CodeBreakers Magazine
Security & Anti-Security - Attack & Defense
Volume 1, Issue 2, 2006

Guide on how to play with processes memory, writing loaders, and Oraculumns

Shub Nigurrath
May 2006

Abstract

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.

This disclaimer is not meant to sidestep the responsibility for the material we will share with you, but rather is designed to emphasize the purpose of this CodeBreakers Magazine feature, which is to provide information for your own purposes. The subjects presented have been chosen for their educational value. The information contained herein consists of Secure Software Engineering, Software Security Engineering, Software Management, Security Analysis, Algorithms, Virus-Research, Software-Protection and Reverse Code Engineering, Cryptanalysis, White Hat and Black Hat Content, and is derived from authors of academically institutions, commercials, organizations, as well as private persons. The information should not be considered to be completely error-free or to include all relevant information; nor should it be used as an exclusive basis for decision-making. The user understands and accepts that if CodeBreakers Magazine were to accept the risk of harm to the user from use of this information, it would not be able to make the information available because the cost to cover the risk of harms to all users would be too great. Thus, use of the information is strictly voluntary and at the users sole risk.

The information contained herein is not a license, either expressly or impliedly, to any intellectual property owned or controlled by any of the authors or developers of CodeBreakers Magazine. The information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of CodeBreakers Magazine hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the contribution.

ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO CODEBREAKERS MAGAZINE.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF CodeBreakers Magazine BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR PUNITIVE OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO CODEBREAKERS MAGAZINE, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

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, 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.

At this point some of you might ask why I won't use the debugAPI which allows easily setting breakpoints, stopping and running 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, transparently"...the debugging APIs are a little more invasive (see also section "

Discussion" at page 46 of this document).

1.1 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.. ☺

An Oraculum is then rather than an instrument a concept, a way to patch an application with a different behaviour in mind.

1.2 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 and what is called a Silent Oraculum, written in ASM by my friend **Gabri3l (of ARTeam)**. These two sections will satisfy I think most of the experts around, those with more programming skills who will to use C++ and those instead used to ASM.

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. These reading are important to fully understand what I'm talking about, so who already know can skip them. For those who didn't I'll tell where it's time for reading.

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 and the Silent Oraculums are also written using ASM: consider that anyway for simple code the two languages are equivalent but consider also what happens when the situation get more complex. All the code mentioned in this document is fully available online at http://tutorials.accessroot.com

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 and it's short enough. I adapted it in C, to make things easier for you understanding the final C++ code (those not accustomed to ASM read the original one, but you will anyway miss some important comments here).
Anyway a complete walk-through on loaders is available at [9].

2.1 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 :)

2.2 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.

2.3 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.
                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);
```

[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".

```
                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 code from scratch 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).

2.4 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, Ebp, Eip, Esp) plus many more. Moreover as MSDN also confirms you do not need to attach the target process to any debugger. This is really important to skip anti-debugger tricks!

The CONTEXT structure is very complex and not all its members could be interesting: you must specify what to get/set through the ContextFlags, the value of the ContextFlags member of this structure specifies which portions of a thread's context are retrieved.

Well, it's time to go with the code again. Reading also the come comments I think you might easily understand it.

```
<-------------Code Snippet second_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
        //immediatly suspend 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.
                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);

                ////////////////////////////////////////////////////////////////////////
                Sleep(5000); //simple way just to be sure that the application is ready & running
                ////////////////////////////////////////////////////////////////////////

                CONTEXT context;

                //Before getting the thread context it's useful to suspend the thread,
                //it's not needed by the GetThreadContext API, but to have all the
                //registy coherent.
                SuspendThread(processinfo.hThread);

                //Set up permissions to get the context.
                //NOTE: the documentation can be found in WINNT.H file and not inside MSDN.
                //These values are highly dependant from the processor used (for x86 families are
```

```
                //always the same, but for alpha family changes). Usually below values are enough
                //but there's another that can be used, CONTEXT_EXTENDED_REGISTERS.
                context.ContextFlags = CONTEXT_FULL | CONTEXT_FLOATING_POINT |
                                       CONTEXT_DEBUG_REGISTERS;

                GetThreadContext(processinfo.hThread, &context);

                //Do some printf just to demonstrate the usage of GetThreadContext.
                //These values are of course useless for us at the moment.
                printf("\nProcess Registers Dump:\n");
                printf("-----------------------\n");
                printf("Edi=%X\n",context.Edi);
                printf("Esi=%X\n",context.Esi);
                printf("Ebx=%X\n",context.Ebx);
                printf("Edx=%X\n",context.Edx);
                printf("Ecx=%X\n",context.Ecx);
                printf("Eax=%X\n",context.Eax);
                printf("Ebp=%X\n",context.Ebp);
                printf("Eip=%X\n",context.Eip);
                printf("Esp=%X\n",context.Esp);

                //Before terminating the process, remember to again let it run freely!
                ResumeThread(processinfo.hThread);

                //Closes everything.
                TerminateProcess(processinfo.hThread,uExitCode);

        }
        ExitProcess(1);
}

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

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

I think that now the situation should be clear and you should have understood the essentials things.

Once important step further understanding what's needed to write loders is to manage memory read/write permissions.

2.5 Checking and setting the accessing rights to memory locations

What we learnt up to now isn't enough: if you try to write to a protected memory section (for example the resources sections), you might fail. Each memory location inherits its memory page permissions, but fortunately there is a function VirtualProtectEx which helps us to solve the situation.

```
BOOL VirtualProtectEx(
    HANDLE hProcess,          // Handle to the process whose memory protection is to be changed
    LPVOID lpAddress,         // Pointer to the base address of the region of pages whose access
                              // protection attributes are to be changed
    SIZE_T dwSize,            // Size of the region whose access protection attributes are changed,
                              // in bytes
    DWORD flNewProtect,       // Memory protection.
    PDWORD lpflOldProtect     // Pointer to a variable that receives the previous access protection
                              // of the first page in the specified region of pages
);
```

For those of you who doesn't have access to MSDN I also report here the complete reference to the parameters with some comments (in italic).

hProcess. Handle to the process whose memory protection is to be changed. The handle must have PROCESS_VM_OPERATION access. For more information on PROCESS_VM_OPERATION, see

OpenProcess. *In our case the CreateProcess API already does it for us, so don't worry the process handle in our case is already ok and we can directly use it as a parameter.*

lpAddress. Pointer to the base address of the region of pages whose access protection attributes are to be changed. All pages in the specified region must be within the same reserved region allocated when calling the VirtualAlloc or VirtualAllocEx function using MEM_RESERVE. The pages cannot span adjacent reserved regions that were allocated by separate calls to VirtualAlloc or VirtualAllocEx using MEM_RESERVE. *In our case the addresses are directly taken from the target process using Ollydbg or other tecniques, so are meaningful by definition. Again we shouldn't have problems.*

dwSize. Size of the region whose access protection attributes are changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the lpAddress parameter to (lpAddress+dwSize). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed. *Quite obvious I think, it's the size in bytes of what we want to write.*

flNewProtect. Memory protection. This parameter can be one of the memory protection options, along with PAGE_GUARD or PAGE_NOCACHE, as needed. *There are plenty of memory protection flags, in our case for almost all the cases we want read/write access, leaving the other rights unchanged. So the most used by us will be PAGE_EXECUTE_READWRITE.*

lpflOldProtect. Pointer to a variable that receives the previous access protection of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails. *Remember that the API fails if you set this parameter to NULL, so we'll need a dummy variable to store this useless value (useless when exiting from our loader of course).*

NOTE
Usually when you access the executable sections of a process there's no need to use VirtualProtectEx, because those sections usually are already readable (to let the execution engine to read the instructions) and writable (to let to store data, self modifying code and so on). The need arise when you plan to access to other sections which usually should only be read, such as the program's resources. So in the most of the examples here to keep things shorter I'll do not do checks rights, except for this specific section.

The set of APIs to be usd for handling memory rights is completed APIs used to test if a specified memory address can be read or written: IsBadStringPtr, IsBadCodePtr, IsBadStringPtr, IsBadWritePtr. All have similar prototypes:

- `BOOL IsBadStringPtr( LPCTSTR lpsz, UINT_PTR ucchMax);` The IsBadStringPtr function verifies that the calling process has read access to a range of memory pointed to by a string pointer.
- `BOOL IsBadCodePtr( FARPROC lpfn);` The IsBadCodePtr function determines whether the calling process has read access to the memory at the specified address.
- `BOOL IsBadWritePtr( LPVOID lp, UINT_PTR ucb);` The IsBadWritePtr function verifies that the calling process has write access to the specified range of memory.
- `BOOL IsBadCodePtr( FARPROC lpfn);` The IsBadCodePtr function determines whether the calling process has read access to the memory at the specified address.

And the usage is quite simple: just before using ReadProcessMemory or WriteProcessMemory use the proper function to test then use VirtualProtectEx to set.
Remember that if the calling process does not have read access to the specified memory, the return value is nonzero and the last parameter byteswritten or bytesread is 0. To get extended error information, call GetLastError.

NOTE
The approach based on isBad*Ptr functions works for the specific case of loaders, where the loader launches the target process through the CreateProcess API. isBad*Ptr APIs are not process aware: being the only parameters they receive, a memory address and a size they are not aware of processes, the only think they do is to test access to a memory location. Generically speaking if you want to write a code interacting with an already running process, you should follow a little different approach where, instead of CreateProcess you should use OpenProcess, and in place of isBad*Ptr functions you should use VirtualQueryEx which provides information about a range of pages within the virtual address space of a specified process. I will not cover this argument also to keep this document shorter! On MSDN there are several document explaining these concepts.

2.5.1 SEH - Structured Exception Handling

This indeed a very long argument for which you might find several tutorials around, starting from the MSDN site[2].
To support SEH, Microsoft extends the C and C++ languages with four new keywords:

- __except
- __finally
- __leave
- __try

Because these keywords are non-standard language extensions, you must compile with such extensions enabled (/Fa option off).
I wouldn't talk about them here except as a more efficient or alternative way to trap errors, deriving from wrong read/write memory access rights. The mechanism is essentially similar to the C++ exceptions..
The following example shows the essential things you must know for this tutorial: the RaiseException function causes an exception in the guarded body of a termination handler.

```
BOL DummyFailingFunction() {
    printf("1"); // just do something
    return FALSE;
}

void main() {
    __try {
       if( !DummyFailingFunction() )
           RaiseException( 1,    // exception code
                0,               // continuable exception
                0, NULL);        // no arguments
    }
    __except ( ) {
        printf("2\n"); // do something again
    }
}
```

[2] http://msdn.microsoft.com/library/en-us/debug/base/structured_exception_handling.asp

Well, so it's time to write a new example. This time the victim will be the standard Calc.exe stored inside system32 folder. We want to change the program's caption at runtime, the caption is stored into the resources, so will be loaded into a read-only memory section. Copy calc.exe into the same where is the code I provided.

2.5.2 How to find the Calculator's memory address to patch

To find where the "Calculator" caption is stored in memory you need to do some steps (briefly):

1. Search with a Hex Editor all the "Calculator" UNICODE strings into the exe file and find which is the one that is used for the titlebar (modifying them one by one till you see the correct one).
2. Take note of the surrounding bytes and open Ollydbg with the Calc.exe inside.
3. Search in the resource section of the process (use ALT-M to open Ollydbg's memory page and the doubleclick on .rsrc)

01000000	00001000	calc	01000000 (itself)		PE header	Imag	R	RWE
01001000	00013000	calc	01000000	.text	code,import	Imag	R	RWE
01014000	00002000	calc	01000000	.data	data	Imag	R	RWE
01016000	00009000	calc	01000000	.rsrc	resources	Imag	R	RWE

4. Search the same byte pattern you found using the HexEditor and take note of its address.
5. For the Calc application hit CTRL-B in the memory window and search for the UNICODE string "jSciCalc": the correct "Calculator" string is beside, at the address 0x01017486.

