



CodeBreakers Magazine

Security & Anti-Security - Attack & Defense

Volume 1, Issue 1, 2006



Writing Loaders for DLLs: Theory and Techniques

Shub-Nigurrath [ARTeam]
January 2006

Abstract

Generally speaking several applications have a protection implemented into one of its DLLs. This is the case for example of out-of-the-shelf protectors used by developers, such as TimeLock, for which the protection checks are implemented into protected DLLs, stored in the Windows' system32 folder. In other cases the application itself is an add-on or a plug-in of another application, so usually a Dll. Finally there are also situations where protection resides into a Dll for specific design reasons.

1 Introduction

This tutorial aims to introduce some different approaches to writing loaders for those applications where the protection or the registration checks resides into one or more DLLs. The tutorial can be read as standalone, but it's a natural follow-up of my previous tutorials, written also with **ThunderPwr** [1, 2].

As usual I will provide sample code with this tutorial, and non-commercial sample victims. All the sources have been tested with Win2000/XP and Visual Studio 6.0.

The techniques described here are general and not specific to any commercial applications. The whole document must be intended as a document on programming advanced techniques, how you will use these information will be totally up to your responsibility.

Generally speaking several applications have a protection implemented into one of its DLLs. This is the case for example of out-of-the-shelf protectors used by developers, such as TimeLock, for which the protection checks are implemented into protected DLLs, stored in the Windows' system32 folder. In other cases the application itself is an add-on or a plug-in of another application, so usually a DLL. Finally there are also situations where protection resides into a DLL for specific design reasons.

In all these cases usually a normal patch is enough and solves most of the cases, but like happens with normal applications there are situations where a loader is a better solution for different reasons, already explained in [1]: portability, protection of the DLL, dimension of the dumped DLL too great or simply because we have fun writing a working loader!

Moreover the potentials of debug loaders, sort of mini debuggers, are extremely large and only your imagination actually is a limit for such technique.

Before continue reading I suggested that you have already understood how a program is loaded into memory, what is a loader, what it does and how and which are the differences among Debug and Standard Loaders. I suggest reading [1] and [2] to better understand the rest of this tutorial.

2 Possible methods to take control of DLLs

Generally speaking in order to patch a DLL through a loader (then dynamically in memory) you have to take control of the DLL: by definition a loader is something that takes care or controls the target's loading/execution process in order to perform its actions.

Generally speaking there are two different ways to take control of a DLL loaded by a program:

- Take Control of the whole application. This allows performing the proper actions when the target DLL is loading into the application's memory space. This approach is the most simple to think (because we are accustomed to debuggers), but requires the most complex code to write and involves Debug Loaders. Moreover for very large applications the final result might slow down too much the program or require too much memory: you are in this case debugging the whole application.
- Write a DLL proxy. With this approach you will write a proxy DLL, a DLL written by you exposing exactly the same interface of the original DLL to the original application. The application then loads the false DLL and invokes its methods as it would have done with the real DLL. The proxy DLL then performs the required actions (patches) and calls the original DLL passing the parameters coming from the application. This approach requires a less complex code but also some tricks I'll explain. The advantage is that you will not debug the whole application.

I just tell now that the two methods are not equivalent of course: there are drawbacks for each. For example taking control of the whole application on the one hand means to slow down it or eat memory too much, on the other hand it is much more powerful than writing a Proxy DLL. It will be up to your sensibility and experience to decide where to use one or another technique.

3 Target Application

In order to have a liet-motif through the entire document and to not use commercial applications I coded a simple application for which the serial number check is inside an external Dll, called by a front-end GUI.

The real application is simply a front-end which takes the serial number from the user and asks to the Dll to check if it's correct, then reports the answer.

The schema is not much different, in its essentials, from several commercial applications, so take the relative conclusions on your own ;-)

The Target Application code included into this tutorial will not be described in details, just because it is relatively easy to understand and because would divert the focus. Moreover it is good to approach to these examples as any other application: think as you do not have sources.

NOTE

The code of this program is under the Simple_Client_1\ folder

The program looks like in Figure 1.

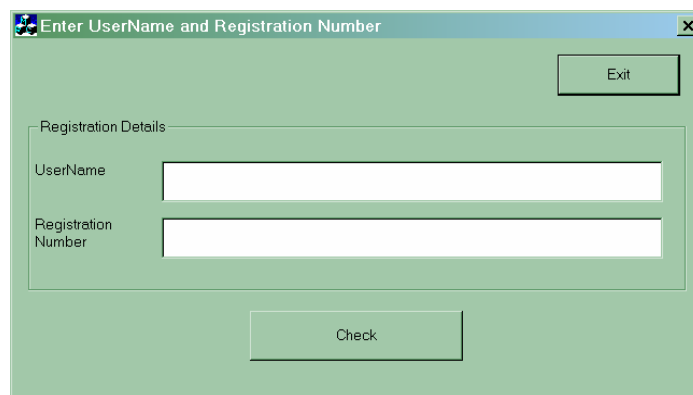


Figure 1 - Interface of the used sample target program.

The example works as following:

1. enter the registration details,
2. press Check to see if the serial is correct.

First of all let's try to debug it a little before going further.. as usually with Olly. The program is composed by two parts: Clients.exe and RegistrationDll.Dll which exposes a method called CheckRegistrationNumber (what the hell could be the use of this method? ☺)

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

Open OllyDbg and let the target program run freely (using F9). You can configure Olly to stop on new modules so as to see when the DLL is loaded or simply let the program run freely till it shows its interface. Now follow these steps:

1. Insert two values as in Figure 2.

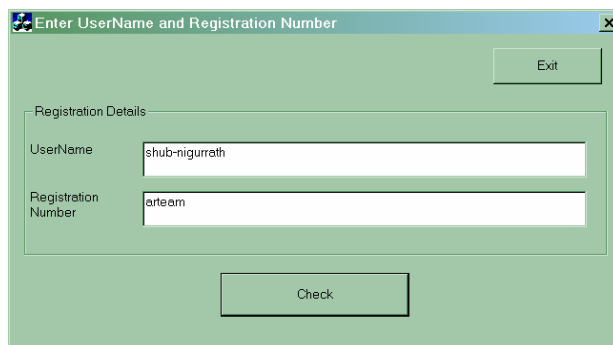


Figure 2 - values used for the debugging.

Now press "Check" then suspend the application in Olly when the badboy MessageBox is shown. Go to Olly and browse the Call Stack (CTRL-K) until you arrive at the JNZ where the goodboy or the badboy message is shown (Figure 3).

004015BB	- 55	PUSH EBP	
004015BC	- E8 7F000000	CALL <JMP.<RegistrationDll.CheckRegistrationNumber>	
004015C1	- 83C4 08	ADD ESP,8	
004015C4	- 8BCE	MOV ECX,ESI	
004015C6	- 8BE8	MOV EBP,EAX	
004015C8	- 6A FF	PUSH -1	
004015CA	- E8 81020000	CALL <JMP.<MFC42.#5572>	
004015CF	- 6A FF	PUSH -1	
004015D1	- 8BCF	MOV ECX,EDI	
004015D3	- E8 78020000	CALL <JMP.<MFC42.#5572>	
004015D8	- 83FD 01	CMP EBP,1	
004015DB	- 6A 00	PUSH 0	
004015DD	- 68 80304000	PUSH Client.00403080	ASCII "Shub-Nigurrath"
004015E2	- 75 07	JNZ SHORT Client.004015EB	ASCII "Compliments, well done the serial is correct"
004015E4	- 68 50304000	PUSH Client.00403050	
004015E9	- EB 05	JMP SHORT Client.004015F0	ASCII "Nahh, you have to get a valid serial somehow!"
004015EB	- 68 20304000	PUSH Client.00403020	hOwner = NULL
004015F0	- 6A 00	PUSH 0	MessageBoxA
004015F2	- FF15 E8214000	CALL DWORD PTR DS:[<<USER32.MessageBoxA>]	

Figure 3 - Check routine of the Client

A little above in the code there's the call to the DLL's method.

