



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

Shub-Nigurrath of ARTeam

Version 1.2 - June 2005

1.	Introduction.....	1
1.2	What the hell is an Oraculum?.....	2
1.3	What is this tutorial about.....	3
2.	How to code a loader.....	3
2.2	What is a loader, and why do we need it?.....	3
2.3	How does a loader work?.....	3
2.4	A loader example.....	5
2.5	Getting the Process Context.....	6
2.6	Checking and setting the accessing rights to memory locations.....	8
2.6.1	SEH - Structured Exception Handling.....	10
2.6.2	How to find the Calculator's memory address to patch.....	11
2.6.3	Writing a loader accessing protected memory pages.....	11
2.7	Using the EB FE Trick to set Breakpoint.....	14
2.7.1	Fishing a serial from Fishme.exe.....	15
2.7.2	How to build a basic Oraculum for Fishme.exe.....	16
3.	The framework for building Oraculums.....	18
3.2	The FishMe_Oraculum.....	19
3.3	Main methods of Oraculums C++ framework.....	21
3.4	Support methods of COraculum.....	24
4.	Writing the DoubleLook for PalmDesktop Oraculum: a normal unpacked application, frequently updated.....	27
5.	Writing the RoomRover Oraculum: an UPX packed, self checking application.....	29
6.	Writing an Oraculum for a simple application in Assembler.....	33
7.	Appendix 1: cracking DoubleLook for PalmDesktop.....	36
8.	Appendix 2: cracking RoomRover.....	41
8.2	Dumping the program.....	41
8.3	Patching the self-checking "protection".....	43
8.4	Finding the serial code.....	45
	References.....	47
	Conclusions.....	48
	History.....	48

1. Introduction

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



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

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

At this point some of you might ask why I won't use the debugAPI which allows to easily set breakpoints, stop and run the application and so on. My approach doesn't use 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.

NOTE

I chosen in this tutorial to use C/C++ because I consider this language much more elegant and powerful of ASM (being an higher level language is not an opinion indeed but a matter of facts), but at this level of difficulty the programs we write can be coded in either ways, so it's a matter of programming experience and tastes what language you'll use. If you really understand the concepts it will not be that difficult to write on your own new oraculums with whatever language you like most.

Anyway as comparison a simple ASM oraculum is included at the end: consider that anyway for simple code the two languages are equivalent but consider also what happens when the situation get more complex..

1.2 What the hell is an Oraculum?

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

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

Informatically speaking, an oraculum is a loader, an external program which executes the target program and does some memory patching in order to obtain some information such as usually the serial code, and then it reports those things to the user.

An Oraculum is not a self-keygen (an application patched to reveal its real serial), because the original application isn't patched on disk, isn't only a loader because the application is closed when the required information are found (usually the real serial) and the application isn't patched to avoid limitations, it is something different, it's simply an Oraculum.. ☺



1.3 What is this tutorial about

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

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

2. How to code a loader

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

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

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

2.3 How does a loader work?

Ok, grab your MSDN and fasten your seatbelt :)

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

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

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

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

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



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

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

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

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

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

NOTE

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

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

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

to let it run, and

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

to pause it again.

The hThread handle can be found in the LPPROCESS_INFORMATION struct.

NOTE

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

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

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

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



To read from the process :

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

These information should be enough to understand the following example.

2.4 A loader example

In the example below I will code a program which open a changeme.exe file (included in this archive), change the text inside the dialog to "Shub-Nigurrath!" and finally show the original and the new strings in the DOS window.

So here we are patching a simple string into an external process, but with the same method we can also patch code bytes or dwords of course ;)

The target of the loader is to change the string shown below the OK button.

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

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

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

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

STARTUPINFO startupinfo;
PROCESS_INFORMATION processinfo;

unsigned long byteswritten;
int uExitCode;

void main() {

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

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

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

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

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



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

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

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

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

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

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

NOTE

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

That's all it takes to create your first loader.

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

2.5 Getting the Process Context

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

```
BOOL GetThreadContext(
    HANDLE hThread,           //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 (`Eax`, `Ecx`, `Edx`, `Ebx`, `Esi`, `Edi`),



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
        //registry 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("Eax=%X\n", context.Eax);
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 loaders is to manage memory read/write permissions.

2.6 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 techniques, 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 used 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 thing 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.

2.6.1 SEH - Structured Exception Handling

This indeed a very long argument for which you might find several tutorials around, starting from the MSDN site².

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
    }
}
```

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

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



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.

2.6.2 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	Image R	RWE	
01001000	00013000	calc	01000000	.text	code, import	Image R	RWE	
01014000	00002000	calc	01000000	.data	data	Image R	RWE	
01016000	00009000	calc	01000000	.rsrc	resources	Image 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.....üü...

2.6.3 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 2.4);
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() {
```



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

```
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.
//OldProtection is !=0 if the previous VirtualProtectEx has been done.
if(OldProtection!=0)
    VirtualProtectEx(processinfo.hProcess, (LPVOID)PatchAddress, PATCH_SIZE,
        OldProtection, &dummyProtection);

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

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.
//OldProtection 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.

2. 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.
3. Be able to set a breakpoint without using debugger's APIs so without setting a system breakpoint. Remember that we 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!

