IMPORT_BY_NAME thunkData = (PIMAGE_IMPORT_BY_NAME)(codeBase + *nameRef);
    *symbolRef = (DWORD)GetProcAddress(handle, (LPCSTR)&thunkData->Name);
    if (*symbolRef == 0)
    {
        handleImportError();
        return;
    }
}
```

3.5. Protect memory

Every section specifies permission flags in its *Characteristics* entry. These flags can be one or a combination of

IMAGE_SCN_MEM_EXECUTE

The section contains data that can be executed.

IMAGE_SCN_MEM_READ

The section contains data that is readable.

IMAGE_SCN_MEM_WRITE

The section contains data that is writeable.

These flags must get mapped to the protection flags



- PAGE_NOACCESS
- PAGE_WRITECOPY
- PAGE_READONLY
- PAGE_READWRITE
- PAGE_EXECUTE
- PAGE_EXECUTE_WRITECOPY
- PAGE_EXECUTE_READ
- PAGE_EXECUTE_READWRITE

Now, the function *VirtualProtect* can be used to limit access to the memory. If the program tries to access it in a unauthorized way, an exception gets raised by Windows.

In addition the section flags above, the following can be added:

IMAGE_SCN_MEM_DISCARDABLE

The data in this section can be freed after the import. Usually this is specified for relocation data.

IMAGE_SCN_MEM_NOT_CACHED

The data in this section must not get cached by Windows. Add the bit flag *PAGE_NOCACHE* to the protection flags above.

3.6. Notify library

The last thing to do is to call the DLL entry point (defined by *AddressOfEntryPoint*) and so notifying the library about being attached to a process.

The function at the entry point is defined as

```
typedef BOOL (WINAPI *DllEntryProc)(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved);
```

So the last code we need to execute is

```
DllEntryProc entry = (DllEntryProc)(baseAddress + PEHeader->OptionalHeader.AddressOfEntryPoint);
(*entry)((HINSTANCE)baseAddress, DLL_PROCESS_ATTACH, 0);
```

Afterwards we can use the exported functions as with any normal library.

4. Exported functions

If you want to access the functions that are exported by the library, you need to find the entry point to a symbol, i.e. the name of the function to call.

The directory entry 0 of the *DataDirectory* in the *OptionalHeader* contains information about the exported functions. It's defined as follows:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
```




```
DWORD Name;  
DWORD Base;  
DWORD NumberOfFunctions;  
DWORD NumberOfNames;  
DWORD AddressOfFunctions; // RVA from base of image  
DWORD AddressOfNames; // RVA from base of image  
DWORD AddressOfNameOrdinals; // RVA from base of image  
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

First thing to do, is to map the name of the function to the ordinal number of the exported symbol. Therefore, just walk the arrays defined by *AddressOfNames* and *AddressOfNameOrdinals* parallel until you found the required name.

Now you can use the ordinal number to read the address by evaluating the n-th element of the *AddressOfFunctions* array.

5. Freeing the library

To free the custom loaded library, perform the steps

- Call entry point to notify library about being detached:

```
DllEntryProc entry = (DllEntryProc)(baseAddress + PEHeader->  
OptionalHeader.AddressOfEntryPoint);  
(*entry)((HINSTANCE)baseAddress, DLL_PROCESS_ATTACH, 0);
```

- Free external libraries used to resolve imports.
- Free allocated memory.

Certain memory stacks allocated by *MemoryLoadlibrary* are not restored with the call of *MemoryFreeLibrary*, Some pages of memory not deallocated when call *MemoryFreeLibrary*. The *FreeSections* function is the following.

```
void FreeSections(PIMAGE_NT_HEADERS old_headers, PMEMORYMODULE module)  
{  
    int i, size;  
    unsigned char *codeBase = module->codeBase;  
    unsigned char *dest;  
    bool b;  
    PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(module->headers);  
    for (i=0; i<module->headers->FileHeader.NumberOfSections; i++, section++)  
    {  
        if (section->SizeOfRawData == 0)  
        {  
            size = old_headers->OptionalHeader.SectionAlignment;  
            if (size > 0)  
            {  
                b=VirtualFree(codeBase + section->VirtualAddress,  
                    size,  
                    MEM_DECOMMIT  
                );  
                section->Misc.PhysicalAddress = 0;  
                ;  
            }  
            continue;  
        }  
    }  
}
```