```
01017474  j.S.c.i.C.a.l.c...C.a.l.c.u.l.a.t.o.r.
010174B4  D.l.g...........▀▀▀P▀.▀.▀.▀.▀▀..ÿÿ▀...
```

2.5.3 Writing a loader accessing protected memory pages.

To accomplish our task the general strategy will be:

1. modify the protection's right of the patched address to PAGE_EXECUTE_READWRITE;
2. do our patching job, reading and writing whatever we need (as did in section 2.3);
3. restore the previous protection's right

And the resulting code is (note that UNICODE strings are not supported by C, so I have to print and set them as array of bytes):

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

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

char FileName[]=".\\Calc.exe";
char notloaded[]="It did not work :-(";
char Letsgo[]="The process is started\nLet's change smthg and run it now :-)";
char OldText[20];
//   N.i.g.u.r.r.a.t.h.! in UNICODE
char NewText[]={0x4E, 0x00, 0x69, 0x00, 0x67, 0x00, 0x75, 0x00, 0x72, 0x00, 0x72, 0x00,
                              0x61, 0x00, 0x74, 0x00, 0x68, 0x00, 0x21};

int PATCH_SIZE = 19;
DWORD PatchAddress = 0x01017486;

STARTUPINFO startupinfo;
PROCESS_INFORMATION processinfo;
```

```
unsigned long byteswritten=0;
int uExitCode=0;

void main() {

        DWORD OldProtection=0;
        //It is not really used. The VirtualProtectEx function always requires a valid
        //variable to hold the old page protection values, otherwise fails. When restoring the
        //protection values of the page, on the existing of this program, the old values are not
        //important of course.
        DWORD dummyProtection=0;

        //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 Calc in it, and
        //immediately suspend the thread (pauses 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);
                ExitProcess(1);
        }

        MessageBox(NULL, Letsgo, NULL, MB_OK); //Display Message

        //Before doing anything at the specified address I must properly set the accessing rights
        VirtualProtectEx(processinfo.hProcess, (LPVOID)PatchAddress, PATCH_SIZE,
                        PAGE_EXECUTE_READWRITE, &OldProtection);

        //Before doing the changes I will read the original value of the string.
        ReadProcessMemory(processinfo.hProcess, (LPVOID)PatchAddress, OldText, PATCH_SIZE, NULL);

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

        //Restore the previous protections of the patched address
        VirtualProtectEx(processinfo.hProcess, (LPVOID)PatchAddress, PATCH_SIZE,
                        OldProtection, &dummyProtection);

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

        printf("Bytes written %d\n",byteswritten);

        //Are UNICODE strings so it's impossible to write them using C functions,
        //which do not support UNICODE. I have to write a special loop writing only valid
        //characters from the buffer.
        printf("Original caption: ");
        for(int i=0; i<PATCH_SIZE; i+=2)
                printf("%c", OldText[i]);
        printf("\n");

        printf("New caption: ");
        for(i=0; i<PATCH_SIZE; i+=2)
                printf("%c", NewText[i]);
        printf("\n");

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

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

If you want to use a protective programming, inserting controls to ensure you always have the correct access rights, the most reliable way is to code also some testing before writing and reading the process memory. So the central lines of the third_loader.cpp would become:

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

...
//Before doing anything at the specified address I must properly set the accessing rights
//The first version uses the IsBadReadPtr and IsBarWritePtr to change or not the rights
if( IsBadReadPtr((LPVOID)PatchAddress, PATCH_SIZE) ||
       IsBadWritePtr((LPVOID)PatchAddress, PATCH_SIZE) )
       VirtualProtectEx(processinfo.hProcess, (LPVOID)PatchAddress, PATCH_SIZE,
                     PAGE_EXECUTE_READWRITE, &OldProtection);

//Before doing the changes I will read the original value of the string.
ReadProcessMemory(processinfo.hProcess, (LPVOID)PatchAddress, OldText, PATCH_SIZE, NULL);

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

//Restore the previous protections of the patched address.
//OlProtection is !=0 if the previous VirtualProtectEx has been done.
if(OldProtection!=0)
       VirtualProtectEx(processinfo.hProcess, (LPVOID)PatchAddress, PATCH_SIZE,
              OldProtection, &dummyProtection);

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

3 Using the SEH instead lead us to the final code which is used inside COraculum class.

1. Instead of the normal ReadProcessMemory and WriteProcessMemory I used two bytes identical local functions (identical for the callers), _ReadProcessMemory and _WriteProcessMemory. Doing this I will not worry anymore about access to the memory and rights. I live in peace with memory rights and do worry anymore about them!

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

...
MessageBox(NULL, Letsgo, NULL, MB_OK); //Display Message

//Before doing the changes I will read the original value of the string.
_ReadProcessMemory(processinfo.hProcess, (LPVOID)PatchAddress, OldText, PATCH_SIZE, NULL);

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

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

printf("Bytes written %d\n",byteswritten);
...

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

2. Code the two new, almost identical, functions. I report here only _ReadProcessMemory to save space.

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

BOOL _ReadProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPVOID lpBuffer,
           DWORD nSize, LPDWORD lpNumberOfBytesRead)
{
    DWORD OldProtection=0;
    //It is not really used because the VirtualProtectEx function always requires a
    //valid variable to hold the old page protection values, otherwise fails. When
    //restoring the protection values of the page, on the existing of this program,
    //the old values are not important of course.
    DWORD dummyProtection=0;

    BOOL bVal=FALSE;

    int tries=0;
    //Do 3 tries loop, so as not the block forever..
    while(tries<3) {
           __try {
                  tries++;
                  bVal=ReadProcessMemory(hProcess, (LPCVOID)lpBaseAddress, lpBuffer, nSize,
                         lpNumberOfBytesRead);
                  if(!bVal)
                         //The RaiseException function raises an exception in the calling
                         //thread.
                         RaiseException(1,  // exception code
                                0,        // continuable exception (non death exception)
                                0, NULL); // no arguments
           }
           __except(TRUE) {
                  if(IsBadReadPtr(lpBaseAddress, nSize) || IsBadWritePtr(lpBaseAddress, nSize))
                         VirtualProtectEx(hProcess, lpBaseAddress, nSize,
                                PAGE_EXECUTE_READWRITE, &OldProtection);
                  continue;
           }
           break;
    }
```

```
    //Restore the previous protections of the patched address.
    //OlProtection is !=0 if the previous VirtualProtectEx has been done.
    if(OldProtection!=0)
           VirtualProtectEx(hProcess, lpBaseAddress, nSize,
                  OldProtection, &dummyProtection);

    return bVal;
}

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

See fourth_loader.cpp for the whole code.

Inside COraclum there are two methods called ReadProcessMemory and WriteProcessMemory with the same prototypes of the Win32 ones, but with additional checks.

Now what we would do is to use these nice windows' API to do something evil, but before going further on with our examples I'll define which our new objective is.

- Our target is to be able to stop the application on the point we want, patch the code in that point and then let the application freely run with our patches.
- Be able to set a breakpoint without using debugger's APIs so without setting a system breakpoint. Remember that the want to use the APIs seen up to now and all of them are able to work even when the target program is not being debugged. This is a real nice feature we want to keep…but how to set breakpoints then?

NOTE
We now finished introducing the main aspects of writing loaders so we are now ready to understand what an Oraculum is .. still awake!? Ok it's time for a coffee break! Then, for whom didn't read it before read the yAtEs tutorial on "Creating Loaders & Dumpers - Crackers Guide To Program Flow" available here http://www.reteam.org/papers/e54.pdf or in this tutorial archive also under yAtEs folder. Take you time to do it because it's needed to understand the following … see you later here!

3.1 Using the EB FE Trick to set Breakpoint

Once read the yAtEs tutorial you are ready to understand what I'm going to describe. For this task I created a small program (FishMe.exe), a very simple program for which you should fish the serial. It's very simple indeed, anyway for beginners I will explain in the following section how to fish the real serial using Ollydbg and then we'll make a loader which will report to us the serial from the program registers. Anyway also not beginners give a look at least at the conclusions of section 3.1.1.

3.1.1 Fishing a serial from Fishme.exe

Fire up Ollydbg on Fishme and search the Good Boy message using the “Search for” -> “All referenced text strings” you should land here:

```
00401421   .  E8 82020000    CALL <JMP.&MFC42.#4160>
00401426   .  8B4C24 08      MOV ECX,DWORD PTR SS:[ESP+8]
0040142A   .  8B46 60        MOV EAX,DWORD PTR DS:[ESI+60]
0040142D   .  8D7E 60        LEA EDI,DWORD PTR DS:[ESI+60]
00401430   .  51             PUSH ECX                          s2 = "Næ\x90|Lm\x81|þ\xFF\xFF\xFF\t"
00401431   .  50             PUSH EAX                          s1 = NULL
00401432   .  FF15 A8214000( CALL DWORD PTR DS:[<&MSVCRT._mbscmp>]  _mbscmp
00401438   .  83C4 08        ADD ESP,8
0040143B   .  85C0           TEST EAX,EAX
0040143D   .v 75 0F          JNZ SHORT Fishme.0040144E
0040143F   .  68 38304000    PUSH Fishme.00403038              ASCII "OK You got the real serial!"
00401444   .  8D4E 64        LEA ECX,DWORD PTR DS:[ESI+64]
00401447   .  E8 20020000    CALL <JMP.&MFC42.#860>
0040144C   .v EB 14          JMP SHORT Fishme.00401462
0040144E   >  68 20304000    PUSH Fishme.00403020              ASCII "None, trial it again!"
00401453   .  8D4E 64        LEA ECX,DWORD PTR DS:[ESI+64]
00401456   .  E8 11020000    CALL <JMP.&MFC42.#860>
0040145B   .  8BCF           MOV ECX,EDI                       ntdll.7C910738
0040145D   .  E8 40020000    CALL <JMP.&MFC42.#2614>
00401462   >  6A 00          PUSH 0
00401464   .  8BCE           MOV ECX,ESI
00401466   .  E8 31020000    CALL <JMP.&MFC42.#6334>
0040146B   .  8D4C24 08      LEA ECX,DWORD PTR SS:[ESP+8]
0040146F   .  C74424 14 FFI MOV DWORD PTR SS:[ESP+14],-1
00401477   .  E8 24010000    CALL <JMP.&MFC42.#800>
0040147C   .  8B4C24 0C      MOV ECX,DWORD PTR SS:[ESP+C]
00401480   .  5F             POP EDI                           kernel32.7C816D4F
00401481   .  5E             POP ESI                           kernel32.7C816D4F
00401482   .  64:890D 0000( MOV DWORD PTR FS:[0],ECX
00401489   .  83C4 10        ADD ESP,10
0040148C   .  C3             RETN
0040148D      90             NOP
```

Place a breakpoint on

```
00401421   .  E8 82020000    CALL <JMP.&MFC42.#4160>
```

And run the application. After this, enter any serial you like and will land in the breakpoint.

Press F8 two times, in order to move the EIP at this instruction:

```
0040142D   .  8D7E 60        LEA EDI,DWORD PTR DS:[ESI+60]
```

And look the registers, you have EAX pointing to the serial you entered and ECX pointing to the correct serial.