2.7 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 2.7.1.

2.7.1 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	
00401431	. 50	PUSH EAX	
00401432	. FF15 A8214000	CALL DWORD PTR DS:[<&MSVCRT._mbscmp>]	s2 = "Na\x90 Lm\x81 p\xFF\xFF\xFF\t" s1 = NULL _mbscmp
00401438	. 83C4 08	ADD ESP,8	
0040143B	. 85C0	TEST EAX,EAX	
0040143D	. 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	. 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 FF	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 ??

2.7.2 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
```



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

```
//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! ;-)

3. 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 some Oraculums, for the fishme.exe program and for two real applications: DoubleLook for PalmDesktop³ and RoomRover⁴.

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

3.2 The FishMe_Oraculum

I think that the best way to start is to write the same Oraculum of section 2.7.2, 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 exe file 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.
    //
}
```

³ DoubleLook for Palm Desktop any version, <http://www.companionlink.com> or http://intechhosting.com/~access/ARTeam/tools/DoubleLook_20.2587.rar

⁴ RoomRover version 1.2, <http://gamewand.com/roomrover.htm> or http://intechhosting.com/~access/ARTeam/tools/RoomRover_12.rar



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

```
// - 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.

3.3 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 2.7.2, 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. 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)⁵.

FindPatterns() returns the number of matches found.

⁵ What I added is the C++ conversion and the memory searching, plus some other optimizations..



- **DoPatch()**, calls the first callback (*DoPatch_callback*) and resumes the process, then stay in a loop waiting for the program to fall into the EBF 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 triplete of callbacks..

3.4 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 5). 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:x0C: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:x0C: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 3.2).
- `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 4 and 5).
- `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!
```

4. Writing the DoubleLook for PalmDesktop Oraculum: a normal unpacked application, frequently updated

I'll make this step simple, assuming you already read the Appendix 1. This application has some interesting characteristics by the Oraculum point of view:

1. The real serial appears clearly into one of the registers during registration
2. The application is often updated with minor builds, this oblige normal patches to be updated also, because the offset of the application changes and so it's difficult to write an universal self-keygen patch.
3. Writing a keygenerator is a waste of time and we don't want to do it.
4. it's not compressed at all.
5. You can install it and then deinstall without problems so you can try just to install it for the sake of this tutorial only, even if you don't need it.

Start immediately from the code of the main program and then continue with the callbacks (the whole code is similar to the FishMe Oraculum already seen).

<-----Code Snippet DoubleLook_Oraculum.cpp----->

```
#include "Oraculum.h"

//Register the callback function names.
DECLARE_CALLBACK_PROTOTYPES(DoPatch_callback, DoActionPatch_callbackStop,
                             DoActionPatch_callbackResume);

int main(int argc, char** argv)
{
    COraculum Oraculum;

    if(argc==1) {
        printf("Instruction:\n\n programName <path_of_DoubleLook.exe>\n\nThe program launches
        DoubleLook normally.\nYou have to press in DoubleLook the \"Register\" menu and enter any
        long enough serial.\nThe Oraculum will tell you the real serial.\nLaunch this program from a
        DOS window to see more infos.\n\nTested on versions from 1.3 to 2.x\n-----
        -----\n&8~) Shub-Nigurrath of ARTeam\n");
        return 0;
    }
}
```



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

```
if(argc!=1)
    Oraculum.SetPath(argv[1]);

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

////////////////////////////////////
//Search the offset of the real place where to patch the destination process
//set up the search pattern once and for all
//
// 005FE93A . 83EC 58      SUB ESP,58
// 005FE93D . 53          PUSH EBX
// 005FE93E . 56          PUSH ESI
// 005FE93F . 57          PUSH EDI
// 005FE940 . 8BF9       MOV EDI,ECX
// 005FE942 . 33DB       XOR EBX,EBX
// 005FE944 . 6A 01      PUSH 1
//
//Which is in bytes: 83EC 58 53 56 57 8BF9 33DB 6A 01
//
//This unique string is found in several versions and for all of them 1D4 bytes
//forward there's the point to patch!
//
//If the string isn't found the program tries to find this other one instead
//
// 0057993A . 83EC 40      SUB ESP,40
// 0057993D . 53          PUSH EBX
// 0057993E . 56          PUSH ESI
// 0057993F . 57          PUSH EDI
// 00579940 . 8BF1       MOV ESI,ECX
// 00579942 . 33DB       XOR EBX,EBX
// 00579944 . 6A 01      PUSH 1
//
//Which is in bytes: 83EC 40 53 56 57 8BF1 33DB 6A 01
//
//This unique string is found in several versions and for all of them 17B bytes
//forward there's the point to patch!
//
//Add the search patterns to the list of things to search. These values are searched in
//this order till one gives a single match. More than a match is considered as a
//wrong "unique pattern" and it is ignored.

