Guide on How to play with processes memory, write loaders and Oraculums

Shub-Nigurrath of ARTeam

Version 1.2 - June 2005

1. Introduction

This tutorial aim is to do a whole flight over loaders, memory patching and how to build them. Told this you might think that there's nothing new in this, because there are several excellent tutorials (not that many anyway) already around, which already cover this argument, but the real final target of this tutorial is to teach how to write an **"Oraculum"**, and to write an Oraculum is impossible without first of all understanding all the things about loaders, processes and memory patching of applications.

At the same time reading this requires a little of knowledge of the C programming language. All the examples I provide have been written in C (and tested using Visual C++ 6.0), but I tried to leave things as much easy as possible.

I must admit anyway that this is a really long tutorial, the longest I ever written, but I wanted to take by hand all the possible readers giving them also the path to understand all the concepts. At the same time inside this tutorial there are some advanced concepts and quite complex C++ sources which are included in this tutorial's archive. So the tutorial is meant for several level readers; if you want you are free of course to skip early chapters, introducing the argument, and directly go those presenting the Oraculum concept. I also wrote two appendixes to help understanding the target applications with a traditional Ollydbg-based approach.

At this point some of you might ask why I won't use the debugAPI which allows to easily set breakpoints, stop and run the application and so on. My approach doesn't uses debugAPIs because several programs can detect if the program is being debugged, quite easily and in different ways, while they are not able (most of the times simply doesn't do it) to detect direct memory writing/reading (except for CRCs, but can be "skipped"). The concept I applied here is "let the program run freely, normally and trap what you want from it, trasparently"...the debugging APIs are a little more invasive.

NOTE
I chosen in this tutorial to use C/C++ because I consider this language much more elegant and powerful of ASM (being an higher level language is not an opinion indeed but a matter of facts), but at this level of difficulty the programs we write can be coded in either ways, so it's a matter of programming experience and tastes what language you'll use. If you really understand the concepts it will not be that difficult to write on your own new oraculums with whatever language you like most. Anyway as comparison a simple ASM oraculum is included at the end: consider that anyway for simple code the two languages are equivalent but consider also what happens when the situation get more complex..

1.2 What the hell is an Oraculum?

But.. What is an Oraculum?? Well, literally "Oraculum", an ancient Latin term, means (Oxford Dictionary):

Oraculous - \O*rac"u*lous\, a. Oracular; of the nature of an oracle. [R.] ``Equivocations, or oraculous speeches." --Bacon. ``The oraculous seer." --Pope. -- O*rac\"u*lous*ly, adv. -- O*rac\"u*lous*ness, n.

Informatically speaking, an oraculum is a loader, an external program which executes the target program and does some memory patching in order to obtain some information such as usually the serial code, and then it reports those things to the user.
An Oraculum is not a self-keygen (an application patched to reveal its real serial), because the original application isn't patched on disk, isn't only a loader because the application is closed when the required information are found (usually the real serial) and the application isn't patched to avoid limitations, it is something different, it's simply an Oraculum.. ☺

1.3 What is this tutorial about

This tutorial discusses about loaders, memory patching of processes and finally oraculous. The final part of the tutorial will introduce a C/C++ framework I wrote to assist you writing a new Oraculum. I will also give you two examples written using this framework: an Oraculum of a not packed application and one of an UPX packed application with CRC and antidump tricks.

To make the whole argument shorter and not to repeat what has already been written elsewhere I will also point you to the right tutorials where to get the missing information (this tutorial is also included in this tutorial archive). These reading are important to fully understand what I'm writing about, so who already know can skip them. For those who didn't I'll tell where it's time for reading.

2. How to code a loader

This part of the tutorial has been heavily based on an original tutorial of Detten, available at: http://biw.rult.at/coding/loader.htm. I reused it here because it's useful to introduce some of the things required but I adapted it in C, to make things easier for you understanding the final C++ code..