```
EAX 00333FE0 ASCII "1111111"
ECX 00334030 ASCII "1234567890"
EDX 00000002
EBX 00000001
ESP 0012F800
EBP 0012F824
ESI 0012FE8C
EDI 0012FE8C
```

Well solved (simple he) but now what about writing a loader which takes out these values? What about writing our first Oraculum ??

3.1.2 How to build a basic Oraculum for Fishme.exe

Let write an action plan of what our program must do:

1. Create the process in suspended mode
2. Read the process's memory at the address 0040142D and save this buffer for a later restoring (we want let the application go on unchanged after we fished what we want).
3. Write the process's memory at the address 0040142D with an EB FE value (see yAtEs's tutorial above).
4. Resume the process.
5. Check whenever the EIP is equal to 0040142D, using a cycle with a GetThreadContext inside.
6. Get the value of the ECX register from the Context structure.
7. Read the memory pointed by ECX, because ECX is the address of the buffer containing the serial we want to fish.
8. Restore the original application's bytes read at the beginning.
9. Report the ECX pointed string to the user.

Are you ready? Ok, see the code below!

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

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

char FileName[]=".\\Fishme.exe";

char notloaded[]="It did not work :-(";
char Letsgo[]="The process is started\nLet's change smthg and run it now :-)";
BYTE OriginalCode[2];

//This is a JMP -2, see the yAtEs tutorial to understand what this code is used for.
//I used the BYTE type which is indeed an unsigned char, a number from 0 to 255, so a byte.
const BYTE HALT_CODE[2] = {0xEB,0xFE};

//Number of bytes to write, it's the length of the HALT_CODE matrix
const int HALT_SIZE = 2;

//size of the buffer which will receive the serials read from the victim's target.
const int SERIAL_SIZE = 20;

//location to patch, it has been found using Ollydbg
const DWORD PatchLocation = 0x0040142D;

STARTUPINFO startupinfo;
PROCESS_INFORMATION processinfo;
CONTEXT processcontext;

unsigned long byteswritten;
int uExitCode;

void main() {

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

        //Create a process and load the fishme in it, and
        //immediately suspend 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);
        ExitProcess(1);
}

MessageBox(NULL, Letsgo, NULL, MB_OK); //Display Message

//Before doing the changes I will read the original value of the location.
//for our purpose is not useful because I'll terminate the program
ReadProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation, OriginalCode,
                HALT_SIZE, NULL);

//write the HALT_CODE at the proper address. The last parameter byteswritten
//is set to the number of bytes written, this parameter can be NULL and in that
//case will be ignored.
//Note that to cast the BYTE array to a void* I used a trick which takes
//the pointer of the first byte of the array and convert it to a
//void pointer: (void*)&HALT_CODE. This is required to fool compiler's type checking.
WriteProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation,
                (void*)&HALT_CODE, HALT_SIZE, &byteswritten);

//Set up permissions to get the context.
processcontext.ContextFlags = CONTEXT_FULL | CONTEXT_FLOATING_POINT |
                              CONTEXT_DEBUG_REGISTERS;

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

//////////////////////////////////////////////////////////////////////
//Now, according to the yAtEs tutorial the program will run happily till
//it fall into our properly placed JMP -2 code then the EIP will not change
//anymore and the program will stay there forever.
//The following code checks this using a loop and the GetThreadContext to get
//the EIP value.

//When the GetThreadContext fails means that the application has been closed,
//for example it happens when for some reason the application doesn't pass
//through our trap.
while(GetThreadContext(processinfo.hThread, &processcontext))
{
        //when we reach the proper point we are in the place we patched!
        if(processcontext.Eip==PatchLocation) {
                //this buffer is used to store memory location taken from the target process,
                //SERIAL_SIZE has been defined at the beginning.
                char buffer[SERIAL_SIZE];

                //fills with zero the buffer:
                //it is a trick to automatically terminate the string after the last
                //read character
                memset(buffer,0,SERIAL_SIZE);

                //refresh the thread context with the last registers values
                GetThreadContext(processinfo.hThread, &processcontext);

                //read the memory pointed by EAX and places into a buffer.
                //Remember that EAX contains the address of the serial's string.
                ReadProcessMemory(processinfo.hProcess,
                                (LPVOID)(processcontext.Eax),
                                &buffer, SERIAL_SIZE, NULL);
                printf("you inserted this serial: %s\n", buffer); //print out the value

                //read the memory pointed by ECX and places into a buffer.
                ReadProcessMemory(processinfo.hProcess,
                                (LPVOID)(processcontext.Ecx),
                                &buffer, SERIAL_SIZE, NULL);
                printf("but your real serial is: %s\n", buffer); //print out the value

                //Restore the original applications bytes so the application can continue.
                WriteProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation,
                                (void*)&OriginalCode,
                                HALT_SIZE, &byteswritten);

                break; //we did the job so we can exit
        }
        Sleep(10); //sleep a while among requests so as to not slowdown the system
```

```
        }
        ////////////////////////////////////////////////////////////////////////
}

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

Have you understood everything? The most important new thing is the while cycle which stand till the GetThreadContext API return not NULL (meaning that the process is still alive) and every 10 seconds checks if the EIP register is equal to the patched location. If it is, then reads the two process's memory location pointed by EAX and ECX.

Again, to compile use this command line: `cl first_oraculum.cpp /link user32.lib`

Well I imagine you got the real power of this technique, but as you should notice reading the above code, there are some still left problems:

1. The location to be patched must be known in advance and if the process is relocated it will change.
2. There's quite a lot of code to write for creating a new Oraculum and when the application get's complicate (e.g. a packed application) the process will not be that simple.

To solve these problems here's what we'll need:

1. Some function to search for specific patterns in the executable file and on memory: most of the times new versions of programs still can be patched in the same way, what changes are the file offsets of the same patched bytes. So finding a common unique pattern which remains unchanged among versions is a very useful think: a pattern must not have relative addresses, such as JMP instructions because they often change, and must be the same numbers of bytes away from the real patch location (for example be 1D bytes before the bytes to be patched).
2. Some functions to convert to RVAs and file offsets: once you find the file offset of a byte pattern it has to be converted into a real memory address, according to the PE Header values.

Definitely we need a framework which will take care of the repetitive parts and make easier to write a new Oraculum! ;-)

4 The framework for building Oraculums

Well with a framework I mean a set of generic classes that can be used in order to easily perform some programming task, an informatics monkey wrench.

It's quite complex indeed to write a framework of classes general and at the same time easy to use but hey, no one is perfect and giving source to you I explicitly express the desire that someone else also will improve my code. By the way if you improve these classes please send me the new version and write somewhere my name (for my eternal glory ;-)).

What we'll do in the remaining of this tutorial is to explain the framework and use it to write an Oraculums for the fishme.exe program.

Take a coffee, relax a while and then go forward and .. do not leave me alone here in the middle ;-)

4.1 The FishMe_Oraculum

I think that the best way to start is to write the same Oraculum of section 3.1.2, using my framework, before going inside the description of its methods and tricks and blah blah.. I think that it's better to start from the source of the main() method…here it follows.
Carefully read also the code comments, because several anticipations about how the COraculum class works are already there!

First of all this time there are several classes (we definitely moved to C++ code) thus we need a VisualC++ project (.dsp file) to compile. Look into the "source_oraculums" folder. Do not look at the several files for the moment, but concentrate on the file FishMe_Oraculum.cpp

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

#include "Oraculum.h" //main include of the framework..
#define SERIAL_SIZE 10

char FileName[]=".\\Fishme.exe";

//Standard prototypes of the callbacks used by COraculum
void DoPatch_callback(COraculum *oraculum, DWORD dwMemoryPatchAddr);
void DoActionPatch_callbackStop(COraculum *oraculum);
void DoActionPatch_callbackResume(COraculum *oraculum);

void main() {

        COraculum Oraculum;

        //Setup the exefile on which to operate.
        Oraculum.SetPath(FileName);

        //Setup the exe scanning library and control structure
        Oraculum.Setup();

        //Add the pattern to be searched.
        //The code we want to stop on is:
        //0040142A  |.  8B46 60       MOV EAX,DWORD PTR DS:[ESI+60]
        //0040142D  |.  8D7E 60       LEA EDI,DWORD PTR DS:[ESI+60] ; this is the point
        //the instruction at 0040142A occurs only once in the executable.
        //
        //The COraculum class supports the pattern searching rather than the direct memory
        //address patching.
        //The AddPattern method allows to specify add a pattern (supplied using a specific
```

```
        //syntax which separates each byte with a :x couple of characters)
        //
        //In this case the unique pattern is: 8B4660 (which is present in the exe file only once)
        //and the relative offset is 3 bytes (which lead to a final address 0040142D).

        Oraculum.AddPattern(":x8B:x46:x60", 0x3);

        //The callbacks are functions used by the class to perform specific actions.
        //- DoPatch_callback is called when the victim is launched and still not running.
        //
        //- DoActionPatch_callbackStop is called when the program reaches the first EBFE loop
        //
        //- DoActionPatch_callbackResume is called before just resuming the process.

        Oraculum.SetCallBacks( DoPatch_callback,
                DoActionPatch_callbackStop,
                DoActionPatch_callbackResume);

        //Patching working functions
        //Do the patch at the desired location, and create the suspended process, using
        //CreateProcess API.
        if(Oraculum.IsValid()) //all initialization have been done?
                if(Oraculum.FindPatterns()>0) //find one of the patterns given with AddPattern
                        //do the hard work and invokes callbacks when required.
                        if(Oraculum.DoPatch()>=0)
                                Oraculum.KillProcess(); //closes the process

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

        cout << "Messages Stack dump, read information in reverse order.." << endl
                << "-----------------------------------------------------------" << endl;

        //The COraculum accumulates messages into a private stack that can be flushed when
        //everything is finished. Below cycle does exactly this.
        while(!Oraculum.MessageStack().empty()) {
                int idx=Oraculum.MessageStack().size();
                TextString str=Oraculum.MessageStack().top();
                Oraculum.MessageStack().pop();
                printf("%d> %s\n", idx--, str.c_str());

        }

}

//This callback is called by COraculum.
//- DoPatch_callback is called when the victim is launched and still not running.
//  Usually it is used to place our EBFE loops in the proper addresses and do all the
//  initialization you need BEFORE the process is resumed from initial creation.
//  The caller automatically provides a memory address which is where one of the
//  patterns (given using AddPattern) is found. For the moment consider it to be used just
//  for only ONE EBFE loop.
void DoPatch_callback(COraculum *oraculum, DWORD dwMemoryPatchAddr) {

        oraculum->WriteProcessMemory(oraculum->GetPI()->hProcess,
                (LPVOID)dwMemoryPatchAddr,
                (void*)&COraculum::HALT_CODE, HALT_SIZE, NULL);     // OEP HOOK

}

//- DoActionPatch_callbackStop is called when the program reaches the first EBFE loop
//  (usually the one set by the corresponding DoPatch_callback)
//  When the COraculum class identifies that the EBFE loop is reached it Suspend the process
//  get the Context and calls this callback
void DoActionPatch_callbackStop(COraculum *oraculum) {

        //ECX contains the real serial code
        oraculum->ReadProcessMemory(oraculum->GetPI()->hProcess,
                (LPVOID)oraculum->GetProcessContext()->Ecx,
                oraculum->GetProcessBuffer(),SERIAL_SIZE, NULL);
}

//- DoActionPatch_callbackResume is called before just resuming the process.
//  For example it is used to write back the original bytes of the application or to
//  write the collected results.
```

```
void DoActionPatch_callbackResume(COraculum *oraculum) {

	//Writes the message!
	char str[256];
	sprintf(str,"Shub-Nigurrath says: your correct serial is %s",oraculum->GetProcessBuffer());
	oraculum->pushMessage(str);

}

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

Quite complex he? Well it is less linear than before, but consider that the main() is essentially almost the same all the times and that what changes from target to target are the callbacks.

Of course the compiled program is bigger because in order to include support for most complex cases I added a lot of code which is not used for this little example.

4.2 Main methods of Oraculums C++ framework

The above code is quite commented and some of the basic concepts of the COraculum class should have been understood a little, but explain what isn't clear enough..

- **The SetPath() method**. Specify the program (must be an executable) on which to work. It's required to call it at the beginning.

- **The Setup() method.** This method setup memory, buffer and other internal things. It's should be called at the beginning.

- **The AddPattern() method**. COraculum works on the concept of patterns, which are unique bytes sequences found inside the program file (the .exe) and their relative offsets.
  The Coraculum class supports the pattern searching rather than using direct memory addresses to patch. This is more reliable across different target versions because most of the times the bytes to be patched doesn't change, what change instead are the offsets of these bytes. So a search&replace patch is more reliable. The AddPattern method allows to add a pattern (supplied using a specific syntax which separates each byte with a :x couple of characters) to a stack of patterns which will be searched in the victims exe file (on disk) and a relative offsets with respect to which the real patch is done. All the patterns are stored into a stack to support multiple versions Oraculums: suppose that a pattern is present only till a specific version and then another one appears from that version on.
  Remember that the patterns are considered as OR in the order of addition to the stack (LIFO the last added is the first searched then if not found search for the second and so on): being these patterns stored into an internal stack must be wrote in the c++ file in reverse order of importance: the last added is the first searched!
  Remember the patterns are supposed to be unique (use an Hex editor to see if they are unique or not).
  The pattern offset is used as a relative offset, because most of the times a reliable pattern isn't exactly where we want to patch. The relative offset is added to the address where the pattern is found.

  The complete syntax for patterns is the following one.
  The sequence of bytes is entered using some control characters. The control characters can be entered by using a ':' in the string followed by the ASCII value of the character. The value

is entered using a ':' followed by three decimal digits or ':x' followed by two hex numbers. To enter a colon (:) in the search pattern use '::'.
Example: To search for the string :foo ('o' is 111 decimal, 6F in hex) use the search options: ::foo or ::fo:111 or ::fo:x6F
If you want to search for a string with spaces in it, surround the expression with quotes.

- **The SetCallBack() method and the callbacks.** This is the most complex concept. If you look carefully at the code presented in 3.1.2, you should note that there are some generic actions that might be repeated for any Oraculum and some other which are victim specific. For example, the operation before the while loop