//NB: being a stack must be added in reverse order of importance.
//The last added is the first searched!
Oraculum.AddPattern(":x83:xEC:x40:x53:x56:x57:x8B:xF1:x33:xDB:x6A:x01", 0x17B);
Oraculum.AddPattern(":x83:xEC:x58:x53:x56:x57:x8B:xF9:x33:xDB:x6A:x01", 0x1D4); //first

Oraculum.SetCallbacks( DoPatch_callback,
                      DoActionPatch_callbackStop,
                      DoActionPatch_callbackResume);

//Do the patch at the desired location, and create the suspended process,
//using CreateProcess API.
if(Oraculum.IsValid())
    if(Oraculum.FindPatterns()>0)
        if(Oraculum.DoPatch()>=0)
            Oraculum.KillProcess();

//Return accumulated messages!
cout << "Messages Stack dump, read information in reverse order.." << endl
    << "-----" << endl;

while(!Oraculum.MessageStack().empty()) {
    int idx=Oraculum.MessageStack().size();
    TString str=Oraculum.MessageStack().top();
    Oraculum.MessageStack().pop();
    printf("%d> %s\n", idx--, str.c_str());
}

return 0;
}

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



The callbacks instead are the following:

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

void DoPatch_callback(COraculum *oraculum, DWORD dwMemoryPatchAddr) {

    //005FEB0E . 59          POP ECX contains the real serial!
    oraculum->WriteProcessMemory(oraculum->GetPI()->hProcess,
        (LPVOID)dwMemoryPatchAddr,
        (void*)&COraculum::HALT_CODE, HALT_SIZE, NULL);    // places the EBFE loop
}

void DoActionPatch_callbackStop(COraculum *oraculum) {

    //EDX contains the pointer to real code!
    oraculum->ReadProcessMemory(oraculum->GetPI()->hProcess,
        (LPVOID)oraculum->GetProcessContext()->Edx,
        oraculum->GetProcessBuffer(), 100, NULL);
}

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----->
```

If you arrived still with a working brain here, it should be evident how the Oraculum is simple to write: if you cut away the comments what remains are few lines of code.

5. Writing the RoomRover Oraculum: an UPX packed, self checking application

I will assume you already read the Appendix 2 and you have already understood the difficulties of writing an Oraculum for this program.

The main program is almost identical to all the others, as already told the most complex things happen inside the callbacks.

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

int main(int argc, char** argv)
{
    COraculum Oraculum;

    if(argc==1) {
        printf("Instruction:\n\n programName <path_of_RoomRover.exe>\n\nThe program launches
RoomRover normally.\nYou have try to register in RoomRover and enter any long enough
serial.\nThe Oraculum will tell you the real serial.\nLaunch this program from a DOS
window to see more infos.\n-----\n&8~) Shub-Nigurrath of
ARTeam\n");
        return 0;
    }

    if(argc!=1)
        Oraculum.SetPath(argv[1]);

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




```
//I will set an EBFE loop exactly at JMP to the OEP, found using Ollydbg, right at the
//end of the UPX unpacking procedure.
//Bytes: E9B0ADFDFE //this is the unique identifying bytes pattern, found in the exe file
//Offset: (mem) 0x4740DC (file) 0x2A4DC
//imageBase: 00400000
//baseofCode: 0004A000
Oraculum.AddPattern(":xE9:xB0:xAD:xFD:xFF:x00", 0x0);

Oraculum.SetCallbacks( DoPatch_callbackOEP,
                      DoActionPatch_callbackStopOEP,
                      DoActionPatch_callbackResumeOEP );

//Do the patch at the desired location, and create the suspended process, using
//CreateProcess API.
//It's a chain of cross validation so this chain of iff..
if(Oraculum.IsValid())
    if(Oraculum.FindPatterns()>0)
        if(Oraculum.DoPatch()>=0)
            Oraculum.KillProcess();

//Return accumulated messages!