2.2 What is a loader, and why do we need it?

A loader is a little standalone program that is used to load another program. Of course we will only use a loader if we want to change something in the program after it is loaded in the memory. (patching in memory) A well known example of a loader is a trainer used to cheat in games.
The reasons why we choose for a loader instead of a regular patch can be various. We might only want to change something after the CRC is done, or we might want to change something and later in the program restore the original bytes,...I'm sure you can find some use for it :)

2.3 How does a loader work?

Ok, grab your MSDN and fasten your seatbelt :)
First of all, the loader has to create a new process and start the target. We will use the CreateProcess API for that (pretty obvious ;) When the target is loaded in memory we want to pause the process, so we can change the things we want.

Let's check out what MSDN can tell us about this API :

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,                 // pointer to name of executable module
    LPTSTR lpCommandLine,                      // pointer to command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // pointer to process security attributes
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  // pointer to thread security attributes
    BOOL bInheritHandles,                      // handle inheritance flag
    DWORD dwCreationFlags,                     // creation flags
    LPVOID lpEnvironment,                      // pointer to new environment block
    LPCTSTR lpCurrentDirectory,                // pointer to current directory name
    LPSTARTUPINFO lpStartupInfo,               // pointer to STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation // pointer to PROCESS_INFORMATION
);
```

Check all the API's I mention here in your MSDN documentation, because I will only discuss the things that are important for our loader.

lpApplicationName is the path + name of our target program. (eg c:\somedir\crackme.exe)

lpCommandLine can be used if you want to add some commandline parameters to the target (we will set this always to NULL, eventually including the command-line in lpApplicationName).

dwCreationFlags is important for us, because we want to pause the process as soon as it is loaded. To accomplish that, we use CREATE_SUSPENDED here.

lpStartupInfo points to a struct with startup information (again check win32.hlp for more info)

lpProcessInformation points to an empty struct that will be filled when the target is loaded in memory.

This struct contains the process handle, thread handle and process/thread ID.

NOTE
The advantage of using the process handle instead of the thread handle, is that when using the process handle you have PROCESS_ALL_ACCESS access to the process object. Meaning that you have read/write access for the entire process. When using the thread handle, you will need to enable write access manually.

Ok, now that the target is loaded, we can easily let the thread run/pause with the following API's :

```
DWORD ResumeThread(
    HANDLE hThread // identifies thread to restart
);
```

to let it run, and

```
DWORD SuspendThread(
    HANDLE hThread // handle to the thread
);
```

to pause it again.

The hThread handle can be found in the LPPROCESS_INFORMATION struct.

NOTE
Remember that any process has a thread, the main thread, so some API will work on threads while other will work on the process handles. A thread has it's own memory location and variables, but all of them are shared in the process's space. So note which API works on threads and which works on the whole process (different handles types).

Finally we can read and write from/to the process using these API's :

```
BOOL WriteProcessMemory(
    HANDLE hProcess,          // handle to process whose memory is written to
    LPVOID lpBaseAddress,     // address to start writing to
    LPVOID lpBuffer,          // pointer to buffer to write data to
    DWORD nSize,              // number of bytes to write
    LPDWORD lpNumberOfBytesWritten    // actual number of bytes written
);
```

This is pretty self-explanatory. The hProcess handle is the one from the LPPROCESS_INFORMATION struct.

To read from the process :

```
BOOL ReadProcessMemory(
    HANDLE hProcess,          // handle of the process whose memory is read
    LPCVOID lpBaseAddress,    // address to start reading
    LPVOID lpBuffer,          // address of buffer to place read data
    DWORD nSize,              // number of bytes to read
    LPDWORD lpNumberOfBytesRead       // address of number of bytes read
);
```

These information should be enough to understand the following example.

2.4 A loader example

In the example below I will code a program which open a changeme.exe file (included in this archive), change the text inside the dialog to "Shub-Nigurrath!" and finally show the original and the new strings in the DOS window.
So here we are patching a simple string into an external process, but with the same method we can also patch code bytes or dwords of course ;)
The target of the loader is to change the string shown below the OK button.

As preparation for the loader we need to know the address where the caption string is saved in the target. So fire up your favourite disassembler, and you'll find this address: 0x00403020 (might be different on your PC of course, due to relocations)[1].

