CodeBreakers Magazine
Security & Anti-Security - Attack & Defense
Volume 1, Issue 1, 2006


# Cracking with Loaders: Theory, General Approach, and a Framework

Shub-Nigurrath [ARTeam], ThunderPwr [ARTeam]
January 2006

**Abstract**
*This tutorial aim is to describe the work we did on loaders, to introduce you to the problem and to describe two different approaches to write loaders. We'll also present a framework we used for several many patches which worked goodwill and that you can re-use as you like. This paper reading requires a little of knowledge of the C/C++ programming language.*

# Table of Contents



# 1 Introduction

This tutorial aim is to describe the work we did on loaders, introduce you to the problem and to describe two different approaches to write loaders. We'll also present a framework we used for several patches which worked well and that you can re-use as you like.

This paper reading requires a little of knowledge of the C/C++ programming language, all the code which we reported into the following chapters had been written in C (and tested using Visual C++ 6.0 with Console Application project type).
We also release with this tutorial a framework written in C++, which can be used to more rapidly write generic and complex loaders for applications. We didn't want to release a library, just because to write loaders at least you should be able to understand a little C or C++, so do you homework also..

As a practical examples we will also present an approach to VB applications serial sniffing through loaders, beside some notes about VB cracking magically performed without using the remote thread technique or DLL injection like in [2], [3] [4] and [5].

If you already know how to code a loader on your own you can skip to section 4. If you don't know how to write a debugger loader start skip to section 3.3, otherwise relax, take your time and read it all, it's a long story to tell.

# 2 What's a loader?

For all of you which do not know anything about loaders and how a program is loaded into memory we suggest reading [1] and [8] to better understand the rest of this tutorial; this paper will only cover few of the base concepts, because readers should already know them.

## *2.1 Loader classification and behaviour*

A loader is a program able to load in memory and running another program. Every time you start a program the standard window loader make this work for you in invisible way. There is many type of loader but basically every loader can be classified in two classes:

- Standard loader
- Debugger loader

### 2.1.1 Standard Loaders

Standard loader is able to create a process in memory from a target which is into the disk, then a standard loader must be able to handle CreateProcess API function and then use ReadProcessMemory, WriteProcessMemory to read/write the memory space of the process and also run or stop the process by using SuspendThread and ResumeThread API function. Other useful API function is related to the process context. More generally the context of a process reflect the state of the process itself in every instruction cycle, imagine to stop the process and look at the registers value (then EAX, EBX, ECX and so on) all the registers and flag examined at the same time keep the process context and all the values is stored into a CONTEXT structure then a CONTEXT structure contains processor-specific register data, the system uses CONTEXT structures to perform various internal operations (refer to the header file WinNT.h for definitions of this structure for each processor architecture). Figure 1 reports what we have just described as a flowchart.

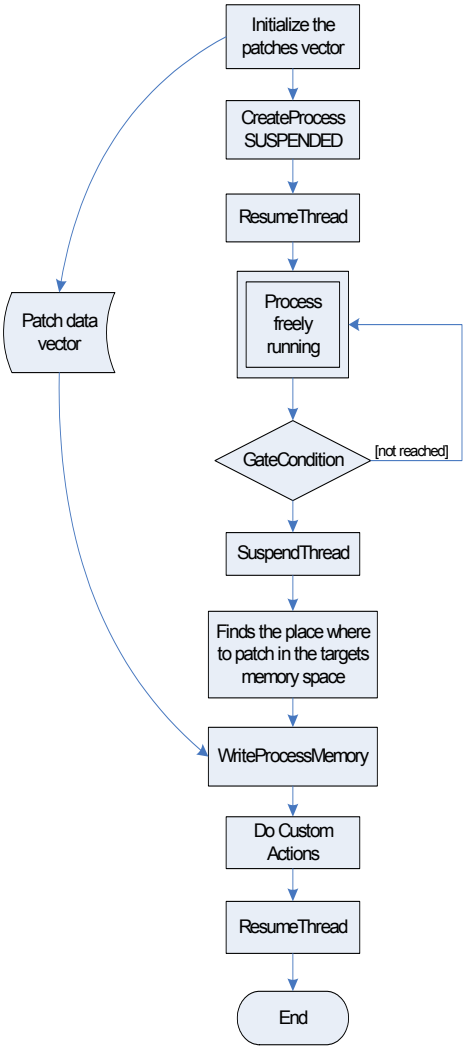



*Figure 1 - Generic simple loader's flowchart*

The Loader launches the program as suspended, or generically speaking in a controlled mode, then the GateCondition checks if the target reaches a wake-up condition (e.g. the display of a nag, a specific pattern is present in the target's memory, or a specific window has been created) then writes to the target's memory the patches we want to do (previously identified with Olly for example) and perform some custom actions (or example read/write the Context), the resume the thread.
Obviously this schema is simple and doesn't keep in consideration cases such as multi-threaded applications, but in these cases the actions changes a little bit, but the overall concept remains.

Figure 2 reports the same as UML sequence diagram for those of you who's able to understand it (very simple anyway).

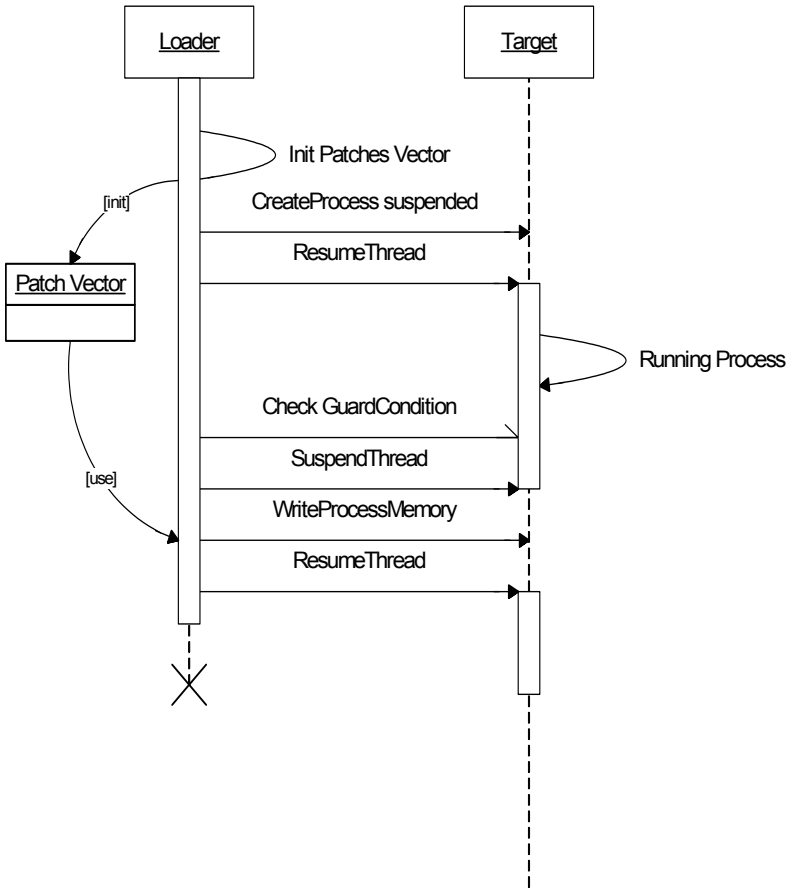



*Figure 2 Sequence Diagram of a simple loader*

### 2.1.2 Debugger Loader

Debugger loaders have basically the same feature of the standard loader and can also debug the target process by using some specialized API function, debugging can be from a process which has to run or from a running process by using the debugger attach feature.

The debugging functions can be used to create a basic, event-driven debugger. *Event-driven* means that the debugger is notified every time certain events occur in the process being debugged. Notification enables the debugger to take appropriate action in response to the events (for example exception which is generated from the target) then you can wait without do anything until some events occur and then take action or just pass the event handling to the target itself. Essentially the main body of such program is a big switch-case construct which have in its "case" the handled events. It's the Operative System debugging environment that worries to send the debugger events to the registered debugger for that process. As a matter of facts an important step of such a loader is to register the loader as a debugger of the target process. This can be easily done through Windows APIs of course (anticipating there's a special switch of the CreateProcess's API).

Generally speaking then independently of the type of loader you'll choose you can gain the process control and then make some changes into its memory, the basic question when use standard loader instead of debugger loader, is strictly dependant from the target and related to the task which we have to do.

Essentially a standard loader is able to interact with the program without using the system's debug APIs while a debug loader works more or less like a ring3 debugger, like OllyDbg, intercepting debug events and interacting the program this way. The choice among the two approaches is completely application's dependant. Of course in the case you will choose to use a debugger's loader you will also have to hide the loader to the application, more or less like you are normally doing using OllyDbg. There is a simple way to hide that a program is being debugged and we will use this approach before doing anything with the loader.
Generally speaking the usefulness of these two types of loaders is the same.

# 3 Write your first loader

For the first example we want to focus on an application protected with Asprotect ealier than 2.0. The application itself it is not important, because the only thing that's tied to the application is the patches vector. For this particular application the standard loaders or the debugger loaders are both fine, so we'll write them both. You'll be able to understand the code at its simplest level.

| **NOTE** |
| --- |
| Writing loaders for AsProtect with different versions will be argument of some following tutorials, for specific real applications. See as usual http://tutorials.accessroot.com for details. |

## *3.1 Patches Vector*

First of all we need to create a proper C structure to store the patches. Generally speaking what we need are: the original byte, the patched byte, the offset. The original byte is required because we want to add a little control before writing a patch into the victim.

| **NOTE** |
| --- |
| **Blind Loaders** are those loaders which are not doing these additional checks! It is important to add these checks (e.g. also the CRC check of the target) to be sure to patch the correct target's version. |

In the example below we used a C++ class called Patch, but a C structure would have worked fine as well. Even 3 simple vectors of BYTES or DWORD for offset, original bytes, patched bytes would have done the work. The concept here is to build up the data-structures properly so as to write a simpler code after.

```
<-------------Code Snippet----------------->
 // A little class (C++) which is useful to store the single patch data. It's a facility
// to use a C++ class, but any other structure is also usable, depending on your knowledge.
class Patch {
public:
        Patch() {orig=address=patch=0;}
        Patch(DWORD dw, BYTE bt) { orig=0; address=dw; patch=bt; }
        DWORD address;
        BYTE patch;
        BYTE orig;
};

// The patch vector is made of Patch objects (there are 15 patches for this specific example).
Patch crk[15];

// Fill in the patch vector with the values we want to patch.
crk[0]=Patch(0x0044337C, 0xEB);
crk[1]=Patch(0x004795F0, 0xC3);
```

```
crk[2]=Patch(0x004795F1, 0x90);
crk[3]=Patch(0x004795F2, 0x90);
crk[4]=Patch(0x004795F3, 0x90);
crk[5]=Patch(0x004795F4, 0x90);
crk[6]=Patch(0x005E478E, 0x90);
crk[7]=Patch(0x005E478F, 0x90);
crk[8]=Patch(0x005E4790, 0x90);
crk[9]=Patch(0x005E4791, 0x90);
crk[10]=Patch(0x005E4792, 0x90);
crk[11]=Patch(0x005E5669, 0xEB);
crk[12]=Patch(0x005F1552, 0xEB);
crk[13]=Patch(0x005E626E, 0xEB);
crk[14]=Patch(0x005E67D0, 0xEB);
<-------------End Code Snippet----------------->
```

As already told, the point is not the patches used in the example. In this very first example we have not used the "orig" bytes, because we are writing a blind loader.
Given this piece of code common to both loaders types, we can go.

## *3.2 Standard Loader*

As usual we present immediately the core structure of Standard Loader as we presented it so far.

```
<-------------Code Snippet----------------->
int main(int argc, char** argv) {

        //Handle of the victim main window
        HWND VictimDlghWnd=NULL;

        Patch crk[15];
        crk[0]=Patch(0x0044337C, 0xEB);
        crk[1]=Patch(0x004795F0, 0xC3);
        crk[2]=Patch(0x004795F1, 0x90);
        crk[3]=Patch(0x004795F2, 0x90);
        crk[4]=Patch(0x004795F3, 0x90);
        crk[5]=Patch(0x004795F4, 0x90);
        crk[6]=Patch(0x005E478E, 0x90);
        crk[7]=Patch(0x005E478F, 0x90);
        crk[8]=Patch(0x005E4790, 0x90);
        crk[9]=Patch(0x005E4791, 0x90);
        crk[10]=Patch(0x005E4792, 0x90);
        crk[11]=Patch(0x005E5669, 0xEB);
        crk[12]=Patch(0x005F1552, 0xEB);
        crk[13]=Patch(0x005E626E, 0xEB);
        crk[14]=Patch(0x005E67D0, 0xEB);
        //These are process'specific structures
        PROCESS_INFORMATION pi;
        STARTUPINFO si;
        memset(&pi, 0, sizeof(PROCESS_INFORMATION));
        memset(&si, 0, sizeof(STARTUPINFO));
        si.cb=sizeof(si);

        if( !::CreateProcess( ".\\TargetProcess.exe", // No module name (use command line).
                NULL,               // Command line.
                NULL,             // Process handle not inheritable.
                NULL,             // Thread handle not inheritable.
                NULL,             // Set handle inheritance to FALSE.
                CREATE_SUSPENDED, // suspended creation flags.
                NULL,             // Use parent's environment block.
                NULL,             // Use parent's starting directory.
                &si,              // Pointer to STARTUPINFO structure.
                &pi )             // Pointer to PROCESS_INFORMATION structure.
                )
        {
                char szBuf[80];
                GetLastErrorMsg(szBuf);
                MessageBox(NULL, szBuf, MSG_CAPTION, MB_OK);
                return 1;
        }
```

```
	ResumeThread(pi.hThread);

	// CheckGuardCondition implementation
	// Execute the FindConsole function that locates the console
	while(VictimDlghWnd==NULL) {
		EnumDesktopWindows(NULL, EnumWindowsProc, (LPARAM)&VictimDlghWnd);
		if(VictimDlghWnd!=NULL) {

			::MessageBox(NULL,"Victim's window found",MSG_CAPTION, MB_OK);

			HANDLE hProcess=NULL;
			hProcess = pi.hProcess;

			SuspendThread(pi.hThread);

			//find the memory addresses to patch!
			unsigned long byteswritten[15];
			unsigned long bytesread[15];
			char errors[15][256];

			for(int i=0; i<15; i++) {
				bytesread[i]=0;
				byteswritten[i]=0;
				strcpy(errors[i],"");
			}


			for (int idx=0; idx<15;idx++) {
				ReadProcessMemory(hProcess,
					(LPVOID)(crk[idx].address),
					(LPVOID)(&(crk[idx].orig)), 1,
					&bytesread[idx]);
				if(bytesread[idx]==0)
					GetLastErrorMsg(errors[idx]);
				else
					strcpy(errors[idx],"OK");

				WriteProcessMemory(hProcess,
					(LPVOID)(crk[idx].address),
					(LPVOID)(&(crk[idx].patch)), 1,
					&byteswritten[idx]);
				if(byteswritten[idx]==0)
					GetLastErrorMsg(errors[idx]);
				else
					strcpy(errors[idx],"OK");
			}

			ResumeThread(pi.hThread);

			char str[10000];
			strcpy(str,"");

			break;

		}
	}
	return 0;
}

void GetLastErrorMsg(char *szBuf)
{
    TCHAR szBuf[80];
    LPVOID lpMsgBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );
```

```
    wsprintf(szBuf, "Loader failed with error %d: %s", dw, lpMsgBuf);

    LocalFree(lpMsgBuf);
}

BOOL CALLBACK EnumWindowsProc(
        HWND hWnd,      // handle to parent window
        LPARAM lParam   // application-defined value
) {
        char ClassName[256];

        GetClassName(hWnd,ClassName, 256);
        char caption[256];
        GetWindowText(hWnd, caption,256);

        if(strstr(caption,"Main Target Window Caption")!=0 && _stricmp(ClassName,"TMainForm")==0)
        {
                HWND *hw=(HWND*)lParam;
                *hw=hWnd;
                return FALSE;
        }
        return TRUE;
}

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

The loader has the same structure of Figure 1 but in this case the guard condition is something that in most applications works just fine. The check is simple: if the main application's window is already among the windows that are on the desktop (visible or not) then the application is ready to be patched. This guard is used just because any Windows' application has a so called message pump that allows the application to handle messages coming from the GUI and generally implements the event-driven architecture of Windows. In Windows, just being brief, the only things that have message pumps are the windows (either visible or not). So any application to perform some graphical interface requires always a window. If your patch can be applied to the program once uncompressed in memory, a reliable method to understand that the program is ready in memory and unpacked, is to check for the presence of its main window. Well, what the above code does is to enum the windows starting from the desktop, using the following instruction:

```
EnumDesktopWindows(NULL, EnumWindowsProc, (LPARAM)&VictimDlghWnd)
```

What this instruction does is to call for all the windows on the desktop the EnumWindowProc with the handle of the currently examined window and a custom parameter, which is VictimDlgHwnd in our case. If you have a look at what the `EnumWindowsProc` does you will see that it simply uses two APIs, `GetClassName` and `GetWindowText` to get the caption of the window and check if it's the victim's window we are searching (that is of type TMainForm for the example and has a specific caption). Returning FALSE the cycle stops and the control returns to the main function.

Then the program applies one by one the patches of the patches vector.

Obviously there are some assumptions at the base of such a simple loader:

- the victim is single thread;
- the used packer once the application is unpacked in memory doesn't do much checks;
- the memory of the target process can be written;
- the security context of the victim allows us to operate on it;
- the victim doesn't have complex anti-tampering protections (see [10]). For example with Armadillo and COPYMEM2 this approach won't work.

All these limitations can be overcome, but of course make the sources more complicated.

## *3.3 Debugger Loader*

First of all, the loader has to create/attach a new/existing process and work on the target memory space. Because I've to talk about debugging a running process we have to search for some API able to open a process which is still active in memory and perform the attach feature. Once the process was attached the loader can start to wait for some event by settings a suitable debugging loop, in this loop all the event which came from the target is passed to the loader for debugging and finally the loader have to pass the control to the target or close the target or detach from the target and leave this one to run freely (this last feature is available only with Windows XP).

More schematically we have:

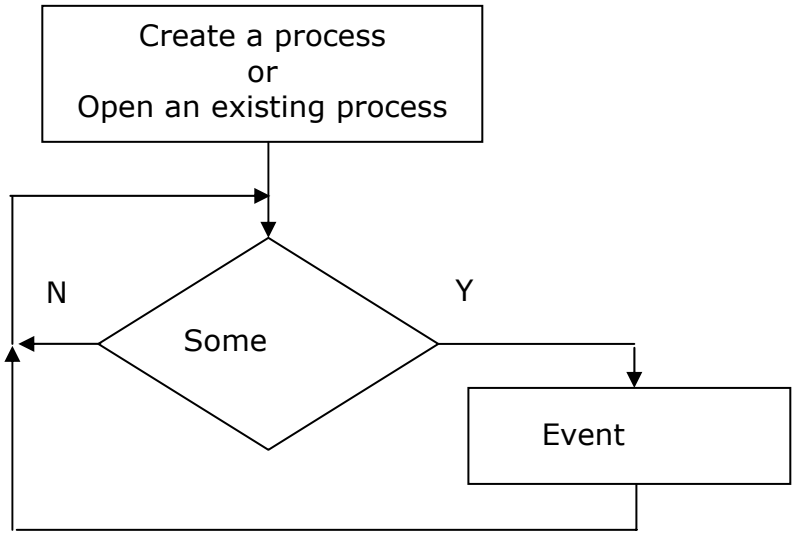


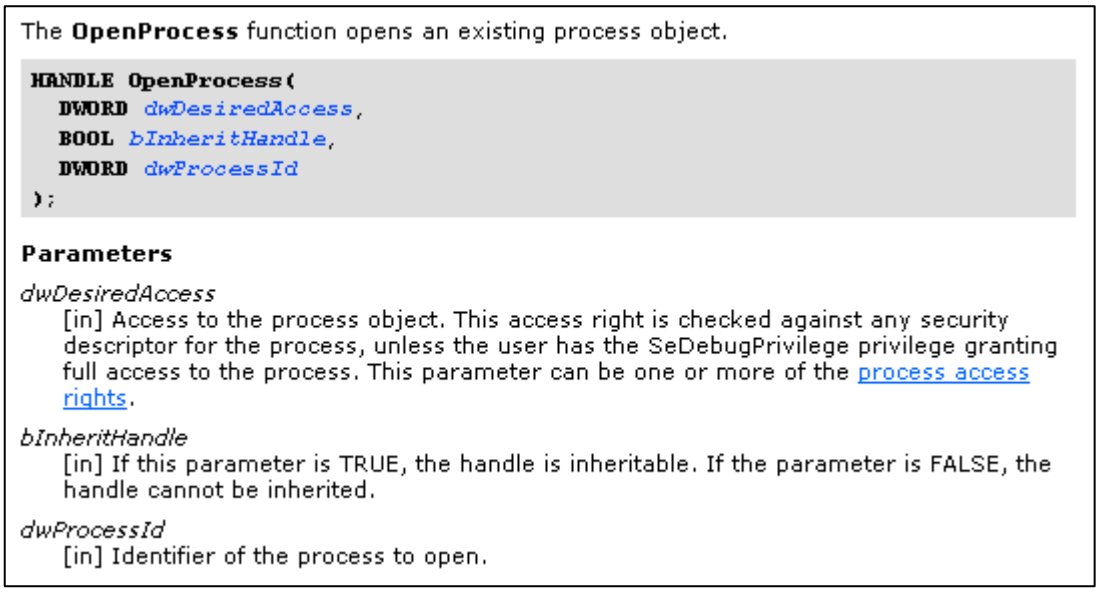

*Figure 3 - Debugger loader main cycle*

The ***CreateProcess*** function enables a debugger to start a process and debug it, specifying a proper creation parameter.

The ***OpenProcess*** function enables a debugger to obtain the identifier (PID or process identifier) of an existing process. (The ***DebugActiveProcess*** function uses this identifier to attach the debugger to the process.) Typically, debuggers open a process with the PROCESS_VM_READ and PROCESS_VM_WRITE flags. Using these flags enables the debugge\r to read from and write to the virtual memory of the process by using the well knows ***ReadProcessMemory*** and ***WriteProcessMemory*** functions.

The CreateProcess function should already be known, from MSDN library and [1], so we'll describe only the latter one.

MSDN states the following about the ***OpenProcess*** API :



The **OpenProcess** function opens an existing process object.

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  DWORD dwProcessId
);
```

**Parameters**

*dwDesiredAccess*
[in] Access to the process object. This access right is checked against any security descriptor for the process, unless the user has the SeDebugPrivilege privilege granting full access to the process. This parameter can be one or more of the process access rights.

*bInheritHandle*
[in] If this parameter is TRUE, the handle is inheritable. If the parameter is FALSE, the handle cannot be inherited.

*dwProcessId*
[in] Identifier of the process to open.

*Figure 4 - OpenProcess API description*

The first point to understand is about the *dwProcessId* parameter, this one is a unique identifier (namely process ID or PID) of the running process to open moreover ID Process numbers are reused, so they only identify a process for the lifetime of that process.

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a base priority, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

| **NOTE** |
| --- |
| From above consideration we have another very important remark about the address space of the target process, this one is different from the address space of the loader, every application have is own address space which is different from other, this is a key point and must be keep in mind in order to understand following consideration about accessing the process space from the loader space. |

Then now we have to look for a methods able to retrieve the (process) identifier related to our target process, this goal can be achieved by using the process enumeration and more in detail all the feature which came from the PSAPI.DLL.

### 3.3.1 Hiding a debugger to the target process

Of course if you're going to debug a program or to attach a debugger to a running process, the first thing to worry about is to hide the debugger to the process's controls. There are plenty of ways for a program to check if it is being debugged or not and not all of them can be easily fooled (see also [9]). What we are going to insert here is the fooling of the most common (easy) anti-debugging check, that lazy programmers use anywhere. As already introduced in [1], the most used API is IsDebuggerPresent, which returns 1 if yes otherwise 0 (false). The problem with a loader is that the API must be fooled into the target's process's space, thus the Hiding function will be a little different.

It comes really handy at this point to understand a little a structure each running process has, called PEB (*Process Environment Block*). This structure has several fields which are of interest to us especially the `BeingDebugged` element. We reported the TEB structure with the relative offsets of its elements, which are always useful while coding:

```
TEB
Offset    Elements name              Type
+0x000    InheritedAddressSpace      :                                                    UChar
+0x001    ReadImageFileExecOptions   :                                                    UChar
+0x002    BeingDebugged              :                                                    UChar
+0x003    SpareBool                  :                                                    UChar
+0x004    Mutant                     :                                Ptr32               Void
+0x008    ImageBaseAddress           :                                Ptr32               Void
+0x00c    Ldr                        :                              Ptr32                 _PEB_LDR_DATA
+0x010    ProcessParameters          :                    Ptr32                          _RTL_USER_PROCESS_PARAMETERS
+0x014    SubSystemData              :                                Ptr32               Void
+0x018    ProcessHeap                :                                Ptr32               Void
+0x01c    FastPebLock                :                           Ptr32                    _RTL_CRITICAL_SECTION
+0x020    FastPebLockRoutine         :                                Ptr32               Void
+0x024    FastPebUnlockRoutine       :                                Ptr32               Void
+0x028    EnvironmentUpdateCount     :                                                    Uint4B
+0x02c    KernelCallbackTable        :                                Ptr32               Void
+0x030    SystemReserved             :                                [1]                 Uint4B
+0x034    ExecuteOptions             :                 Pos                0,  2          Bits
+0x034    SpareBits                  :                 Pos                2, 30          Bits
+0x038    FreeList                   :                           Ptr32                    _PEB_FREE_BLOCK
+0x03c    TlsExpansionCounter        :                                                    Uint4B
+0x040    TlsBitmap                  :                                Ptr32               Void
+0x044    TlsBitmapBits              :                                [2]                 Uint4B
+0x04c    ReadOnlySharedMemoryBase   :                                Ptr32               Void
+0x050    ReadOnlySharedMemoryHeap   :                                Ptr32               Void
+0x054    ReadOnlyStaticServerData   :                     Ptr32                  Ptr32    Void
+0x058    AnsiCodePageData           :                                Ptr32               Void
+0x05c    OemCodePageData            :                                Ptr32               Void
+0x060    UnicodeCaseTableData       :                                Ptr32               Void
+0x064    NumberOfProcessors         :                                                    Uint4B
+0x068    NtGlobalFlag               :                                                    Uint4B
+0x070    CriticalSectionTimeout     :                                                  _LARGE_INTEGER
+0x078    HeapSegmentReserve         : Uint4B
```

This is instead the formal declaration of PEB structure, in case you might need it for your code:

```
typedef struct _PEB {
  BOOLEAN                 InheritedAddressSpace;
  BOOLEAN                 ReadImageFileExecOptions;
  BOOLEAN                 BeingDebugged;
  BOOLEAN                 Spare;
  HANDLE                  Mutant;
  PVOID                   ImageBaseAddress;
  PPEB_LDR_DATA           LoaderData;
  PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
  PVOID                   SubSystemData;
  PVOID                   ProcessHeap;
  PVOID                   FastPebLock;
  PPEBLOCKROUTINE         FastPebLockRoutine;
```

```
  PPEBLOCKROUTINE         FastPebUnlockRoutine;
  ULONG                   EnvironmentUpdateCount;
  PPVOID                  KernelCallbackTable;
  PVOID                   EventLogSection;
  PVOID                   EventLog;
  PPEB_FREE_BLOCK         FreeList;
  ULONG                   TlsExpansionCounter;
  PVOID                   TlsBitmap;
  ULONG                   TlsBitmapBits[0x2];
  PVOID                   ReadOnlySharedMemoryBase;
  PVOID                   ReadOnlySharedMemoryHeap;
  PPVOID                  ReadOnlyStaticServerData;
  PVOID                   AnsiCodePageData;
  PVOID                   OemCodePageData;
  PVOID                   UnicodeCaseTableData;
  ULONG                   NumberOfProcessors;
  ULONG                   NtGlobalFlag;
  BYTE                    Spare2[0x4];
  LARGE_INTEGER           CriticalSectionTimeout;
  ULONG                   HeapSegmentReserve;
  ULONG                   HeapSegmentCommit;
  ULONG                   HeapDeCommitTotalFreeThreshold;
  ULONG                   HeapDeCommitFreeBlockThreshold;
  ULONG                   NumberOfHeaps;
  ULONG                   MaximumNumberOfHeaps;
  PPVOID                  *ProcessHeaps;
  PVOID                   GdiSharedHandleTable;
  PVOID                   ProcessStarterHelper;
  PVOID                   GdiDCAttributeList;
  PVOID                   LoaderLock;
  ULONG                   OSMajorVersion;
  ULONG                   OSMinorVersion;
  ULONG                   OSBuildNumber;
  ULONG                   OSPlatformId;
  ULONG                   ImageSubSystem;
  ULONG                   ImageSubSystemMajorVersion;
  ULONG                   ImageSubSystemMinorVersion;
  ULONG                   GdiHandleBuffer[0x22];
  ULONG                   PostProcessInitRoutine;
  ULONG                   TlsExpansionBitmap;
  BYTE                    TlsExpansionBitmapBits[0x80];
  ULONG                   SessionId;
} PEB, *PPEB;
```

So the trick is to always set to 0 the byte `BeingDebugged` of the above structure. The problem is of course on how to find the PEB starting address. The PEB block is stored into another structure called *Thread Environment Block* (TEB), also known as *Thread Information Block* (TIB).

The operating system maintains a structure called *Thread Environment Block* (TEB) for every thread running in the system. The FS segment register is always set such that the address FS:0 points to the TEB of the thread being executed (as also reported in Figure 5).

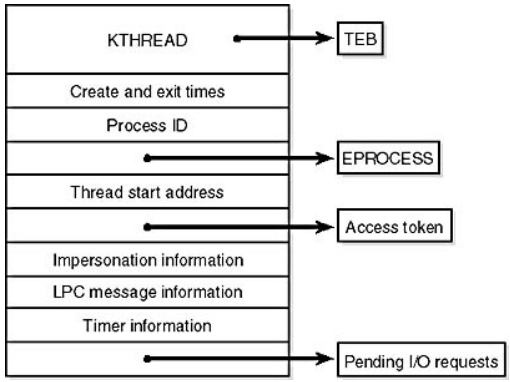



*Figure 5 - Structure of the executive thread block*

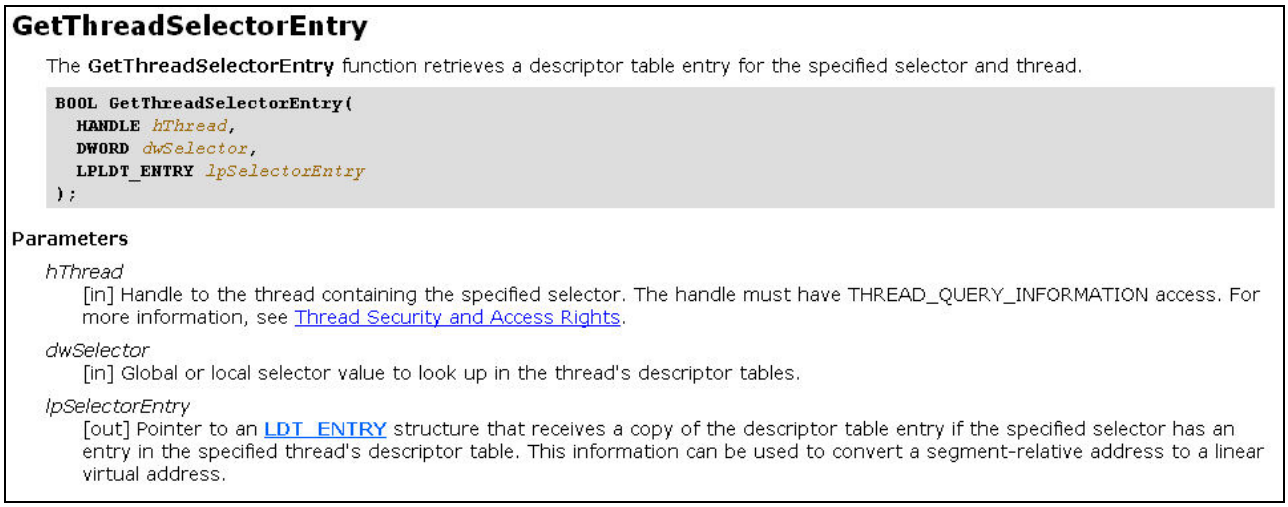
Its structure is the following one, for reference use:

```
typedef struct _TEB {
  NT_TIB                  Tib;
  PVOID                   EnvironmentPointer;
  CLIENT_ID               Cid;
  PVOID                   ActiveRpcInfo;
  PVOID                   ThreadLocalStoragePointer;
  PPEB                    Peb;
  ULONG                   LastErrorValue;
  ULONG                   CountOfOwnedCriticalSections;
  PVOID                   CsrClientThread;
  PVOID                   Win32ThreadInfo;
  ULONG                   Win32ClientInfo[0x1F];
  PVOID                   WOW32Reserved;
  ULONG                   CurrentLocale;
  ULONG                   FpSoftwareStatusRegister;
  PVOID                   SystemReserved1[0x36];
  PVOID                   Spare1;
  ULONG                   ExceptionCode;
  ULONG                   SpareBytes1[0x28];
  PVOID                   SystemReserved2[0xA];
  ULONG                   GdiRgn;
  ULONG                   GdiPen;
  ULONG                   GdiBrush;
  CLIENT_ID               RealClientId;
  PVOID                   GdiCachedProcessHandle;
  ULONG                   GdiClientPID;
  ULONG                   GdiClientTID;
  PVOID                   GdiThreadLocaleInfo;
  PVOID                   UserReserved[5];
  PVOID                   GlDispatchTable[0x118];
  ULONG                   GlReserved1[0x1A];
  PVOID                   GlReserved2;
  PVOID                   GlSectionInfo;
  PVOID                   GlSection;
  PVOID                   GlTable;
  PVOID                   GlCurrentRC;
  PVOID                   GlContext;
  NTSTATUS                LastStatusValue;
  UNICODE_STRING          StaticUnicodeString;
  WCHAR                   StaticUnicodeBuffer[0x105];
  PVOID                   DeallocationStack;
  PVOID                   TlsSlots[0x40];
  LIST_ENTRY              TlsLinks;
  PVOID                   Vdm;
  PVOID                   ReservedForNtRpc;
  PVOID                   DbgSsReserved[0x2];
  ULONG                   HardErrorDisabled;
  PVOID                   Instrumentation[0x10];
  PVOID                   WinSockData;
  ULONG                   GdiBatchCount;
  ULONG                   Spare2;
  ULONG                   Spare3;
  ULONG                   Spare4;
  PVOID                   ReservedForOle;
  ULONG                   WaitingOnLoaderLock;
  PVOID                   StackCommit;
  PVOID                   StackCommitMax;
  PVOID                   StackReserved;
} TEB, *PTEB;
```

The most interesting element for us is the Peb one, which is at an offset of 0x30 (which is the summed size of all the preceding elements into the TEB structure).

Figure 6 shows the correct API to use. Indeed there's an undocumented function NtCurrentTeb[1], which will give directly the TEB, but explaining how to use it would take us out of scope.



**GetThreadSelectorEntry**

The **GetThreadSelectorEntry** function retrieves a descriptor table entry for the specified selector and thread.

```
BOOL GetThreadSelectorEntry(
  HANDLE hThread,
  DWORD dwSelector,
  LPLDT_ENTRY lpSelectorEntry
);
```

**Parameters**

*hThread*
[in] Handle to the thread containing the specified selector. The handle must have THREAD_QUERY_INFORMATION access. For more information, see Thread Security and Access Rights.

*dwSelector*
[in] Global or local selector value to look up in the thread's descriptor tables.

*lpSelectorEntry*
[out] Pointer to an LDT_ENTRY structure that receives a copy of the descriptor table entry if the specified selector has an entry in the specified thread's descriptor table. This information can be used to convert a segment-relative address to a linear virtual address.

*Figure 6 - GetThreadSelectorEntry API*

The returned value is another structure `LDT_ENTRY` (not described here, but essentially it is used to store the address in a special way, able to handle very big values, because of all the valid addressing space of windows is huge). Anyway once the `LDT_ENTRY` returned by `GetThreadSelectorEntry` is converted into a linear value, it can be used to access the TEB and then the PEB, and then again the `BeingDebugged` element and set it to 0.