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

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());
}

return 0;
}
```

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

The most interesting things happen inside the callbacks that, how you could see, aren't so simple. The RoomRoverCallbaks.cpp file appears like following:

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

#include "RoomRoverCallBacks.h"
#define SERIAL_SIZE 100

char OEPBuffer[HALT_SIZE];
DWORD dwOEPAddr;
DWORD dwFishingAddress;

void DoPatch_callbackOEP(COraculum *oraculum, DWORD dwMemoryPatchAddr) {

    //004740DC  .^E9 B0ADFDFE  JMP RoomRove.0044EE91
    //0044EE91  > /6A 60      PUSH 60
    //Places in a safe place the real bytes of the application..later will be restored.
    oraculum->ReadProcessMemory(oraculum->GetPI()->hProcess,
        (LPVOID)dwMemoryPatchAddr,
        (void*)&OEPBuffer, HALT_SIZE, NULL);

    dwOEPAddr=dwMemoryPatchAddr;

    oraculum->WriteProcessMemory(oraculum->GetPI()->hProcess,
        (LPVOID)dwMemoryPatchAddr, //JMP to OEP , seen in Olly.
        (void*)&COraculum::HALT_CODE, HALT_SIZE, NULL);    // OEP HOOK
}

void DoActionPatch_callbackStopOEP(COraculum *oraculum) {

    //it's in the OEP, so it's time to write our patches and restore the jmp to OEP
    oraculum->WriteProcessMemory(oraculum->GetPI()->hProcess,
        (LPVOID)dwOEPAddr,
        (void*)&OEPBuffer, HALT_SIZE, NULL); // restore instruction
}
```



```
//We'll modify:
//004020A5 /74 6C JE SHORT RoomRove.00402113
// with:
//004020A5 /EB 6C JMP SHORT RoomRove.00402113
//and you get to the registration dialog without annoying exceptions.
//To find the thing uses the memory search part of the class.
//The unique pattern to be searched is
// 004020A5 /74 6C JE SHORT RoomRove.00402113
// 004020A7 13BFB CMP EDI,EBX
// 004020A9 174 05 JE SHORT RoomRove.004020B0
// 004020AB 1C607 00 MOV BYTE PTR DS:[EDI],0
//which corresponds to this pattern: :x74:x6C:x3B:xFB:x74:x05:xC6:x07:x00

//Specification tells that the first executable section is just after the headers section.
//It is calculated as ImageBase + SizeofHeaders
DWORD startAddr= oraculum->GetPE()->GetImageBase() +
    oraculum->GetPE()->GetSizeOfHeaders();

//Specification tells that the last executable section finish at the code section.
//It is calculated as ImageBase + BaseofCode + SizeofCode
DWORD endAddr=oraculum->GetPE()->GetImageBase() +
    oraculum->GetPE()->GetBaseOfCode() +
    oraculum->GetPE()->GetSizeOfCode();

DWORD PatchAddress=0;

int nMatches=oraculum->BMG_MemorySearch(
    oraculum->GetPI()->hProcess, startAddr, endAddr,
    ":x74:x6C:x3B:xFB:x74:x05:xC6:x07:x00");

if(nMatches==1) {
    PatchAddress=(DWORD)oraculum->GetLastOffset() + startAddr;
}

//Do the patch now
BYTE patch1=0xEB;
oraculum->WriteProcessMemory(oraculum->GetPI()->hProcess, (LPVOID)PatchAddress,
    (void*)&patch1, 1, NULL);

////////////////////////////////////
//Hook right place for Serial Fishing!
//The serial will be placed inside the stack, so I need to copy it into a register.
//Doing so I'm able to get it from context, without worrying of memory page permissions

//This is the patch!
// 0041E9A5 8B8424 DC000000 MOV EAX,DWORD PTR SS:[ESP+DC] ; moves the serial into EAX
// 0041E9AC - EB FE JMP SHORT RoomRove.0041E9AC ; the halt code
//
//The original code at this address is:
// 0041E9A5 8A45 02 MOV AL,BYTE PTR SS:[EBP+2] ; SerialFish here!
// 0041E9A8 84C0 TEST AL,AL
// 0041E9AA 75 1D JNZ SHORT RoomRove.0041E9C9
//
//which corresponds to this pattern: :x8A:x45:x02:x84:x00:x75:x1D

//The fasted way to solve the situation is to find two patterns, one from the original
//process's memory, the other is instead the changed bytes. To ensure that the original
//pattern is unique within the specified memory range, use CTRL-B to search in the
//specified memory range or restrict the memory range.
//
//To make it shorter I specified to search into the whole process' memory space, but you
//can do smarter memory windows selections!
nMatches=oraculum->BMG_MemorySearchReplace(
    oraculum->GetPI()->hProcess, startAddr, endAddr,
    ":x8A:x45:x02:x84:x00:x75:x1D",":x8B:x84:x24:xDC:x00:x00:x00:EB:xFE");

if(nMatches==1) {
    PatchAddress=(DWORD)oraculum->GetLastOffset() + startAddr;
}

//According to the patch just done calculate the fishing address, which is the
//address of the last patch done, at 0041E9AC
//It is calculated at PatchAddress + (size of the MOV at 0041E9A5).
dwFishingAddress=PatchAddress + 7;
```



```
}

void DoActionPatch_callbackResumeOEP(COraculum *oraculum) {

    char buffer[SERIAL_SIZE];
    while(GetThreadContext(oraculum->GetPI()->hThread, oraculum->GetProcessContext()))
    {
        if(oraculum->GetProcessContext()->Eip==dwFishingAddress) {
            //do the real work
            //now, according to my patch, EAX contains the serial.
            oraculum->ReadProcessMemory(oraculum->GetPI()->hProcess,
                (LPVOID)(oraculum->GetProcessContext()->Eax),
                &buffer, SERIAL_SIZE, NULL);
            break;
        }
        Sleep(10);
    }

    char msg[256];
    sprintf(msg, "Shub-Nigurrath says that your serial is: \"%s\"", buffer);
    oraculum->pushMessage(msg);
}

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

Well, need to comment? Not much, it should be clear now what I did. Anyway here the essential things to take care of..

1. DoPatch_callbackOEP places an EBFE loop at the JMP to OEP, at the end of the UPX unpacking procedure (004740DC) and stores away the original bytes.
2. DoActionPatch_callbackStopOEP is so complex because the bytes to be patched exist only after the UPX unpacking procedure has written them (at the corresponding offset the exe file has nothing). It does some different things:
 - a. Restore the OEP bytes back to let the application continue
 - b. Calculates the memory range used by the application's code.
The specifications tell that the first executable section is just after the headers section, so it is calculated as ImageBase + SizeofHeaders

```
DWORD startAddr = oraculum->GetPE()->GetImageBase() +
    oraculum->GetPE()->GetSizeOfHeaders();
```

Specifications tell on the other hand that the last executable section finishes at the end of the last code section, so it is calculated as ImageBase + BaseofCode + SizeofCode.