2. Place a breakpoint at the DLL method's call (0x4015BC) and restart the application (CTRL-F2) then press F9 to run it. Re-enter the username and serial used before.
3. You should land at the above said call. On the call stack there are the values you entered (the username and the regnumber):

0012F800	00374000	ASCII "shub-nigurrath"
0012F804	00374050	ASCII "arteam"
0012F808	0012FE8C	

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

- Follow the call (F7) and you will land into the Dll (passing through the JMP of the IAT). The disassembly of the routine is quite easy to follow:

1000106F	. 894C24 24	MOV DWORD PTR SS:[ESP+24],ECX	
10001073	> OFFE5434 10	MOVSB EDI, BYTE PTR SS:[ESP+ESI+10]	
10001078	. 52	PUSH EDI	
10001079	. E8 92000000	CALL Registra.10001110	
1000107E	. 884434 14	MOV BYTE PTR SS:[ESP+ESI+14],AL	
10001082	. OFFE4434 08	MOVSB EAX, BYTE PTR SS:[ESP+ESI+8]	
10001087	. 50	PUSH EAX	
10001088	. E8 83000000	CALL Registra.10001110	
1000108D	. 83C4 08	ADD ESP,8	
10001090	. 884434 04	MOV BYTE PTR SS:[ESP+ESI+4],AL	
10001094	. 46	INC ESI	
10001095	. 83FE 0C	CMP ESI,0C	
10001098	. 7C D9	JL SHORT Registra.10001073	This cycle simple erase the memory locations
1000109A	. 33F6	XOR ESI,ESI	
1000109C	> 8A4C34 04	MOV CL, BYTE PTR SS:[ESP+ESI+4]	
100010A0	. 51	PUSH ECX	
100010A1	. E8 6AFF0000	CALL Registra.10001010	
100010A6	. OFFED0	MOVSB EDI,AL	
100010A9	. 52	PUSH EDI	
100010AA	. E8 61000000	CALL Registra.10001110	
100010AF	. 83C4 08	ADD ESP,8	
100010B2	. 884434 1C	MOV BYTE PTR SS:[ESP+ESI+1C],AL	AL contains the rotated letter which is copied in memory
100010B6	. 46	INC ESI	increments the variable used for the loop
100010B7	. 83FE 0C	CMP ESI,0C	check the exit condition. Note the 0C value.
100010BA	. 7C E0	JL SHORT Registra.1000109C	
100010BC	. 8D7424 10	LEA ESI, DWORD PTR SS:[ESP+10]	
100010C0	. 8D4424 1C	LEA EAX, DWORD PTR SS:[ESP+1C]	
100010C4	> 8A10	MOV DL, BYTE PTR DS:[EAX]	Here the real serial appears on the register EAX
100010C6	. 8ACA	MOV CL,DL	
100010C8	. 3A16	CMP DL, BYTE PTR DS:[ESI]	
100010CA	. 75 28	JNE SHORT Registra.100010F4	
100010CC	. 84C9	TEST CL,CL	
100010CE	. 74 14	JE SHORT Registra.100010E4	
100010D0	. 8A50 01	MOV DL, BYTE PTR DS:[EAX+1]	
100010D3	. 8ACA	MOV CL,DL	
100010D5	. 3A56 01	CMP DL, BYTE PTR DS:[ESI+1]	
100010D8	. 75 1A	JNE SHORT Registra.100010F4	
100010DA	. 83C0 02	ADD EAX,2	
100010DB	. 83C5 02	ADD ESI,2	
100010DE	. 84C9	TEST CL,CL	
100010E2	. 75 E0	JNE SHORT Registra.100010C4	

Figure 4 - portion of the disassembly of CheckRegistrationNumber.

Figure 4 reports a commented section of the Dll method. The interesting point is the value of EAX when we are at the instruction at 0x100010C4.

```

EAX 0012F7F0 ASCII "fuho:{vthee"
ECX 00000000
EDX 00000000
EBX 0012FE8C
ESP 0012F7D4
EBP 00374000 ASCII "shub-nigurrath"
ESI 0012F7E4 ASCII "arteam"
EDI 0012FE8C

```

Figure 5 - values of registers when stopped at 0x100010C4.

As you can see EAX points to the correct serial which is in this case "fuho:{vthee".

- Write down the real serial, reopen the application and insert it to check if it is ok.

NOTE

I suggest printing the Dll sources and compare them with this disassembly to see how the compiler optimizes the sources and which the correspondences between ASM and C structures are. It's an exercise I suggest to do at least once in your RCE life ;-)

Well we have successfully fished a serial, now what can we do? We could patch the application in different ways, but of course the better way is to patch the Dll, which is probably changing less among releases (this is generally true for applications using out-of-the-shelf protectors). Anyway remember that the general rule is: patch at the deepest level possible!

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

For this document we will patch the Dll in the simplest way possible: changing the conditional jump at 0x100010CA:

```
100010CA    /75 28                JNZ SHORT Registra.100010F4
```

Becomes:

```
100010CA    /75 18                JNZ SHORT Registra.100010E4
```

100010C4	> 8A10	MOV DL,BYTE PTR DS:[EAX]	Here the real serial appears on the register EAX
100010C6	- 8ACA	MOV CL,DL	
100010C8	- 3A16	CMP DL,BYTE PTR DS:[ESI]	compares the real serial chars with the serial inserted
100010CA	- 75 18	JNZ SHORT Registra.100010E4	if a char is different then jumps away
100010CC	- 84C9	TEST CL,CL	
100010CE	- 74 14	JZ SHORT Registra.100010E4	
100010D0	- 8A50 01	MOV DL,BYTE PTR DS:[EAX+1]	
100010D3	- 8ACA	MOV CL,DL	
100010D5	- 3A56 01	CMP DL,BYTE PTR DS:[ESI+1]	
100010D8	- 75 1A	JNZ SHORT Registra.100010F4	
100010DA	- 83C0 02	ADD EAX,2	
100010DD	- 83C6 02	ADD ESI,2	
100010E0	- 84C9	TEST CL,CL	
100010E2	- 75 E0	JNZ SHORT Registra.100010C4	
100010E4	- 33C0	XOR EAX,EAX	destination of the routine in case it's all fine
100010E6	- 33C9	XOR ECX,ECX	
100010E8	- 85C0	TEST EAX,EAX	
100010EA	- 0F94C1	SETC CL	
100010ED	- 8BC1	MOV EAX,ECX	
100010EF	- 5E	POP ESI	
100010F0	- 83C4 24	ADD ESP,24	
100010F3	- C3	RETN	
100010F4	- 1BC0	SBB EAX,EAX	destination of the routine in case the serial is not ok
100010F6	- 5E	POP ESI	
100010F7	- 83D8 FF	SBB EAX,-1	
100010FA	- 33C9	XOR ECX,ECX	
100010FC	- 85C0	TEST EAX,EAX	
100010FE	- 0F94C1	SETC CL	
10001101	- 8BC1	MOV EAX,ECX	
10001103	- 83C4 24	ADD ESP,24	
10001106	- C3	RETN	

Figure 6 - CheckRegistrationNumber patched disassembly details

NOTE

We can also write an Oraculum which is always a good solution in these cases, but for the moment we will simply force the check to always return TRUE. The difference with the code here presented is only that, instead of patching you will have to get the application's context and then store it away.

Suppose, for the sake of simplicity that we cannot patch the Dll on disk (for example because it's heavily protected); the only solution would be to code a loader which takes control of the application or of the Dll and patch it at the right time (that is after it has been loaded in memory and before execution).

4 Take Control of the whole application

The idea is pretty simple I think, especially for those of you who already read the tutorial on Loaders [1]. In the Appendix 1 "Complete example of a debug-loader cycle" of document [1] (note that this appendix has been added since version 1.2) I already have introduced the argument, but here I will extend it.

The basic concept is: we want to write a debug-loader which takes control of the whole application more or less like a ring3 debugger, like Olly then. The loader will reach specifically to the debug event `LOAD_DLL_DEBUG_EVENT`, coming from the application.

Then the gate-condition (activation of the loader) this time will be the loading in memory of a specific DLL. The next thing the loader should do is to understand where in memory the DLL is loaded and then properly apply the patch.

All the following code can be written quite easily and once for all with the framework I introduced in [1] (The real advantage of the framework is to maximize code reusability), but for this tutorial I will not use my framework and will use instead simple C code, which being simpler should help to understand faster.

4.1 Writing first debug loader and introducing the code structure

This paragraph will extend what document [1] introduced, in order to successfully patch the DLL as told in Section 3. The code presented is always well commented. I will only focus on the most important parts.

NOTE

I used some functions of the DDK package. Just because I didn't want to install the whole DDK I wrote some wrappers. You can find them into the `NTInternals.h` and `NTInternals.cpp` files. These two files are different version of the two similar found in [1], because have been converted in C (thus less readable). I will not comment on them, just read the sources if you are curious.

The only comment is that these wrappers have been written to allow the programmer who's writing the `main()` to use them transparently, as they were normal Win32 APIs. So indeed it's not strictly required to know how they are implemented.

NOTE

Note that the code reported here is simplified and that I left out all the debug events not occurring with this target program.

The code of this program is under the `Simple_Client_Loader_2\` folder.

The complete loader with also all the debug events erased by this simple loader is in the folder `Complete_Client_Loader_3\` or can be found in [1].