The whole operation is in the code of the `HideDebugger` function reported here.
This time you need to pass to the `HideDebugger` two target's handles, the thread and the process handles. Both of them will be explained later on.

```
<-------------Code Snippet----------------->
BOOL HideDebugger(HANDLE thread, HANDLE hproc)
{
        CONTEXT victimContext;

        // This function is used to patch the IsDebuggerPresent API which might be called from
        // debugged program (e.g. ASProtect) in order to detect debugger presence. This function
        // is mainly based on FS:[0] treating.
        // In an x86 environment, the FS register points to the current value of the Thread
        // Information Block (TIB) structure.
        // One element in the TIB structure is a pointer to an EXCEPTION_RECORD structure, which
        // in turn contains a pointer to an exception handling callback function. Thus, each
        // thread has its own exception callback function.
        // The x86 compiler builds exception-handling structures on the stack as it processes
        // functions. The FS register always points to the TIB, which in turn contains a pointer
        // to an EXCEPTION_RECORD structure.
        // The EXCEPTION_RECORD structure points to the exception handler function.

        // EXCEPTION_RECORD structures form a linked list: the new EXCEPTION_RECORD structure
        // contains a pointer to the previous EXCEPTION_RECORD structure,
        // and so on. On Intel-based machines, the head of the list is always pointed
        // to by the first DWORD in the thread information block, FS:[0]

        //77E5276B >  64:A1 18000000  MOV EAX,DWORD PTR FS:[18]
        //77E52771    8B40 30         MOV EAX,DWORD PTR DS:[EAX+30]
        //77E52774    0FB640 02       MOVZX EAX,BYTE PTR DS:[EAX+2]
        //77E52778    C3              RETN
```

[1] NTSYSAPI PTEB NTAPI NtCurrentTeb( );

```
        // Set up the victimContex access flag
        victimContext.ContextFlags = CONTEXT_SEGMENTS;
        // Fill the victim context structure with process data
        if (!GetThreadContext(thread, &victimContext))
                return FALSE;

        // GetThreadSelectorEntry is only functional on x86-based systems.
        // For systems that are not x86-based, the function returns FALSE.
        // The GetThreadSelectorEntry function fills this structure with
        // information from an entry in the descriptor table. You can use this information
        // to convert a segment-relative address to a linear virtual address.
        // The base address of a segment is the address of offset 0 in the segment.
        // To calculate this value, combine the BaseLow, BaseMid, and BaseHi members

        LDT_ENTRY sel;
        if (!GetThreadSelectorEntry(thread, victimContext.SegFs, &sel))
                return FALSE;

        DWORD fsbase = (sel.HighWord.Bytes.BaseHi << 8| sel.HighWord.Bytes.BaseMid) << 16|
                        sel.BaseLow;
        DWORD RVApeb;
        SIZE_T numread;

        if (!ReadProcessMemory(hproc, (LPCVOID)(fsbase + 0x30), &RVApeb, 4, &numread) ||
                numread != 4)
                return FALSE;

        WORD beingDebugged;
        if (!ReadProcessMemory(hproc, (LPCVOID)(RVApeb + 2), &beingDebugged, 2, &numread)
                || numread != 2)
                return FALSE;

        beingDebugged = 0;

        if (!WriteProcessMemory(hproc, (LPVOID)(RVApeb + 2), &beingDebugged, 2, &numread)
                || numread != 2)
                return FALSE;

        return TRUE;
}
<-------------End Code Snippet----------------->
```

### 3.3.2 Process Status Helper (PSAPI.DLL)

Looking into the MSDN, we find a lot of useful info about process investigation by using the PSAPI.DLL (process status API) functions.
The process status API (PSAPI) provides sets of functions for retrieving the following information:

- Process Information
- Module Information
- Device Driver Information
- Process Memory Usage Information
- Working Set Information
- Memory-Mapped File Information

The system maintains a list of running processes (the one you see is when you see open the task manager). You can retrieve the identifiers (PID) for these processes by calling the ***EnumProcesses*** function. This function fills an array of DWORD values with the identifiers of all processes which is currently running in the system.
Many functions in PSAPI require a process handle. A handle is a pointer to an object which is controlled by the system. To obtain a process handle for a running process, we have to pass its process identifier (obtained from ***EnumProcesses***) to the ***OpenProcess*** function. Remember also to call the ***CloseHandle*** function when you are finished with the process handle (this don't close

the process but simply free the memory related to the opened handle, and allows to keep the system more stable).

A module is an executable file or a DLL. Each process consists of one or more modules. You can retrieve the list of module handles for a process by calling the ***EnumProcessModules*** function. This function fills an array of HMODULE values with the module handles for the specified process. The first module is the executable file. Remember that these module handles are most likely from some other process, so you cannot use them with functions such as ***GetModuleFileName***. However, you can use PSAPI functions to obtain information about a module from another process. To obtain module information:

- Call the ***GetModuleBaseNam***e function. This function takes a process handle and a module handle as input and fills in a buffer with the base name of a module (for example, KERNEL32.DLL). A related function, ***GetModuleFileNameEx***, takes the same parameters as input but returns the full path to the module (for example, C:\WINNT\SYSTEM32\KERNEL32.DLL).

- Call the ***GetModuleInformation*** function. This function takes a process handle and a module handle and fills a MODULEINFO structure with the load address of the module, the size of the linear address space it occupies, and a pointer to its entry point.

Using this information we can write a code snippet able to find the process ID of the victim process and also to enumerate all the modules used by the process, below we will report a first snippet code to perform the process enumeration, module enumeration will be show later.

First of all we need to use the PSAPI function then we have to build a valid pointer for each function which we have to use then:

```
<-------------Code Snippet----------------->
hPsapi = LoadLibrary("psapi.dll");

if (!hPsapi) {
    printf("Cannot load psapi.dll :-(\n");
    return;
}

pEnumProcessModules = (BOOL (WINAPI *)(HANDLE,
                       HMODULE *,
                       DWORD,
                       LPDWORD)) GetProcAddress(hPsapi, "EnumProcessModules");
pGetModuleBaseName = (DWORD (WINAPI *)(HANDLE,
                       HMODULE,
                       LPTSTR,
                       DWORD)) GetProcAddress(hPsapi, "GetModuleBaseNameA");
pGetModuleInformation=(BOOL (WINAPI *)(HANDLE,
                       HMODULE,
                       LPMODULEINFO,
                       DWORD)) GetProcAddress(hPsapi, "GetModuleInformation");
pEnumProcesses = (BOOL (WINAPI *)(DWORD*,
                       DWORD,
                       DWORD*)) GetProcAddress(hPsapi, "EnumProcesses");

// Make some simple check about right pointer assignment
if ( (pEnumProcessModules == NULL) || (pGetModuleBaseName  == NULL) ) {
               printf("Cannot load psapi functions\n");
               FreeLibrary(hPsapi);
               return;
}
<-------------End Code Snippet------------->
```

Now we have to collect the list of all the running processes and then for each one check if it is equal to our victim process. This task must be performed after the victim process is running then we have to sure about user has really started our process:

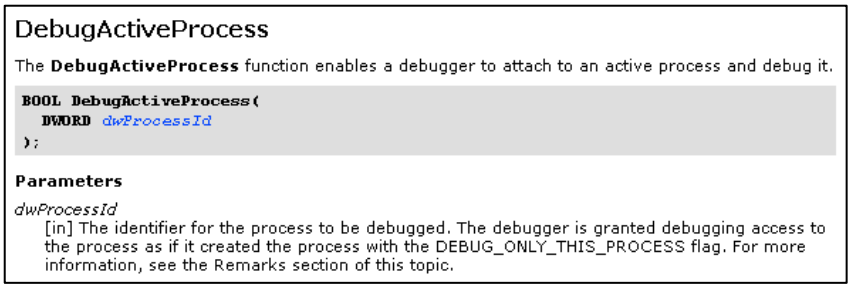
```
<-------------Code Snippet----------------->
// ----------------------------------------
// Wait for user confirmation
// ----------------------------------------
sprintf(szMsgText,"\tStart the installer and press OK when you are into the registration window");
MessageBox(NULL, szMsgText, szMsgCapt, MB_OK);
printf("now go into the registration\nwindow and insert a fake serial...\n");
<-------------End Code Snippet------------->
```

Then we can start to look for all the running processes and check if at least one of them is the one we were searching, while saving the handler for future uses:

```
<-------------Code Snippet----------------->
// ----------------------------------------
// Get the list of process identifiers.
// ----------------------------------------
TCHAR szProcessName[MAX_PATH] = TEXT("<unknown>");

if ( !pEnumProcesses( aProcesses, (DWORD)sizeof(aProcesses), &cbNeeded ) )
{
        if (hPsapi != NULL)
                FreeLibrary(hPsapi);
        return;
}
           cProcesses = cbNeeded / sizeof(DWORD); // Calculate how many process identifiers were returned.
for ( i = 0; i < cProcesses; i++ )      // Print the name and process identifier for each process.
{
        hTmpProcess = OpenProcess( PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
                                   FALSE,
                                   aProcesses[i] );  // Get a handle to the process.
        if (NULL != hTmpProcess ) // Get the process name.
        {
                              if ( pEnumProcessModules( hTmpProcess, &hMod, sizeof(hMod), &cbNeededTmp) )
                    pGetModuleBaseName(    hTmpProcess,
                                           hMod,
                                           szProcessName,
                                           sizeof(szProcessName)/sizeof(TCHAR) );
                }

                                           // Print the process name and identifier.
                                                        if (bDebugStage)
                    printf("%s  (PID: %u)\n", szProcessName, aProcesses[i] );

                                   // Search for victim process name and retrieve the process ID
                                        if ( strcmp(szProcessName,szVictimProcessName) == 0)
                {
                        bVictimPIDfound = true;
                        aVictimProcessId = aProcesses[i];
                }
                CloseHandle( hTmpProcess );   // Close the process handle
        }

if (bVictimPIDfound == false)
{
        MessageBox( NULL,
                    "\tVictim process ID not found!\n You've to start the installation before!",
                    szMsgCapt, MB_OK);

        if (hPsapi != NULL)
                FreeLibrary(hPsapi);
        return ;
}
else {
        if (bDebugStage)
                MessageBox(NULL, "\tVictim process ID found!", szMsgCapt, MB_OK);
}
<-------------End Code Snippet------------->
```

### 3.3.3 The debugging stage (the attach stage)

Now we found the target process, the next step is about debugging this target, before going into the main debugging task we have to attach the target and this can be done by using the ***DebugActiveProcess*** API function.



DebugActiveProcess

The **DebugActiveProcess** function enables a debugger to attach to an active process and debug it.

```
BOOL DebugActiveProcess(
  DWORD dwProcessId
);
```

**Parameters**

*dwProcessId*
[in] The identifier for the process to be debugged. The debugger is granted debugging access to the process as if it created the process with the DEBUG_ONLY_THIS_PROCESS flag. For more information, see the Remarks section of this topic.

*Figure 7 - DebugActiveProcess API description*

The process is debugged with DEBUG_ONLY_THIS_PROCESS privilege; for the sake of clarity we have these two distinctions:

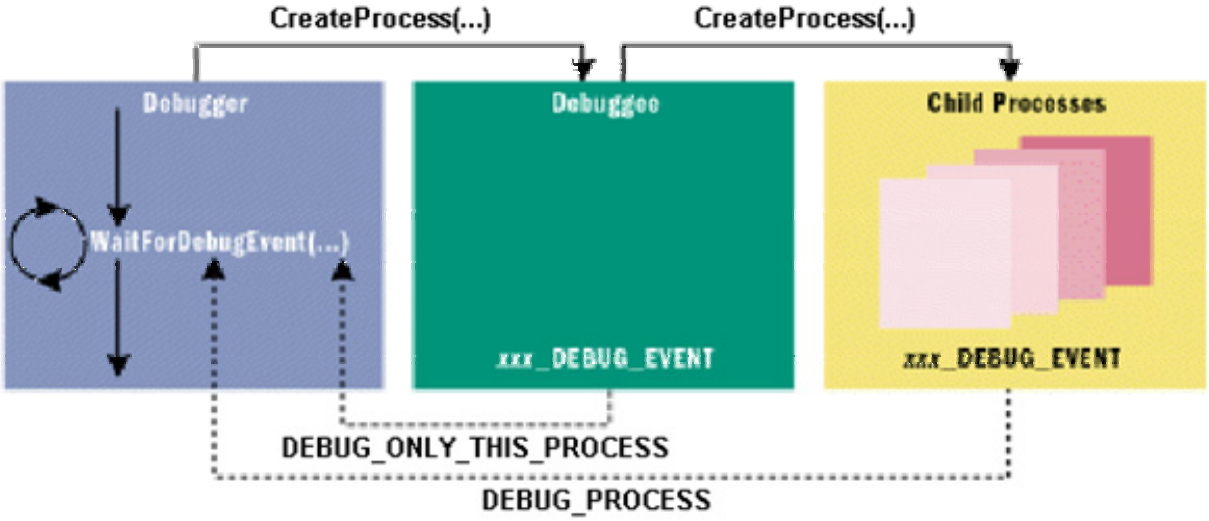
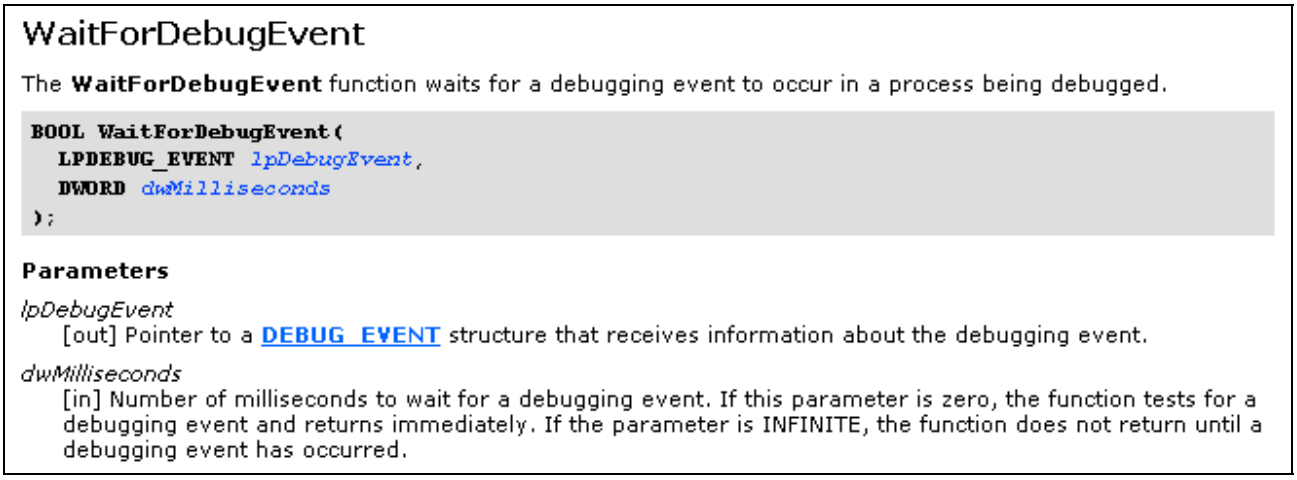



*Figure 8 - DEBUG process situation.*

From MSDN we have: the debugger must have appropriate access to the target process in order to read and write the process memory, and the debugger must be able to open the process for PROCESS_ALL_ACCESS. On Windows Me/98/95, the debugger has appropriate access if the process identifier is valid. On other versions of Windows, ***DebugActiveProcess*** can fail if the target process is created with a security descriptor that grants the debugger anything less than full access. If the debugging process (our loader) has the SE_DEBUG_NAME privilege granted and enabled, it can debug any process.

After successfully execution of this function the process (debuggee) can be debugged and the debugger is expected to wait for debugging events by using the ***WaitForDebugEvent*** function.



## WaitForDebugEvent

The **WaitForDebugEvent** function waits for a debugging event to occur in a process being debugged.

```
BOOL WaitForDebugEvent(
  LPDEBUG_EVENT lpDebugEvent,
  DWORD dwMilliseconds
);
```

**Parameters**

*lpDebugEvent*
[out] Pointer to a **DEBUG_EVENT** structure that receives information about the debugging event.

*dwMilliseconds*
[in] Number of milliseconds to wait for a debugging event. If this parameter is zero, the function tests for a debugging event and returns immediately. If the parameter is INFINITE, the function does not return until a debugging event has occurred.

*Figure 9 - WaitForDebugEvent function API.*

This function should be called in two ways, first one have *dwMilliseconds* value set from 0 to some value in this mode this function wait some event for a specified amount of millisecond and then return back the control to the debugger, the second way is by using the INFINITE constant; in this case the function doesn't return until some event occurs (during this time the target runs freely and the debugger is inactive or frozen).

```
<-------------Code Snippet----------------->
// ------------------------------------------
// Main debugger cycle
// ------------------------------------------

DEBUG_EVENT DebugEv;                    // debugging event information
DWORD dwContinueStatus = DBG_CONTINUE; // exception continuation
HMODULE hDLL;                           // temp handle used for target function offset calculation

                                                        for(;;) {
        // Wait for a debugging event to occur. The second parameter indicates
        // that the function does not return until a debugging event occurs.
        // We are waiting for infinite time, then wait for each Debug Event.

        WaitForDebugEvent(&DebugEv, INFINITE);

        // If we're into the first event save the process thread handle
        if (!bFirstEvent)
        {
                hVictimThreadHandle = DebugEv.u.CreateProcessInfo.hThread;
                bFirstEvent = true;
        }
        // Process the debugging event code.
        switch (DebugEv.dwDebugEventCode) {
                // Event handler …
        }
}
<-------------End Code Snippet------------->
```

### 3.3.4 The debugging stage (the DEBUG_EVENT structure)

When the attach stage is finished the system send to the debugger a CREATE_PROCESS_DEBUG_EVENT debugging event, when the ***WaitForDebugEvent*** function return to the debugger the system fill the DebugEv structure with the process data.

Now is time to give a close look to the DEBUG_EVENT structure that describes a debugging event:

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
    EXCEPTION_DEBUG_INFO Exception;
    CREATE_THREAD_DEBUG_INFO CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO ExitThread;
    EXIT_PROCESS_DEBUG_INFO ExitProcess;
    LOAD_DLL_DEBUG_INFO LoadDll;
    UNLOAD_DLL_DEBUG_INFO UnloadDll;
    OUTPUT_DEBUG_STRING_INFO DebugString;
    RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT,
*LPDEBUG_EVENT;
```

*Figure 10 - The DEBUG_EVENT structure*

This structure give a complete situation of the event which have triggered the ***WaitForDebugEvent*** function and also keep more interesting parameters as the CREATE_PROCESS_DEBUG_INFO structure which is a member of the main DEBUG_EVENT structure. The CREATE_PROCESS_DEBUG_INFO structure contains process creation information that can be used by a debugger, from MSDN we have:

```
typedef struct _CREATE_PROCESS_DEBUG_INFO {
  HANDLE hFile;
  HANDLE hProcess;
  HANDLE hThread;
  LPVOID lpBaseOfImage;
  DWORD dwDebugInfoFileOffset;
  DWORD nDebugInfoSize;
  LPVOID lpThreadLocalBase;
  LPTHREAD_START_ROUTINE lpStartAddress;
  LPVOID lpImageName;
  WORD fUnicode;
} CREATE_PROCESS_DEBUG_INFO,
*LPCREATE_PROCESS_DEBUG_INFO;
```

*Figure 11 - The CREATE_PROCESS_DEBUG_INFO structure.*

| **NOTE** |
| --- |
| A more important thing to do is about the *hThread* parameter because this handle is related to the main thread creation and is also useful when we have to read or modify the CONTEXT for the process examined, this it is the right time to save this handle because for all the future event this parameter will be set to NULL. |

# 4 An unifying C++ framework for writing loaders

Now it's time to understand better what we did playing with C++ around loaders. After having written several loaders, we tried to cut out the complex or repetitive parts of all loaders placing them inside a C++ framework which will hide most of the complexity. We are going now to explain how to code a loader using such a framework while the internals are left to the included sources (commented). The resulting framework isn't that simple indeed and took a little to code it. You'll have to integrate this document with the comments in the code.

Obviously there are some assumptions at the base of this framework:

- the security context of the victim can be modified by the user to allows us to write on process's memory;
- the victim doesn't have complex anti-tampering protections (see [10]). For example with Armadillo and COPYMEM2 this approach won't easily work.

## *4.1 Generics on the framework*

First of all for those of you which have already read [1] there are a lot of classes used also in this framework which I already used for Oraculums (and will not explain again). Oraculums are indeed special loaders, with a specific scope in mind!

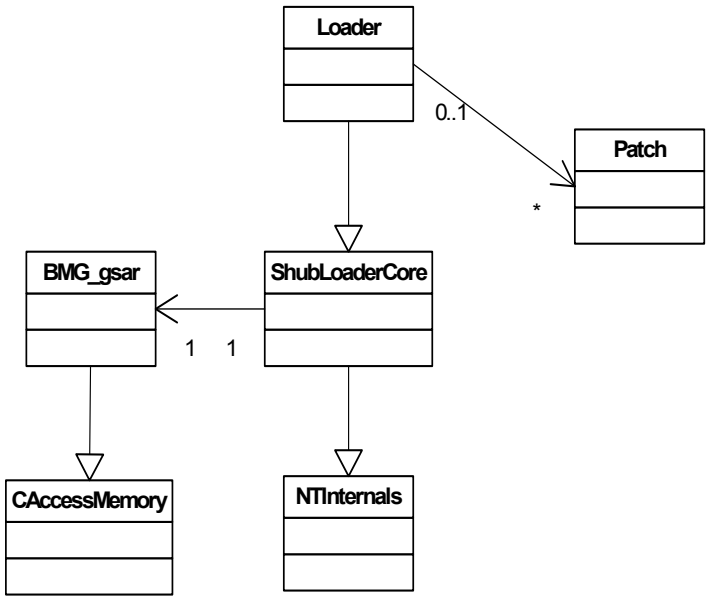



*Figure 12 - Main framework classes structure (class diagram)*

Figure 12 reports the main classes' structure of the framework using an UML notification. There are more classes behind these, but are not so important for us and indeed quite complicated to explain.

We will go in details for each one. Anyway briefly:

- NTInternals class. It is the base class for all the loader's classes and exposes some useful NT methods that are available in the Kernel32, but not exposed by the compiler (such as SuspendProcess, DebugActiveProcessStop).
- ShubLoaderCore class. It is the real core of the framework where all the work is performed.

- Loader class. It is the top part of the loader where all the application specific code is. This is or should be the only class that a developer should modify and where the applications specific things should be coded. This class is quite complex, but left alone is not able to do anything. What it needs is a derived class which instructs the engine on how/when patch the victim.
- Patch is a class, a little more complex than the one presented in section 3.1, which is used to easily store the patches of the program. Being the patches application specific by definition, this class must be initialized into the Loader's class.
- BMG_gsar class. It is a class I have already used in [1] which implements a really fast memory patterns searching algorithm (see [1] for details). It is used by the ShubLoaderCore class to search patches faster.
- CAccessMemory class. It is the base class for BMG_gsar, which gives to this class the methods for a controlled access to memory (handling read/write rights of accessed pages).

As for all the Object Oriented Programming the guiding concept behind the whole framework is the encapsulation of problems. As I already told usually the only thing a developer should modify is the Loader class.

Now we will present a little the most important classes: NTInternals, ShubLoaderCore and a sample Loader class.

### 4.1.1 NTInternals

| NTInternals |
| --- |
| |
| +DebugActiveProcessStop(in dwProcessId : unsigned long)<br>+GetProcessId(inout Process : void*)<br>+HideDebugger(inout thread : void*, inout hproc : void*)<br>+ZwResumeProcess(inout Process : void*)<br>+ZwSuspendProcess(inout Process : void*)<br>+ZwSuspendThread(inout hThread : void*, inout pSuspendCount : unsigned long*) |

This class implements few wrappers of the NT internals functions the loader will use. The functions are directly taken from exports in the system's dlls, because Microsoft doesn't officially give support for these APIs (you cannot find the prototypes in the standard Visual Studio distributions) or because we didn't want to install the whole DDK package (Driver Developer Kit). We implemented them here in the following basic way (for example for `DebugActiveProcessStop`):

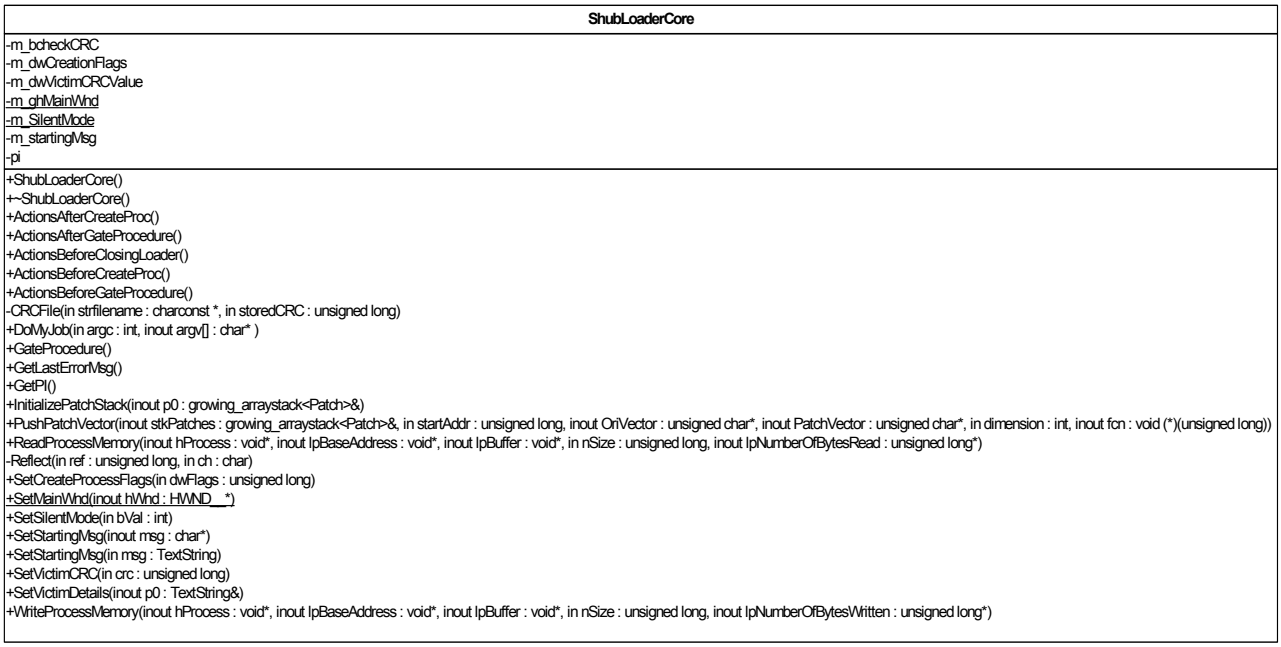
```
<-------------Code Snippet------------->
//Function pointer to the export.
typedef WINBOOL  (STDCALL *fcnDebugActiveProcessStop)(DWORD dwProcessId);

WINBOOL STDCALL NTInternals::DebugActiveProcessStop(DWORD dwProcessId)
{
        FARPROC addrIDP;
        HINSTANCE hKer;

        fcnDebugActiveProcessStop fcn;

        hKer = GetModuleHandle("Kernel32");
        addrIDP = GetProcAddress(hKer, "DebugActiveProcessStop");

        //Check API
        if (addrIDP!=NULL) {
                //gives to the function pointer the parameters.
                fcn=(fcnDebugActiveProcessStop)addrIDP;
                return fcn(dwProcessId);
        }
        return 0;
}
<-------------End Code Snippet------------->
```

Note that the Windows' API `DebugActiveProcessStop` is available on Windows only since the XP release. Using the NTInternals class, ensures the compatibility of the loader with all the Windows systems (9x/NT/2000), just because if the function is present in the system it is used (the addIDP variable is not NULL) otherwise the function simply does nothing, returning 0.

An important note is about the inclusion in this class of the HideDebugger API already described in section 3.3.1. This gives to you the possibility to add extra hiding might be required deriving this API into a derived class. Simply you can write code such this:

```
<------------- Code Snippet------------->
BOOL HideDebugger(HANDLE thread, HANDLE hproc) {

	//TODO: Add you own extra hiding customization here
	return NTInternals:: HideDebugger(thread, hproc);
}
<-------------End Code Snippet------------->
```

### 4.1.2 ShubLoaderCore




This class is quite complex. All of its methods can be classified in two.

- Virtual methods (see a C++ manual for the exact meaning of "virtual methods" of a class): briefly this means that if the derived class (Loader) implements them then this implementation is used, otherwise a dummy implementation is instead. Virtual functions are functions for which a given class has only a default implementation. If a derived class implements one of them, then the derived implementation is used, otherwise the default one. This mechanism is essential to allow derived classes to specify a different behaviour for a given method, thus to customize the loader's behaviour.
- Help methods, which can be used from within the virtual methods implementation to easily do common operations.

#### 4.1.2.1 DoMyJob

The main flowchart is implemented into the DoMyJob method, which is the real core of the class.

- `int DoMyJob(int argc, TCHAR* argv[])`. This is the core part of the loader, does all the hard work. The parameters are the command-line parameters of the loaders which are passed to the victim as well (usually they are coming from parameters with same names from the loader's main). If you don't need them simply set all of them to NULL. A loader is usually a DOS or a Win32 application, which command line parameters can be passed to the DoMyJob method. The function then will pass them transparently to the victim process. This is really useful when the loader is applied to a victim that uses command line parameters.

The DoMyJob method is the only one that the main() function of the loader must call in order to start the loader. See following sections where a complete loader writing process is described.

With respect to Figure 1 we modified a little the flow chart, inserting some more custom control points which usually are needed to perform a loader in most situations. Figure 13 reports the new flowchart where the additional methods are coloured differently. These methods are the virtual methods mentioned previously that the Loader class can implement to customize the whole loader behaviour.

The most important place where to insert the applications dependant things is the GateProcedure which is a function that should return TRUE when the application is ready to be patched. The GateProcedure then is a continuous test on the victim to find if a patching condition is met. All the other virtual functions are ancillary, meant to prepare the things.

The source code of the class is heavily commented, thus for further clarification take a look at those comments.

#### 4.1.2.2 Virtual Methods

Pure virtual methods, MUST be overwritten by the class derived from ShubLoaderCore which implements specific actions for the specific loader, such as patches, application path, and a specific gate condition.

Only ActionsBeforeCreateProc() and ActionsAfterCreateProc() are not pure virtual, because several times you don't need to do anything special here inside (derived classes are not obliged to implement them).

- `virtual BOOL SetVictimDetails(/*OUT*/ TextString &victimFileName)`. Set the Victim's name and it's CRC (optional, using SetVictimCRC()). The TextString is an OUT parameter, must be set by this function if you don't call SetVictimCRC from within the CRC isn't checked.

- `virtual BOOL InitializePatchStack(/*OUT*/ growing_arraystack<Patch> &stkPatches)`. Add to the patches stack the patches to do. The stkPatches variable is an OUT parameter and must be filled by the function. You can also use matrix of consecutive binary data, such for example coming to a dump or a long patch. In this case use the `PushPatchVector` which pushes on the Patch stack a whole matrix of consecutive patches, starting from an initial address. All the patches are stored into a stack of patches, which is internally handled. This logic allows adding whatever patches you like. If the order of patches is important, consider the stack logic, so the first patch added is the last applied. The variable holding the stack is the `stkPatches`, which must be used.

- `virtual BOOL ActionsBeforeCreateProc`(). Invoke an action just before the call to CreateProcess.
- `virtual BOOL ActionsAfterCreateProc()`. Invoke an action just after the call to CreateProcess, while it is still SUSPENDED
- `virtual BOOL ActionsBeforeGateProcedure()`. Actions performed just before calling the gatecondition, are useful to prepare it if needed.
- `virtual BOOL GateProcedure`(). It's the condition till the Loader waits before Suspending the process and applying patches. Returns TRUE when ready to patch.
- `virtual BOOL ActionsAfterGateProcedure()`. Actions performed just after the GateProcedure to clean eventually the special settings made to reach the GateProcedure. This operation is done after having applied all the patches but before resuming the process.
- `virtual BOOL ActionsBeforeClosingLoader()`. Invoke an action just before closing the loader.

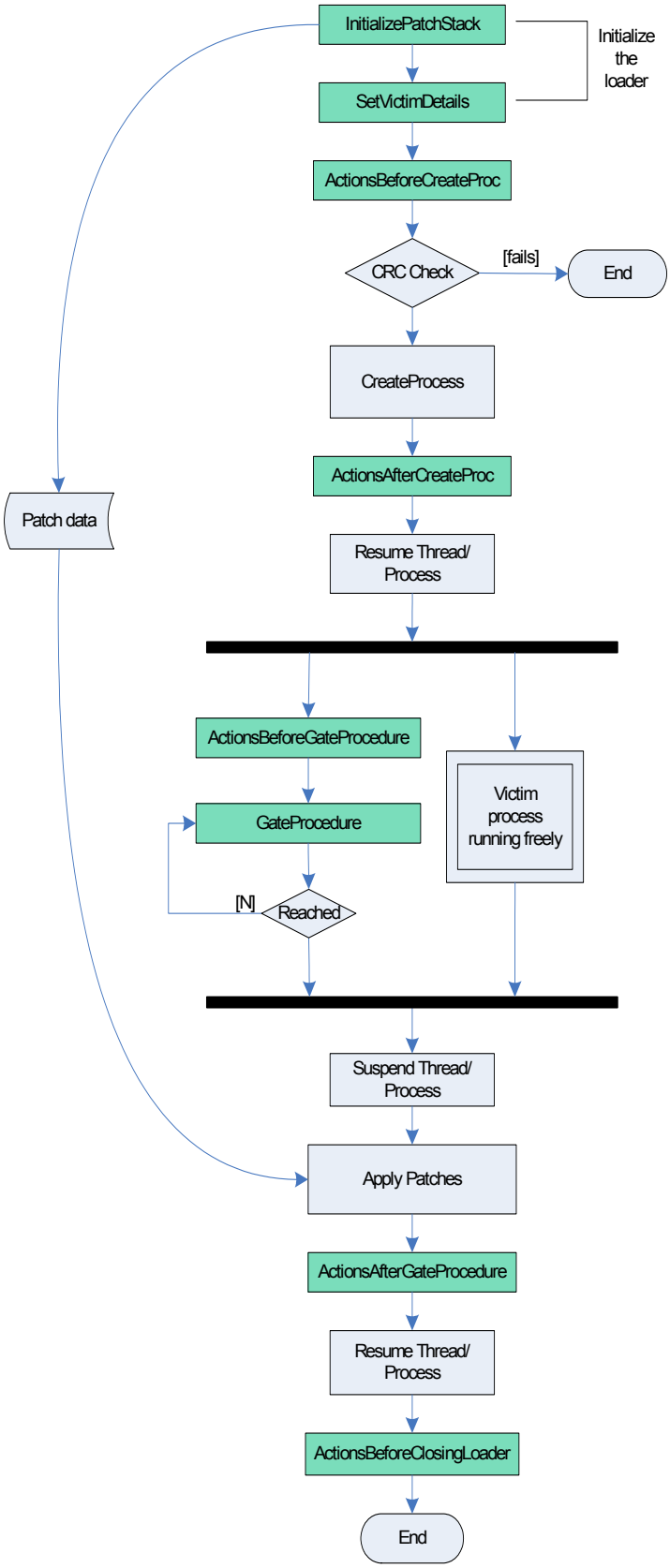



*Figure 13 - Modified flowchart od the ShubLoaderCore::DoMyJob method*

#### 4.1.2.3 Helper Methods

Helper methods are available to be used into the virtual methods to perform some initializing actions (e.g. setting the CRC of the victim, or setting the loader to be silent, not returning any message windows).

- `TextString GetLastErrorMsg()`. Retrieves a formatted message of the last system error message. Use for you own error checking/reporting in the derived classes.

- `static void SetMainWnd(HWND hWnd)`. This function might be used from inside the GateCondition procedure to set the real main HWND of the application. When a program is difficult to suspend, the Loader tries to suspend the whole process using undocumented low level APIs, which requires, in order to be executed, the handle of the main process' window. If this method isn't called these undocumented tentatives to stop the victim process are not used. Do not call if you are not experiencing problems suspending the victim's process.

- `void SetCreateProcessFlags(DWORD dwFlags)`. Used to define new creation flags to be passed to CreateProcess API. Default value is `CREATE_SUSPENDED` and you don't need to call this method to set it. Otherwise if you want to specify something else, call it properly. For example if you are coding a debugger loader you'll surely need to call this method with proper parameters. Can be called in any function before the call to CreateProcess, thus one of the following: `PushPatchVector` || `SetVictimDetails` || `InitializePatchStack` || `ActionsBeforeCreateProc`. The most logical place is anyway `ActionsBeforeCreateProc`. For example to create a debugger loader use this combination: `DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS | CREATE_NEW_CONSOLE`.

- `PROCESS_INFORMATION* GetPI()`. Use this function in all the derived classes to get the PROCESS_INFORMATION structure. If it's NULL means that the process has not been already started or something went wrong!

- `void SetVictimCRC(DWORD crc)`. Set the victim's CRC. If invoked the loader will check against the real victim's CRC (calculated on the whole victim's file).

- `void SetSilentMode(BOOL bVal)`. This function must eventually be called into the derived class and modify the whole behavior of the program. If it is defined the loader does not issue most of the errors messages which are usually issued. This is useful for those cases with which the dialogs are disturbing the program or for those cases where error messages are useless. An example: suppose that a victim program creates another internal thread which closes the main thread and continue running from that thread or from that thread launches another instance of itself (it's a quite common custom protection). In this case the loader couldn't be able to suspend the thread/process because it would not be active anymore. An error message will be issued. But anyway properly writing the `GateProcedure()` the loader would still work (for example waiting for the main victim's windows to appear) and the error would be not meaningful. In this case you would use the `SILENT_MODE` set to TRUE. By default is set to FALSE!

- `void SetStartingMsg()`. Used to modify the starting message of the loader. If not used the loader uses a standard string. This function should be called for example in the `SetVictimDetails` method or in the derived class constructor.

- `int PushPatchVector(growing_arraystack<Patch> &stkPatches, DWORD startAddr, BYTE *OriVector, BYTE *PatchVector, int dimension, fcnPatchCallBack fcn )`. Add a whole Vector of patch data. This

function takes two BYTEs vectors and pushes each value to a stack of Patch objects, the `fcnPatchCallBack` is applied to the last Patch of the vector, so as eventually the action is performed at the end of the operation.

Input parameters:

- `stkPatches` stack of Patch elements where the values are pushed
- `startAddr`, the address where the vector starts.
- `OriVector`, vector of original bytes, if NULL Patch objects will not check original values
- `PatchVector`, vector of new bytes
- dimension, dimension of the vector (the two vectors should be long the same)
- `fcn`, this callback will be applied to the first pushed values of the vector (due to the stack logic will be the last applied). It's is simply a function callback which is called after the array of patches has been applied, which allows to perform any custom operation, just after the application of a "mega" patch. Most times for simple loaders is NULL.

Returns 0 if all went fine, otherwise an error code number <0 (see implementation for codes).
This function is extremely useful when you have to patch several consecutive bytes, so you might write a piece of code such this:

```
<-------------Code Snippet------------->
typedef void (*fcnPatchCallBack) (DWORD addr);
fcnPatchCallBack fcn(DWORD addr) {
    char str[256];
    sprintf(str,"Patch applied at address %X", addr);
    ::MessageBox(NULL, str, DEFAULT_MSG_CAPTION,
    MB_OK|MB_ICONEXCLAMATION|MB_APPLMODAL);
    }