```
DWORD endAddr = oraculum->GetPE()->GetImageBase() +
    oraculum->GetPE()->GetBaseOfCode() +
    oraculum->GetPE()->GetSizeOfCode();
```

- c. Patch the JE at 004020A5 to JMP (see Appendix 2) to avoid exceptions (this is done using MemorySearch()).
 - d. Do a second patch at 0041E9A5. This is required to transfer the real serial from the application's stack (at the address ESP+DC) to one of the registers; we will then be



able to get it through GetThreadContext API (this is done using MemorySearchReplace()).

The original code at this address is:

```
0041E9A5 8A45 02 MOV AL,BYTE PTR SS:[EBP+2] ; SerialFish here from [ESP+DC]
0041E9A8 84C0 TEST AL,AL
0041E9AA 75 1D JNZ SHORT RoomRove.0041E9C9
```

This is the patch:

```
0041E9A5 8B8424 DC000000 MOV EAX,DWORD PTR SS:[ESP+DC] ; moves serial into EAX
0041E9AC EB FE JMP SHORT RoomRove.0041E9AC ; the halt code
```

e. Set the dwFishingAddress variable to point to the correct address

3. DoActionPatch_callbackResumeOEP simply read the memory address pointed by EAX and places a message in the messages stack.

NOTE

A comment about code organization: I used the globally visible variable with a file scope, to simulate private properties of C++ classes in C (block programming): the variables EPBuffer, dwOEPAddr, dwFishingAddress, declared outside any function's body, have a scope that coincide with the file: all the functions declared into this file can access them and not others.

6. 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 HackerRMaN 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 accompaning 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
```



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

```
        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 ?
```



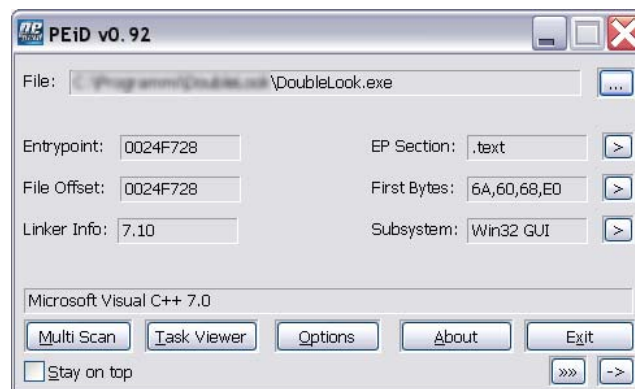
hInstance dd ?

#####

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

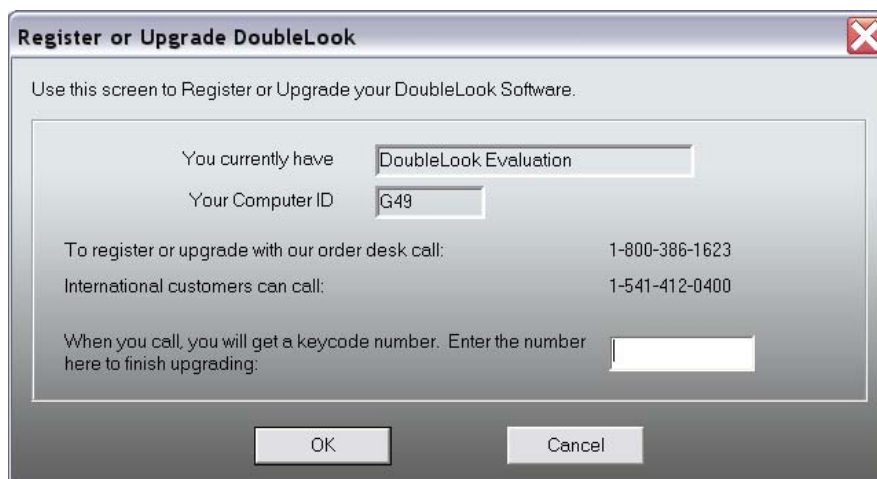
7. Appendix 1: cracking DoubleLook for PalmDesktop

As usual the first thing to check is the program's compression, if any packer has been used on the target.



Well, it's not packed then an easier target. The second usual think to do is to take time to study the protection a little to understand what happens.

The program runs unregistered and asks to the user a registration number in a dialog like the one below:



The registration uses an identification number, much probably calculated directly starting from the computer characteristics (it's not important how indeed).

If you enter 5 digits then it shows this bad boy dialog box



Guess what? We will now launch Ollydbg on the target, and using F9 will go directly to the bad boy message. Once there press pause in Ollydbg.