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

<----- Start Code Snippet ----->

```
int main(int argc, char* argv[])
{
    STARTUPINFO      si;
    PROCESS_INFORMATION pi;

#ifdef _DEBUG
    MODULEINFO      mi;
#endif

    //ISDEBUGGERPRESENT PATCH INFO:
    DWORD DebugPatch[] = {0x00000000};
    //Patch byte info:
    //Search (read) byte
    BYTE scanbytd[] = {0x02};
    //Found (write) byte
    BYTE replbytd[] = {0x01};

    //////////////////////////////////////
    //MAIN PROGRAM PATCH INFO:
    //Patch Address info: # elements in following arrays must be synchronized for Address/scan/replace
    DWORD AddressOfPatch[] = {0x100010CA, 0x100010CB};

    //Patch byte info:
    //Search (read) byte. Original bytes read from the Dll in memory (attn: # elements must be the same
    //of AddressOfPatch)
    BYTE scanbyte[] = {0x75, 0x28};
    //Found (write) byte. New patch bytes to be written in memory (attn: # elements must be the same
    //of AddressOfPatch)
    BYTE replbyte[] = {0x75, 0x18};

    //Target file system information
    char FileName[] = "..\\Simple_Client\\bin\\Client.exe";
    //specific Dll for loaddll event (if required). This variable must be UPPERCASE.
    char *mydll = "REGISTRATIONDLL.DLL";
    //////////////////////////////////////

    BYTE DataRead[] = {0};
    char b[1024];
    BOOL contproc;
    HMODULE hMods[2048];
    DWORD cbNeeded;
    DWORD nMods;
    DWORD nPatches;
    DWORD Pid[2];
    DWORD dwPid;
    HANDLE hSaveFile;
    HANDLE hSaveProcess;
    HANDLE hSaveThread;
    LPVOID lpMsgBuf;
    DWORD dwRead;
    DWORD dwWritten;
    DWORD dwContinueStatus;
    char *dbdll = "KERNEL32.DLL"; //IsDebuggerPresent dll for loaddll event (if required)
    FARPROC ProcAdd;
    unsigned int i;
    unsigned int j;
    unsigned int k = 0; //count # of processes
    unsigned int l = 0; //count # of breakpoints
    unsigned int modlist[200];

    DEBUG_EVENT DebugEv; //debugging event information
    contproc = TRUE; //default; continue processing = TRUE
    dwContinueStatus = DBG_CONTINUE;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    char commandline[1024];
    //This function builds up the commandline string to be given to CreateProcess
    BuildCommandLine(argc, argv, FileName, commandline);

    //Start the child process.
    if(!CreateProcess( NULL, //No module name (use command line).
        commandline, //Command line.
        NULL, //Process handle not inheritable.
        NULL, //Thread handle not inheritable.
        TRUE, //Set handle inheritance.
        DEBUG_PROCESS, //No creation flags (use for more than 1 process
        //DEBUG_PROCESS+DEBUG_ONLY_THIS_PROCESS, //(use for only 1 process)
        NULL, //Use parent's environment block.
        NULL, //Use parent's starting directory.
```


WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
&si,           //Pointer to STARTUPINFO structure.
&pi ))         //Pointer to PROCESS_INFORMATION structure.
{
    MessageBox(0,"Unexpected load error","Create Process Failed",MB_OK+MB_TASKMODAL+MB_ICONERROR);
    return 1;
}

//Hides the debugger more deeply to the target. IsDebuggerPresent patch is often not enough
//to effectively hide the loader.
HideDebugger(pi.hThread, pi.hProcess);

while (TRUE) {
    //Wait for a debugging event to occur. The second parameter indicates
    //that the function does not return until a debugging event occurs.

    if (WaitForDebugEvent(&DebugEv, 1000)) { //wait 1 second
        //Process the debugging event code.

        switch (DebugEv.dwDebugEventCode) {

            case CREATE_PROCESS_DEBUG_EVENT: {
                //As needed, examine or change the registers of the process's initial thread with
                //the GetThreadContext and SetThreadContext functions; read from and write to the
                //process's virtual memory with the ReadProcessMemory and WriteProcessMemory functions;
                //and suspend and resume thread execution with the SuspendThread and ResumeThread
                //functions. Be sure to close the handle to the process image file with CloseHandle.

                contproc = TRUE;
                dwContinueStatus = DBG_CONTINUE;
                hSaveFile = DebugEv.u.CreateProcessInfo.hFile;
                hSaveProcess = DebugEv.u.CreateProcessInfo.hProcess;
                hSaveThread = DebugEv.u.CreateProcessInfo.hThread;

                Pid[k] = GetProcessId(hSaveProcess);
                dwPid = Pid[k];

                //more than 1 process
                if (k > 0)
                {
                    //DebugActiveProcessStop(Pid[0]);

                    //OpenProcess(
                    //  PROCESS_ALL_ACCESS, //access flag
                    //  FALSE, //handle inheritance flag
                    //  dwPid //process identifier
                    //);

                    //DebugActiveProcess(dwPid);

                    //no need to go further
                    contproc = FALSE;
                }

                k++;

                //include process info
                sprintf( b, "hFile:%X\n"
                    "ProcessId:%X\n"
                    "hProcess:%X\n"
                    "hThread:%X\n"
                    "lpBaseOfImage:%08X\n"
                    "dwDebugInfoFileOffset:%d\n"
                    "nDebugInfoSize:%d\n"
                    "lpThreadLocalBase:%08X\n"
                    "lpStartAddress:%08X\n"
                    "lpImageName:%08X\n"
                    "fUnicode:%d",
                    DebugEv.u.CreateProcessInfo.hFile, Pid[k-1], DebugEv.u.CreateProcessInfo.hProcess,
                    DebugEv.u.CreateProcessInfo.hThread, DebugEv.u.CreateProcessInfo.lpBaseOfImage,
                    DebugEv.u.CreateProcessInfo.dwDebugInfoFileOffset,
                    DebugEv.u.CreateProcessInfo.nDebugInfoSize,
                    DebugEv.u.CreateProcessInfo.lpThreadLocalBase,
                    DebugEv.u.CreateProcessInfo.lpStartAddress,
                    DebugEv.u.CreateProcessInfo.lpImageName, DebugEv.u.CreateProcessInfo.fUnicode
                );

                MessageBox(NULL, b, "CREATE_PROCESS_DEBUG_EVENT",MB_OK+MB_TASKMODAL);
            }
            break;

            case LOAD_DLL_DEBUG_EVENT: {
                //Read the debugging information included in the newly loaded DLL.
                //Be sure to close the handle to the loaded DLL with CloseHandle.
```



WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
contproc = TRUE;
dwContinueStatus = DBG_CONTINUE;

if (DebugEv.u.LoadDll.hFile == NULL) {
    break;
}

//EnumProcessModules returns an array of hMods for the process
//Fails first time for ntdll.dll
//Do not worry it's a normal behavior
if (!EnumProcessModules(hSaveProcess, hMods, sizeof(hMods), &cbNeeded)) {
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //Default language
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
    );

#ifdef _DEBUG
    //Display any error msg.
    ::MessageBox(NULL, (const char*)lpMsgBuf, "EnumProcessModules Error",
        MB_OK+MB_TASKMODAL);
#endif

    //Free the buffer.
    LocalFree( lpMsgBuf );
    SetLastError(ERROR_SUCCESS);

    //close handle to load dll event
    CloseHandle(DebugEv.u.LoadDll.hFile);
    break;
}

//Calculate number of modules in the process
nMods = cbNeeded / sizeof(HMODULE);

for ( i = 0; i < nMods; i++ ) {
    HMODULE hModule = hMods[i];
    char szModName[MAX_PATH];

    //GetModuleFileNameEx is like GetModuleFileName, but works in other process
    //address spaces. Get the full path to the module's file.
    GetModuleFileNameEx( hSaveProcess, hModule, szModName, sizeof(szModName));

    if ( 0 == i ) { //First module is the EXE. Add to list and skip it.
        modlist[i] = i;
    }
    else //Not the first module. It's a DLL
    {
        //Determine if this is a DLL we've already seen
        if ( i == modlist[i] ) {
            continue;
        }
        else {
            //We haven't see it, add it to the list
            modlist[i] = i;

#ifdef _DEBUG
            //Get the module information
            GetModuleInformation(
                hSaveProcess,
                hModule,
                &mi,
                cbNeeded
            );
            //include DLL entry, name and base image address, etc. info
            sprintf( b, "DLL entry:%d\n"
                "DLL module:%s\n"
                "Load address:%08X\n"
                "Size of image:%08X\n"
                "Entry Point:%08X", i, szModName, hModule, mi.SizeOfImage, mi.EntryPoint
            );
            MessageBox(NULL, b, "LOAD_DLL_DEBUG_EVENT",
                MB_OK+MB_TASKMODAL+MB_ICONINFORMATION);
#endif

            //Find the last '\\' to obtain a pointer to just the base module name part
            //(i.e. mydll.dll w/o the path)
            PSTR pszBaseName = strrchr( szModName, '\\' );
            if ( pszBaseName ) { //We found a path, so advance to the base module name
                pszBaseName++;
            }
        }
    }
}
```

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```

    }
    else {
        pszBaseName = szModName; //No path. Use the same name for both
    }

    //optionally, if module name = "DB.DLL"
    if (strcmp(strupr(pszBaseName), dbdll)==0) {

        //Get the address of the specified exported dynamic-link library
        //(DLL) function
        ProcAdd = GetProcAddress(
            hModule, //handle to DLL module
            "IsDebuggerPresent" //name of function
        );

        //Add offset 0x0C to ProcAddress
        if (NULL != ProcAdd) {
            DebugPatch[0] = (DWORD) ProcAdd + 0x0C;

            //apply the IsDebuggerPresent patch
            ReadProcessMemory(hSaveProcess, (LPVOID) DebugPatch[0], DataRead,
                sizeof(BYTE), &dwRead);

            if(DataRead[0] == scanbytd[0]) {
                WriteProcessMemory(hSaveProcess, (LPVOID) DebugPatch[0],
                    &replbytd[0],
                    sizeof(BYTE), &dwWritten);
                #ifdef _DEBUG
                sprintf( b, "Patch applied at address: %08X (%02X -> %02X)",
                    (LPVOID) DebugPatch[0], (LPVOID) scanbytd[0],
                    (LPVOID) replbytd[0] );
                MessageBox(0, b, "Attention", MB_OK+MB_TASKMODAL);
                #endif
            }
        }
    }