```
b=VirtualFree(codeBase + section->VirtualAddress,
              section->SizeOfRawData,
              MEM_DECOMMIT
              );

    section->Misc.PhysicalAddress = 0;
}
```

6. MemoryModule

MemoryModule is a C-library that can be used to load a DLL from memory.

The interface is very similar to the standard methods for loading of libraries:

```
typedef void *HMEMORYMODULE;

HMEMORYMODULE MemoryLoadLibrary(const void *);
FARPROC MemoryGetProcAddress(HMEMORYMODULE, const char *);
void MemoryFreeLibrary(HMEMORYMODULE);
```

6.1. Downloads

Latest development release can be grabbed at <https://leviathan.joachim-bauch.de/cgi-bin/viewcvs.cgi/MemoryModule/trunk/?root=misc> (it's not a 24/7 server and is often offline during nights or on weekends)

All released versions can be downloaded from the list below.

Version 0.0.2 (MPL release)

<http://www.joachim-bauch.de/tutorials/downloads/MemoryModule-0.0.2.zip>

Version 0.0.1 (first public release)

<http://www.joachim-bauch.de/tutorials/downloads/MemoryModule-0.0.1.zip>

6.2. Known issues

- All memory that is not protected by section flags is gets committed using *PAGE_READWRITE*. I don't know if this is correct.

6.3. License

The MemoryModule library is released under the Mozilla Public License (MPL). It is provided as-is without ANY warranty. You may use it at your own risk.



7. Additional Examples

Following I added some examples which were not in the original work, and which will surely help to apply the method for lazy readers ;-)

7.1. An MFC Client and Dll

Folder `example_2\` contains an MFC Client and Dll. The Client loads the Dll using the *MemoryLoadLibraryEx*, an extension of the original methods I added which has exactly the same interface of the *LoadLibrary* (receives the filename as `char*`). The Dll simply exposes a method called *ShowADialog* which receives two strings and uses them as title and text of a normal messagebox.

NOTE

The code of this program is under the `example_2\` folder

Here's the code of the *OnCheck* method of the Client of this example:

```
void CClientDlg::OnCheck()
{
    char* username, *regnumber;

    HMEMORYMODULE hMod=MemoryLoadLibraryEx("../bin\\SampleDll.dll");

    if(hMod!=NULL) {
        //These are MFC things used to transform a CString object to a simple C buffer
        username=m_title.GetBuffer(m_title.GetLength());
        regnumber=m_content.GetBuffer(m_content.GetLength());

        //declares a function pointer of the same type of the exported method of the Dll
        BOOL (*fpShowADialog)(char*, char*)=NULL;

        //gets the proc address from the Dll
        fpShowADialog= (BOOL (*)(char*, char*))MemoryGetProcAddress(hMod, "ShowADialog");

        //invokes the method
        if(fpShowADialog!=NULL) {
            BOOL bRet=fpShowADialog(username, regnumber);

            //Releases the buffer previously allocated.
            m_title.ReleaseBuffer();
            m_content.ReleaseBuffer();

            //Resets the interface for another try if needed!
            m_title.Empty();
            m_content.Empty();
            UpdateData(FALSE);
        }

        //Free up all the allocated buffer and memory
        MemoryFreeLibrary(hMod);
    }
}
```

The *MemoryLoadLibraryEx* instead simply loads a given file in memory and then calls the *MemoryLoadLibrary* seen in previous sections.

```
HMEMORYMODULE MemoryLoadLibraryEx(char *filename) {
```



```
FILE *fp=NULL;
unsigned char *data=NULL;
size_t size;

fp = fopen(filename, "rb");
if (fp == NULL)
{
    char b[1024];
    sprintf(b, "Can't open DLL file \"%s\".", filename);
    OutputDebugString(b);
    return NULL;
}

fseek(fp, 0, SEEK_END);
size = ftell(fp);
data = (unsigned char *)malloc(size);
fseek(fp, 0, SEEK_SET);
fread(data, 1, size, fp);
fclose(fp);

stAllocatedMemoryBuffer=data;