The following picture shown the stack we see (ALT-K):

Address	Stack	Procedure / arguments	Called from	Frame
0012E188	77D193F5	Includes ntdll.KiFastSystemCallRet	USER32.77D193F3	0012E18C
0012E18C	77D3EA24	USER32.WaitMessage	USER32.77D3EA1F	0012E18C
0012E1C0	77D2688A	USER32.77D3E895	USER32.77D26885	0012E18C
0012E1E8	77D3B7C5	USER32.77D267D4	USER32.77D3B7C0	0012E1E4
0012E4A8	77D3B12B	USER32.SoftModalMessageBox	USER32.77D3B126	0012E4A4
0012E5F8	77D65FDF	USER32.77D3AFB6	USER32.77D65FDA	0012E5F4
0012E650	77D66084	USER32.MessageBoxTimeoutW	USER32.77D6607F	0012E64C
0012E684	77D50598	? USER32.MessageBoxTimeoutA	USER32.77D50593	0012E680
0012E6A8	77D50550	? USER32.MessageBoxExA	USER32.77D50548	0012E6A0
0012E6A8	00000000	hOwner = NULL		
0012E6AC	0141C770	Text = "Incorrect Key Code Number.		
0012E6B0	014164D8	Title = "DoubleLook"		
0012E6B4	00000030	Style = MB_OK MB_ICONEXCLAMATION		
0012E6B8	00000000	LanguageID = 0 (LANG_NEUTRAL)		
0012E6C0	005FFC32	? USER32.MessageBoxA	DoubleLo.005FFC2C	0012E6B8
0012E6C4	00000000	hOwner = NULL		
0012E6C8	0141C770	Text = "Incorrect Key Code Number.		
0012E6CC	014164D8	Title = "DoubleLook"		
0012E6D0	00000030	Style = MB_OK MB_ICONEXCLAMATION		
0012E748	0066240E	Includes DoubleLo.005FFC32	DoubleLo.0066240B	0012E744
0012E758	006625F1	DoubleLo.006623D1	DoubleLo.006625EC	0012E754

Of course we want to jump where the bad boy message starts to be assembled to investigate the conditions that cause this message to appear.

Double click on the line highlighted above on the Procedure /arguments columns (the 3rd column) and you land here:

005FFC1A	. 83C1 F0	ADD ECX, -10	
005FFC1D	. E8 1F18E0FF	CALL DoubleLo.00401441	
005FFC22	> 6A 30	PUSH 30	Style = MB_OK MB_ICONEXCLAMATION MB_APPLMODAL
005FFC24	. FF75 DC	PUSH DWORD PTR SS:[EBP-24]	Title
005FFC27	. FF75 E8	PUSH DWORD PTR SS:[EBP-18]	Text
005FFC2A	. 6A 00	PUSH 0	hOwner = NULL
005FFC2C	. FF15 78856C00	CALL DWORD PTR DS:[&USER32.MessageBoxA	MessageBoxA

Here the message is assembled. So start to investigate what happens just above the MessageBoxA invocation..

It's a very long routine as you might realise soon, but it's filled on really interesting strings, such as all those of the type "Now configure as ..." ..we apparently will have the possibility to choose how to register our program, just choosing where to go! But first of all we need to find the start of this routine.

To find it have a look at the address from which this invocation comes from, double click on the "Called from" column in the calling stack view, and land into 0066240B:



006623F4	. 83C0 CB	ADD EAX, -35	Switch (cases 35..40)
006623F7	. 83F8 0B	CMP EAX, 0B	
006623FA	. 56	PUSH ESI	
006623FB	~ 0F87 AE000000	JR DoubleLo.006624AF	
00662401	> FF2485 B7246	JMP DWORD PTR DS:[EAX*4+6624B7]	
00662408	> 8B4D 08	MOV ECX, DWORD PTR SS:[EBP+8]	Case 35 ('5') of switch 006623F4
0066240B	. FF55 14	CALL DWORD PTR SS:[EBP+14]	
0066240E	~ E9 98000000	JMP DoubleLo.006624AB	
00662413	> 8B4D 08	MOV ECX, DWORD PTR SS:[EBP+8]	Case 36 ('6') of switch 006623F4
00662416	. FF55 14	CALL DWORD PTR SS:[EBP+14]	
00662419	~ E9 8B000000	JMP DoubleLo.006624A9	
0066241E	> FF75 0C	PUSH DWORD PTR SS:[EBP+C]	Case 37 ('7') of switch 006623F4
00662421	~ EB 75	JMP SHORT DoubleLo.00662498	
00662423	> FF75 0C	PUSH DWORD PTR SS:[EBP+C]	Case 38 ('8') of switch 006623F4
00662426	~ EB 7B	JMP SHORT DoubleLo.006624A3	
00662428	> 8B45 18	MOV EAX, DWORD PTR SS:[EBP+18]	Case 39 ('9') of switch 006623F4
0066242B	. FF30	PUSH DWORD PTR DS:[EAX]	
0066242D	. 8B4D 08	MOV ECX, DWORD PTR SS:[EBP+8]	
00662430	. FF70 04	PUSH DWORD PTR DS:[EAX+4]	
00662433	. FF55 14	CALL DWORD PTR SS:[EBP+14]	
00662436	~ EB 73	JMP SHORT DoubleLo.006624AB	
00662438	> 8B45 18	MOV EAX, DWORD PTR SS:[EBP+18]	Case 3A (':') of switch 006623F4

As you can see the called procedure is part of a switch which is invoked any time you press a character in the registration number editbox, in the program. Evidently is an OnChange handler.

Place a Breakpoint here and rerun the program. When you press the first character of the registration number you land on the BP just set. Using F7 follow the call and land into the top of the routine which contains the MessageBox responsible to show the bad boy msg.