    //////////////////////////////////////
    //If module name = "MY.DLL". Here is where the patch to the DLL is applied.
    //The real core of the loader is all into this if!
    if (strcmp(strupr(pszBaseName), mydll)==0)
    {

        //Apply the patches to the *.exe or *.dll module
        //Calculate number of patches / addresses (not always this formula works,
        //but here it is)
        nPatches = sizeof(AddressOfPatch) / sizeof(AddressOfPatch[0]);

        for ( j = 0; j < nPatches; j++ ) {
            LPVOID CurrentAddress= (LPVOID) (AddressOfPatch[j]);
            ReadProcessMemory(hSaveProcess, CurrentAddress, DataRead,
                sizeof(BYTE), &dwRead);

            if(DataRead[0] == scanbyte[j])
            {
                WriteProcessMemory(hSaveProcess, CurrentAddress, &replbyte[j],
                    sizeof(BYTE), &dwWritten);
                #ifdef _DEBUG
                sprintf( b, "One Patch applied at address: %08X (%02X -> %02X)",
                    CurrentAddress, (LPVOID) scanbyte[j], (LPVOID) replbyte[j] );
                MessageBox(0, b, "Attention", MB_OK+MB_TASKMODAL);
                #endif
            }
        }
    } //end MY.DLL patch! If you need break here to jump to DebugActiveProcessStop!

}

}

//close handle to load dll event
CloseHandle(DebugEv.u.LoadDll.hFile);
} /// end case LOAD_DLL_DEBUG_EVENT
break;

default: {
    contproc = TRUE;
    dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;
}
break;

} //end switch DebugEv.dwDebugEventCode

```

2

3

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
if (!contproc)
    break;

ContinueDebugEvent(DebugEv.dwProcessId, DebugEv.dwThreadId, dwContinueStatus);

} //end if

} //end while

//Close process and thread handles.
CloseHandle( pi.hProcess );
CloseHandle( pi.hThread );

//Detach from debugger/debuggee
sprintf( b, "ProcessId1:%X\nProcessId2:%X", Pid[0], Pid[1]);

MessageBox(NULL, b, "Process Ids", MB_OK+MB_TASKMODAL+MB_ICONASTERISK);
DebugActiveProcessStop(Pid[0]);
DebugActiveProcessStop(Pid[1]);

ExitProcess(0);

return 0;
}

<----- End Code Snippet ----->
```

The most interesting part is the `case LOAD_DLL_DEBUG_EVENT` (the most important points are shown beside the sources). This case is quite simple in its logic and if you follow step-by-step the debug version of this program in your C compiler, you would understand what it does:

1. When the debug event is raised it checks what the loaded DLL is through `EnumProcessModules`,
2. if it's name (without path) matches with the one we want to patch applies the patch using `WriteProcessMemory`.
3. Before writing to memory it does an additional check to see if the original bytes are those we expect to have.

This code section of the program has also a particular attention in case the loaded DLL is `KERNEL32.DLL` (stored in the `dbdll` variable). When this occurs it also patches the `IsDebuggerPresent` API.

We'll try to run the already compiled programs attached to this tutorial (either Debug or Release) to see what happens (see Figure 7).

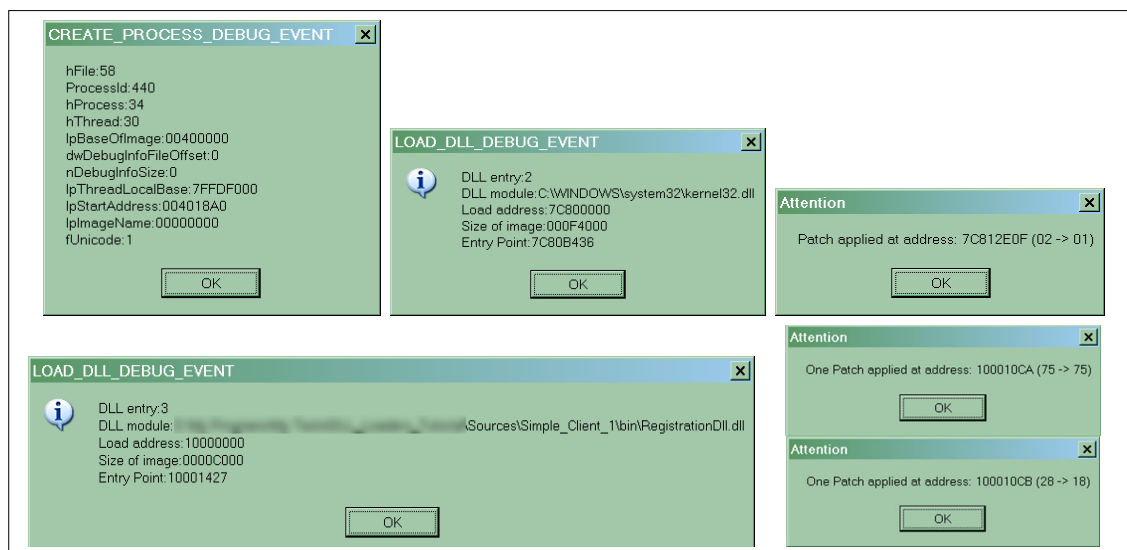


Figure 7 – Part of the sequence of dialogs in the Simple_Client_Loader Debug's build (from left to right).

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

This code, despite working excellently, has a little problem. It's not sensible to Dll relocations in memory. The variable `AddressOfPatch` is used blindly when reading and writing to memory. We obtained that address using OllyDbg, thus the assumption this code does it that the Dll is loaded by the program in exactly the same place where it was loaded into OllyDbg.

We will solve this problem using some math!

As you can see from Figure 7 we have pretty much information to use either for the process or for each Dll. The most important of which are the "Load Address" (This is the `hModule`: the handle of the module is by definition the address where it has been loaded in memory. OllyDbg calls this "Base Address") and the "Size of Image". The "Entry Point" is obtained for the Dll as the address of the `DllMain` method each Dll has.

Well, to accomplish portability, with the meaning told, it's enough to convert absolute patch addresses to offsets relative to the Load Address (`hModule`).

```
DWORD AddressOfPatch[] = {0x10CA, 0x10CB}; //becomes a relative offset
```

And the code where the target Dll is patched (point 2 in the previous code snippet) changes a little:

```
LPVOID CurrentAddress= (LPVOID)((DWORD)hModule + (DWORD)AddressOfPatch[j]);
```

Always remember to convert address to `DWORD` before adding them, otherwise the compiler uses the C arithmetic of pointers, and the final result is something different.

NOTE

The given examples are not stopping when the application exits, but generally speaking if the loader is running on an XP system it can use the `DebugActiveProcessStop` API and terminate just after the patch is done, leaving the application run freely. To enable this insert where you want to stop the instruction `contproc = FALSE`;
Moreover using the NTInternals files I wrote, ensures compatibility with older system, because if the operative system doesn't support this API, it will not do anything.

4.2 Write a debug loader for a protected Dll

It is now clear what can be done taking full control of the application, but what happens when the Dll is protected with some packer? To try it out I protected both the `Client.exe` and the `RegistrationDll.dll` using `ASProtect` (just to try it out, not for commercial purposes). Let see in this case what happens.

NOTE

The code of this program is under the `Simple_Client_Protected_4\` folder

I protected both the client and the Dll with `ASProtect SKE 2.11 build 03.13`, using the settings of Figure 8.

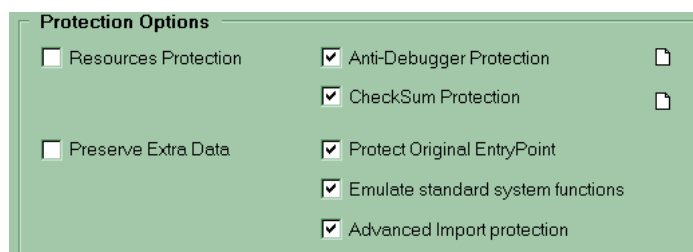


Figure 8 - Settings used to protect Client and Dll with `ASProtect`

Well, before writing a loader doing the same things of Section 4.1, just take time to analyze what ASProtect did to our program; it will also be the occasion to write some considerations on ASProtect SKE 2.11

4.3 Analysis of the protected program

Launch as usual your program into Olly and follow the same approach we did on Section 3: launch the program, skip all the exceptions, insert a fake serial and when the badboy message appears press pause in Olly. Now, inside Olly, take a look at the Calling Stack. You should have something like the following:

Address	Stack	Procedure / arguments	Called from	Frame
0012F118	77D49418	Includes ntdll.KiFastSystemCallRet	USER32.WaitMessage+0A	0012F14C
0012F11C	77D5E2A2	USER32.WaitMessage	USER32.77D5E29D	0012F14C
0012F150	77D561C6	USER32.77D5E113	USER32.77D561C1	0012F14C
0012F173	77D6A92E	USER32.77D6A110	USER32.77D6A929	0012F174
0012F438	77D6A294	USER32.SoftModalMessageBox	USER32.77D6A28F	0012F434
0012F588	77D95F8B	USER32.77D6A11F	USER32.77D95F86	0012F584
0012F5E0	77D96060	USER32.MessageBoxTimeoutW	USER32.MessageBoxTimeoutA+97	0012F5DC
0012F614	77D90577	USER32.MessageBoxTimeoutA	USER32.MessageBoxExA+16	0012F610
0012F634	77D8052F	USER32.MessageBoxExA	USER32.77D8052A	0012F630
0012F638	00000000	hOwner = NULL		
0012F63C	00403020	Text = "Nahh, you have to get a valid serial somehow!"		
0012F640	00403080	Title = "Shub-Nigurrath"		
0012F644	00000000	Style = MB_OK MB_APPLMODAL		
0012F648	00000000	LanguageID = 0 (LANG_NEUTRAL)		

A first consideration: the call stack is extremely different from the not protected application, because there are no elements of it which are inside the client application. Indeed ASProtect transforms all the normal calls to the Windows APIs to Windows Messages, through a dispatching service it added to the program. This makes the program's flow not linear and thus not so easy to follow for complex applications.