return MemoryLoadLibrary(data);
}
```

stAllocatedMemoryBuffer is a static global variable to the *MemoryModule.c* module, which behaves in C more or less like a C++ class's property.

7.2. MFC Client with the Dll included into its Resources

This is at the end the real example you should be interested to see in real word: an exe with a Dll inserted as a resource, which doesn't uses any temporary file and which loads the Dll directly from memory.

NOTE

The code of this program is under the example_3\ folder

First of all it's needed a little of explanation on how to add a custom resource to a Win32 program from Visual Studio (Note that I refer to Microsoft Visual C++ 6.0). Well, the program must of course already have some resources, being a Win32 client.

NOTE

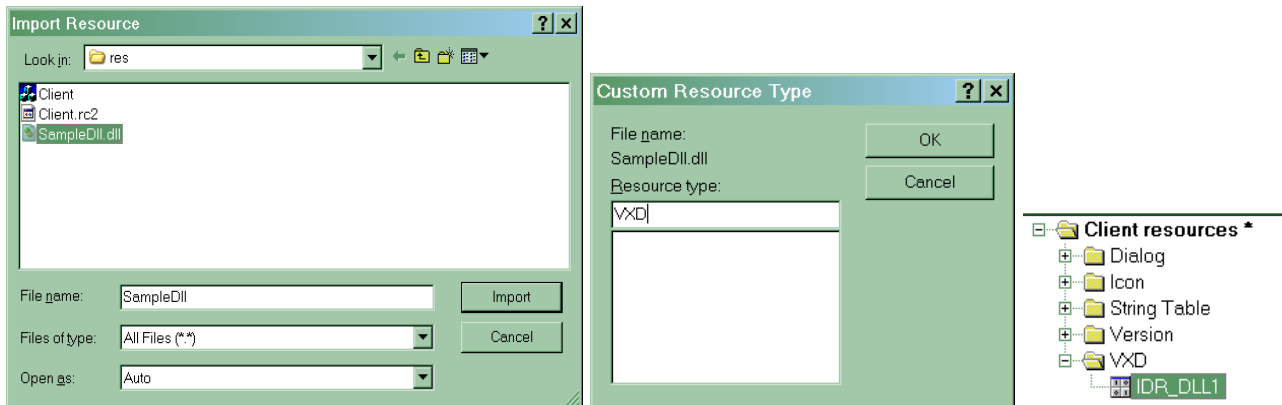
You can include much more easily resources also with a resource editor and then use the code to get resources inside the program, which are reported later on this section.

First of all you need to add custom resources to the Client as following (follow the sequence of snapshots from left to right):

1. Select with the right menu on an already existing resource "Import" and select the Dll you want to insert
2. Select the resource type as VXD (I'll tell you why later)
3. You should have then a new type of resource called VXD containing a resource which you renamed to IDR_DLL1 or whatever you like



4. If you try to open this resource you would see our DLL.



```
000000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
000010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....
000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000030 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00 .....
000040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 .....!..L.!Th
000050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
000060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
000070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...$.
000080 A2 CC 15 D4 E6 AD 7B 87 E6 AD 7B 87 E6 AD 7B 87 .....{...{...{
000090 65 B1 75 87 F2 AD 7B 87 D0 8B 71 87 DE AD 7B 87 e.u...{...q...{
0000a0 E6 AD 7A 87 A7 AD 7B 87 84 B2 68 87 E5 AD 7B 87 ..z...{...h...{
0000b0 D0 8B 70 87 E5 AD 7B 87 19 8D 7F 87 E7 AD 7B 87 ..p...{...{...{
0000c0 52 69 63 68 E6 AD 7B 87 00 00 00 00 00 00 00 00 Rich...{...
0000d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000e0 50 45 00 00 4C 01 04 00 22 A7 15 43 00 00 00 00 PE..L...".C...
0000f0 00 00 00 00 E0 00 0E 21 0B 01 06 00 00 50 00 00 .....!.....P..
000100 00 60 00 00 00 00 00 00 47 14 00 00 00 10 00 00 .....G.....
000110 00 60 00 00 00 00 00 10 00 10 00 00 00 10 00 00 .....
000120 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 .....
000130 00 C0 00 00 00 10 00 00 00 00 00 00 02 00 00 00 .....
000140 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00 .....
000150 00 00 00 00 10 00 00 00 90 6A 00 00 58 00 00 00 .....j..X...
000160 08 66 00 00 28 00 00 00 00 00 00 00 00 00 00 00 ..f..(.....
000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000180 00 B0 00 00 1C 05 00 00 00 00 00 00 00 00 00 00 .....
000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0001b0 00 00 00 00 00 00 00 00 00 60 00 00 CC 00 00 00 .....
0001c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0001d0 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 .....text...
0001e0 0A 41 00 00 00 10 00 00 00 50 00 00 00 10 00 00 ..A.....P.....
```

Why we choose to insert the Dll as a VXD type of resource? The problem is that each resource has a Resource Id and a Resource Type. In our example the Resource Id is IDR_DL1 which is defined somewhere in the code, and the Resource Type is "VXD". While it's always possible to define your own resource types, it's simpler to use one of those already defined by Windows. Inside the file *winuser.h* there are the definitions of the predefined resources types.

One of the resource types already defined is:

```
#define RT_VXD MAKEINTRESOURCE(20)
```

This value is used to identify VXD resources.

Then looking at MSDN resource management (I will not tell you the whole story) you should be able to write this code. It takes from the resources of the current module (the one which the code belongs) and write its dump into a locked memory section where it can be used for your own usage.

```
//Get a pointer to memory location where the specific resource is stored.
//This pointer then has to be passed to MemoryLoadLibrary
////////////////////////////////////
HRSRC hResLoad=NULL; // handle to loaded resource
```



```
HRSRC hRes=NULL;           // handle/ptr. to res. info. an hEXE
char *lpResLock=NULL;      // pointer to resource data

//first parameter is NULL means this module.
hRes = FindResource(NULL, MAKEINTRESOURCE(IDR_DLL1), RT_VXD);
if (hRes != NULL)
{
    // Load the resource into global memory.
    hResLoad = (HRSRC)LoadResource(NULL, hRes);
    if (hResLoad != NULL)
    {
        // Lock the resource into global memory.
        lpResLock = (char*)LockResource(hResLoad);
        if (lpResLock != NULL)
        {
            hMod=MemoryLoadLibrary(lpResLock);
        }
    }
}
else {
    TRACE0("Could not locate Internal Resource.");
    return;
}
////////////////////////////////////
```

The above code finds the resource using *FindResource*, then load the resource somewhere in memory, out of the .rsrc executable section, using *LoadResource*. The value returned from *LoadResource* is already a pointer which can be freely used from *MemoryLoadLibrary*, being a pointer to the extracted resource.

But there's a subtle problem indeed, the resources are shared among all the thread of the executable and thus calling *MemoryLoadLibrary* directly passing to it the *hResLoad* handle is not safe and might cause crashes. Thus I used an additional call to *LockResource* which does what its name tells: locks the resource for access from other threads. Now we are ready then to call *MemoryLoadLibrary* passing it the *lpResLock* value.

We are then ready to finish the code of the *OnCheck* routine as following:

```
void CClientDlg::OnCheck()
{
    char* username, *regnumber;
    BOOL bRet=FALSE;

    HMEMORYMODULE hMod=NULL;

    //Get a pointer to memory location where the specific resource is stored.
    //This pointer then has to be passed to MemoryLoadLibrary
    //////////////////////////////////////
    HRSRC hResLoad=NULL;      // handle to loaded resource
    HRSRC hRes=NULL;          // handle/ptr. to res. info. an hEXE
    char *lpResLock=NULL;     // pointer to resource data

    //first parameter is NULL means this module.
    hRes = FindResource(NULL, MAKEINTRESOURCE(IDR_DLL1), RT_VXD);
    if (hRes != NULL)
    {
        // Load the resource into global memory.
        hResLoad = (HRSRC)LoadResource(NULL, hRes);
        if (hResLoad != NULL)
        {
            // Lock the resource into global memory.
            lpResLock = (char*)LockResource(hResLoad);
        }
    }
}
```



```
        if (lpResLock != NULL)
        {
            hMod=MemoryLoadLibrary(lpResLock);
        }
    }
else {
    TRACE0("Could not locate Internal Resource.");
    return;
}
////////////////////////////////////

if(hMod!=NULL) {
    //These are MFC things used to transform a CString object to a simple C buffer
    username=m_username.GetBuffer(m_username.GetLength());
    regnumber=m_regnumber.GetBuffer(m_regnumber.GetLength());