BYTE iPatchDataInj[87] ={
     0x90, 0x90, 0x90, 0x90, 0x90, 0xE9, 0x01, 0x00, 0x00, 0x00, 0xBC, 0x8B, 0x45, 0xFC, 0x8B,
     0x40, 0x14, 0xE8, 0xC2, 0x21, 0xED, 0xFF, 0x8B, 0x45, 0xFC, 0x8B, 0x58, 0x1C, 0x85, 0xDB,
     0x74, 0x10, 0x8B, 0xC3, 0xE8, 0x01, 0xB1, 0xF5, 0xFF, 0x8B, 0xD0, 0x8B, 0xC3, 0xE8, 0x84,
     0xB3, 0xF5, 0xFF, 0x8B, 0x45, 0xFC, 0x8B, 0x40, 0x20, 0x85, 0xC0, 0x74, 0x07, 0x33, 0xD2,
     0xE8, 0x17, 0x7C, 0xE8, 0xFF, 0xC6, 0x45, 0xFB, 0x01, 0x8B, 0x45, 0xFC, 0xC6, 0x40, 0x19,
     0x00, 0xEB, 0x04, 0x80, 0x8E, 0x8C, 0x06, 0x90, 0x90, 0x90, 0x90, 0x90 };

PushPatchVector(stkPatches, 0x005C684C, NULL, iPatchDataInj, 87, fcn);
<-------------End Code Snippet------------->
```

This piece of code applies the whole matrix (87 bytes) of values starting from the address `0x005C684C` and at the end calls the function `fcn`, which shows a messagebox.

- `BOOL ReadProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPVOID lpBuffer, DWORD nSize, LPDWORD lpNumberOfBytesRead).`
  `BOOL WriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPVOID lpBuffer, DWORD nSize, LPDWORD lpNumberOfBytesWritten).`
  These two reflectors of the similar methods of the BMG_gsar class (see [1]), allows a controlled access to memory automatically handling right to access memory pages and errors. The two methods behave exactly like their Windows counterparts, and the programmer writing the derived class can use these two functions exactly like in normal code, the C++ inherits properties will call these functions instead. Usually hence there's no need to place further controls when calling these two methods from derived classes.

#### 4.1.2.4 When could happen to dump a big chunk of memory from a process?

A very common case where you have to patch a long vector of consecutive bytes is when you have an asprotected program using an encrypted section of its code, as described in [12], and a valid key for the program (in the so lucky case that you or a friend brought the program). In this case

you already know from [12] that there's no way to decode the encrypted instructions unless you use a brute-force attack. Anyway in this case of course you don't want to share your key, a solution then is to run the program as fully registered and examine its memory. In this case the encrypted sections of the program are completely decrypted resulting in real working code.

What you have to do then is to save into a textual file (using OllyDbg and a tool we did, as shown in section 4.2.1 in following pages) the memory section from the registered program and insert it into a loader which loads the un-registered program (run without the "legal" key) and overwrite the same memory portion, substituting the encrypted memory block with the decrypted one.

### 4.1.3 Loader

| Loader |
|---|
| |
| +Loader()<br>+~Loader()<br>+ActionsAfterCreateProc()<br>+ActionsAfterGateProcedure()<br>+ActionsBeforeClosingLoader()<br>+ActionsBeforeCreateProc()<br>+ActionsBeforeGateProcedure()<br>+GateProcedure()<br>+InitializePatchStack(inout stkPatches : growing_arraystack<Patch>&)<br>+SetVictimDetails(inout victimFileName : TextString&) |

As already told this class should concentrate all the victim's specific things, and should drive the ShubLoaderCore class from which it is derived.

This class usually is derived from of ShubLoaderCore, implements all or some of the parent's virtual methods (depending on the needed customizations), using some of the parent's helping methods.

The better way to describe it is to directly see the sources of a working loader (see section 4.2). Our experience tells that once you wrote a single loader you'd be able to write the following in a snap.

### 4.1.4 Patch Class

| Patch |
|---|
| +address<br>+bytesread<br>+byteswritten<br>+checkorigByte<br>+fcnCallBack<br>+msg<br>+OnlyDoCallback<br>+orig<br>+patch |
| +Patch(in p0 : unsigned long, in p1 : unsigned char)<br>+Patch(in p0 : unsigned long, in p1 : unsigned char, in p2 : unsigned char)<br>+Patch(in p0 : unsigned long, in p1 : unsigned char, in p2 : unsigned char, inout p3 : void (*)(unsigned long))<br>+Patch(in p0 : unsigned long, in p1 : unsigned char, inout p2 : void (*)(unsigned long))<br>+Patch(in p0 : unsigned long, inout p1 : void (*)(unsigned long))<br>+Patch()<br>+~Patch() |

The Patch class is simple in its meaning, it is a class used to store the patch details, composed of offset, original byte and patched byte. The Patch class represents a single byte patch: each object of type Patch represents a single patched byte. There's also the possibility to perform a custom action (callback) for each single patch applied: the framework worries to eventually call the callback after the patch has been applied.

The class has several constructors which are used to perform the different types of patches you can have. Usually all these Patch object are pushed into a Patch object stack, in the `InitializePatchStack` method.

Properties of the class:

- `address` – is the RVA address of the patch
- `byteswritten` – number of bytes written in the process
- `bytesread` – number of bytes read from the process
- `checkorigByte` – flag value used to check against the original byte read from the process.
- `fcnCallBack` – callback called after the patch has been applied, can be different for each single Patch object
- `msg` – a message reporting the result of the patch up to now, it is set by the framework automatically and can be taken to understand the status of a specific patch
- `OnlyDoCallback` – flag to specify to only call the callback and do not write patches. Useful in some cases when you need special actions to be performed.
- `orig` – is the original byte read from the application
- `patch` – is the new byte to substitute

Methods of the class:

- `Patch()` – This one shouldn't ever be used, it' useless. It's present only for C++ syntax.
- `Patch(DWORD addr, BYTE ptc)` – Use this when you want to write a single byte at a specific location, regardless of the original byte.
- `Patch(DWORD addr, BYTE ptc, fcnPatchCallBack fcn)` – Use this when you want to only write a byte at a specified address and perform a callback after.
- `Patch(DWORD addr, BYTE ori, BYTE ptc)` – Use this when you want to also to check the original byte value then patch if matches (otherwise the patch is not applied and the `msg` member is set accordingly).
- `Patch(DWORD addr, BYTE ori, BYTE ptc, fcnPatchCallBack fcn)` – Use this if you want also to call a specified callback after having done a patch.
- `Patch(DWORD addr, fcnPatchCallBack fcn)` – Use this when you want to only do a specific callback without having to read/write anything. This is more or less like a "virtual" patch, where you are not patching anything. Value `addr` is passed to the callback and can be used by this function for whatever scopes you want.

#### 4.1.4.1 Callbacks

The framework uses in different places some callbacks, they always must be functions with a specified prototype or of a specific custom type.
This is the prototype of functions actions that can be performed to any patch.

```
typedef void (*fcnPatchCallBack) (DWORD addr);
```

for example then:

```
fcnPatchCallBack fcn(DWORD addr) {
	//do whatever you like here
}
```

The function receives the address of the patch as unique argument and can then perform any operation you like. The callback mechanism is very powerful and flexible so as there's the possibility to have a single callback for each single patched byte.

## *4.2 How to write a loader using the framework*

We perfectly understand that writing a loader might be simpler than using the framework we are proposing, but the complexity you felt is due to the general approach we wanted to keep. The framework allows you to write very complex loaders without changing a line of the core code. Thus if a simple loader is your target then the framework might be an additional complexity not really needed, but as a matter of facts loaders written using the framework we proposed here are almost always the same (for simple cases) and we found in everyday RCEing that once you took time to write the first loader, it's a snap to write the following, making the effort of writing them to the minimum. So we felt that would have been extremely important to add a section to this long tutorial where to teach a step-by-step process for creating a loader using the framework here proposed. So, go on with another chapter...

Generally speaking the steps to write a loader are:

1. Patch the program using OllyDbg; write down the offsets, the original and the modified bytes (or only the offset and the modified bytes).
2. Calculate the CRC of the victim, for example using the CRCCalculator program we provide in this tutorial's archive.
3. Create a project with Visual C++, generally a DOS CRT Program is enough and shorter than a graphical Win32 program, including all the required sources from the framework
4. Rename the original executable to something else. We're used to rename the original exe as _originalname.exe, placing a leading "_" in the filename.
5. Fill in the main() program.
6. Customize the loader behaviour creating a derived class from ShubLoaderCore, called normally Loader or whatever you like.

Step 1 is easy or not the target of this tutorial, so we will skip them, except for the usage of OllyTranslator. For the step 2 you can use the CRC calculator we provide which is very easy to use, just drag & drop the .exe over it to get the CRC value. Step 3 is given as already known because it is an everyday operation using Visual C++. Step 4 is easy (☺), step 5 is where the things start to be interesting. The 6$^{th}$ is the more complex one..

### 4.2.1 How to use OllyDumpTranslator

This simple program [11] has been made to automatically transform the OllyDump file format (txt format) into a corresponding C patch data matrix, ready to be used for loaders.
This utility is able to take an Olly file, like as this one:

```
005EFD7F  90 90 90 90 90 E9 01 00 00 00 B5 8B C3 E8 0B D2  □□□□□é_...µ‹ÃèÒ
005EFD8F  FF FF EB 04 EA 04 86 E6 90 90 90 90 90           ÿÿë_ê_†æ□□□□□
```

and translate it into this C language slice of code:

```
<-------------Code Snippet------------->
// ======================================
// Olly File Translator 1.0 by ThunderPwr
// 03/03/2005 22.02.03
// translating file utility
// ======================================

#define IMAXINDEXINJ 29// Patch size

// ------------------------------------------------------------------------------------------
// Definition about the addresses where to apply the patches.
// ------------------------------------------------------------------------------------------

DWORD dwPatchaddrInj[IMAXINDEXINJ] = {  0x005EFD7F, 0x005EFD80, 0x005EFD81, 0x005EFD82,
                                        0x005EFD83, 0x005EFD84, 0x005EFD85, 0x005EFD86,
                                        0x005EFD87, 0x005EFD88, 0x005EFD89, 0x005EFD8A,
                                        0x005EFD8B, 0x005EFD8C, 0x005EFD8D, 0x005EFD8E,
                                        0x005EFD8F, 0x005EFD90, 0x005EFD91, 0x005EFD92,
                                        0x005EFD93, 0x005EFD94, 0x005EFD95, 0x005EFD96,

                                        0x005EFD97, 0x005EFD98, 0x005EFD99, 0x005EFD9A,
                                        0x005EFD9B };
// ------------------------------------------------------------------------------------------
// Definition about the patching value
// ------------------------------------------------------------------------------------------
int iPatchDataInj[IMAXINDEXINJ] ={  0x90, 0x90, 0x90, 0x90,
                                    0x90, 0xE9, 0x01, 0x00,

                                    0x00, 0x00, 0xB5, 0x8B,
                                    0xC3, 0xE8, 0x0B, 0xD2,
                                    0xFF, 0xFF, 0xEB, 0x04,
                                    0xEA, 0x04, 0x86, 0xE6,
                                    0x90, 0x90, 0x90, 0x90,
                                    0x90 };
<-------------End Code Snippet------------->
```

You can then directly use the last matrix in the framework, using the `PushPatchVector` method as described in section 4.1.2.3 (in this case the `dwPatchaddrInj` is useless):

```
PushPatchVector(stkPatches, 0x005EFD7F, NULL, iPatchDataInj, IMAXINDEXINJ, NULL);
```

or using a loop, if the patched addresses are not all adjacent:

```
for (int i=0; i<IMAXINDEXINJ; i++)
    stkPatches.push(Patch(dwPatchaddrInj[i], (BYTE)iPatchDataInj[i]));
```

For sake of completeness, the whole process is as follow:

The Ollydbg dump files can be obtained using it like in Figure 14.

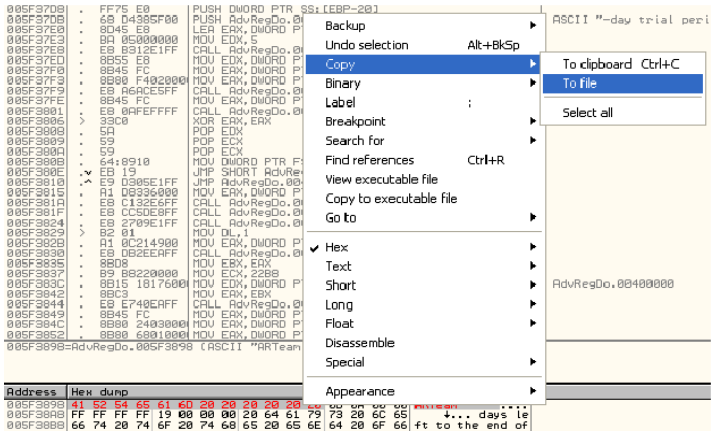



*Figure 14 - how to dump to file a binary section from Ollydbg*

Then launch the OllyDumpTranslator and press buttons 1, choose the file and then 2, as in Figure 15.


*Figure 15 - Main window of OllyDumpTranslator*

The program then creates in the same folder of the original dump file, another file with the same name plus the suffix "_translated"

### 4.2.2 Write the main() function of the loader

This step is always the same and there's need that much to discuss: you have to call the `DoMyJob` method of the Loader class you derived from ShubLoaderCore. I report here and example:

```
<-------------Code Snippet------------->
#include "Loader.h"
int main(int argc, char** argv)
{
        Loader loader;
        int nRetCode=loader.DoMyJob(argc, argv);

        return nRetCode;
}
<-------------End Code Snippet------------->
```

### 4.2.3 Write the derived Loader Class

As you know the C++ classes are divided into a declaration of the class and an implementation. The declaration normally goes into a .h file, while the implementation normally into a .cpp file (could also be in the .h file indeed).

Here below a declaration of the Loader class:

```
<-------------Code Snippet LoaderActions.h------------->
#include "ShubLoaderCore.h"

class Loader: public ShubLoaderCore {
public:

        Loader();
        ~Loader();

        BOOL SetVictimDetails(/*OUT*/ TextString &victimFileName);
        BOOL InitializePatchStack(/*OUT*/ growing_arraystack<Patch> &stkPatches);
        BOOL ActionsBeforeCreateProc();
        BOOL ActionsAfterCreateProc();
        BOOL ActionsBeforeGateProcedure();
        BOOL GateProcedure();
        BOOL ActionsAfterGateProcedure();
        BOOL ActionsBeforeClosingLoader();

};
<-------------End Code Snippet LoaderActions.h------------->
```

As described in section 4.1.2 the class redefine the virtual methods of ShubLoaderCore being publicly derived from it.

Here below the implementation of the Loader class (the patches values don't refer actually to any real application):

```
<-------------Code Snippet LoaderActions.cpp------------->
#include "LoaderActions.h"

////////////////////////////////////////////////////////////////////////

Loader::Loader()
{
        //TODO: insert specific actions if you require additional initialization
        SetStartingMsg("Loader working...wait a little\nCreditz 2 Shub-Nigurrath & ThunderPwr [at] ARTEam");
}

Loader::~Loader()
{
        //TODO: insert specific actions if you require additional de-initialization
}

////////////////////////////////////////////////////////////////////////

//Receives
//- the Stack of Patch elements that must be properly filled. The variable to use is stkPatches!
//- the victim file name, containing a valid path to the patched file
BOOL Loader::InitializePatchStack(growing_arraystack<Patch> &stkPatches)
{
        ////////////////////////////////////////////////////////////////////////
        // This is the filling of the patches stack.
        // you can use one of the constructors available.
        // - The first only requires the patch address and the new byte so no controls will be
        //   performed later, the loader will only do a simply write to that memory section,
        //   regardless of the read value.
        // - The second way, used here is to also add the original bytes, doing so the loader
        //   will also check if the byte read at the memory location specified is equal to the
        //   original byte you expected to be there. If not the patch is not applied and the msg
        //   buffer is set according.
```

```
	// - The third one allows to specify a callback which is called when trying to perform
	//   the patch.
	// Note that the patches are all applied subsequently after the gate condition is met
	// (see GateProcedure())

	//NB 0x00 must explicitly be casted to BYTE because otherwise the complier confuses
	//it with a NULL pointer and doesn't know which constructor of class Patch to use.

	// Example patches which also checks against the original bytes. If the original byte is
	// different the Loader will issue and error BEFORE applying the patch
	stkPatches.push(Patch(0x0044337C, 0x74, 0xEB));
	stkPatches.push(Patch(0x005E5669, 0x75, 0xEB));
	stkPatches.push(Patch(0x005F1552, 0x75, 0xEB));
	stkPatches.push(Patch(0x005E626E, 0x75, 0xEB));
	stkPatches.push(Patch(0x005E67D0, 0x75, 0xEB));
	stkPatches.push(Patch(0x005E6921, 0x7E, 0xEB));

	//Example of patches which don't check against the original bytes
	stkPatches.push(Patch(0x005F3898, 0x41)); //A
	stkPatches.push(Patch(0x005F3899, 0x52)); //R
	stkPatches.push(Patch(0x005F389A, 0x54)); //T
	stkPatches.push(Patch(0x005F389B, 0x65)); //e
	stkPatches.push(Patch(0x005F389C, 0x61)); //a
	stkPatches.push(Patch(0x005F389D, 0x6D)); //m
	stkPatches.push(Patch(0x005F389E,(BYTE)0x00)); //end string

	//0x00 must explicitly be casted to BYTE because otherwise the complier confuses
	//it with a NULL pointer and doesn't know which constructor of class Patch to use.
	stkPatches.push(Patch(0x005F37C2, 0x75, (BYTE)0x00));

	//Injected code sections.
	BYTE iPatchDataInj[87] ={
		0x90, 0x90, 0x90, 0x90, 0x90, 0xE9, 0x01, 0x00, 0x00, 0x00, 0xBC, 0x8B, 0x45, 0xFC,
		0x8B, 0x40, 0x14, 0xE8, 0xC2, 0x21, 0xED, 0xFF, 0x8B, 0x45, 0xFC, 0x8B, 0x58, 0x1C,
		0x85, 0xDB ,0x74, 0x10, 0x8B, 0xC3, 0xE8, 0x01, 0xB1, 0xF5, 0xFF, 0x8B, 0xD0, 0x8B,
		0xC3, 0xE8, 0x84, 0xB3, 0xF5, 0xFF, 0x8B, 0x45, 0xFC, 0x8B, 0x40, 0x20, 0x85, 0xC0,
		0x74, 0x07, 0x33, 0xD2, 0xE8, 0x17, 0x7C, 0xE8, 0xFF, 0xC6, 0x45, 0xFB, 0x01, 0x8B,
		0x45, 0xFC, 0xC6, 0x40, 0x19, 0x00, 0xEB, 0x04, 0x80, 0x8E, 0x8C, 0x06, 0x90, 0x90,
		0x90, 0x90, 0x90 };
	PushPatchVector(stkPatches, 0x005C684C, NULL, iPatchDataInj, 87, NULL);

	return TRUE;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

//Simply used to specify the victim's filename, received the storing variable.
BOOL Loader::SetVictimDetails(TextString &victimFileName)
{
	victimFileName=TextString(".\\_TargetProgram.exe");

	//Set this parameter to true when you want the loader to check the CRC of the file!
	SetVictimCRC(0x8281dfe6);

	return TRUE;
}

// It is called just before calling the GateProceduce, then should contain steps required to perform the action or
special settings..
BOOL Loader::ActionsBeforeGateProcedure()
{
	return TRUE;
}

// The function GateProcedure must always be defined with this prototype.
// Returned value is TRUE when the matching condition required to start the patch is met.
// Often this function simply checks against a specified DWORD value in a specified
// memory location or the presence of a specific window, after which the patch can be successfully
// applied.
BOOL Loader::GateProcedure()
{
	BOOL bRet=FALSE;
	//Enum all the windows starting from the desktop, one by one, also the
```