```
ReadProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation, OriginalCode,
               HALT_SIZE, NULL);
```

and the following one

```
WriteProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation,
       (void*)&HALT_CODE, HALT_SIZE, &byteswritten);
```

are specific of the victim process and also the following operations, which are into the while loop, are victim's specific:

```
ReadProcessMemory(processinfo.hProcess, (LPVOID)(processcontext.Eax),
       &buffer, SERIAL_SIZE, NULL);
printf("you enteret this serial: %s\n", buffer); //print out the value

ReadProcessMemory(processinfo.hProcess, (LPVOID)(processcontext.Ecx),
       &buffer, SERIAL_SIZE, NULL);
printf("but your real serial is: %s\n", buffer); //print out the value

WriteProcessMemory(processinfo.hProcess, (LPVOID)PatchLocation,
       (void*)&OriginalCode, HALT_SIZE, &byteswritten);
```

We can group these operations into 3 functions which are called in three different moments:

1. when the process has just been created and before resuming it from initial creation;
2. when the process reach the EBFE loop and the information are collected from Context and generally speaking from the victim process;
3. when the process exits from our modificatin and returns on its own normal execution path.

We then need three callbacks which will be different for each victim program.

- ✓ *DoPatch_callback* is called when the victim is launched and still not running (created suspended). Usually it is used only to place our EBFE loops in the proper addresses and to eventually save the original bytes. It does all the initialization you need BEFORE the process is resumed from initial creation. The caller automatically provides a memory address to this callback which is where the first of the patterns (given using AddPattern) has been found found. It must be used for just one EBFE loop because refers to one unique pattern only. Note that this function must properly place an EBFE trap somewhere, otherwise our Oraculum will loop forever and the victim will work normally (that it's not what we want).
- ✓ *DoPatch_callbackStop* is called when the program reaches the first EBFE loop (the one set by the corresponding DoPatch_callback). When the COraculum class identifies that the EBFE loop is reached it suspends the process, get the Context and calls this callback. You

can use it to retrieve the information from the Context of the victim process. Note that *DoPatch_callbackStop* can also modify the registers of the application. COraculum class worries of updating the victim's Context before resuming its execution (for example you might need to change EIP so as to simulate a JMP or to change some other register).

- ✓ *DoPatch_callbackResume* is called just before resuming the process. For example it is used to write back the original bytes of the application (so as it will continue undisturbed) or to write the collected results on the COraculum message stack.

Resuming, the actions of the three callbacks are then:

1. set the EBFE loop DoPatch_callback
2. reach the loop and stop DoPatch_callbackStop
3. resume the process DoPatch_callbackResume

➲ The 3 callbacks are joint and must be used to work on a single EBFE loop. If you want to set other callbacks for different patterns / EBFE loops, you must reset the pattern stack (using FlushPatternStack() method) and then add the new callbacks using SetCallBack() again.

```
Oraculum.AddPattern(...); //add pattern 1 information
Oraculum.SetCallBacks( DoPatch_callback_1,
DoActionPatch_callbackStop_1, DoActionPatch_callbackResume_1);

//perform patching actions for pattern_1 using FindPattern, DoPatch as in examples before

Oraculum.FlushPatternStack();

Oraculum.AddPattern(...); //add pattern 2 information
Oraculum.SetCallBacks( DoPatch_callback_2,
DoActionPatch_callbackStop_2, DoActionPatch_callbackResume_2);

//perform patching actions for pattern_2
```

➲ The COraculum class worries about setting access rights to the memory, before calling the callbacks, and restores original rights just after. Then virtually these callbacks can access to any location of the process's memory.

➲ You as developer must use the COraculum's ReadProcessMemory and WriteProcessMemory methods, which also checks for permission rights transparently. These two methods must be used exactly as they original Win32 counterparts.

➲ Often writing an Oraculum means writing these callbacks only.

- **Patching methods.** I grouped together the methods IsValid(), FindPatterns(), DoPatch() and KillProcess() because often they are used as following:

```
if(Oraculum.IsValid())
       if(Oraculum.FindPatterns()>0)
              if(Oraculum.DoPatch()>=0)
                     Oraculum.KillProcess();
```

  - **FindPatterns(),** using the patterns stack (filled using AddPattern()) finds the first pattern match in the executable's file.
    It uses the Boyer/Moore/Gosper-assisted "egrep" search, with delta0 table as in original paper (CACM, October, 1977) and following adaptations (Horspool, Soft. Prac. Exp., 1982 and Apostolico/Giancarlo, Siam. J. Comput., February 1986). I improved and adapted the

sources of the GSAR command (GSAR: General Search and Replace Utility, available also in this tutorial's archive)[3].
FindPatterns() returns the number of matches found.

- **DoPatch(),** calls the first callback (*DoPatch_callback*) and resumes the process, then stay in a loop waiting for the program to fall into the EBFE loop (that is EIP is constant and equal to the patched address).
  When the trap is reached (the victim process loops on the same EIP), it suspend the thread and call the other two callbacks (*DoPatch_callbackStop* and *DoPatch_callbackResume*).
  These two latter callbacks are called using a code like this one:

```
SuspendThread(m_pi.hThread);
GetThreadContext(m_pi.hThread, &m_processContext);

////////////////////////////////////////////////////////////////////////
m_fpDoActionPatch_callbackStop(this); //function pointer to the Stop callback
////////////////////////////////////////////////////////////////////////

SetThreadContext(m_pi.hThread, &m_processContext);
ResumeThread(m_pi.hThread);

////////////////////////////////////////////////////////////////////////
m_fpDoActionPatch_callbackResume(this); //function pointer to the Resume callback
////////////////////////////////////////////////////////////////////////
```

Returned value is >=0 if all is fine.

- **KillProcess(),** simply kill the victim's process.

➲ While these three last functions might be quite simple what it is not is the common way to use them. Consider the following piece of code:

```
//Do the patch at the desired location, and create the suspended process, using
//CreateProcess API.
if(Oraculum.IsValid()) //all initialization have been done?
        if(Oraculum.FindPatterns()>0) //find one of the patterns given with AddPattern
                //do the hard work and invokes callbacks when required.
                if(Oraculum.DoPatch()>=0)
                        Oraculum.KillProcess(); //closes the process
```

The program check if the Oraculum has been initialized properly (pattern added, callback assigned and a valid exe file given), and then find the patterns. If one of them is present in the executable file then do the patching calling the callbacks. At the end kill the process or start with another triplette of callbacks..

[3] What I added is the C++ conversion and the memory searching, plus some other optimizations..

4.3 Support methods of COraculum

Writing a callback might become complex depending on the complexity of the patch (look for example the RoomRover's example in section **Fehler! Verweisquelle konnte nicht gefunden werden.**). To help getting all the required information COraculum class exposes different things you might need.

NOTE
1. All these functions are meant to be used from within the callbacks, so all of the assume you have access to the memory locations you want to patch, otherwise you must directly use the VirtualProtectEx API. 2. The callbacks receive a pointer to the calling COraculum object so you can access these functions using that object (see example in 3.2).

- `PROCESS_INFORMATION* GetPI();` returns the PROCESS_INFORMATION structure, it's useful for any direct process manipulation you might need (e.g. used for ReadProcessMemory and WriteProcessMemory)

- `BMG_gsar* GetBMG_GSar();` returns a pointer to the BMG_gsar class previously described which has some public methods (see the file gsar.h in this archive). I placed it for future needs but still never found a case where I would need direct access to this internal object. So you should never use it.

- `PE_EXE* GetPE();` returns a pointer to the internal object used to handle PE Header stuffs. It's a code originally written by Matt Pietrek, which I adapted and expanded a little. It has a lot of interesting public methods you could directly use into your callbacks for special purposes (are quite intuitive).
  - ✓ Functions returning IMAGE_NT_HEADERS fields:
    - `GetMachine( void )`
    - `GetNumberOfSections( void )`
    - `GetTimeDateStamp( void )`
    - `GetCharacteristics( void )`
  - ✓ Functions returning IMAGE_OPTIONAL_HEADER fields:
    - `GetSizeOfCode( void )`
    - `GetSizeOfInitializedData( void )`
    - `GetSizeOfUninitializedData( void )`
    - `GetAddressOfEntryPoint( void )`
    - `GetBaseOfCode( void )`
    - `GetBaseOfData( void )`
    - `GetImageBase( void )`
    - `GetSectionAlignment( void )`
    - `GetFileAlignment( void )`
    - `GetMajorOperatingSystemVersion( void )`
    - `GetMinorOperatingSystemVersion( void )`
    - `GetMajorImageVersion( void )`
    - `GetMinorImageVersion( void )`
    - `GetMajorSubsystemVersion( void )`
    - `GetMinorSubsystemVersion( void )`
    - `GetSizeOfImage( void )`
    - `GetSizeOfHeaders( void )`
    - `GetSubsystem( void )`
    - `GetSizeOfStackReserve( void )`
    - `GetSizeOfStackCommit( void )`

- `GetSizeOfHeapReserve( void )`
- `GetSizeOfHeapCommit( void )`
- `GetDataDirectoryEntryRVA( DWORD id )`
- `GetDataDirectoryEntryPointer( DWORD id )`
- `GetDataDirectoryEntrySize( DWORD id )`
- `GetReadablePointerFromRVA( DWORD rva )`

✓ Useful functions assisting in address translations:

- `RVAToFileOffset( DWORD rva )`
- `FileOffsetToRVA(DWORD offset)`

- `GetProcessContext();` return the process Context (already read by the COraculum class), you often need it to get registers values.

- `GetProcessBuffer();` return the buffer which COraculum uses to store information read from the registers (for example through a ReadProcessMemory into a the memory location pointed by a register). This buffer could be reused as many times you wish. It's length is equal to system's constant MAX_PATH.

- `GetLastOffset();` return the last memory offset where one of the pattern has been found, it is the memory location which will be passed to *DoPatch_callback*.

- `FileOffsetToRVA(DWORD offset);` utility function useful to convert file offsets to RVA addresses (it calls the corresponding method in PE_EXE).

- `RVAToFileOffset( DWORD rva );` utility function useful to convert RVA addresses to file offsets (it calls the corresponding method in PE_EXE).

- `BMG_MemorySearch(HANDLE hProcess, DWORD startAddr, DWORD endAddr, TextString SearchPatt);` this function does a pattern search in memory between the startAddr and the endAddr locations. SearchPatt must follow the same syntax used by AddPattern.
  This method returns the number of matches while the offset of the last result must be retrieved using the GetLastOffset(). For example:

```
nMatches=oraculum->BMG_MemorySearch(
                oraculum->GetPI()->hProcess, startAddr, endAddr,
                ":x8A:x45:x02:x84:xC0:x75:x1D");
DWORD foundMemoryOffset=(DWORD)oraculum->GetLastOffset();
printf("substitution done at %X\n", foundMemoryOffset);
```

- `BMG_MemorySearchReplace(HANDLE hProcess, DWORD startAddr, DWORD endAddr, TextString SearchPatt, TextString ReplacePatt);` it has the same syntax of the BMG_MemorySearch function. The ReplacePatt is replaced starting from the first byte of the SearchPatt. The two patterns could have different lengths. The memory offset at which the operation has been done must be retrieved using the GetLastOffset().

```
nMatches=oraculum->BMG_MemorySearchReplace(
                oraculum->GetPI()->hProcess, startAddr, endAddr,
                ":x8A:x45:x02:x84:xC0:x75:x1D",
               ":x8B:x84:x24:xDC:x00:x00:x00:xEB:xFE");
DWORD foundMemoryOffset=(DWORD)oraculum->GetLastOffset();
printf("substitution done at %X\n", foundMemoryOffset);
```

- `pushMessage(TextString str);` this function pushes a string into the internal messages stack. When the process ends you can pop all the messages from this stack in order to retrieve what happened (see example in section 4.1).

- `pushMessage(char *str);` do the same thing with normal C character's array

- `DECLARE_CALLBACK_PROTOTYPES(DoPatch_cb, DoPath_cbStop, DoPatchcbResume)` it's a macro you should use before the main() method of an Oraculum to declare callbacks function names (see the examples in sections **Fehler! Verweisquelle konnte nicht gefunden werden.** and **Fehler! Verweisquelle konnte nicht gefunden werden.**).

- `ReadProcessMemory and WriteProcessMemory` are wrappers of their standard Win32 counterparts, but access rights and exceptions are controlled before doing the operation.

- `KillisDebuggerPresent;` it is a method that patches the isDebuggerPresent kernel API in memory so as to always return false. It is especially useful when debugging a loader (usually when you are writing it) on a victim which checks if it is being debugged (for example packed programs). The COraculum doesn't uses debugAPIs so it will not needed when you'll release the oraculum. This method returns the number of written bytes, if returns 0 something has gone wrong . The function is quite simple so I also provided a separate isDebugPresent self-explanatory example where you have two buttons (look in this package). You can take the function out of course of the framework for your uses. This API can be called anywhere in the loader you are writing before or after the initial call to Setup().

NOTE

The framework is meant to be **MFC-free**, so I had to use another string manipulation array; the TextString class, you might have noticed above, is a basic string manipulation array with some handy operators (it works like normal C++ streams) so you can write things like the following:

```
TextString str;
DWORD dwPatchAddress = 0x004023D3;
char buffer[5]=”huh!”;
...
str << “Message “ << idx << “) You have patched the address: “ << dwPatchAddress
       << “ and its value was: “
       << buffer << “\n”;
```

For which the output would be:

```
Message 1) You have patched the address: 0x004023D3 and its value was: huh!
```

5 Writing an Oraculum for a simple application in Assembler

For those of you really accustomed to ASM and not still able to program in C/C++ there's here an oraculum written in ASM for reference and comparison. An Oraculum indeed in it's simpler form is nothing more than a loader with a specific scope and it's not quite complex to write it in assembler, but making an oraculum able to wait unpack the application, and performing search& replace functions might be more complex.

NOTE

Thanks to **HackeRMaN** for these sources. You can find the whole sources and the fishme used to test the approach (the same used before in this document) into the folder `oraculum_asm\` of the sources accompanying this tutorial.

First of all the target is another time the Fishme.exe used before in this document

```
<-------------Code Snippet oraculum.Asm------------->
.386
.model flat, stdcall  ;32 bit memory model
option casemap :none  ;case sensitive
include oraculum.inc

.code

start:

        invoke GetModuleHandle,NULL
        mov    hInstance,eax
        invoke InitCommonControls
        invoke DialogBoxParam, hInstance, IDD_DIALOG1, NULL, addr DlgProc,NULL
        invoke ExitProcess, 0

;#########################################################################

DlgProc proc hWin:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM

        mov eax,uMsg
        .if eax==WM_INITDIALOG

        .elseif eax==WM_COMMAND

        .if wParam == 1001
                ;zeroes the memory..
                invoke RtlZeroMemory, addr processinfo, sizeof PROCESS_INFORMATION
                invoke RtlZeroMemory, addr Startup, sizeof STARTUPINFO
                invoke RtlZeroMemory, addr processcontext, sizeof CONTEXT

                ;Create a process and load the fishme in it, and
                ;immediately suspend the thread (pause it)
                invoke CreateProcess, ADDR filename, NULL, NULL, NULL, NULL, CREATE_SUSPENDED, NULL,
                                NULL, ADDR Startup, ADDR processinfo

                ;Creation of new process failed?
                cmp eax,0
                je hell ;jump to error message if it has failed

                ;File is found
                invoke MessageBox, hWin, addr filefound, addr found, MB_ICONINFORMATION

                ;Read the original value of the location [0040142D]
                invoke ReadProcessMemory, processinfo.hProcess , 0040142Dh,
                                addr OriginalCode, HALT_SIZE, NULL

                ;Using JMP -2 trick by yAtEs
                ;Write the HALD_CODE[EBFE] in the same location
                invoke WriteProcessMemory, processinfo.hProcess, 0040142Dh,
                                addr HALT_CODE, HALT_SIZE,byteswritten
```

```
		;Set up permissions to get the context.
		mov processcontext.ContextFlags, CONTEXT_FULL

		;Let the process run happily ;)
		invoke ResumeThread, processinfo.hThread

		;Now, according to the yAtEs tutorial the program will run happily till
		;it fall into our properly placed JMP -2 code then the EIP will not change
		;anymore and the program will stay there forever.
		;The following code checks this using a loop and the GetThreadContext to get
		;the EIP value.
		;When the GetThreadContext fails means that the application has been closed,
		;for example it happens when for some reason the application doesn't pass
		;through our trap.
	hello:
		invoke GetThreadContext, processinfo.hThread, addr processcontext
		test eax,eax
		je hell

		;When we reach the proper point we are in the place we patched!
		.if processcontext.regEip==0040142Dh

			;Fills with zero the buffer
			invoke RtlZeroMemory, addr buffer,SERIAL_SIZE

			;Refresh the thread context with the last registers values
			invoke GetThreadContext, processinfo.hThread, addr processcontext

			;Read the memory pointed by EAX and places into a buffer.
			;Remember that EAX contains the address of the serial's string.
			invoke ReadProcessMemory, processinfo.hProcess, processcontext.regEax,
					addr buffer,SERIAL_SIZE, NULL

			;Show the inputten Serial
			invoke MessageBox, hWin, addr buffer, addr yrserial, MB_ICONINFORMATION

			;Fills with zero again to make our job clean
			invoke RtlZeroMemory, addr buffer, SERIAL_SIZE
			;Read the memory pointed by ECX and places into a buffer.
			invoke ReadProcessMemory, processinfo.hProcess, processcontext.regEcx,
				addr buffer, SERIAL_SIZE, NULL

			;Show the real serial
			invoke MessageBox,hWin, addr buffer, addr realserial,MB_ICONINFORMATION

			;Restore the original applications bytes so the application can
			;continue running.
			invoke WriteProcessMemory, processinfo.hProcess, 0040142Dh,
					addr OriginalCode, HALT_SIZE, byteswritten
			;job is done so we can exit
			invoke ExitProcess, 0
		.endif

		;Sleep a while among requests so as to not slowdown the system
		invoke Sleep, 10
		jmp hello
	hell:
		invoke MessageBox, hWin, addr filenot, addr found, MB_ICONINFORMATION
		ret
		;////////////////////////////////////////////////////////////////////

	.endif
	.elseif eax==WM_CLOSE
		invoke EndDialog, hWin, 0
	.else
		mov eax,FALSE
		ret
	.endif
	mov eax,TRUE
	ret

DlgProc endp
```

```
end start

<-------------End Code Snippet oraculum.Asm------------->
```

And the .inc file is as following

```
<-------------Code Snippet oraculum.Inc------------->

include windows.inc
include kernel32.inc
include user32.inc
include Comctl32.inc
include shell32.inc

includelib kernel32.lib
includelib user32.lib
includelib Comctl32.lib
includelib shell32.lib

DlgProc                PROTO   :HWND,:UINT,:WPARAM,:LPARAM

.const

IDD_DIALOG1            equ 101

;#########################################################################
.data
        Startup         STARTUPINFO <>
        processinfo     PROCESS_INFORMATION <>
        processcontext  CONTEXT <>
        filefound    db "Fishme.exe is found mate, lets carry on!",0
        found        db "www.hackerman.knows.it",0
        filenot      db "File Is Not Found",0
        yrserial     db "Your Have Inserted This Serial:",0
        realserial   db "But The Real Serial Is:",0
        filename     db "Fishme.exe",0
        HALT_SIZE    db 2
        SERIAL_SIZE  db 20
        OriginalCode dd 4 dup(?)
        buffer       dd 20 dup(?)
        HALT_CODE    dd 0FEEBh ;in reverse order "EBFE"

.data?
        byteswritten dd ?
        hInstance    dd ?