Anyway place a breakpoint in the top level call of the stack at 0x77D8052A into User32.dll, where the MessageBox is prepared by the system for display. Let run the application and insert another time the serial. You should land on the BP. Look the Call Stack, is empty!

Do not worry, there's always the Data Stack which helps us. Look the data stack into Olly, you should have something like below:

0012F638	00000000	hOwner = NULL
0012F63C	00403020	Text = "Nahh, you have to get a valid serial somehow!"
0012F640	00403080	Title = "Shub-Nigurrath"
0012F644	00000000	Style = MB_OK MB_APPLMODAL
0012F648	00000000	LanguageID = 0 (LANG_NEUTRAL)
0012F64C	00000000	
0012F650	00C5A5A4	
0012F654	00000000	
0012F658	00403020	ASCII "Nahh, you have to get a valid serial somehow!"
0012F65C	00403080	ASCII "Shub-Nigurrath"
0012F660	00000000	
0012F664	0012F66C	
0012F668	00402300	Client.00402300
0012F66C	0012F804	
0012F670	00000001	
0012F674	73DD24C0	RETURN to mfc42.73DD24C0
0012F678	00402300	Client.00402300
0012F67C	00000111	
0012F680	0012F834	
0012F684	73DD23BF	RETURN to mfc42.73DD23BF from mfc42._AfxDispatchCmdMsg

What we notice immediately is the call to the _AfxDispatchCmdMsg function of the mfc42 Dll. Well this is an internal part of the dispatching mechanism of Windows and it's the door through which we will find where the hell the routine, calling our Dll, is gone.

Press enter into the Data Stack to land to this API and place a breakpoint here:

```
73DD23BA      E8 7F000000      CALL mfc42._AfxDispatchCmdMsg
```

Now let the application again run freely (F9) and insert another time the username and regnumber. Notice that each time you do something in the interface (not just moving the mouse) you stop at the above breakpoint. To make it shorter just enter a single letter username and regnumber and press check (you should have stopped at the BP 3 times then).

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

After having pressed Check you land on the BP, this time we will follow the call to discover where the message is dispatched.

Step by step the function arrives at the place shown below:

73DD2482	75 69	JNZ SHORT mfc42.73DD24ED	
73DD2484	8B45 18	MOV EAX,DWORD PTR SS:[EBP+18]	
73DD2487	FF30	PUSH DWORD PTR DS:[EAX]	
73DD2489	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]	
73DD248C	FF70 04	PUSH DWORD PTR DS:[EAX+4]	
73DD248F	FF55 14	CALL DWORD PTR SS:[EBP+14]	Client.00401590
73DD2492	E9 A8000000	JMP mfc42.73DD253F	
73DD2497	8B45 18	MOV EAX,DWORD PTR SS:[EBP+18]	
73DD249A	FF30	PUSH DWORD PTR DS:[EAX]	
73DD249C	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]	
73DD249F	FF70 04	PUSH DWORD PTR DS:[EAX+4]	
73DD24A2	FF55 14	CALL DWORD PTR SS:[EBP+14]	Client.00401590
73DD24A5	E9 97000000	JMP mfc42.73DD2541	
73DD24AA	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]	
73DD24AD	FF55 14	CALL DWORD PTR SS:[EBP+14]	Client.00401590
73DD24B0	E9 8A000000	JMP mfc42.73DD253F	
73DD24B5	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
73DD24B8	EB 45	JMP SHORT mfc42.73DD24FF	
73DD24BA	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]	
73DD24BD	FF55 14	CALL DWORD PTR SS:[EBP+14]	Client.00401590

The call at 0x73DD24BD is really important, follow it!

You land into a part of the Client code, which should look like below:

00401590	53	DB 53	CHAR 'S'
00401591	8B	DB 8B	
00401592	D9	DB D9	
00401593	55	DB 55	CHAR 'U'
00401594	56	DB 56	CHAR 'V'
.....	..		

Just remove the Analysis and magically we will see the place where things are called:

00401590	53	PUSH EBX	
00401591	8BD9	MOV EBX,ECX	
00401593	55	PUSH EBP	
00401594	56	PUSH ESI	Client.00402300
00401595	8B43 64	MOV EAX,DWORD PTR DS:[EBX+64]	
00401598	8D73 64	LEA ESI,DWORD PTR DS:[EBX+64]	
0040159B	57	PUSH EDI	
0040159C	8BCB	MOV ECX,ESI	Client.00402300
0040159E	8B40 F8	MOV EAX,DWORD PTR DS:[EAX-8]	
004015A1	50	PUSH EAX	
004015A2	E8 AF020000	CALL Client.00401856	JMP to mfc42.#2915
004015A7	8B4B 60	MOV ECX,DWORD PTR DS:[EBX+60]	
004015AA	8D7B 60	LEA EDI,DWORD PTR DS:[EBX+60]	
004015AD	8BE8	MOV EBP,EAX	
004015AF	8B41 F8	MOV EAX,DWORD PTR DS:[ECX-8]	Client.004010F3
004015B2	8BCF	MOV ECX,EDI	
004015B4	50	PUSH EAX	
004015B5	E8 9C020000	CALL Client.00401856	JMP to mfc42.#2915
004015BA	50	PUSH EAX	
004015BB	55	PUSH EBP	
004015BC	E8 7F000000	CALL Client.00401640	JMP to registra.CheckRegistrationNumber
004015C1	83C4 08	ADD ESP,8	
004015C4	8BCB	MOV ECX,ESI	Client.00402300
004015C6	8BE8	MOV EBP,EAX	
004015C8	6A FF	PUSH -1	
004015CA	E8 81020000	CALL Client.00401850	JMP to mfc42.#5572
004015CF	6A FF	PUSH -1	
004015D1	8BCF	MOV ECX,EDI	
004015D3	E8 78020000	CALL Client.00401850	JMP to mfc42.#5572
004015D8	83FD 01	CMPP EBP,1	
004015DB	6A 00	PUSH 0	
004015DD	68 80304000	PUSH Client.00403080	ASCII "Shub-Nigurrath"
004015E2	75 07	JNZ SHORT Client.004015EB	
004015E4	68 50304000	PUSH Client.00403050	ASCII "Compliments, well done the serial is correct"
004015E9	EB 05	JMP SHORT Client.004015F0	
004015EB	68 20304000	PUSH Client.00403020	ASCII "Nahh, you have to get a valid serial somehow!"
004015F0			

Figure 9 - Check routine of the protected Client

You should have recognized the place where we are now. Compare Figure 3 and Figure 9, something has changed, but leave all these considerations for another tutorial.

The most important thing is that we found again the checking routine (I could have done it faster, but I preferred to follow this way to better illustrate some concepts).