    //declares a function pointer of the same type of the exported method of the Dll
    BOOL (*fpShowADialog)(char*, char*)=NULL;

    //gets the proc address from the Dll
    fpShowADialog= (BOOL (*)(char*,char*))
        MemoryGetProcAddress(hMod,"CheckRegistrationNumber");

    //invokes the method
    if(fpShowADialog!=NULL) {
        bRet=fpShowADialog(username, regnumber);

        //Releases the buffer previously allocated.
        m_username.ReleaseBuffer();
        m_regnumber.ReleaseBuffer();

        //Resets the interface for another try if needed!
        m_username.Empty();
        m_regnumber.Empty();
        UpdateData(FALSE);

        if(bRet==TRUE)
            ::MessageBox(NULL,"Compliments, well done the serial is correct",
                "Shub-Nigurrath",MB_OK);
        else
            ::MessageBox(NULL,"Nahh, you have to get a valid serial somehow!",
                "Shub-Nigurrath",MB_OK);
    }

    MemoryFreeLibrary(hMod);
}
}
```

Compile everything and try to execute it from a place where the Dll is not available, it works.

7.2.1 How the program looks from inside OllyDbg

I have just a curiosity, how the program looks from OllyDbg. This program comes from a previous tutorial I wrote [2]: in that document I did a discussion on it, so I will start from there.

1. Run the program into OllyDbg and press F9 to let it go freely.
2. Enter username (shub-nigurrath) and regnumber (arteam), press Check
3. Suspend the application into OllyDbg
4. Press ALT-K to see the Call Stack



5. Select the right call (you know which), and land here:

00401632	68 90404000	PUSH Client.00404090	ASCII "CheckRegistrationNumber"
00401632	52	PUSH EDX	
00401633	894424 1C	MOV DWORD PTR SS:[ESP+1C],EAX	
00401637	E8 F4040000	CALL Client.00401B30	MFC42.73EB0378
0040163C	83C4 08	ADD ESP,8	
0040163F	85C0	TEST EAX,EAX	
00401641	74 56	JE SHORT Client.00401699	
00401643	8B4C24 14	MOV ECK,DWORD PTR SS:[ESP+14]	
00401647	51	PUSH ECK	
00401648	55	PUSH EBP	
00401649	FDD0	CALL EAX	
0040164B	83C4 08	ADD ESP,8	
0040164E	8BC8	MOV ECK,ESI	
00401650	8BE8	MOV EBP,EAX	
00401652	6A FF	PUSH -1	
00401654	E8 31080000	CALL <JMP.<MFC42.#5572>	
00401659	6A FF	PUSH -1	
0040165B	8BCF	MOV ECK,EDI	
0040165D	E8 28080000	CALL <JMP.<MFC42.#5572>	
0040165E	8BC8	MOV ECK,ESI	
00401664	E8 1B080000	CALL <JMP.<MFC42.#2614>	
00401669	8BCF	MOV ECK,EDI	
0040166B	E8 14080000	CALL <JMP.<MFC42.#2614>	
00401670	6A 00	PUSH 0	
00401672	8BCB	MOV ECK,EBX	
00401674	E8 05080000	CALL <JMP.<MFC42.#6334>	
00401679	83FD 01	CMF EBP,1	
0040167C	6A 00	PUSH 0	
0040167E	68 80404000	PUSH Client.00404080	ASCII "Shub-Nigurath"
00401681	71 07	JNE SHORT Client.0040168C	
00401685	E8 50404000	PUSH Client.00404050	ASCII "Compliments, well done the serial is correct"
0040168A	EB 05	JNE SHORT Client.00401691	
0040168C	> E8 20404000	PUSH Client.00404020	ASCII "Nahh, you have to get a valid serial somehow!"
00401691	> 6A 00	PUSH 0	hOwner = NULL
00401693	FF15 20324000	CALL DWORD PTR DS:[<USER32.MessageBoxA>]	MessageBoxA

6. The string “CheckRegistrationNumber” appears at 0x0040162D, but its just a string pushed on the data stack

- At this stage open the Executables Module view into OllyDbg and you should have something like the following:

Base	Size	Entry	Name	(system)	File version	Path
00400000	00011000	00401E00	Client		1, 0, 0, 1	C:\Sources\example_3\bin\Client.exe
10000000	00000000	10001400	PPSYS			
50090000	00097000	50093200	COMCTL32	(system)	5.82 (xpsp_sp2-...	C:\WINDOWS\system32\COMCTL32.dll
60000000	00015000	60002E00	SynTPFcs	(system)	7.12.3.880104	C:\WINDOWS\system32\SynTPFcs.dll
70000000	000F0000	70007C00	MFC42	(system)	6.02.4131.0	C:\WINDOWS\system32\MFC42.dll
74720000	00048000	74721300	MSCIIF	(system)	5.1.2600.2180	C:\WINDOWS\system32\MSCTF.dll
77130000	0008C000	77121E00	OLEAUT32	(system)	5.1.2600.2180	C:\WINDOWS\system32\OLEAUT32.dll
773D0000	00102000	773D4200	comctl32	(system)	6.0 (xpsp_sp2_r...	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.2180_x-ww_319c2b12_6595b641-6.0.2600.2180_x-ww_319c2b12_x-ww_319c2b12\comctl32.dll
774E0000	0013D000	774F0000	ole32	(system)	5.1.2600.2666	C:\WINDOWS\system32\ole32.dll
77D40000	00030000	77D41100	VERSION	(system)	5.1.2600.2180	C:\WINDOWS\system32\VERSION.dll
77FC0000	00058000	77C1F200	nsuport	(system)	7.0.0.2600.2180	C:\WINDOWS\system32\nsuport.dll
78000000	00090000	78004E00	USER32	(system)	5.1.2600.2666	C:\WINDOWS\system32\USER32.dll
77D00000	0009B000	77D07000	ADVAPI32	(system)	5.1.2600.2180	C:\WINDOWS\system32\ADVAPI32.dll
77E00000	00091000	77E02E00	RPCRT4	(system)	5.1.2600.2180	C:\WINDOWS\system32\RPCRT4.dll
777F0000	00040000	77F16300	GDI32	(system)	5.1.2600.2180	C:\WINDOWS\system32\GDI32.dll
77F60000	00076000	77F65100	SHLWAPI	(system)	6.00.2900.2713	C:\WINDOWS\system32\SHLWAPI.dll
700F0000	00074000	700F0400	kernel32	(system)	5.1.2600.2180	C:\WINDOWS\system32\kernel32.dll
70080000	00080000	70081300	ntdll	(system)	5.1.2600.2180	C:\WINDOWS\system32\ntdll.dll
7C9C0000	00014000	7C9C7376	SHELL32	(system)	6.00.2900.2620	C:\WINDOWS\system32\SHELL32.dll

There are no additional modules or Dlls loaded, of course!

8. The real call to the *CheckRegistrationNumber* API is at 0x00401649 (CALL EAX). And we will be able to find again the patch to be done at