005FE930	. B8 353C6B00	MOV EAX, DoubleLo.006B3C35
005FE935	. E8 9EC60400	CALL DoubleLo.0064AFD8
005FE93A	. 83EC 58	SUB ESP, 58
005FE93D	. 53	PUSH EBX
005FE93E	. 56	PUSH ESI
005FE93F	. 57	PUSH EDI
005FE940	. 8BF9	MOV EDI, ECX
005FE942	. 33DB	XOR EBX, EBX
005FE944	. 6A 01	PUSH 1
005FE946	. 897D B0	MOV DWORD PTR SS:[EBP-50], EDI
005FE949	. 895D EC	MOV DWORD PTR SS:[EBP-14], EBX
005FE94C	. E8 77560600	CALL DoubleLo.00663FC8
005FE951	. 8DB7 84000000	LEA ESI, DWORD PTR DS:[EDI+84]
005FE957	. 8B06	MOV EAX, DWORD PTR DS:[ESI]
005FE959	. 8B40 F4	MOV EAX, DWORD PTR DS:[EAX-C]
005FE95C	. 83F8 06	CMP EAX, 6
005FE95F	~ 0F85 CF130000	JNZ DoubleLo.005FFD34

Note the presence of the JNZ at 005FE95F, it directly jmps to the function's end and if you trace it does this till all the serial number have been collected. Then moves inside..

How to be sure that this is the correct start: because the stack of calls told so to us. As you can see the function is very long. How to get the clue? You have two options:

1. trace with Ollydbg and understand where are going the jmps;
2. get an bird flight overview of the whole routine. I'm just using here this opportunity which is IMHO more interesting for the less trained of you..

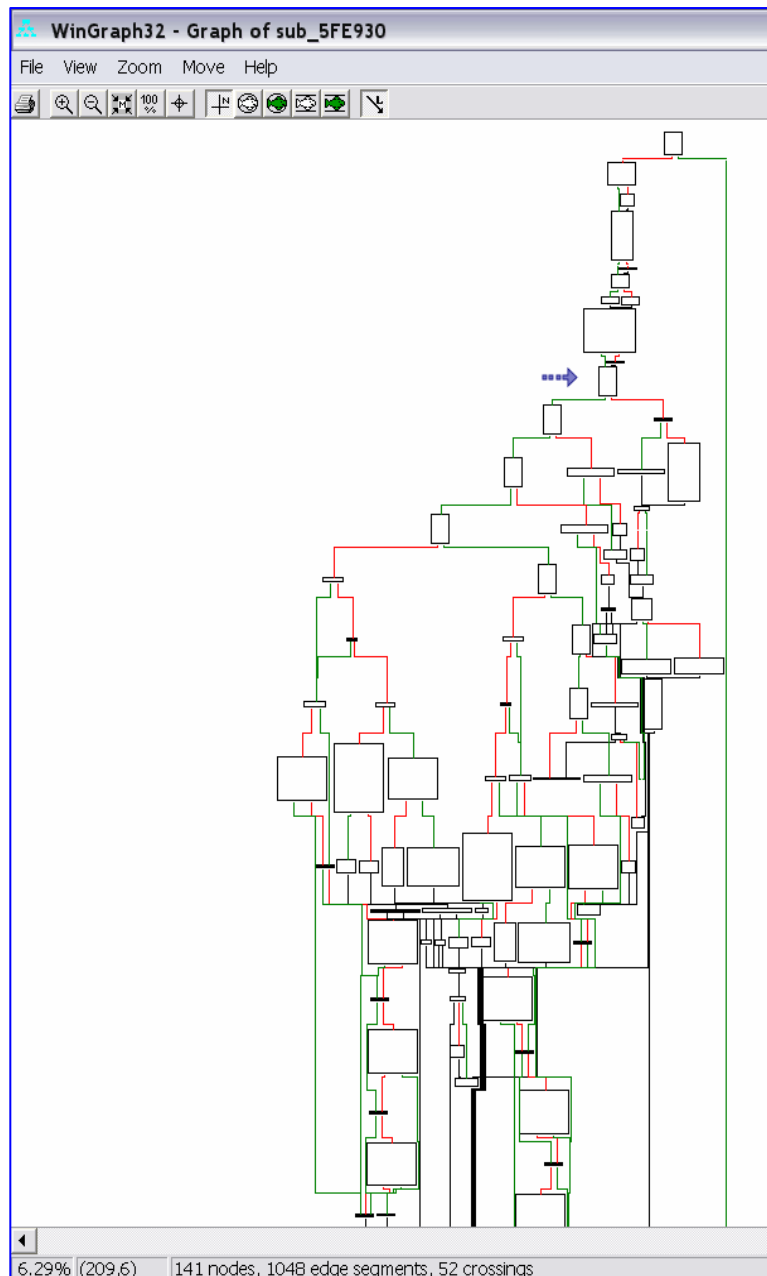
Ok, open IDA and gives him to eat the doublelook.exe (relax, because might be long, being the executable an 8Mb long file). I'm not explaining everything with IDA, because otherwise we would get a 1 hundred pages tutorial, filled of several more snapshots.

Anyway shortly here's what you have to do, without snapshots, once IDA finishes analyzing.

1. go to the address 005FE930 pressing G
2. on that address press P to create a function from here (IDA being this a dynamic handler misses the real start of the function. We are telling it now).
3. Now press F12 to launch the WinGraph application on that function.