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
004015BC      E8      7F000000      CALL     Client.00401640      ;      JMP      to
registra.CheckRegistrationNumber
```

After this point the story is exactly the same, the address where to patch is exactly the same as before!

4.4 Writing the loader for the protected program

Try to run the loader we just created with the protected program and see what happens. There are two news, the first is good and the second is bad:

- The good news is that ASProtect doesn't complain about being debugged by the loader. We wrote the HideDebugger properly.
- The bad news is that the loader won't patch our Dll. Why?

Consider that the patch is applied when the event `LOAD_DLL_DEBUG_EVENT` is raised, but if the Dll is protected at that time it has only been loaded in memory and not yet unpacked. So we are patching too early and we will need to change the patch condition.

NOTE

The code of this loader is under the `Simple_Client_Loader_Protected_5\` folder
The protected version of the demo Client is under the folder `Simple_Client_Protected_4\`

The way I chose, (but it's only to do something different), is to use the Hardware Breakpoints and place a Hardware Breakpoint on write, at the memory location of one of the patches (just one, the first for example). Other valid approaches are those described in [5, 6].

Then when the loader reaches the breakpoint it applies the patches.

NOTE

Please refer to [3] to understand Debug Registers, Hardware Breakpoint vs Software Breakpoints Then look [4] to understand the meaning of the bit fields of DR7 (chapter 15.2.4 specifically).

Anyway here I wrote the essential things.

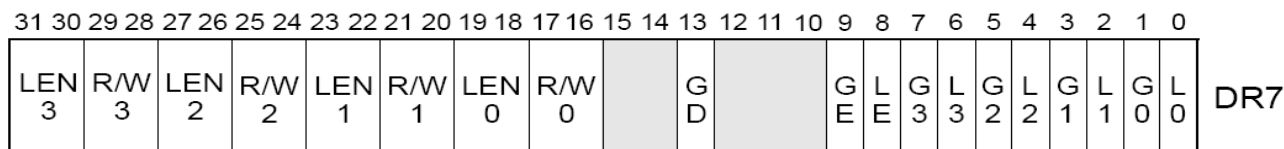


Figure 10 - Bytes position for register DR7

Figure 10 show the bytes meaning of DR7 register [4].

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions. The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6)** — Enable (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated L_n flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7)** — Enable (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

detected and its associated Gn flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.

- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29)** — Specifies the breakpoint condition for the corresponding breakpoint. For the Intel386™ and Intel486™ processors, bits are interpreted as follows:
 - 00 — Break on instruction execution only.
 - 01 — Break on data writes only.
 - 10 — Undefined.
 - 11 — Break on data reads or writes but not instruction fetches.
- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31)** — Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:
 - 00 — 1-byte length
 - 01 — 2-byte length
 - 10 — Undefined (or 8 byte length, see note below)
 - 11 — 4-byte length

So, according to Intel documentation if we want a breakpoint with these characteristics:

- a break which location is stored in DR0,
- a local break,
- which breaks on execution,
- for which the corresponding DRn is a byte.

We have to set DR7 bits as following:

Bit number	0	1	16, 17	18, 19
Field Name	L0	G0	R/W0	LEN0
Value	1	0	01	00

The final DR7 value will be: 00100000000000000001 binary → **0x20001 hex**

The code of Point 2 of the code snippet of Section 4.1 then becomes the following:

```
<----- Start Code Snippet ----->

////////////////////////////////////
//If module name = "MY.DLL". Here is where the patch to the DLL is applied.
//The real core of the loader is all into this if!
if (strcmp(strupr(pszBaseName), mydll)==0)
{

    //Apply the patches to the *.exe or *.dll module
    //Calculate number of patches / addresses (not always this thing works, but here it is)
    DWORD FirstPatchAddress= (DWORD)hModule + (DWORD)AddressOfPatch[0];

    //This is requires because when we will apply the patch the hModule will be different
    lpBaseofDll=hModule;

    _CONTEXT processContext;
    //CONTEXT_DEBUG_REGISTERS must be explicitly added to the code.
    processContext.ContextFlags = CONTEXT_FULL | CONTEXT_FLOATING_POINT | CONTEXT_DEBUG_REGISTERS;

    GetThreadContext(pi.hThread, &processContext);

    try {
        throw 1;
    }
    catch(int) {
        processContext.Dr0=FirstPatchAddress;
        processContext.Dr7=0;

        //According to Intel documentation I use these values:
        //| bits # | 0 | 1 | 16, 17 | 18, 19 |
        //| field name | L0 | G0 | RW0 | LEN0 |
        //| value | 1 | 0 | 01 | 00 |
        //
        //This way DR7 is:
```

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
// - a break which location is stored in DR0,  
// - a local break,  
// - which breaks on execution,  
// -for which the corresponding DRn is a byte.  
//Value: 001000000000000000000001b -> 0x20001  
  
processContext.Dr7 = (DWORD)0x20001;  
SetThreadContext(pi.hThread, &processContext);  
  
}  
  
} //end MY.DLL patch!  
  
<----- End Code Snippet ----->
```

When a Hardware Breakpoint is reached by the program the CPU raises an EXCEPTION_SINGLE_STEP and just after an EXCEPTION_BREAKPOINT exception. Thus we need to add two cases to the switch contained into the case EXCEPTION_DEBUG_EVENT.

These two switches will handle our breakpoint properly

The new parts of the code become:

```
<----- Start Code Snippet ----->  
  
while (TRUE) {  
    // Wait for a debugging event to occur. The second parameter indicates  
    // that the function does not return until a debugging event occurs.  
  
    if (WaitForDebugEvent(&DebugEv, 1000)) { // wait 1 second  
        // Process the debugging event code.  
  
        switch (DebugEv.dwDebugEventCode) {  
  
            case EXCEPTION_DEBUG_EVENT: {  
                // Process the exception code. When handling  
                // exceptions, remember to set the continuation  
                // status parameter (dwContinueStatus). This value  
                // is used by the ContinueDebugEvent function.  
                switch (DebugEv.u.Exception.ExceptionRecord.ExceptionCode) {  
  
                    1 case EXCEPTION_SINGLE_STEP: {  
                        // First chance: Update the display of the  
                        // current instruction and register values.  
  
                        if (DebugEv.u.Exception.dwFirstChance) {  
                            contproc = TRUE;  
                            dwContinueStatus = DBG_CONTINUE;  
                        }  
                        else {  
                            contproc = FALSE;  
                        }  
                    }  
  
                    sprintf( b, "Exception Code: %08X\nException address: %08X",  
                        DebugEv.u.Exception.ExceptionRecord.ExceptionCode,  
                        DebugEv.u.Exception.ExceptionRecord.ExceptionAddress);  
                    #ifdef _DEBUG  
                        ::MessageBox(NULL, b, "EXCEPTION_SINGLE_STEP", MB_OK+MB_TASKMODAL+MB_ICONWARNING);  
                    #endif  
                }  
                break;  
  
                    2 case EXCEPTION_BREAKPOINT: {  
                        // First chance: Display the current  
                        // instruction and register values.  
                        l++;  
  
                        if (DebugEv.u.Exception.dwFirstChance) {  
                            contproc = TRUE;  
                            //l == # of bp's note: (ntdll has debugbreak X 2 open processes)  
                            //note: if ONLY 1 process and want to avoid INT3 debugger trick,  
                            //change to (l > 1) below:  
                            if (l > 2) {  
                                dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
        else {
            dwContinueStatus = DBG_CONTINUE;
        }
    }
    else {
        contproc = FALSE;
    }
}

sprintf( b, "Exception Code: %08X\nException address: %08X",
    DebugEv.u.Exception.ExceptionRecord.ExceptionCode,
    DebugEv.u.Exception.ExceptionRecord.ExceptionAddress);
#ifdef _DEBUG
::MessageBox(NULL, b, "EXCEPTION_BREAKPOINT", MB_OK+MB_TASKMODAL+MB_ICONWARNING);
#endif

//////////////////////////////////////////////////
if(lpBaseofDll!=NULL) {
    //Apply the patches to the *.exe or *.dll module
    //Calculate number of patches / addresses (not always this thing works,
    //but here it is)
    nPatches = sizeof(AddressOfPatch) / sizeof(AddressOfPatch[0]);

    DWORD AppliedPatches=0;
    for ( j = 0; j < nPatches; j++ ) {
        //MODULEINFO.lpBaseOfDll is exactly the same meaning of hModule.
        LPVOID CurrentAddress= (LPVOID)((DWORD)lpBaseofDll + (DWORD)AddressOfPatch[j]);
        ReadProcessMemory(hSaveProcess, CurrentAddress, DataRead,
            sizeof(BYTE), &dwRead);

        if(DataRead[0] == scanbyte[j])
        {
            WriteProcessMemory (hSaveProcess, CurrentAddress, &replbyte[j],
                sizeof(BYTE), &dwWritten);
#ifdef _DEBUG
            sprintf( b, "One Patch applied at address: %08X (%02X -> %02X)",
                CurrentAddress, (LPVOID) scanbyte[j], (LPVOID) replbyte[j] );
            ::MessageBox(0, b, "Attention", MB_OK+MB_TASKMODAL);
#endif
            AppliedPatches++;
        }
    }

    //Have we successfully patched all the things? If yes clean what we did before.
    if(AppliedPatches==nPatches) {
        //To avoid repeating the patch this variable must be set to NULL again.
        lpBaseofDll=NULL;

        //Remove the breakpoint from target's context.
        _CONTEXT context;
        context.ContextFlags = CONTEXT_FULL | CONTEXT_FLOATING_POINT |
            CONTEXT_DEBUG_REGISTERS;
        GetThreadContext(pi.hThread, &context);
        context.Dr0=0;
        context.Dr7=0;
        SetThreadContext(pi.hThread, &context);
    }
}

break;
```

<....>

<----- End Code Snippet ----->

For the complete source code see the folder Simple_Client_Loader_Protected_5\ into the sources accompanying this tutorial.

Some comments on few code points:

1. When an HW Breakpoint is reached the CPU raises a EXCEPTION_SINGLE_STEP exception and falls there. We do nothing special here.
2. After having the processor raises an EXCEPTION_BREAKPOINT this is the place where to do patches.
3. lpBaseofDll was the HMODULE of RegistrationDll.dll we stored before, when we set the HW Breakpoint, this value is used when the loader stops at the breakpoint to correctly find where to patch.
4. Once all the patches are done we can delete the unused variables and remove the HW Breakpoint.