;#########################################################################

<-------------End Code Snippet oraculum.Inc------------->
```

6 Creating an Invisible Oraculum in Assembler

This tutorial is also available at [10].
The goal is to make the Oraculum virtually invisible to the end-user. To do so the Oraculum will be designed to function with little user input and no output. And the design of the Oraculum will be changed to allow for direct memory manipulation.
The Oraculum will be developed in such a way that it will function as a framework, allowing it to be adapted to any target by changing only a few variables.
Finally the Oraculum will be tested on two separate targets. This will demonstrate the adaptability of the framework and the overall flexibility of an Oraculum.

6.1 The Problem

Probably the most common program registration scheme is the serial registration. A serial is either unique or non-unique depending on the registration method. When working with non-unique serials it is simple enough to find and distribute those serials allowing for multiple registrations. A unique serial is a harder task. A unique serial is calculated based on one, or more, unique identifiers. These identifiers can be the registration name, computer name, or even hardware id. There are a few options that can be taken when a program uses a unique serial registration.
The first option is to calculate a serial based on a unique identifier. This identifier&serial combination can then be used to register multiple copies of the program. This attack fails when the unique identifier is either unable to be modified, complicated to modify, or even unknown. This attack also fails when the identifier&serial combination becomes known to the programmer. The programmer can then add either the serial or identifier to a blacklist, disallowing any registration with that identifier&serial for any future program versions.
The next option is to patch the registration scheme allowing for any serial to be used to register the program. This requires the analysis of the registration scheme, and modification of the original executable. This attack will fail if the executable saves the identifier&invalid-serial to a specific location. If the program is updated the registration scheme is not modified within the new executable, the program will then read the invalid serial from a file or the registry. This will cause the program to revert back to unregistered. Another flaw in this attack is that programs that update frequently often check for the presence of a valid non-modified executable before allowing an update. Even replacing the modified executable with a non-modified version may not keep the program registered. For every update there will be the need to re-analyze and re-patch the executable.
Another attack method is the development of a keygen. This will calculate a specific key based on submitted unique identifiers. This is the most effective method of defeating a registration scheme because there is no modification of the program, there is no single identifier and/or serial that can be blacklisted, and it will work for future updates until the registration scheme is changed. This method, however, also requires the most work. The keygen developer needs to first understand the registration scheme, find out how to effectively reverse the scheme to allow for creating unique keys, and then they need to code the keygen. This can often be a time consuming project.

6.2 The Solution

Often times when a program is protected by a serial registration scheme the unique serial is compared against the invalid serial. When that takes place, many times the serial is fully calculated and resident in memory. This is a weakness that we can, and will, take advantage of. If we find out where in memory the serial is located we can read it out and register the program. These are often

the steps taken when serial fishing a program. But as we know, the serial fishing method has its limitations. We want to develop a method to easily read that serial from memory.
This is a program that loads an executable, stops execution at a determined spot, and then reads the valid serial out of memory. The unique serial is displayed to the user. The program then continues as normal allowing the user to enter in the valid unique serial. The paper included both the use of, and framework for, the Oraculum. Developing an Oraculum is an easy method to allow the end-user to generate a valid unique serial. There is no modification of the original executable and no risk of blacklisting.

6.3 The Goal

In this section we will take the Oraculum approach one step further, we will remove any steps the end-user may need to take to register the program. The valid serial is found by reading it out of a specific memory location or register. We then replace the invalid serial by writing in memory overtops of the invalid serial location. Rather than displaying the unique serial to the end-user we will find the unique serial in memory and, before any comparison takes place, replace the invalid serial with the valid one. This way any registration techniques become "invisible" to the user. In this paper we will take the Oraculum code written by Hackerman and adapt it to be a framework for our invisible Oraculum. This will allow us to adjust this Oraculum for any target, just by changing a few variables. We will then write a few Oraculums for different targets to demonstrate the methods used in the Oraculum and to display the ease of using a framework for this task.

6.4 Part1: Creating a framework.

A framework is code that almost already completely written, yet it is written in an abstract way that will allow you to easily modify and develop it. To begin our framework we need to decide how we want our new Oraculum to work. The current function of the Oraculum is shown below. To the right of it is the desired function of our Invisible Oraculum:

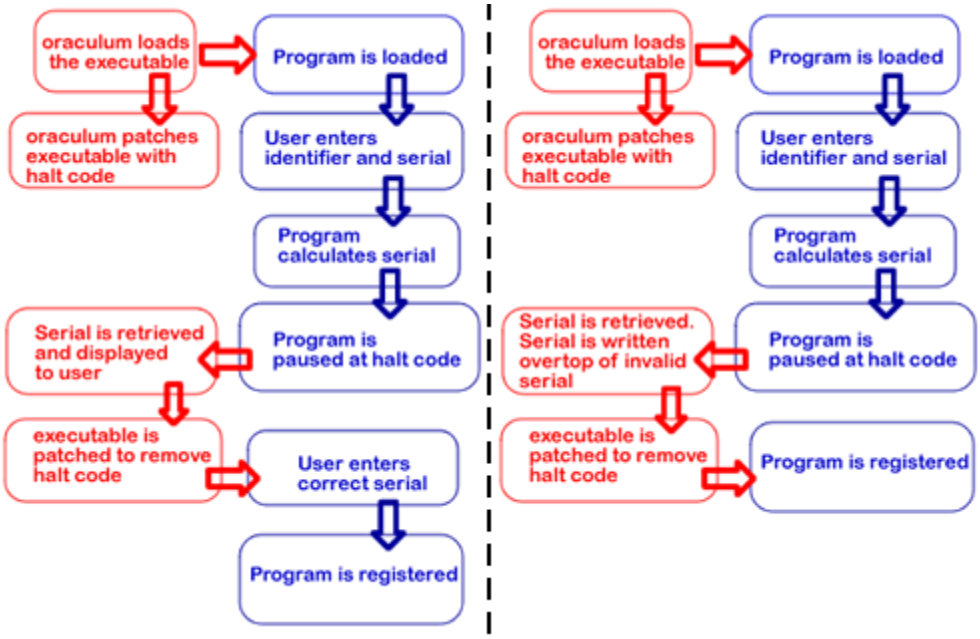


Figure 1 - Oraculum flowchart versus Invisible Oraculum flowchart

Developing a framework for the invisible Oraculum will be easy since we have much of the source already available to us, starting from the code already described in previous sections.
To begin you will need the MASM32 package [11] and RadASM [12]. We will modify in this section the code already used in Section 5 (you can find it in the document version with sources, in the folder Sources/oraculum_asm/).
Begin by opening RadASM. From the File menu choose "Open Project". Navigate to the oraculum_asm/ folder and select **oraculum.rap**. You should now see the following Figure 2:

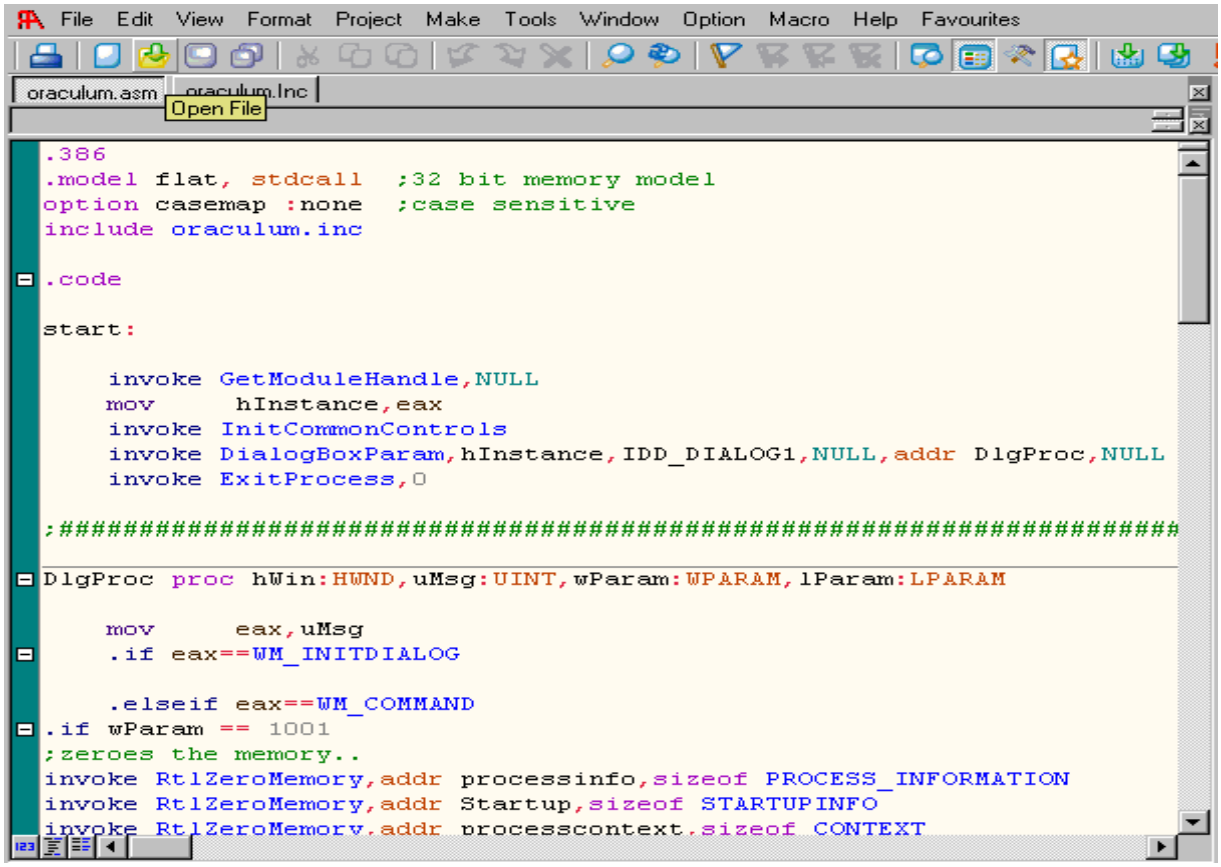


Figure 2 – Oraculum.asm code

NOTE
Within the project you will see 4 files: Oraculum.asm: Our main body of code. Contains all the functions that we will use to open, modify, and register the program Oraculum.inc: This is the include file. This holds our defined variables and our undefined variables that will be used in oraculum.asm. This file is imported into oraculum.asm with the line: include oraculum.inc Oraculum.dlg: This is our dialog menu. This is the menu you will see when you run the program. Oraculum.rc: This file contains the resources to be included in your dialog menu.

Once our project is open we are going to begin modifying it to function as a framework. We will begin by editing **oraculum.asm**. Refer back to the flowchart which illustrates how we desire our Oraculum to function. With this flowchart we can quickly determine what functions we need and do not need.

The first step will be to remove any functions that we will not use. If we want the Oraculum to be effectively invisible there is no need to display the entered serial. I removed the two invoked functions that read and displayed the entered serial. These are the lines removed:

```
;Read the memory pointed by EAX and places into a buffer.
;Remember that EAX contains the address of the serial's string.
invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEax,addr buffer, SERIAL_SIZE, NULL
;Show the inputed Serial
invoke MessageBox,hWin,addr buffer,addr yrserial,MB_ICONINFORMATION
```

Now, to build our framework we are going to begin by moving all variables outside of the **oraculum.asm** and into **oraculum.inc**. These will be any values that will change depending on what program we develop our Oraculum for.

The values I moved from **oraculum.asm** to **oraculum.inc** are:

- Location to set the breakpoint: Now defined as variable **Breakpoint**
- Location of good serial: Defined as variable **Goodlocation**
- Location of bad serial: Defined as variable **Badlocation**

oraculum.inc file now looks like in Figure 3:

```
;#########################################################################

.data
Startup         STARTUPINFO <>
processinfo     PROCESS_INFORMATION <>
processcontext  CONTEXT <>
filefound    db "Exe file found. Begin",0
found        db "Found",0
filenot      db "File not found",0
yrserial     db "Bad Serial",0
realserial   db "Good Serial",0
filename     db "fishme.exe",0
OriginalCode dd 2 dup(?)     ;Buffer to hold original code
buffer       dd 20 dup(?)    ;Buffer to hold our serial
HALT_CODE    dd 0FEEBh       ;BYTES to write on BP location
HALT_SIZE    dd 2            ;Size of our HALT_CODE
SERIAL_SIZE  dd 10           ;Length of our serial in hexadecimal
Breakpoint   equ 0040249Dh   ;Location to set BP on
Goodlocation dd 00404060h    ;Location of Good Serial
Badlocation  dd 00404070h    ;Location of Bad Serial

.data?
byteswritten dd ?
hInstance           dd ?