```
00ED10DD    /75 28                                JNZ SHORT 00ED1107
```

It seems like the patch is done, but the problem is that this location is now dynamic and thus it is much more complex to write a patcher for this program. Moreover every time we press Check the Dll is loaded and written in memory from scratch thus nullifying our previous patch.

Add to this also these simple tricks and you will get how difficult the situation could become:

- we can force the DLL to be written in memory at different locations each time
- pack/protect the DLL directly when inserted in the client resources
- crypt the strings of the called DLLs export methods
- use numeric export tables for DLLs, not using names
- ...



8. References

- [1] “Portable Executable File Format Compendium”, Goppit, <http://tutorials.accessroot.com>
- [2] “Writing Loaders for DLLs: theory and techniques Version 1.0”, Shub-Nigurath, <http://tutorials.accessroot.com>

9. Conclusions

Well, this is the end of this story, I hope all the things here said will be useful to better understand how process is handled by the OS and in which manners we can keep process control and make debugging with some advanced techniques. I suggest as usual to use this tutorial for learning more in deep how the operative system works and to use these examples to evolve your RCE techniques and not to crack programs.

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.

10. History

- Version 1.0 – first public release!
- Version 1.1 – added the MFC sample to the distribution
- Version 1.2 – expanded the examples and added some more explanations.

11. Greetings

I wish to tank all the ARTeam members of course and who read the beta versions of this tutorial and contributed,.. and of course you, who are still alive at the end of this document!



<http://cracking.accessroot.com>