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

Note that this approach might generally not work for protectors checking the presence of HW breakpoints, in those cases you will have to follow different approaches..

Remember that you have full control of the code and of the debug events/exceptions the program arises; so generally speaking I suggest this blind procedure to rapidly find when to place the patching loop.

1. Compile the loader in debug mode, thus with all the messageboxes appearing. I suggest using the complete debugger cycle I provided in **Complete_Client_Loader_3** which traps all supported exceptions, just to not miss any. If required also trap custom exceptions (the code reports some examples).
2. Run the loader beside a memory inspection tool running into another task (such for example WinHex or MemoryHacker from L.Spiro).
3. Take a look at the memory address you want to patch and when a specific exception occurs check the memory with the other tool and see if the values you expect are already in place.
4. If they are, write the source code that applies patches in the "case" that handles the last exception.

This is a *blind process* as I told, because doesn't suppose you to know what the packer does, it simply checks memory if the expected value is already there.

So, why the loader cannot do the same things directly? Well, the answer is that doing this way the solution is more general and exclude the situation where the searched bytes at a given address would never appear (i.e. because the target version is changed).

5 Write a Dll Proxy

As told in Section 2 the other possible solution to solve our problem on DLLs is to write a Dll Proxy. Well the solution is less powerful than mini-debugger approaches (debugger loaders) but have some advantages: is simple to code, it's faster to execute.

An easy way for hacking APIs is just to replace a DLL with one that has the same name and exports all the symbols of the original one. This technique can be effortlessly implemented using function forwarders. A function forwarder basically is an entry in the DLL's export section that delegates a function call to another DLL's function.

As described in [7, 8] we will use the so called "Export Forwarding".

5.1 Export Forwarding

A particularly slick feature of exports is the ability to "forward" an export to another DLL. For example, in Windows NT®, Windows® 2000, and Windows XP, the KERNEL32 HeapAlloc function is forwarded to the RtlAllocHeap function exported by NTDLL. Forwarding is performed at link time by a special syntax in the EXPORTS section of the .DEF file. Using HeapAlloc as an example, KERNEL32's DEF file would contain:

```
EXPORTS
...
HeapAlloc = NTDLL.RtlAllocHeap
```

How can you tell if a function is forwarded rather than exported normally? It's somewhat tricky. Normally, the EAT contains the RVA of the exported symbol. However, if the function's RVA is inside the exports section (as given by the VirtualAddress and Size fields in the DataDirectory), the symbol is forwarded.

When a symbol is forwarded, its RVA obviously can't be a code or data address in the current module. Instead, the RVA points to an ASCII string of the DLL and symbol name to which it is forwarded. In the prior example, it would be NTDLL.RtlAllocHeap.

Another way you can accomplish this task is using #pragma comment

```
#pragma comment(linker, "/export:DoSomething=DllImpl.ActuallyDoSomething")
```

However, remember that if you decide to employ this method, you should take the responsibility of providing compatibilities with newer versions of the original library. For more details see [8] section "Export forwarding" and [9] "Function Forwarders" (Chapter 20).

5.2 Writing first proxy Dll and introducing the code structure

What I want to do strongly depends on what you already know of the target Dll. Let consider two cases:

- *The Dll is a plug-in of another program.* In this case most of the times its interface methods are well known, and also their prototypes,
- *The Dll is an undocumented Dll.* This is the situation with most programs; the Dll exports are unknown or just known by their name.

In any case what we have to write is a normal Dll which loader the original Dll internally and forward the program's requests to this local instance of the Dll.

So these are the actions:

1. the proxy loads an internal copy of the original Dll
2. patch it in its own memory space or somehow elaborates answer coming from the Dll (for example inverting Booleans),
3. send answers to the client program

Remember that the client has loaded the proxy Dll instead of the original Dll.

NOTE

The code of this part of the document is under the Proxy_Dll_6\ folder, also with a local copy of the client and of the RegistrationDll.dll, renamed to _RegistrationDll.dll

The complete code is shown below..

```
<----- Start Code Snippet ----->

#include "stdafx.h"

//Forward declaration of Prototypes
BOOL PerformPatch();

/////////////////////////////////////////////////////////////////
// Function forwarders to functions in DllWork
#pragma comment(linker, "/export:CheckRegistrationNumber=_RegistrationDll.CheckRegistrationNumber")

//Other things to do in order to create a new project:
//1. add to the linker's path the path where to find the forwarded dll'
//2. add to not use the precompiled headers
//3. add the /force commandline to the linker, to ignore stupid missing include files errors.
// These linker errors will not influence the final output
/////////////////////////////////////////////////////////////////

//Module of the loaded Library, it's a global variable.
HMODULE hDllMod=0;
char b[1024]; //messages buffer

/////////////////////////////////////////////////////////////////
//MAIN PROGRAM PATCH INFO:
//Patch Address info: # elements in following arrays must be synchronized for Address/scan/replace
DWORD AddressOfPatch[] = {0x10CA, 0x10CB};

//Patch byte info:
//Search (read) byte. Original bytes read from the dll in memory (attn: # elements must be the same of AddressOfPatch)
BYTE scanbyte[] = {0x75, 0x28};
//Found (write) byte. New patch bytes to be written in memory (attn: # elements must be the same of AddressOfPatch)
BYTE replbyte[] = {0x75, 0x18};
```



WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
char szDllName[]=".\\_RegistrationDll.dll";
////////////////////////////////////

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);

    switch(ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        {
            hDllMod=::LoadLibrary((LPCSTR)szDllName);
            if(hDllMod==NULL) {
                //Find the last '\\' to obtain a pointer to just the base module name part
                //(i.e. mydll.dll w/o the path)
                PSTR pszBaseName = strrchr( szDllName, '\\' );
                if ( pszBaseName ) { //We found a path, so advance to the base module name
                    pszBaseName++;
                }
                else {
                    pszBaseName = szDllName; //No path. Use the same name for both
                }

                sprintf(b, "%s not found.\r\nHave you renamed it as _%s\r\n Is this dll into the same path?",
                    pszBaseName, (pszBaseName+1));
                ::MessageBox(NULL, b, "Load Library Failed", MB_OK+MB_TASKMODAL+MB_ICONERROR);

                return TRUE;
            }

            PerformPatch();

        }
        break;

        case DLL_PROCESS_DETACH:
        {
            if(hDllMod!=NULL) {
                ::FreeLibrary(hDllMod);
                hDllMod=NULL;
            }
        }
        break;

        case DLL_THREAD_ATTACH: {} break; //not used at the moment
        case DLL_THREAD_DETACH: {} break; //not used at the moment
    }
    return TRUE;
}

BOOL PerformPatch() {

    int i=0, j=0;
    int nPatches=0;
    DWORD dwRead=0;
    DWORD dwWritten=0;
    BYTE DataRead[] = {0};
    int AppliedPatches=0;

    //////////////////////////////////////
    //Apply the patches to the *.exe or *.dll module
    //Calculate number of patches / addresses (not always this thing works, but here it is)
    nPatches = sizeof(AddressOfPatch) / sizeof(AddressOfPatch[0]);

    for ( j = 0; j < nPatches; j++ ) {
        LPVOID CurrentAddress= (LPVOID)((DWORD)hDllMod + (DWORD)AddressOfPatch[j]);

        //Pay attention that you have to arrive to the patch address through the CurrentProcess HANDLE
        //and not through the hDllmod, otherwise access to memory will be denied.
        //The GetCurrentProcess API returns the HANDLE of the process owning the program.
        ReadProcessMemory(GetCurrentProcess(), CurrentAddress, DataRead, sizeof(BYTE), &dwRead);
        if(DataRead[0] == scanbyte[j])
        {
            WriteProcessMemory (GetCurrentProcess(), CurrentAddress, &replbyte[j], sizeof(BYTE), &dwWritten);
            #ifdef _DEBUG
                //Debug output
            #endif
        }
    }
}
```



WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
    sprintf ( b, "One Patch applied at address: %08X (%02X -> %02X)",
              CurrentAddress, (LPVOID) scanbyte[j], (LPVOID) replbyte[j] );
    MessageBox(0, b, "Attention", MB_OK+MB_TASKMODAL);
    #endif
    AppliedPatches++;
}

//Have we successfully patched all the things? If yes clean things and return ok.
if(AppliedPatches==nPatches)
    return TRUE;

return FALSE;
}

<----- End Code Snippet ----->
```

First of all you should note the easiness of this second sources compared to the debug loader. Indeed we are not debugging anything indeed, because we already have an anchor (a system event) where to insert our code: the DllMain DLL_PROCESS_ATTACH is a proper place where to insert the patching process.

There are 3 points for which I would spend few words, before you study the whole code.

1. Note the pragma directive syntax. Beside it you should add to the linker's path where the Dll is located.
2. The real library is loaded by the DllMain when the event DLL_PROCESS_ATTACH is raised. This is also the place where the patches can be done.
3. This time you have to give to the Read/WriteProcessMemory APIs the handle of the process owning the Dll, otherwise the access to memory will fail. This handle is got trough the GetCurrentProcess API.