;#########################################################################
```

Figure 3 - Oraculum.inc file after modifcation

We now need to replace the values in the ASM file with our new variables.

Starting from the beginning of **oraculum.asm**. We will first replace all locations that reference our breakpoint directly. We will replace the absolute value of the breakpoint with our newly defined variable: **Breakpoint**

The first action taken by the oraculum is to read the original bytes out of our breakpoint location. We replace the direct value of our breakpoint with our variable:

```
;Read the original value of the location [0040142D]
invoke ReadProcessMemory,processinfo.hProcess , 0040142Dh,addr OriginalCode,HALT_SIZE,NULL
```

Becomes:

```
;Read the original value of the location [0040142D]
invoke ReadProcessMemory,processinfo.hProcess , Breakpoint,addr OriginalCode,HALT_SIZE,NULL
```

Next we overwrite the bytes at our breakpoint location with our halt code. Modify the code so **Breakpoint** variable replaces the absolute value:

```
;Write the HALT_CODE[EBFE] in the same location
invoke WriteProcessMemory,processinfo.hProcess,0040142Dh,addr HALT_CODE,HALT_SIZE,byteswritten
```

Becomes:

```
;Write the HALT_CODE[EBFE] in the same location
invoke WriteProcessMemory,processinfo.hProcess,Breakpoint,addr HALT_CODE,HALT_SIZE,byteswritten
```

As the program runs we will constantly be checking to see if it has reached our breakpoint. That test is taken place here:

```
;When we reach the proper point we are in the place we patched!
.if processcontext.regEip==0040142Dh
```

And we replace the direct breakpoint reference:

```
;When we reach the proper point we are in the place we patched!
.if processcontext.regEip==Breakpoint
```

When we have successfully replaced the serial we need to restore the original bytes to our breakpoint location so:

```
;Restore the original applications bytes so the application can continue running.
invoke WriteProcessMemory,processinfo.hProcess, 0040142Dh, addr OriginalCode,HALT_SIZE, byteswritten
```

Becomes:

```
;Restore the original applications bytes so the application can continue running.
invoke WriteProcessMemory,processinfo.hProcess, Breakpoint, addr OriginalCode,HALT_SIZE, byteswritten
```

We are finished replacing our breakpoint with our variable we defined in our .inc file. So we will now move onto modifying to code to allow for read and write of memory locations. Specifically, the two new variable locations ***Goodlocation*** *and* ***Badlocation*** *we defined in our .inc file.*

We currently read the correct serial from the registry and then display it to the user. What we want to do is read the serial from a memory location and then write it back to another location. Let's begin by commenting both the invoke ReadProcessMemory and invoke MessageBox lines out.

Next we are going to add a new invoke ReadProcessMemory but instead of reading from processcontext.regEcx, it will read from the location defined in ***Goodlocation*** *and store it in the variable* ***buffer****.*

Now we are going to add a invoke WriteProcessMemory. This will write the value stored in buffer to our location in **Badlocation**. Your code should now look like Figure 4.

```
;Fills with zero again to make our job clean
invoke RtlZeroMemory,addr buffer,SERIAL_SIZE
;###################COMMENTED OUT#################################
;Read the memory pointed by ECX and places into a buffer.
;invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEcx,addr buffer, SERIAL_SIZE, NULL
;Show the real serial
;invoke MessageBox,hWin,addr buffer,addr realserial,MB_ICONINFORMATION
;#################################################################


;Read the good serial from memory location: goodlocation and store it into buffer
invoke ReadProcessMemory,processinfo.hProcess,goodlocation,addr buffer, SERIAL_SIZE, NULL
;Write the good serial from buffer ontop of bad serial located at: badlocation.
invoke WriteProcessMemory,processinfo.hProcess, badlocation, addr buffer, SERIAL_SIZE, NULL


;Restore the original applications bytes so the application can continue running.
invoke WriteProcessMemory,processinfo.hProcess, Breakpoint, addr OriginalCode,HALT_SIZE, byteswritten
```

Figure 4 - Oraculum.asm after modification

One more modification; We will add another invoke WriteProcessMemory within our COMMENTED OUT section. This WriteProcessMemory will write our serial into a register. This will allow us to modify the Oraculum depending on whether the serial is stored in memory or in a register. We can make that modification by just by commenting or uncommenting code.

The final changed code for oraculum.asm should be:

```
;###################COMMENTED OUT#################################
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A REGISTRY

;Read the memory pointed by ECX and places into a buffer.
;invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEcx,addr buffer, SERIAL_SIZE, NULL

;Uncomment to show the real serial in a messagebox
;invoke MessageBox,hWin,addr buffer,addr realserial,MB_ICONINFORMATION

;Write the serial stored in buffer to memory pointed by EAX.
;invoke WriteProcessMemory,processinfo.hProcess,processcontext.regEax,addr buffer, SERIAL_SIZE, NULL
;#################################################################

;#################################################################
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A MEMORY LOCATION

;Read the good serial from memory location: goodlocation and store it into buffer
invoke ReadProcessMemory,processinfo.hProcess,Goodlocation,addr buffer, SERIAL_SIZE, NULL

;Write the good serial from buffer ontop of bad serial located at: badlocation.
invoke WriteProcessMemory,processinfo.hProcess, Badlocation, addr buffer, SERIAL_SIZE, NULL

;#################################################################
```

We have now modified the Oraculum source to function as a framework for our invisible Oraculum. All our variables have been moved into the .inc file, allowing us to change only a few values and adapt the program for other targets.

6.5 Part 2: Proof of Concepts example 1

We have developed our invisible Oraculum and it is now time to put it into practice. Our first target will be a very simple serial fish: deibiz_xxl's Learn The First Few Tricks #1 [13]. To follow along you will need to open your Ollydbg.

Open Ollydbg and then open LTFFT.exe in Olly. You will be in Figure 5.

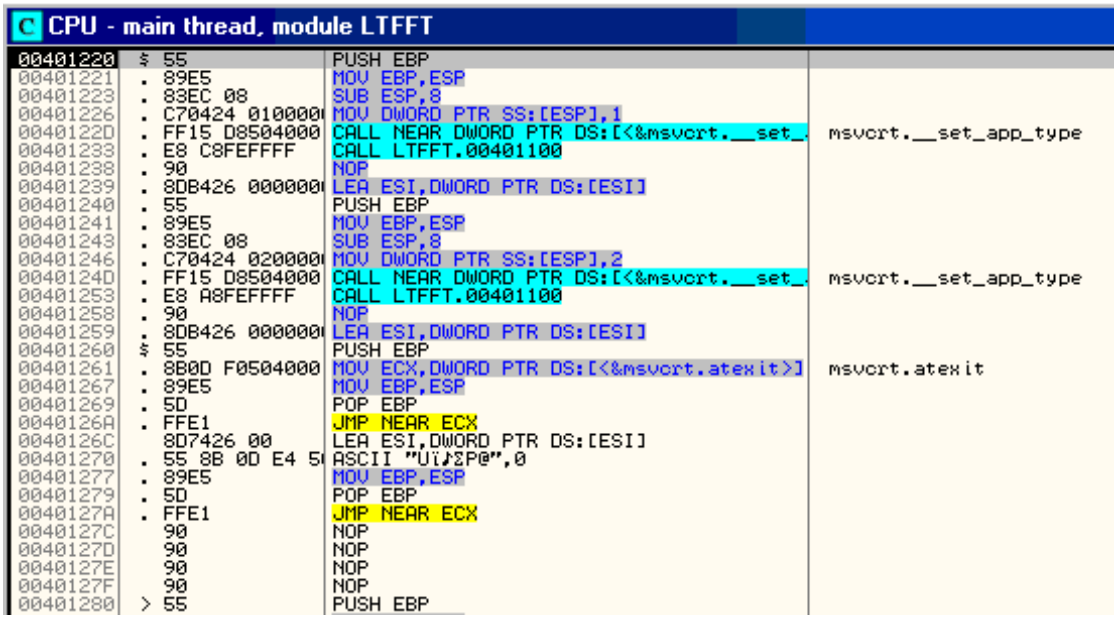


Figure 5 - LTFFT.exe loaded in Ollydbg

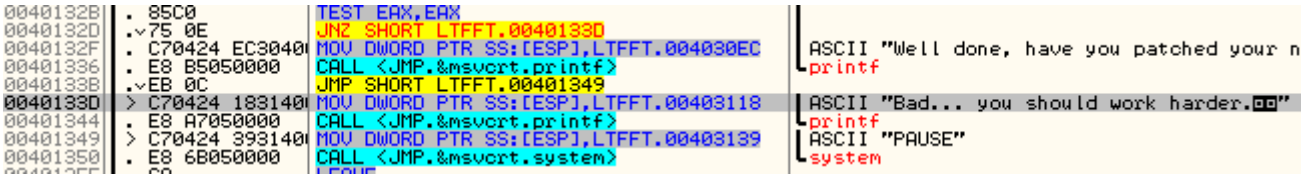
Begin by running the program and entering any serial. You will see the following:

```
Type your password: ARTeam
Bad... you should work harder.
Press any key to continue . . .
```

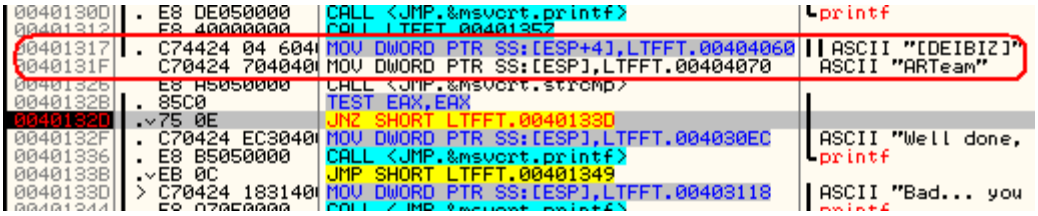
NOTE

If you are reading this paper I will assume you know how to search for this "BadBoy" message in Olly. If you are unsure of how to proceed, visit: http://tutorials.accessroot.com and start from Beginner Olly Tutorial Part 1

*Restart the program in Olly and search for the message: "**Bad... You should work harder**". You will find the BadBoy message being referenced here:*



Looking at the information window, we find that our BadBoy was called by 40132D. So we will set a breakpoint on that location. Run the program again. When prompted; enter any serial. When you break at location 40132D look at the code in Olly:



I have circled the code that we find interesting. The program takes the correct serial stored in location 00404060 and writes it to [ESP+4], it then takes the serial we entered stored at 00404070 and writes it to [ESP]. The program then does a string compare and jumps to GoodBoy or BadBoy accordingly. This little bit of code is all the information we need to write our invisible Oraculum. We will set our BP on 00401317. We know the Goodlocation: 00404060 and we know the Badlocation: 00404070. We also know the length of our serial: 9 bytes (8 bytes + NULL termination byte).

In RadASM we will modify our oraculum.inc file to what you can see in Figure 6:

```
;#########################################################################

.data
Startup         STARTUPINFO <>
processinfo     PROCESS_INFORMATION <>
processcontext  CONTEXT <>
filefound    db "Exe file found. Begin",0
found        db "Found",0
filenot      db "File not found",0
yrserial     db "Bad Serial",0
realserial   db "Good Serial",0
filename     db "LTFFT.exe",0
OriginalCode dd 2 dup(?)     ;Buffer to hold original code
buffer       dd 20 dup(?)    ;Buffer to hold our serial
HALT_CODE    dd 0FEEBh       ;BYTES to write on BP location
HALT_SIZE    dd 2            ;Size of our HALT_CODE
SERIAL_SIZE  dd 9            ;Length of our serial in hexadecimal
Breakpoint   equ 00401317h   ;Location to set BP on
Goodlocation dd 00404060h    ;Location of Good Serial
Badlocation  dd 00404070h    ;Location of Bad Serial

.data?
byteswritten dd ?
hInstance           dd ?