```
        //hidden windows. Each handle is passed to EnumWindowsProc which decides
        //what to do with that handle. Actually it returns if it's the victim's window.
        EnumDesktopWindows(NULL, EnumWindowsProc, (LPARAM)&bRet);

        return bRet;
}

BOOL Loader::ActionsAfterGateProcedure()
{
        //Stop debugger action and let program run freely
        DWORD dwProcessId = GetProcessId(GetPI()->hProcess);
        BOOL  bDbgStopFlag = DebugActiveProcessStop(dwProcessId);

        return TRUE;
}

//This function is called just before the call to CreateProcess. Could be left empty.
BOOL Loader::ActionsBeforeCreateProc()
{
        return TRUE;
}

//This function is called just before the process has been created but it is still in waiting mode
BOOL Loader::ActionsAfterCreateProc()
{
        HideDebugger(GetPI()->hThread, GetPI()->hProcess);
        return TRUE;
}

//This function is called just before closing the loader, after all the actions have been performed.
BOOL Loader::ActionsBeforeClosingLoader()
{
        return TRUE;
}

////////////////////////////////////////////////////////////////////////////
//Callback of EnumDesktopWindows
BOOL CALLBACK EnumWindowsProc(
                  HWND hWnd,      // handle to parent window
                  LPARAM lParam   // application-defined value
                  )
{
        char ClassName[256];

        //Retrieve the classname of the given handle
        GetClassName(hWnd,ClassName, 256);
        char caption[256];
        //Retrieve the caption of the given handle
        GetWindowText(hWnd, caption,256);

        //Check of the window I want to find, It's specific of the application
        //We have to wait till the window is visible because all the checks happens before
        //this point.
        if(strstr(caption,"Application titlebar")!=0 &&
                IsWindowVisible(hWnd) &&
                _stricmp(ClassName,"TMainForm")==0)
        {
                //a little of tricky casting required to return the final BOOL to the caller,
                //via an LPARAM parameter, which after all is a generic LPVOID.
                BOOL *flag=(BOOL*)lParam;
                *flag=TRUE;
                return FALSE;
        }
        return TRUE;
}
<-------------End Code Snippet LoaderActions.cpp------------->
```

The Loader class implementation is not that difficult, being a derived class of `ShubNigurrathCore` and `NTInternals` (see Figure 12), can use directly all their public methods, without special notations (see the C++ inherit proprieties).

Note that the `DebugActiveProcessStop` called in the code above is not the real Windows API (which is available only since Windows XP), but rather the method exposed by the class `NTInternals` (see [1] or section 4.1.1) which also ensure Loader's compatibility with Windows 9x/NT/2000. In those cases it will simply do nothing at all.

In the above example we have a `GateCondition` testing the presence of a specific window (with a specific class and title). The situation is quite common, because even hardly compressed programs (with AsProtect for example) often do all their checks during decompression, inside the AsProtect code. When the first target's window appears (often not visible) the program is completely unprotected in memory (most programs doesn't have anti-tampering protections, see [10]) and can be patched by the loader.

In order to obtain the window's details inside OllyDbg when you are at the OEP (uncompressing inside OllyDbg, at the last exception for AsProtect) see the list of handles belonging to the target and choose the right one.

| **NOTE** |
| --- |
| We successfully tested the trick (wait for a given window before doing the patch) on several targets protected with AsProtect. The resulting loader is much smaller and compatible with all the Windows versions, because the used APIs are available since Windows 9x. Debugger loaders might have some problems on older Windows versions.<br>The framework we did accomplish all the compatibility problems but simply not doing specific operations on not supported Windows. As a result the loader will not crash the system, but might not work as expected. |

## *4.3 Writing a Debugger Loader using the framework*

As told at the beginning the Debugger Loader are special loaders which interact with the target application like debuggers. We already described the essential things you should know (a complete description would take too much) and we are going now to write the skeleton of a debugger loader which you'll be able to reuse (it's also included in the tutorial's archive). The steps are the same used in section 4.2, what differs mostly is the `GateCondition` which is a little more complex.

Theory of the `GateCondition` is the same described in section 2.1.2 and following.

I report here the main differences with the code of section 4.2.3

```
<-------------Code Snippet LoaderActions.cpp------------->
//Simply used to specify the victim's filename, received the storing variable.
BOOL Loader::SetVictimDetails(TextString &victimFileName)
{
        victimFileName=TextString(".\\_TargetProgram.exe");

        //Set this parameter to true when you want the loader to check the CRC of the file!
        SetVictimCRC(0x8281dfe6);

        SetCreateProcessFlags(DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS | CREATE_NEW_CONSOLE);

        return TRUE;
}
<-------------End Code Snippet LoaderActions.cpp------------->
```

Note the `SetCreateProcessFlags` which was not called before, because by default the process is created as SUSPENDED. These parameters are useful to create the process in debug mode.

```
<-------------Code Snippet LoaderActions.cpp------------->
BOOL Loader::GateProcedure()
{
	BOOL bRet=FALSE;

	DEBUG_EVENT DebugEv;                    // debugging event information
	DWORD dwContinueStatus = DBG_CONTINUE; // exception continuation

	// Define the CONTEXT structure used to load the victim process context
	// when debugged process break due to exception event
	CONTEXT victimContext;
	int iExceptionCounter = 0;

	BYTE OridataRead[2];

	try {
		for(;;)
		{
			// Wait for a debugging event to occur. The second parameter indicates
			// that the function does not return until a debugging event occurs.
			// We are waiting for infinite time, then wait for each Debug Event.
			WaitForDebugEvent(&DebugEv, INFINITE);

			// 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.

					// Increment exception counter (not used)
					iExceptionCounter++;

					#ifdef _DEBUG
					// Show the current exception number
					char str[256];
					sprintf(str,"Exception number %d", iExceptionCounter);
					::MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
					#endif

					// Check if this is the right exception by reading the context
					// structure for the victim process. Before to do it set the
					// ContextFlags to READ_ALL
					victimContext.ContextFlags = 0x1003F;

					// Fill the process CONTEXT with the process information
					GetThreadContext(GetPI()->hThread , &victimContext);

					// Now I've to scan the process memory in order to see if I
					// can found the PUSH 0C instruction (19 byte after exception)
					ReadProcessMemory(GetPI()->hProcess,
						(LPVOID)((victimContext.Eip) + 19),
						OridataRead, 2, NULL);

					//6A 0C PUSH 0C
					if ((OridataRead[0] == 0x6A) && (OridataRead[1] == 0x0C))
					{
						// Key location found, now we can apply the patch
						#ifdef _DEBUG
						char str[256];
						sprintf(str,"Found PUSH 0C location");
						MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
						#endif

						throw TRUE; //jump to the catch block at the end
					}

					// Debugger's Exception handler
```

```
                switch(DebugEv.u.Exception.ExceptionRecord.ExceptionCode)
                {
                        case EXCEPTION_ACCESS_VIOLATION: {
                                // First chance: Pass this on to the system.
                                // Last chance: Display an appropriate error.
                                dwContinueStatus = DBG_EXCEPTION_NOT_HANDLED;
                        }
                        break;

                        case EXCEPTION_BREAKPOINT: {
                                // First chance: Display the current
                                // instruction and register values.
                        }
                        break;

                        case EXCEPTION_DATATYPE_MISALIGNMENT: {
                                // First chance: Pass this on to the system.
                                // Last chance: Display an appropriate error.
                        }
                        break;

                        case EXCEPTION_SINGLE_STEP: {
                                // First chance: Update the display of the
                                // current instruction and register values.
                        }
                        break;

                        case DBG_CONTROL_C: {
                                // First chance: Pass this on to the system.
                                // Last chance: Display an appropriate error.
                        }
                        break;

                        default: {
                                // Handle other exceptions.
                        }
                        break;
                }
        }

        case CREATE_THREAD_DEBUG_EVENT: {
                // As needed, examine or change the thread's registers
                // with the GetThreadContext and SetThreadContext functions;
                // and suspend and resume thread execution with the
                // SuspendThread and ResumeThread functions.
        }
        break;

        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.

                dwContinueStatus = DBG_CONTINUE;
        }
        break;

        case EXIT_THREAD_DEBUG_EVENT: {
                // Display the thread's exit code.
        }
        break;

        case EXIT_PROCESS_DEBUG_EVENT: {
                // Target Process is closed from user, then we have
                // to stop the debugger work and exit from loader
                // Exit form loader
                ContinueDebugEvent(DebugEv.dwProcessId,DebugEv.dwThreadId,
                        DBG_CONTINUE);
```

```
                                throw FALSE;
                        }
                        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.
                        }
                        break;

                        case UNLOAD_DLL_DEBUG_EVENT: {
                                // Display a message that the DLL has been unloaded.
                        }
                        break;

                        case OUTPUT_DEBUG_STRING_EVENT: {
                                // Display the output debugging string.
                        }
                        break;

                }

                // Resume executing the thread that reported the debugging event.
                ContinueDebugEvent(DebugEv.dwProcessId,DebugEv.dwThreadId, dwContinueStatus);
        } //end for(;;)

    } //end try

    catch(BOOL bRet) {
            return bRet; //gate condition met, returns to the framework!
    }
}
<-------------End Code Snippet LoaderActions.cpp------------->
```

The `GateCondition` here presented has a quite general structure. If you study the code, you might see that essentially its core is a switch-case-break construct where all the types of debug events are mapped. Several of the "`case`" present are not used indeed in our example; we wrote them anyway to help you understanding where your loaders can place controlling actions following to specific exceptions the target might raise and also to understand which exception types you can catch[2].

The whole switch is inserted into an endless for loop (`for(;;)`) and then into a try-catch block. So to exit from this function there are several ways, but the safer one we decided to implement is to throw an exception. When the condition you are searching for (when the loader can apply patches) is met, you should throw a TRUE value (`throw TRUE;`) caught by the final "`catch`" statement. This ensures a correct stack unwinding and a safer returning from the deepest levels of the `GateCondition`.

A special "case" is the `EXCEPTION_DEBUG_EVENT` which includes another switch-case-break construct used to differentiate among the different debug exceptions might happens.

A little of explanation for this specific example is also needed: the real core of the `GateCondition` is on the `EXCEPTION_DEBUG_EVENT` exception. The code there is thought for a generic AsProtected program with versions 1.2x and earlier.

```
<-------------Code Snippet------------->
// Process the exception code. When handling
// exceptions, remember to set the continuation
```

[2] Of course these are not all the possible exceptions this structure can catch, if you target plays with custom exceptions, it's simple to add them to the loader.

```
// status parameter (dwContinueStatus). This value
// is used by the ContinueDebugEvent function.

// Increment exception counter (not used)
iExceptionCounter++;
#ifdef _DEBUG
// Show the current exception number
char str[256];
sprintf(str,"Exception number %d", iExceptionCounter);
::MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
#endif

// Check if this is the right exception by reading the context
// structure for the victim process. Before to do it set the
// ContextFlags to READ_ALL
victimContext.ContextFlags = 0x1003F;

// Fill the process CONTEXT with the process information
GetThreadContext(GetPI()->hThread , &victimContext);

// Now I've to scan the process memory in order to see if I
// can found the PUSH 0C instruction (19 byte after exception)
ReadProcessMemory(GetPI()->hProcess, (LPVOID)((victimContext.Eip) + 19), OridataRead, 2, NULL);

//6A 0C PUSH 0C
if ((OridataRead[0] == 0x6A) && (OridataRead[1] == 0x0C))
{
        // Key location found, now we can apply the patch
        #ifdef _DEBUG
        char str[256];
        sprintf(str,"Found PUSH 0C location");
        MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
        #endif

        throw TRUE; //jump to the catch block at the end
}
<-------------End Code Snippet LoaderActions.cpp------------->
```

Briefly, the `GateCondition` is trapping all the exceptions coming from AsProtect (case `EXCEPTION_DEBUG_EVENT`), till the last one, which is recognized because there's a PUSH 0C instruction just near the EIP address. Then the condition is met and you can jump out of the debugger's cycle and leave the patcher the rest of the work. Our code is placed at the generic event `EXCEPTION_DEBUG_EVENT` and not for a specific exception type, in order to support all the possible AsProtect exceptions sequences.

Some specific tutorials will follow focusing on writing a loader for the different AsProtect versions or different packers.

**NOTE**

As a general note to the wondering one might have here, we want to tell that you can also patch the program even if the target has some anti-tampering protection in memory, or some memory CRC on the target's code, preventing an easy modification of the process's memory.
The Set/GetThreadContext APIs allow getting and setting all the flags and registry at a given point of execution. So for example the result of a TEST or a CMP may be changed handling the context at a given time (exactly how you do using OllyDbg). The problem of course is to suspend the target program at the right point, passing control the loader. How to do this is not the scope of the present document, anyway generally speaking a smart loader for a complex program may be built without modifying a single byte of code either in memory or on the disk.

For a complete debug cycle refers also to [XX]

Generic method to fish serials from a VB Application
What we will explain here is another nice application loaders will be able to do. We are building this approach without the framework just presented, to show you a comparison of complexity needed to write an articulated loader, performing complex debugging interactions with the target.
The sources are included and explained below in the most crucial parts. Consider anyway that the approach for this specific loader is general for any VB application, because the API that we are trapping is always used by VB programs to compare strings, and often serials (see [13], [14]).
Of course the same program could be coded as well, but we left it out intentionally .. you now have an exercise to do on your own! ^__^

This argument falls into an appendix because it's a little "ancillary" to the main argument of the present tutorial, but not less important or easy!

# 5 Finding the right module and placing a breakpoint

When the first event about the process creation is finished the system loads all the modules used by the target, for each dynamic-link library (DLL) that is currently loaded into the address space of the target process, the system sends a LOAD_DLL_DEBUG_EVENT debugging event.

| **NOTE** |
| --- |
| From above consideration again we point to the fact that each module is loaded into the address space of the target process not into the address space of the debugger, this is more important when we have to set the breakpoint into the target module. |

After all of this is done, the system resumes all threads in the process. When the first thread in the process resumes, it executes a breakpoint instruction that causes an EXCEPTION_DEBUG_EVENT debugging event to be sent to the debugger. All future debugging events are sent to the debugger by using the normal mechanism and rules.

Because our goal is to set a breakpoint into some API function exported from some DLL used by the target now we have to:

1. look for the target module and keep the base address
2. retrieve the function address inside the module and place a breakpoint into the function EP
3. wait when the target call our function by a breakpoint event send from the system to the debugger
4. doing what you want to do in order to patch…
5. restore the EP of the target function
6. back to the victim target and leave it run freely

We have then to look when we are into the system breakpoint, this is the first EXCEPTION_BREAKPOINT event and we can store this by using a flag:

```
<-------------Code Snippet CrackMe.cpp----------------->
                                                case EXCEPTION_BREAKPOINT:
        // First chance: Display the current instruction and register values.
        // This exception will be called during the system breakpoint, then we have to
        // check about system breakpoint, if yes we have to place a breakpoint
        // into the target module and target exported function
       if (!bSystemBreakpoint)
                                                                {
                                                        // Enumerate all module
            iVictimDLLBaseAddress = EnumAllProcesModule( aVictimProcessId,
```

```
                                        szVictimDLLname,
                                        bDebugStage);
            if (!iVictimDLLBaseAddress){
        sprintf(szMsgText,"Can't read proces module\n");
               MessageBox(NULL, szMsgText, szMsgCapt, MB_OK);
               return;
            }
        ...
<-------------End Code Snippet------------->
```

by using the custom function ***EnumAllProcesModule*** we can search and retrieve the base address for the target module, below the code for this function:

```
<-------------Code Snippet CrackMe.cpp----------------->
// ------------------------------------------------------------------------------------------------
// EnumAllProcesModule routine
// ------------------------------------------------------------------------------------------------
FARPROC EnumAllProcesModule(DWORD, char *, BOOL);
FARPROC EnumAllProcesModule(DWORD aVictimProcessId, char * VictimDLLNamePtr, BOOL bDebugFlag)
{
        HANDLE hTmpProcess;    // Handle used when target is open by OpenProcess
        HMODULE hMods[1024];   // Structure filled by all modules base address handle
        DWORD cbNeeded;
        char szModName[MAX_PATH];

        unsigned int j;

        hTmpProcess = OpenProcess( PROCESS_ALL_ACCESS, FALSE, aVictimProcessId );
        if   ( pEnumProcessModules(hTmpProcess, hMods, sizeof(hMods), &cbNeeded))
               {
                                                for ( j = 0; j < (cbNeeded / sizeof(HMODULE)); j++ )
                  {
                    // Get the full path to the module's file.
                    if ( pGetModuleBaseName( hTmpProcess, hMods[j], szModName, sizeof(szModName)))
                       {
                         if (bDebugFlag)
                             printf("\t%s\t(0x%08X)\n", szModName, hMods[j] );
                         if (!strcmp(szModName,VictimDLLNamePtr)) // I've find the module
                             return( (FARPROC)hMods[j]);
                        }
                  }
                }
         CloseHandle( hTmpProcess );  // Close the handle to the process previously opened
         return(0);
}
<-------------End Code Snippet------------->
```

first we have to find a way to refer the victim process space and this is easily obtained by using the ***OpenProcess*** API function (with all the process access rights set by using the PROCESS_ALL_ACCESS flag) which is able to opens an existing process object (in other word a running process). After that we can enumerate all the modules running into the opened process by using the ***EnumProcessModule*** and ***GetModuleBaseName*** API functions.

About ***EnumProcessModule*** we have pinpoint just one note, this API need the *hMods* size to be set large enough to store all the possible modules loaded by the process; a value equal to 1024 is sufficient for almost all applications.

Each element of this array is related to each module loaded into the target process space and it is equal to the base address for each module. Base address is really important in our analysis because starting from it we can find the right address into the target space where we have to place the breakpoint.

Now we have to find the real address (then into the target space) for some function which is inside the searched module.

To do this we use a trick based on a simple consideration: when a DLL is loaded into the process space this is a copy from the original DLL that must reside into the system directory (e.g. SYSTEM32) or into the target process folder or in some other well know process path. The main difference from two copy of the same DLL for different process is into the module base address because the system can place this module into the process space regard the position that we can have in other process address space (relocation), but the fixed point is the offset for the exported function which is the same in each module due to the copy nature.

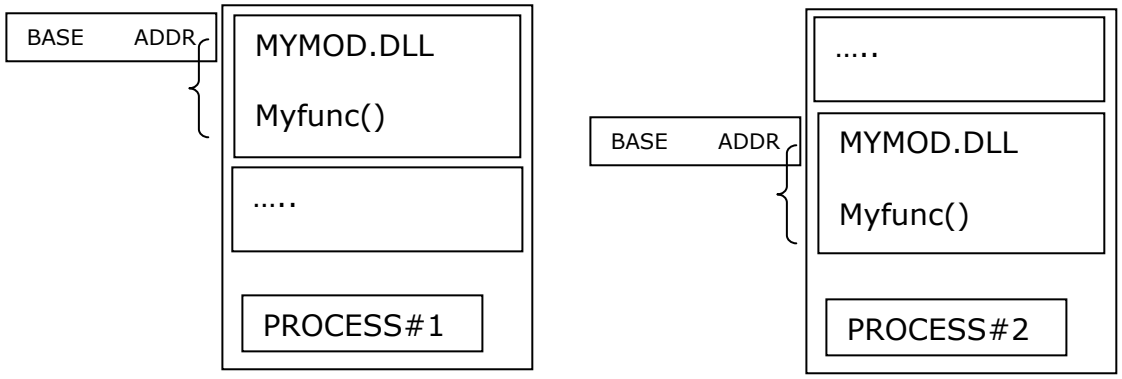



*Figure 16 - Module mapping in different process.*

Then our work consists in finding the offset from the base address and the exported function by using a copy of the module (in our case a DLL) that can be load into the loader address space, then when we have the offset make the sum from this one and the base address which came from the previous module enumeration in order to obtain the real function EP (entry point) address into the target process space.

Summing up the trick is made of these steps:

7. load the DLL into the Loader and store the hModule (starting address of the library)
8. find in the loader the address of the API we want to hook into the target process through *GetProcAddress* (it's an export of the DLL), and store it in hProc
9. calculate the delta, meant as hDelta=hProc – hModule. hDelta represent the offset where the API starts from the beginning of the DLL
10. sum the hDelta to the handle of the already loaded module into the target process. You will this way find the real address in the target process of the API.
11. Place a breakpoint in the target process in the found address: write an INT3 (0xCC) at the address found (see [15])