5.3 Write a proxy for a protected Dll

As did in Section 4.2 we will now worry to write a proxy Dll patching the same Dll protected with ASProtect.

NOTE

The code of this part of the document is under the Proxy_Dll_Protected_7\ folder, also with a local copy of the **protected** client and of the RegistrationDll.dll, renamed to _RegistrationDll.dll

The loader of Section 5.2 doesn't work for this Dll, for exactly the same reasons we had in Section 4.2: at the time the DLL_PROCESS_ATTACH event is issued the Dll is still not uncompressed in memory and the patch would be overwritten by the unpacking procedure of ASProtect. As it will be clearer in a few, we are facing to the point where this technique is less powerful: using this approach we will not have anything can be used to intercept the decryption of the Dll in memory. Using Debug Loaders we used the exceptions: the more exceptions there were in the unpacking process the easier were to find a place where to patch the code.

We can anyway do something. The hypothesis is that we are able to understand at least one prototype of an exported method of the Dll, which is called before the code to be patched is executed. We can use OllyDbg and reverse it using the Call Stack or, as said before, taking it from the plug-in documentation of the main application to which the target Dll applies.

The important assumption is that we are able to understand the prototype of one of the exports of the original Dll. We will implement locally in the proxy Dll this method and forward all the others as before. Note also that export forwarding doesn't require the forwarder Dll to know the prototype of the forwarded function.

Let see just now which the resulting code is and later discuss it a little

WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

<----- Start Code Snippet ----->

```
#include "stdafx.h"

//Forward declaration of Prototypes
BOOL PerformPatch();

//Module of the loaded Library, it's a global variable.
HMODULE hDllMod=0;
char b[1024]; //messages buffer

////////////////////////////////////
//MAIN PROGRAM PATCH INFO:
//Patch Address info: # elements in following arrays must be synchronized for Address/scan/replace
DWORD AddressOfPatch[] = {0x10CA, 0x10CB};

//Patch byte info:
//Search (read) byte. Original bytes read from the dll in memory (attn: # elements must be the same of AddressOfPatch)
BYTE scanbyte[] = {0x75, 0x28};
//Found (write) byte. New patch bytes to be written in memory (attn: # elements must be the same of AddressOfPatch)
BYTE replbyte[] = {0x75, 0x18};

char szDllName[] = ".\\_RegistrationDll.dll";
////////////////////////////////////

BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);

    switch(ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        {
            #ifdef _DEBUG
                //Needed to be able to debug the program from the compiler.
                HideDebugger(GetCurrentThread(), GetCurrentProcess());
            #endif

            hDllMod = ::LoadLibrary( (LPCSTR)szDllName);
            if(hDllMod==NULL) {
                //Find the last '\\' to obtain a pointer to just the base module name part
                //(i.e. mydll.dll w/o the path)
                PSTR pszBaseName = strrchr( szDllName, '\\');
                if ( pszBaseName ) { //We found a path, so advance to the base module name
                    pszBaseName++;
                }
                else {
                    pszBaseName = szDllName; //No path. Use the same name for both
                }

                sprintf(b, "%s not found.\r\nHave you renamed it as _%s\r\n Is this dll into the same path?",
                    pszBaseName, (pszBaseName+1));
                ::MessageBox(NULL, b, "Load Library Failed", MB_OK+MB_TASKMODAL+MB_ICONERROR);

                return TRUE;
            }
        }
        break;

        case DLL_PROCESS_DETACH:
        {
            if(hDllMod!=NULL) {
                ::FreeLibrary(hDllMod);
                hDllMod=NULL;
            }
        }
        break;

        case DLL_THREAD_ATTACH: {} break; //not used at the moment
        case DLL_THREAD_DETACH: {} break; //not used at the moment
    }
    return TRUE;
}

BOOL PerformPatch() {
```



WRITING LOADERS FOR DLLS: THEORY AND TECHNIQUES

```
int i=0, j=0;
int nPatches=0;
DWORD dwRead=0;
DWORD dwWritten=0;
BYTE DataRead[] = {0};
int AppliedPatches=0;

////////////////////////////////////
//Apply the patches to the *.exe or *.dll module
//Calculate number of patches / addresses (not always this thing works, but here it is)
nPatches = sizeof(AddressOfPatch) / sizeof(AddressOfPatch[0]);

for ( j = 0; j < nPatches; j++ ) {
    LPVOID CurrentAddress= (LPVOID) ((DWORD)hDllMod + (DWORD)AddressOfPatch[j]);

    //Pay attention that you have to arrive to the patch address through the CurrentProcess
    //HANDLE and not through the hDllmod, otherwise access to memory will be denied.
    //The GetCurrentProcess API returns the HANDLE of the process owning the program.
    ReadProcessMemory(GetCurrentProcess(), CurrentAddress, DataRead, sizeof(BYTE), &dwRead);
    if(DataRead[0] == scanbyte[j])
    {
        WriteProcessMemory (GetCurrentProcess(), CurrentAddress, &replbyte[j], sizeof(BYTE), &dwWritten);
#ifdef _DEBUG
        sprintf ( b, "One Patch applied at address: %08X (%02X -> %02X)",
            CurrentAddress, (LPVOID) scanbyte[j], (LPVOID) replbyte[j] );
        MessageBox(0, b, "Attention", MB_OK+MB_TASKMODAL);
#endif
        AppliedPatches++;
    }
}

//Have we successfully patched all the things? If yes clean what we did before.
if(AppliedPatches==nPatches)
    return TRUE;

return FALSE;
}

BOOL CheckRegistrationNumber(char *a, char* b) {

    //Performs the patch
    PerformPatch();

    //Call the original method and returns to the caller the result.
    BOOL (*fp)(char *, char*);
    fp = (BOOL (*)(char *, char*)) GetProcAddress(hDllMod, "CheckRegistrationNumber");

    if(fp!=NULL)
        return fp(a,b);
    else
        return FALSE;
}

<----- End Code Snippet ----->
```



As you can see (point 1) I eliminated the forwarder #pragma line used before in example of Section 5.2 (the method forwarded from the proxy Dll to the original Dll). Please note that I eliminated only the one I will explicitly implement in my Proxy Dll here and not all the others: if there were other exports I would have left them forwarded.

I then added a method (point 3) which simply performs the patch and then call the original method from the original (now patched) Dll. The whole thing is done through function pointers and using the GetProcAddress Win32 API.

A note at point 2: I added a call to HideDebugger only in debug mode to be able to debug the code of the proxy Dll, without ASProtect complaining. In Release mode there's no need of this patch, because the Proxy Dll is not using any debugging function.

6 Comparison of the two presented methods

Let summarize the main differences between the two methods.

- Take Control of the whole application. This allows performing the proper actions when the target DLL is loading into the application's memory space. This approach is the most simple to think but requires the most complex code to write and involves the Debug Loaders. Moreover for very large applications the final result might slow down too much the program or require too much memory: you are in this case debugging the whole application.
- Write a DLL proxy. With this approach you will write a proxy DLL, a DLL written by you exposing exactly the same interface of the original DLL to the original application. The application then loads the false DLL and invokes its methods as it would have done with the real DLL. The proxy DLL then performs the required actions (patches) and calls the original DLL passing the parameters coming from the application or just forwards its exports. This approach requires a less complex code but also some tricks. The advantage is that you will not debug the whole application. In case of protected applications there are some limitations which are bypassed only knowing at least one prototype of an export of the original DLL, called before the code to be patched code is executed.
An alternative approach is using API Hooks: if you consider using API Hooking you will have to hook a System API called before the code to be patched and then patch from there the victim DLL.
This approach is also commonly used by trojans to gather information from the system

Of course as always a best approach doesn't exist, just evaluate and possibly mix the two I proposed here.

7 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.

8 References

- [1] "Cracking with Loaders: Theory, General Approach and a Framework, Version 1.2", Shub-Nigurrath, ThunderPwr, <http://tutorials.accessroot.com> or on Code-Breakers Journal Vol.2 No.2
- [2] "Guide on How to play with processes memory, write loaders and Oraculums", Shub-Nigurrath, <http://tutorials.accessroot.com> or on Code-Breakers Journal Vol. 2 No.2
- [3] "Beginner Olly Tutorial #8, Breakpoints Theory", Version 4,1, Shub-Nigurrath, <http://tutorials.accessroot.com>
- [4] "IA-32 Intel Architecture Software Developer's Manual, Volume 3", Section "Debug Registers", Chapter 15, <http://developer.intel.com/design/pentium4/manuals/253668.htm>
- [5] "Writing Loader 2 Patch Apps Protected With Asprotect 2.0 V10", Shub-Nigurrath, Thunderpwr, <http://tutorials.accessroot.com>
- [6] "Writing Loader 2 Patch Apps Protected With Asprotect 1.2x And Earlier V10", Shub-Nigurrath, Thunderpwr, <http://tutorials.accessroot.com>
- [7] "An In-Depth Look into the Win32 Portable Executable File Format, Part 1", Matt Pietrek, <http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx>
- [8] "An In-Depth Look into the Win32 Portable Executable File Format, Part 2", Matt Pietrek, <http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx>
- [9] "Programming Application for MS Windows" , Jeffrey Richter, Microsoft Press

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.