;#########################################################################
```

Figure 6 - Oraculum.inc modified for LTFFT.exe

Once our oraculum.inc file has been modified we can build oraculum.exe. In RadASM choose MAKE from the menu, and then choose BUILD from the drop down menu. The program should then compile.

- If you have errors double check your capitalization, MASM is very particular goodlocation is not the same as Goodlocation.
- Once the Oraculum is built you will find it in the same place as oraculum.rap.

Copy oraculum.exe and place it in the same directory as LTFFT.exe. Run the Oraculum and press the “Fish me” button to load LTFFT.exe. Enter any serial and press Enter:

```
Type your password: ARTeam
Well Done, have you patched your name?
Press any key to continue . . .
```

The Oraculum worked perfectly! It successfully replaced the bad serial located at 404060 with the good serial located at 404070.

6.6 Part 2: Proof of Concepts example 2

To show how easy it is to modify the Oraculum now that we converted it to a framework we are going to try one more target: DaXXoR 101's D_KeygenMe [14].

This keygen stores the serials within the registers. This means we will have to comment out the current code and uncomment the code that modifies the registers. Your Oraculum code should now look like this:

```
;##################COMMENTED OUT#################################
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A REGISTRY

;Read the memory pointed by ECX and places into a buffer.
invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEcx,addr buffer, SERIAL_SIZE, NULL

;Uncomment to show the real serial in a messagebox
;invoke MessageBox,hWin,addr buffer,addr realserial,MB_ICONINFORMATION

;Write the serial stored in buffer to memory pointed by EAX.
invoke WriteProcessMemory,processinfo.hProcess,processcontext.regEax,addr buffer, SERIAL_SIZE, NULL
;################################################################

;################################################################
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A MEMORY LOCATION

;Read the good serial from memory location: goodlocation and store it into buffer
;invoke ReadProcessMemory,processinfo.hProcess,Goodlocation,addr buffer, SERIAL_SIZE, NULL

;Write the good serial from buffer ontop of bad serial located at: badlocation.
;invoke WriteProcessMemory,processinfo.hProcess, Badlocation, addr buffer, SERIAL_SIZE, NULL

;################################################################
```

We begin by loading keygenme.exe into Olly. Run the program and enter any name and serial combination. You will get the message: “Bad Serial”. We search for that message in Olly and we end up here:

```
0040249E  . 56              PUSH ESI
0040249F  . E8 C4470000     CALL keygenme.00406C68
004024A4  . 83C4 08         ADD ESP,8
004024A7  . 85C0            TEST EAX,EAX
004024A9  .v75 15           JNZ SHORT keygenme.004024C0
004024AB  . 6A 40           PUSH 40                          Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004024AD  . 68 0E114100     PUSH keygenme.0041110E           Title = "Good"
004024B2  . 68 13114100     PUSH keygenme.00411113           Text = "Good, Now Make a Keygen!"
004024B7  . 6A 00           PUSH 0                           hOwner = NULL
004024B9  . E8 B2DC0000     CALL <JMP.&USER32.MessageBoxA>   MessageBoxA
004024BE  .vEB 5A           JMP SHORT keygenme.0040251A
004024C0  > 6A 40           PUSH 40                          Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
004024C2  . 68 2C114100     PUSH keygenme.0041112C           Title = "Nope"
004024C7  . 68 31114100     PUSH keygenme.00411131           Text = "Bad Serial"
004024CC  . 6A 00           PUSH 0                           hOwner = NULL
004024CE  . E8 9DDC0000     CALL <JMP.&USER32.MessageBoxA>   MessageBoxA
004024D3  .vEB 45           JMP SHORT keygenme.0040251A
```

We see that the BadBoy was called from 004024A9. So lets set a BP just above that. We will set our BP on the CALL located at 0040249F.

Run the program again and when we break take a look at the registers:

```
EAX 0012FCF8 ASCII "1234567"
ECX 00000007
EDX 00130608
EBX 00000006
ESP 0012FCD0
EBP 0012FD38
ESI 0012FD10 ASCII "-1791928899"
EDI 0012FCF6
EIP 0040249F keygenme.0040249F
```

We see that the ESI register contains the correct serial, while the EAX register contains our entered serial. With our new framework we can easily use our oraculum to overwrite the EAX register with the value from ESI. We must now decide where to put our Halt_code? We will not put it at the same place we put our breakpoint: 0040249F because looking directly above our breakpoint we see that our serials are being PUSHed onto the stack:

```
0040249D   . 50             PUSH EAX
0040249E   . 56             PUSH ESI
0040249F   . E8 C4470000    CALL keygenme.00406C68
```

So instead we will put our Halt_code on 0040249D just after the serial is calculated but before the serials are pushed onto the stack. Back in RadASM modify these two invokes in your oraculum.asm file to the following:

```
;Read the memory pointed by ESI and places into a buffer.
Invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEsi ,addr buffer, SERIAL_SIZE, NULL

;Write the serial stored in buffer to memory pointed by EAX.
Invoke WriteProcessMemory,processinfo.hProcess,processcontext.regEax ,addr buffer, SERIAL_SIZE, NULL
```

Now modify the oraculum.inc file as in Figure 7.

```
;#########################################################################

.data
Startup         STARTUPINFO <>
processinfo     PROCESS_INFORMATION <>
processcontext  CONTEXT <>
filefound    db "Exe file found. Begin",0
found        db "Found",0
filenot      db "File not found",0
yrserial     db "Bad Serial",0
realserial   db "Good Serial",0
filename     db "keygenme.exe",0
OriginalCode dd 2 dup(?)     ;Buffer to hold original code
buffer       dd 20 dup(?)    ;Buffer to hold our serial
HALT_CODE    dd 0FEEBh       ;BYTES to write on BP location
HALT_SIZE    dd 2            ;Size of our HALT_CODE
SERIAL_SIZE  dd 12           ;Length of our serial in hexadecimal
Breakpoint   equ 0040249Dh   ;Location to set BP on
;Goodlocation dd 00404060h    ;Location of Good Serial
;Badlocation  dd 00404070h    ;Location of Bad Serial

.data?
byteswritten dd ?
hInstance           dd ?

;#########################################################################
```

Figure 7 - Oraculum.inc modified for keygenme.exe

With the modified code, build oraculum.exe as you did before and place it in the same folder as keygenme.exe. When you are ready, run the oraculum. Keygenme.exe will load. You can now enter any user name and serial and press Okay.
The result: You get the GoodBoy message!
In less than 10 minutes you have created a file that can defeat this programs protection and will work for every user. There was no modification of the program and no need to analyze the serial generation routine.

7 Discussion

The real usefulness of Oraculums is an argument often raised since the first release of this document. I whish to tell once more that first of all an Oraculum is a concept, a particular type of loaders, which has a specific behaviour, that is reporting to you the real serial of the program (or switch the real and the wrong one silently). This is a thing that any loader, debugger loader or other can do, as long as it's able to read and write the CONTEXT structure at a given point in the program. Essentially then an Oraculum is a serial fisher loader nothing more, the name mean just this after all.

The EBFE trick used is just an alternative way to place a breakpoint into a program. It has some advantages versus simple software/hardware breakpoints (see [8] for further details).

Notes	EBFE Trick	Software Breakpoint	Hardware Breakpoint
Method used to stop the application at the given address.	Place an EBFE instruction at the given address	Places a CC instruction at the given address	Uses the debug registers to stop at the given address
Integrity checks on code (CRCs) might reveal its presence?	Yes	Yes	Not
Might be deleted by the application?	Yes, restoring original code	Yes, restoring original code	Yes, using SEH mechanism, to erase the debug registers
Common method used to reveal it's presence	Integrity checks	▪ Integrity checks, ▪ Presence of a CC instruction	▪ Debug registers not all equal to 0 ▪ Debug register DR7 no equal to 0
Stops the application?	Not	Yes	Yes
Are you debugging the application?	Not	Yes	Yes
It's multithread safe?	Yes	Not always	Not always

Using the above table it comes to evidence that the EBFE trick has just few advantages:

1. Bypasses those anti-debugging tricks which only check against CC byte-code.
2. Doesn't really stop the application so all the concurring thread are not altered by the breakpoint.
3. Do not really debug the application, the application is not being "debugged"
4. The timing tricks which monitor the execution time are often fooled by the fact that the application is running (not stopped) and that it's at the same time blocked.

This would point your attention to the usage of EBFE trick for difficult cases, but as I told before you can use also other breakpoint methods as well.

So summing up the Oraculums are an instruments, a concept you can build as you like, coding as you like: the way Oraculums are coded and the concept are two different things.

Think of these different ways to implement an Oraculum

1. write a patch for the target which inline the code needed to popup a MessageBox, with classical patching methods;
2. write a debug loader [9] for protected programs (e.g. asprotect) which instead of writing a patch inserts at the specified location a BP or an EBFE loop and then fishes the CONTEXT and reports it to you with a MessageBox.
3. use my C++ framework
4. use the asm framework

Consider for example this more complex situation: an application protected with AsProtect which requires a serial number to activate it (as usual). And also consider that the serial number can be fished from inside the application looking at a specific memory address or at a specific CONTEXT value.

In this case the Oraculum you would have to code would be basically a debug loader which:

1. Takes control of the application, using debug events and software breakpoints.
2. Places some software breakpoints to suspend the application in the correct place and to patch its trial limit check, so as, the users will be able to use the loader even after the trial time.
3. Uses the EBFE trick to get the context at the right moment and pop-ups the correct serial to the user (or another software breakpoint)

The base concept is always this: the modifications done by the Oraculum oblige the program to reveal the real serial (application tampering). The differences are: how you do the modifications (ASM vs code or inline patch), where (inside the program vs into a loader) and how much resilient to different releases is your work (if can work on different releases on different OS).

8 References

There are several tutorial about this argument I here will report those I found to be more interesting.

[1] David Litchfield, Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server, 2003, http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf *[the argument is slightly different but the used techniques are the same and there are some interesting news]*

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

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

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

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

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

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

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

[9] "Cracking with loaders: theory, general approach and a framework", Shub-Nigurrath, ThunderPwr, http://tutorials.accessroot.com or CodeBreakers-Journal, Vol. 2, No. 2 (2005).

[10] "Creating the Invisible Oraculum, Program Registration Through Memory Manipulation", Gabri3l of ARTeam, http://tutorials.accessroot.com

[11] MASM32 package, http://www.masm32.com/

[12] RadASM, http://www.radasm.com/

[13] deibiz_xxl's Learn The First Few Tricks #1, http://www.crackmes.de/users/deibiz_xxl/learn_the_first_few_tricks_1/

[14] DaXXoR 101's D_KeygenMe, http://www.crackmes.de/users/daxxor_101/d_keygenme/

On http://tutorials.accessroot.com there are several other tutorials with other examples of Oraculums.
The full framework for writing Oraculums is also available on that site too.

9 Conclusions

I hope you are still here to read these last sentences. I must admit 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 glue to all the concepts … you know when one starts writing you cannot stop till everything has been duly written!
At the same time there are inside this paper some advanced concepts and a quite complex C++ structure which I released together with this document. So the tutorial is also meant for several level readers.
Anyway faced with multiple protection schemes it is important to adapt. It is important to take many different and new approaches to the same problem. The Oraculum is one of those approaches: a simple method now made simpler.

Greetings

Thanks to who had the patience to read the beta versions of this tutorial, and of course to you for being still here.
I whish to tanks also who contributed to improve this tutorial:

- Gabri3l for his silent Oraculums code;
- HackeRMaN for his ASM version of the code;
- the authors of the tutorials which inspired me or which I reused: Detten and yAtEs;
- Everyone else I forgotten.