```
<-------------Code Snippet CrackMe.cpp----------------->
// LoadLibrary in order to know the function address, the right address
// into the victim memory space can be found by using the right offset from
// the base address and the API address when the same DLL is loaded into the
// loader address space (copy is the same then offset is the same).
// When you've the offset to find the real address into the victim space simply
// add this offset to the base address of the target DLL mapped into the victim space
// base address came from EnumAllProcesModule function.

hDLL = LoadLibrary(szVictimDLLname);          // Base address into the loader space
FARPROC addrIDPBreakpoint;                    // Used to store the real address (victim space)
DWORD apiOffset;                              // Used to store the function offset into the victim DLL

// Load the absolute address for the victim function into the loader space
addrIDPBreakpoint = GetProcAddress(hDLL, szVictimDLLfunc);

// Calculate the function offset (same for loader and victim space)
apiOffset = (DWORD)addrIDPBreakpoint-(DWORD)hDLL;
```

```
// Calculate the real address into the victim space
addrIDPBreakpoint = (FARPROC)((DWORD)iVictimDLLBaseAddress + (DWORD)apiOffset);

if (addrIDPBreakpoint != NULL)
   {
        if (bDebugStage)
        {
                sprintf(szMsgText,"__vbaStrComp Address %X",(DWORD)addrIDPBreakpoint);
                MessageBox(NULL, szMsgText, szMsgCapt, MB_OK);
        }
   }
else {

        sprintf(szMsgText,"Can't place breakpoint");
        MessageBox(NULL, szMsgText, szMsgCapt, MB_OK);
        if (hDLL != NULL)
                FreeLibrary(hDLL);
        CloseHandle(hTmpProcess);
        return;
}
<-------------End Code Snippet------------->
```

Now we have into *addrIDPBreakpoint* the function address into the target process space then we can use ***WriteProcessMemory*** to place a breakpoint (INT3) in this EP in order to stop the process execution when this function will be call.

```
<-------------Code Snippet CrackMe.cpp----------------->
// ------------------------------------------------------------------------
// Now we can open the process to place breakpoint (iPatchData[0] = 0xCC -> INT3
// ------------------------------------------------------------------------
hTmpProcess = OpenProcess( PROCESS_ALL_ACCESS | PROCESS_VM_READ | PROCESS_VM_WRITE,
                           FALSE,
                           aVictimProcessId);
if (!WriteProcessMemory(hTmpProcess, (LPVOID)(addrIDPBreakpoint), &iPatchData[0], 1, NULL))
   {
        ErrorExit("WriteProcessMemory ERROR: ");
        MessageBox(NULL, "I can't write process memory :-(", szMsgCapt, MB_OK);
        if (hDLL != NULL)
           FreeLibrary(hDLL);
        CloseHandle(hTmpProcess);
        return;
   }
bSystemBreakpoint = true;

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

After this all we have finish with the first goal, place a breakpoint into a module into the target process space.

# 6 Waiting and handling the breakpoint event in a real case

Now we have place the breakpoint, our work for now is all, we have to wait when the target use our examined function because when this occur a INT3 instruction will be executed on the function EP, system will stop the process and related thread and return the EXCEPTION_BREAKPOINT event to the debugger by the ***WaitForDebugEvent*** function.

First we have to check if the system breakpoint was done, and this can be made by checking the bSystemBreakpoint flag, after this we have to read the process CONTEXT in order to keep the registers value that can be useful for collect information from our target function, then after all the work is done restore the original code and leave the process run freely.

To show in a real case how to act we coded a simple program coded in Microsoft Visual Basic 6.0 which perform a check from the user input and one hardcoded serial (you can find the source attached to this tutorial) but the main things used to fish the right serial is more general and later you can show how to use this concept for fish a serial directly from the installation stage (again this example was for a VB target).

The CrackMeVB has a simple main dialog where the user must write the serial, checking is made by pushing the "Check it!" button.


*Figure 17 - CrackMeVB*

As usual write some fake serial and push the checking button:


*Figure 18 - CrackMeVB example*

You know the target is coded in VB, then an useful function where we can place a breakpoint is ***__vbaStrComp***, then in our serial sniffer we have to search for the MSVBVM60.DLL module and place a breakpoint into the comparing function.

```
<-------------Code Snippet CrackMe.cpp----------------->
// Now I've to read the stack in order to fish the right serial
// then first I've to keep the process CONTEXT

victimContext.ContextFlags = 0x1003F;            // Select all the flag and register

if (!GetThreadContext( hVictimThreadHandle, &victimContext))
    {
      ErrorExit("GetThreadContext ERROR: ");
      return;
    }
```

```
if (bDebugStage)
    {
      printf("Stack pointer ESP = %X\n",victimContext.Esp);
      printf("Stack pointer EIP = %X\n",victimContext.Eip);
    }

// First we have to keep a process handle
hTmpProcess = OpenProcess( PROCESS_ALL_ACCESS | PROCESS_VM_READ | PROCESS_VM_WRITE,
                        FALSE,
                        aVictimProcessId);

// Read the stack into the ESP+8 address, in this address we have
// the pointer to the first argument which is in UNICODE format
DWORD FakeSerialPtr;
if (!ReadProcessMemory(hTmpProcess,(LPVOID)((victimContext.Esp) + 8),
                    &FakeSerialPtr, sizeof(DWORD), NULL))
{
      ErrorExit("ReadProcessMemory ERROR: ");
      MessageBox(NULL, "I can't read process memory :-(", szMsgCapt, MB_OK);
      CloseHandle(hTmpProcess);
      return;
}
else
{
      if (bDebugStage)
printf("Fake serial pointer: %X\n", FakeSerialPtr);
}

// Read the stack into the ESP+12 address, in this address we have
// the pointer to the first argument which is in UNICODE format
DWORD RightSerialPtr;
if (!ReadProcessMemory(hTmpProcess, (LPVOID)((victimContext.Esp) + 12),
                    &RightSerialPtr, sizeof(DWORD), NULL))
     {
       ErrorExit("ReadProcessMemory ERROR: ");
       MessageBox(NULL, "I can't read process memory :-(", szMsgCapt, MB_OK);
       CloseHandle(hTmpProcess);
       return;
     }
else
     {
      if (bDebugStage)
            printf("Right serial pointer: %X\n", RightSerialPtr);
      }
```

```
// Now we have to collect the serial code byte by using ReadProcessMemory, remember
// this is in UNICODE format then we have to check about the string end, this is easily
// achieved by checking the current data byte, if this is 0 we have reached the string end
int iAddrPtr, iBufferPtr;
iAddrPtr=0;
iBufferPtr=0;
do
{
        ReadProcessMemory(hTmpProcess, (LPVOID)(FakeSerialPtr + (DWORD)iAddrPtr),
                                  &iOridataReadOne, 1, NULL);
        iAddrPtr=iAddrPtr+2;
        szFakeSerial[iBufferPtr++]=iOridataReadOne;
}
while (iOridataReadOne != 0);
szFakeSerial[iBufferPtr]='\0';              // Place the string terminator

// Now we have to read the second serial number (same things as the previous one
iAddrPtr=0;
iBufferPtr=0;
do
{
        ReadProcessMemory(hTmpProcess, (LPVOID)(RightSerialPtr + (DWORD)iAddrPtr),
                        &iOridataReadOne, 1, NULL);
        iAddrPtr=iAddrPtr+2;
        szRightSerial[iBufferPtr++]=iOridataReadOne;
}
while (iOridataReadOne != 0);
szRightSerial[iBufferPtr]='\0';             // Place the string terminator

// Now we have to show our serial fishing to the user :)
sprintf(szMsgText,"\tFirst serial: %s\n\tSecond serial: %s\n\tYou know where is the right serial ;-)\n\tWrite it
down to register appz!",szFakeSerial,szRightSerial);
printf(szMsgText);
MessageBox(NULL, szMsgText, szMsgCapt, MB_OK);

// Before finish we have to restore the original value into the
// target DLL and restore the EIP to the breakpoint address
hDLL = LoadLibrary(szVictimDLLname);
FARPROC addrIDPBreakpoint;
DWORD apiOffset;
                                                                                addrIDPBreakpoint      =
GetProcAddress(hDLL, szVictimDLLfunc);
apiOffset = (DWORD)addrIDPBreakpoint-(DWORD)hDLL;
addrIDPBreakpoint = (FARPROC)((DWORD)iVictimDLLBaseAddress + (DWORD)apiOffset);
if (!WriteProcessMemory(hTmpProcess, (LPVOID)addrIDPBreakpoint, &iOridata[0], 1, NULL))
{
        ErrorExit("WriteProcessMemory ERROR: ");
        MessageBox(NULL, "I can't write process memory :-(", szMsgCapt, MB_OK);
        return;
}

if (hDLL != NULL)
        FreeLibrary(hDLL);
if (hPsapi != NULL)
       FreeLibrary(hPsapi);

// Restore the EIP by writing the right value into the process CONTEXT
victimContext.Eip =(victimContext.Eip) - 1;
if (!SetThreadContext( hVictimThreadHandle, &victimContext))
    ErrorExit("GetThreadContext ERROR: ");

// Close the temp handle for the process
CloseHandle(hTmpProcess);

// Run the victim process
ContinueDebugEvent(DebugEv.dwProcessId,DebugEv.dwThreadId, DBG_CONTINUE);

// Stop debugger action and let program run freely (only for WinXP)
BOOL  bDbgStopFlag = DebugActiveProcessStop(aVictimProcessId);

// Exit from debugger
return;

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

Now the most important parts of the sources are over, so it's time to launch the target and then the loader.
First of all there's a message for the user, we can press the OK button because the victim process (CrackMeVB.exe) is already running:

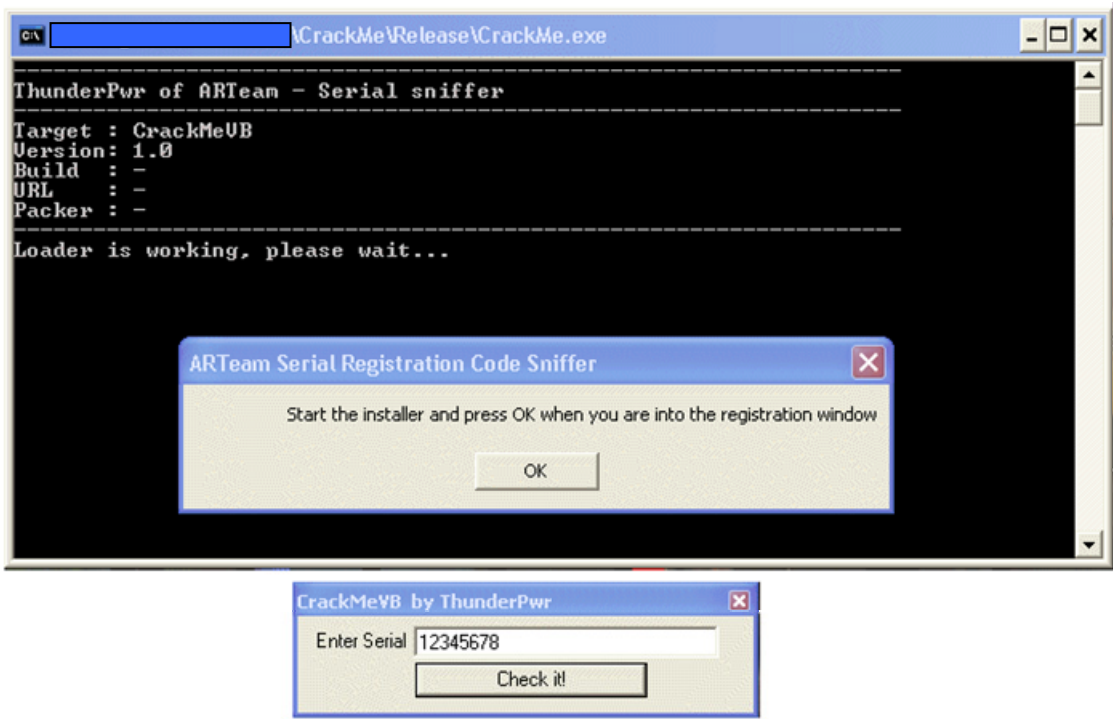



*Figure 19 - Loader in action…*

When you push the OK button a list of all process is shown:

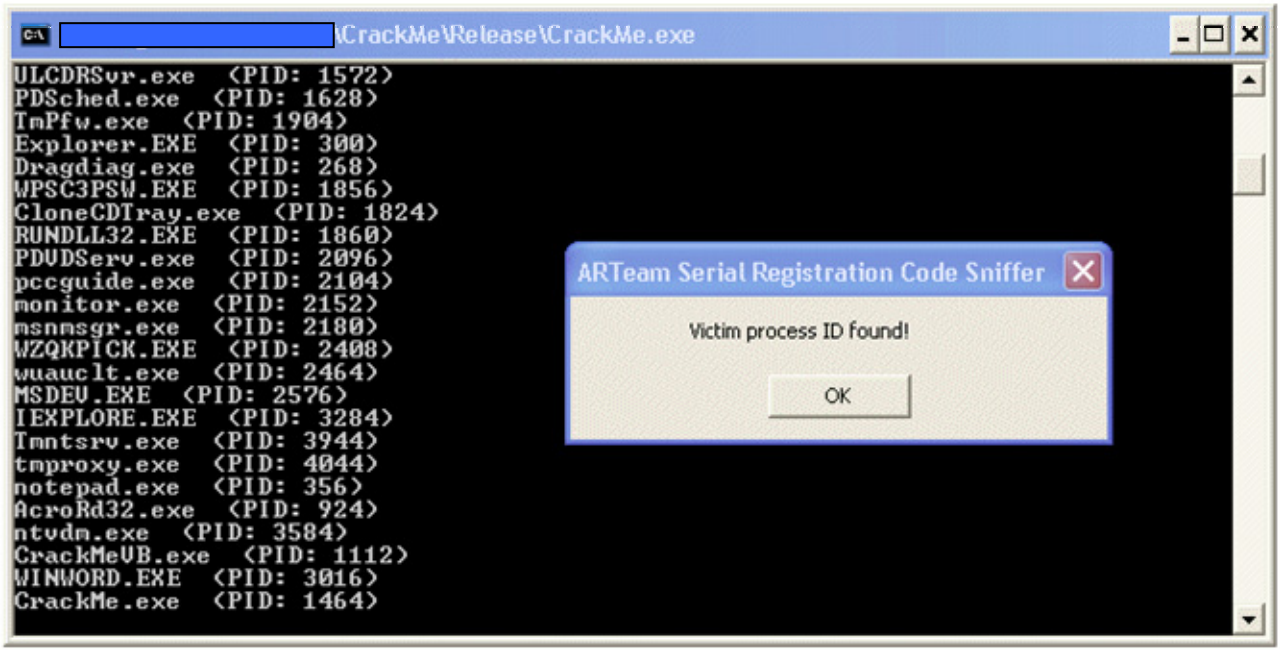



*Figure 20 - The victim process has been found.*

Next step is about the process attach and when all this is done the loader has to do the module enumeration, find the victim module and then the ***__vbaStrComp*** address (the function Entry Point):

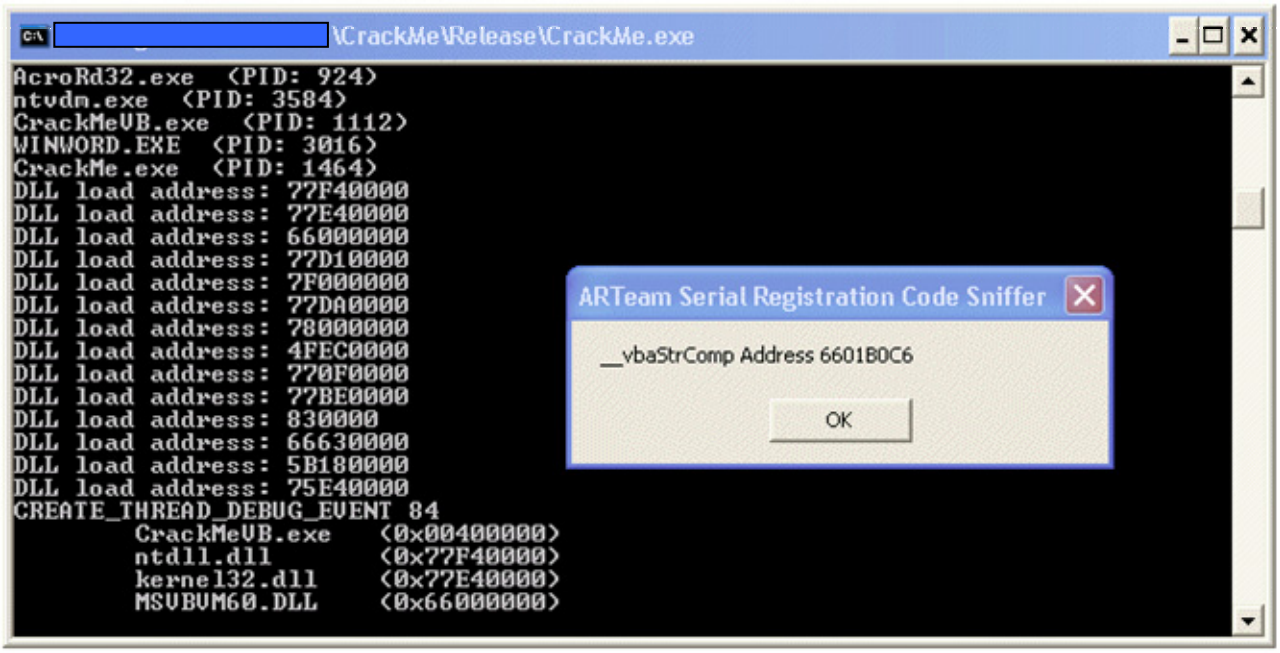



*Figure 21 - Looking for the victim function into the target DLL.*

Press OK, now the target can run freely until you make the serial verification (press again the Check it! button into the CrackMeVB dialog).

Immediately the breakpoint event (INT3) will be triggered by the system to the debugger through the ***WaitForDebugEvent*** function:

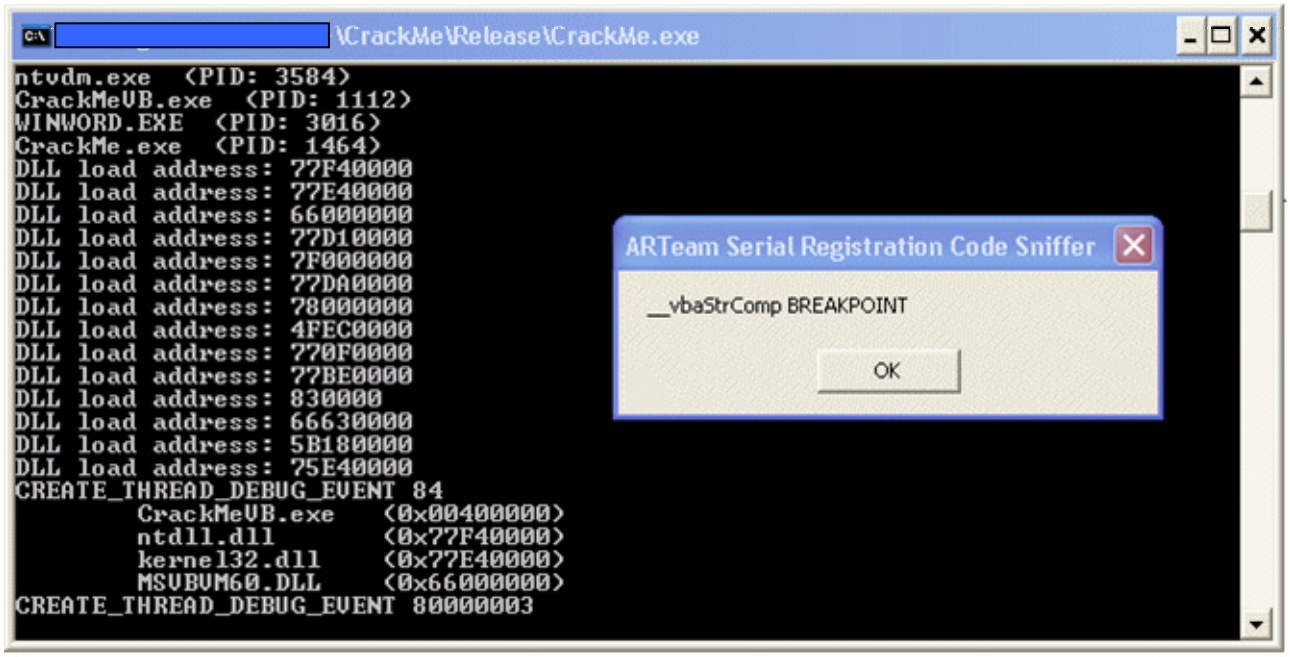



*Figure 22 - The INT3 exception is send to the debugger.*

Push the OK button and we are on the end:

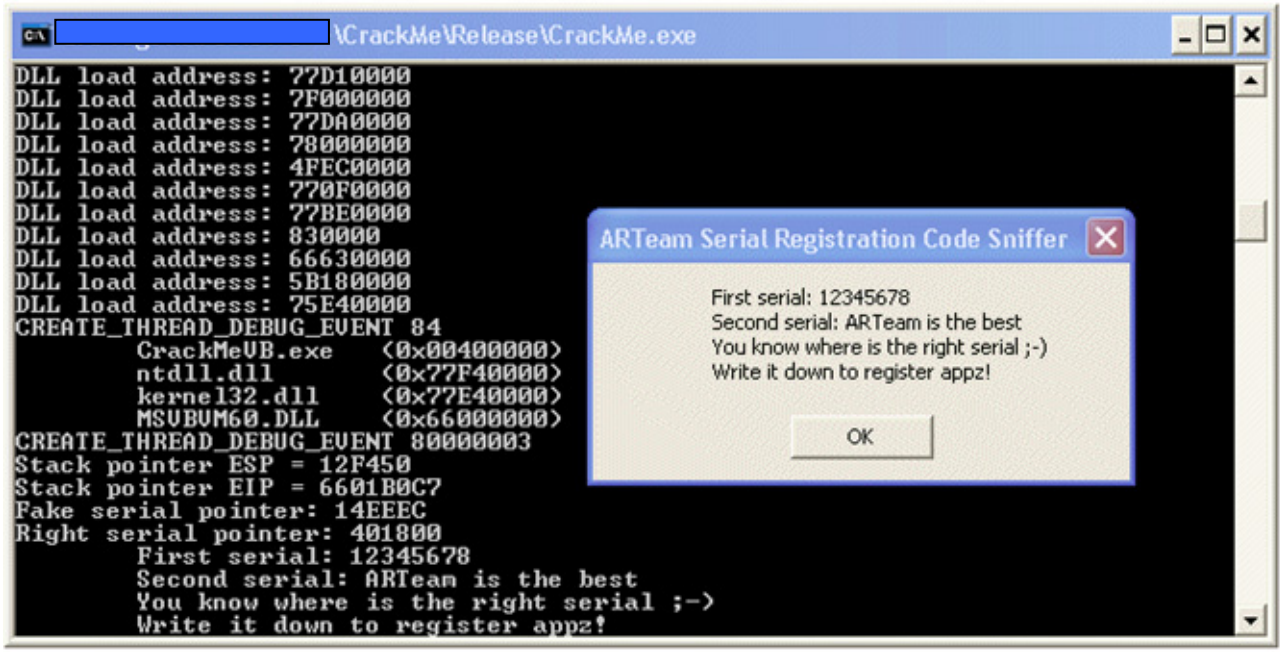
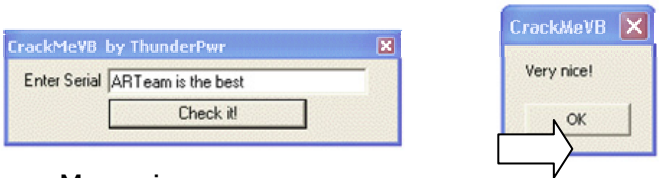



*Figure 23 - Final fishing from the __vbaStrComp function.*

Now is time to check if the fished serial is right if you have some doubt ;-):



Some final word about the target application, this is the source code:

```
<-------------Code Snippet CrackMeVB.frm----------------->
Private Sub Command1_Click()

If ("ARTeam is the best" = txtSerial.Text) Then
    MsgBox "Very nice!"
Else
    MsgBox "Try again!"
End If

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

the argument sequence which is pushed into the stack before call the ***__vbaStrComp*** function is related to the code used to make the verification then:

```
If ( "ARTeam is the best" = txtSerial.Text ) Then
```

gives a different pushed sequence from:

```
If ( txtSerial.Text = "ARTeam is the best") Then
```

## 6.1 Cracking with Olly instead

Below the classical cracking approach using OllyDbg debugger. Just after having loaded the target put a "BP __vbaStrComp" using the Command line plugin and press F9 to run the application. You will land here:

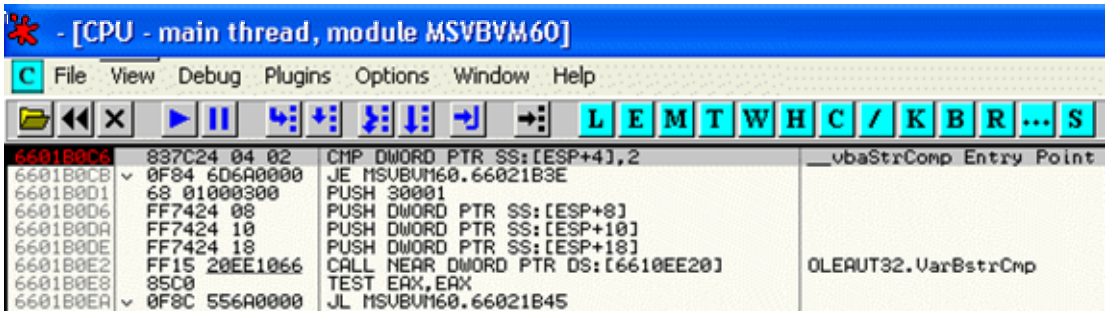



*Figure 25 - OllyDbg breakpoint into the __vbaStrComp function.*

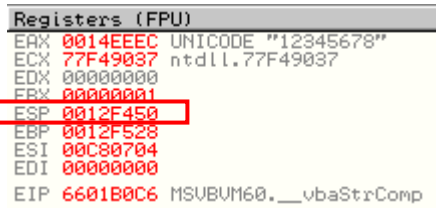



*Figure 26 - Registers window at the breakpoint.*

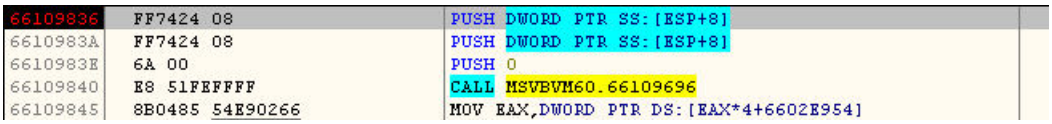

*Figure 27 - Stack window at the breakpoint.*

# 7 Serial fishing example of a real case

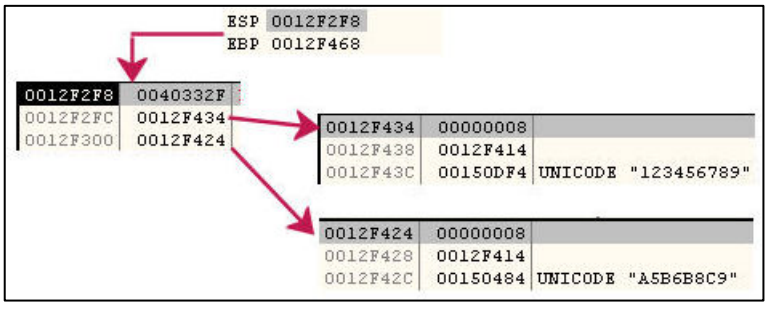
In order to check our theory we can attack for serial fishing a real crackme coded in VB. We will use the same crackme used for [13] (also included in this tutorial), the Abel's 2nd crackme.
For further details see there. The approach to this crackme is interesting because the API used is different than ***__vbaStrComp***. According to analysis described in [13] the API which are used are instead ***__vbaVarTstEq*** or ***__vbaVarTstNe*** (guess what these APIs do). We will modify then the loader just coded to hook the ***__vbaVarTstEq*** API and get the parameters.
First of all this API receives two VARIANT, which are a specific type of strings that are stored in memory in a particular way (string length, address of control, text chars).

Here is how it looks in OllyDbg.

Using the commandline addin place a "BP __vbaVarTstEq" and press F9 to let the program run freely. You will stop here:




And the stack will look similar to the following:




Then what we have to is clear. After the breakpoint has been set, using the already explained code, we will read the two pointers pointed by ESP+4 and ESP+8 (4 is the DWORD size in bytes) and then at an additional offset of 8 from those values we find our UNICODE strings.

What we changed are the program informations:

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

```
// ------------------------------------------------------------------------------------
// General target information
// ------------------------------------------------------------------------------------
char szTargetName[]="abex'2nd crackme";     // Target name
char szTargetVersion[]="1.0";               // Target version
char szTargetBuild[]="-";                   // Target build (if applicable)
char szTargetURL[]="-";                     // Target URL
char szTargetPacker[]="-";                  // Target packer

BOOL bDebugStage = true;              // Set the debug mode (show the message from loader to user)
BOOL bShowExcpNumber=false;           // Exception number flag
BOOL bSystemBreakpoint = false;       // System breakpoint
BOOL bFirstEvent = false;

char szVictimProcessName[]="abexcrackme2.exe"; // Program name
char notloaded[]="Process can be loaded :-(";  // There is one error into loading process stage
char szMsgText[128];                           // Used as a buffer for message to user
char szMsgCapt[]="ARTeam Serial Registration Code Sniffer";
FARPROC iVictimDLLBaseAddress;                // API base address
char szVictimDLLname[]="MSVBVM60.DLL";        // Target DLL
char szVictimDLLfunc[]="__vbaVarTstEq";       // Target function where we have to break
char szFakeSerial[128];                       // Buffer for the fake serial
char szRightSerial[128];
<-------------End Code Snippet------------->
```

Note the different values of *szVictimDLLfunc, szTargetName* and *szVictimProcessName*.

Then the reading of the process's memory, just after the OpenProcess call has changed accordingly to what we saw in Olly

```
<-------------Code Snippet------------->
// Read the pointer to the fake serial
DWORD FakeSerialPtr;
DWORD FakeVariantPtr, RightVariantPtr;

//skip a DWORD
ReadProcessMemory(hTmpProcess, (LPVOID)((victimContext.Esp) + 4), &FakeVariantPtr,
                        sizeof(DWORD), NULL);
ReadProcessMemory(hTmpProcess, (LPVOID)((victimContext.Esp) + 4*2), &RightVariantPtr,
                        sizeof(DWORD), NULL);

ReadProcessMemory(hTmpProcess, (LPVOID)((FakeVariantPtr) + 8), &FakeSerialPtr,
                        sizeof(DWORD), NULL);

if (bDebugStage) {
        printf("Fake serial pointer: %X\n",FakeSerialPtr);
}

// Read the pointer value to the right serial which is in ESP+12
DWORD RightSerialPtr;
ReadProcessMemory(hTmpProcess, (LPVOID)(RightVariantPtr + 8), &RightSerialPtr, sizeof(DWORD), NULL);

if (bDebugStage) {
        printf("Right serial pointer: %X\n",RightSerialPtr);
}
...
<-------------End Code Snippet------------->
```

Just after this we also convert the UNICODE strng into an ANSI one. This thing anyway is always done with VB program and were already in the other crackme.cpp file.

The result is then:


*Figure 28 – Final MessageBox showing the real and the fake serials.*

Those of you who read the tutorial [1] should recognize that what we created is an Oraculum. Oraculums are indeed just specialized loaders which aim is just fishing the real serial for you directly from the program.

# 8 Complete example for a debug-loader cycle

What I introduced before is not a complete cycle of debug a loader can use. There are several events you can handle and to which you can attach you patching actions. In the code reported in the Sections 3.3 and below several has been omitted for sake of brevity. Here instead I report a much more complete source you can use for your own `GateConditions` (thanks also to **condzero**). The whole source is also part of this archive, read it out because including it here would take too much pages.
Anyway you can use that code to see how much additional debug conditions you can add to your `GateCondition` to precisely handle the loading process.

The source included in `Complete_Debug_GateCondition.cpp` is built for a target where what must be patched is one of its DLLs. This time it is a simple C source (but easily convertible to our framework).

What the loader does is to debug the main application (might slow down a little), then apply the patch only when the event `LOAD_DLL_DEBUG_EVENT` is raised.
In this case these are the operations done:

1. extract the name of the DLL being loaded among all those loaded (using `EnumProcessModules` and `GetModuleFileNameEx`)
2. if match the victim that must be patched apply the `isDebuggerPresent` patch (see Section 3.3.1) and patch the DLL location as offset from the base (handle)
3. exits from the debugger and let it run freely.

```
<-------------Code Snippet------------->
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.
        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
```

```
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
        );

        // Display any error msg.
        //MessageBox(NULL, lpMsgBuf, "EnumProcessModules Error", MB_OK+MB_TASKMODAL);

        // 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;


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

                                // 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);

                                                        }
                                                }

                                        }
                                }
                        }
                }

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

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

Once more, as you can see reading the `Complete_Debug_GateCondition.cpp` source, the debugging cycle is quite complex, because there are a lot of cases and nested sub-cases which complicates reading. But consider that once you written it once most of the times you'll re-use the same structure.

Other things that you might find immediately interesting from the above source are:

- **Debugging more than one process at time:**
  when calling the CreateProcess use these parameters..
  ```
  DEBUG_PROCESS,         // No creation flags (use for more than 1 process
  //DEBUG_PROCESS+DEBUG_ONLY_THIS_PROCESS,   //(use for only 1 process)
  ```
- **Avoid locks of debugger loader**
  at the beginning of the debug cycle instead of INFINITE use a timeout, so as to avoid hangs.
  ```
  if (WaitForDebugEvent(&DebugEv, 1000))
  ```
- **How to write current exception address:**
  this for example writes the ExceptionAddress of the ExceptionRecord, but this record contains also a lot of other interesting informations
  ```
  sprintf( b, "Exception address:%08X", DebugEv.u.Exception.ExceptionRecord.ExceptionAddress);
  ```
- **How to print some process' information**
  for example a much more rich printf of process information
  ```
  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
  );
  ```

These are only examples, because as you can see from MSDN and from the code above, the involved structures are one inside the other much like a Matrioska (the Russian dolls) and are very rich of really interesting elements (from the RCE point of view). We think that you can easily find now you own way out of how to print your information and how to write a complex debugger..

| **NOTE** |
| --- |
| The code reported into `Complete_Debug_GateCondition.cpp` is directly compilable, because if you try to compile it there are some misses, some include and libraries declaration, that you must add to your Visual Studio Project. The code is anyway perfectly working. |

# 9 References

There are several tutorial about loader and code injection argument I here will report those I found to be more interesting.


[1] "Guide on How to play with processes memory, write loaders and Oraculums", Shub-Nigurrath of ARTeam, http://tutorials.accessroot.com

[2] "Three Ways to Inject your code into Another Process", Robert Kuster, http://www.codeguru.com/system/winspy.html

[3] "RemoteLib - DLL Injection for Win9x & NT", Abin, http://www.codeproject.com/dll/RemoteLib.asp *[Interesting approach to memory injection into a remote process, which works also for Win9x systems]*

[4] "Injecting a DLL into Another Process's Address Space", Zoltan Csizmadia, http://www.codeguru.com/Cpp/W-P/dll/article.php/c105/

[5] "DLL Injection and function interception tutorial, 2003", CrankHank, , http://www.codeproject.com/dll/DLL_Injection_tutorial.asp

[6] "Creating Loaders & Dumpers - Crackers Guide to Program Flow Control", yAtEs, 2004, http://www.yates2k.net/lad.txt

[7] "9x/NT API Hooking via Import Tables", yAtEs, http://www.yates2k.net/import.html

[8] "Portable Executable File Format Compendium", Goppit, http://tutorials.accessroot.com

[9] "CrackProof your software", Pavol Cerven, Nostarch Press, 2002 (available as bookz)

[10] "Beginner Olly Tutorial #10, Anti-tampering Theory", Shub-Nigurrath of ARTeam, http://tutorials.accessroot.com

[11] "OllyDumpTranslator", ThunderPwr of ARTeam, http://releases.accessroot.com

[12] "Beginner Olly Tutorial #6, Packer's Theory", Shub-Nigurrath of ARTeam, http://tutorials.accessroot.com

[13] "Fishing Primer with SmartCheck", Palaryel, http://tutorials.accessroot.com

[14] "Fishing Primer with SmartCheck number 2", Palaryel, http://tutorials.accessroot.com

[15] "Beginner Olly Tutorial #8, Breakpoints Theory", Shub-Nigurrath of ARTeam, http://tutorials.accessroot.com


# 10 Conclusions

Well, this is the end of this story, we 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 technique. We have also show how to modify code into module which is loaded into the target space.

**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.**