```
<-------------Code Snippet first_loader.cpp----------------->

#include <stdio.h>
#include <windows.h>

char FileName[]=".\\Changeme.exe";
char notloaded[]="It did not work :-(";
char Letsgo[]="The process is started\nLet's change smthg and run it now :-)";
char OldText[20];
char NewText[]="Shub-Nigurrath!";

STARTUPINFO startupinfo;
PROCESS_INFORMATION processinfo;

unsigned long byteswritten;
int uExitCode;

void main() {

        //Initialize correctly the required structures..
        memset(&processinfo, 0, sizeof(PROCESS_INFORMATION));
        memset(&startupinfo, 0, sizeof(STARTUPINFO));
        startupinfo.cb=sizeof(STARTUPINFO);

        //Create a process and load the changeme in it, and immediately suspends
        //the thread (pause it).
        BOOL bRes=CreateProcess(FileName, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED,NULL, NULL,
                                &startupinfo, &processinfo);

        if(bRes== NULL) //Creation of new process failed?
                MessageBox( NULL, notloaded, NULL, MB_ICONEXCLAMATION);
        else {
                 MessageBox(NULL, Letsgo, NULL, MB_OK); //Display Message

                //Before doing the changes I will read the original value of the string.
```

[1] To find the address, open Ollydbg on changeme.exe and search for "All Referenced Strings", then go to the reference of the string "Change me!". You will find a "PUSH Changeme.00403020".

```
                ReadProcessMemory(processinfo.hProcess, (LPVOID)0x00403020, OldText, 26, NULL);

                //I will change the text string in the changeme used in the dialog (0x00403020)
                WriteProcessMemory(processinfo.hProcess, (LPVOID)0x00403020, NewText, 20,
                                   &byteswritten);

                //Let the process run happily ;)
                 ResumeThread(processinfo.hThread);

                printf("Original string: %s\n", OldText);
                printf("New string: %s\n", NewText);
        }
        ExitProcess(1);
}


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

To compile use this command line: `cl first_loader.cpp /link user32.lib`

NOTE
To be able to compile in a DOS command-line you must have VisualC++ 6.0 installed and then go inside one of the installation sub-folders where you should find the vcvars32.bat batch file. To setup all the environment variables requested by VC++60 to complie, execute that bat file (usually is here <Installation folder>\vc98\bin\vcvars32.bat).

That's all it takes to create your first loader.
In the next sections I'll explain how to do some more advanced loader techniques, like reading and changing the process context (all registers and flags).

2.5 Getting the Process Context

Two important APIs GetThreadContext and SetThreadContext are used to get the registers from a running process, see also the MSDN library for a complete help.

```
BOOL GetThreadContext(
    HANDLE hTread,          //Handle to the thread whose context is to be retrieved
    LPCONTEXT lpContext     //Pointer to the CONTEXT structure that receives
                            //the appropriate context of the specified thread
);

BOOL SetThreadContext(
    HANDLE hThread,           //Handle to the thread whose context is to be set.
    const CONTEXT* lpContext  //Pointer to the CONTEXT structure that contains the context
                              //to be set in the specified thread
);
```

MSDN states: "*These functions allow the selective context to be get or set based on the value of the ContextFlags member of the CONTEXT structure. The thread handle identified by the hThread parameter is typically being debugged, but the function can also operate even when it is not being debugged.*" .. note the underlined sentence!

Generally speaking do not try to get/set the context for a running thread; the results are unpredictable. Use the SuspendThread function to suspend the thread before calling SetThreadContext.

As the MSDN help also reports the process must be in a well known fixed state in order to be able to get a meaningful thread context. Anyway everything will be into the CONTEXT structure which contains all the processor-specific register data, so our usual registers (Edi, Esi, Ebx, Edx, Ecx, Eax,