You should get something like the following one.



Well, quite complicated and not readable at all, but I told it would have been a bird flight overview!

Anyway we can learn several things looking at its general structure.

First of all just at the beginning of the function there is the JMP I told you, at the address 005FE95F, which directly jumps to the function's end.

Secondly the function continues, on essentially a single flow, up to the code section pointed by the arrow.

The following execution of the function depends by this piece of code. If you just look at the picture, zooming your view, you should see that the two paths are completely different: while one argues, in a complex way, how the application will be registered with the given number, the second one essentially exits faster.

Well we can guess that something interesting is happening in the code block pointed by the arrow.



```
lea     eax, [ebp-2Ch]
push    6Ch
push    eax
mov     byte ptr [ebp-4], 4
call    sub_59EB04
mov     ebx, offset Default
push    ebx
push    dword ptr [ebp-20h]
mov     byte ptr [ebp-4], 8
call    __mbstomp
add     esp, 18h
test    eax, eax
jnz     short loc_SFEAE9
```

```
005FEAE2:
mov     dword ptr [ebp-28h], 1
```

```
loc_SFEAE9:
push    esi
push    9
push    ecx
lea     eax, [ebp-20h]
mov     [ebp-4Ch], esp
mov     ecx, esp
push    eax
call    sub_401CA7
lea     eax, [ebp-30h]
push    eax
call    sub_5D7D8C
add     esp, 0Ch
push    eax
call    sub_402B53
pop     ecx
pop     ecx
mov     ecx, [ebp-30h]
add     ecx, 0FFFFFF0h
mov     [ebp-00h], al
call    sub_401441
cmp     byte ptr [ebp-00h], 0
jz      loc_SFECE12
```

Beside picture shows the zoomed view. Our code starts at 005FEAE9.

Then now go back into Ollydbg and place a breakpoint on:

```
005FEAE9  > \56      PUSH ESI
```

And restart the application if needed or press F9, depending on where you left Ollydbg.

Step through all these instructions, I will not detail them further, the interesting thing is that when you arrive here

```
005FEB0D  . 59      POP ECX
```

The registers ECX and EDX will magically contain respectively the fake and the real serial numbers!!

Exit, rerun the application, enter it and voilà you are registered.

005FEAE9	> 56	PUSH ESI	the code chunk starts here
005FEAEA	. 6A 09	PUSH 9	
005FEAEC	. 51	PUSH ECX	
005FEAED	. 8D45 E0	LEA EAX, DWORD PTR SS:[EBP-20]	EAX 00000000
005FEAF0	. 8965 B4	MOV DWORD PTR SS:[EBP-4C], ESP	ECX 0141C758 ASCII "111111"
005FEAF3	. 8BCC	MOV ECX, ESP	EDX 0141C588 ASCII "483620"
005FEAF5	. 50	PUSH EAX	EBX 006C88CA DoubleLo.006C88CA
005FEAF6	. E8 AC31E0FF	CALL DoubleLo.00401CA7	ESP 0012E6CC
005FEAFB	. 8D45 D0	LEA EAX, DWORD PTR SS:[EBP-30]	EBP 0012E744
005FEAFE	. 50	PUSH EAX	ESI 0012EEB4
005FEAFF	. E8 8892FDFF	CALL DoubleLo.005D7D8C	EDI 006EE680 ASCII "Software\Microsoft\Windows"
005FEB04	. 83C4 0C	ADD ESP, 0C	EIP 005FEB0D DoubleLo.005FEB0D
005FEB07	. 50	PUSH EAX	
005FEB08	. E8 4640E0FF	CALL DoubleLo.00402B53	
005FEB0D	. 59	POP ECX	fish serial here

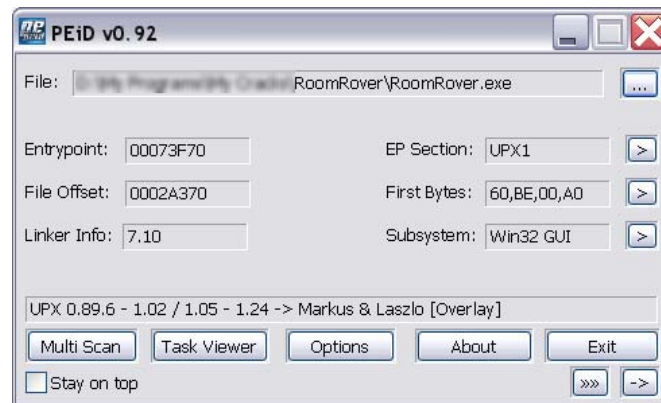
NOTE

During the function other registration codes are calculated, configuring the application according to several licenses types. The mechanism is the same here; the real codes always appear into a register. We are happy anyway with this registration number, which fully registers the application.



8. Appendix 2: cracking RoomRover

As usual see if the target is packed:



Well a normal UPX. This is good! But hey, try to unpack the program and you will have a surprise: it won't run at all, because it complains about a missing script.

8.2 Dumping the program

I wrote some time ago a complete tutorial about how to manually unpack UPX, but here I will follow another way which is the same that I used to write the RoomRover's Oraculum..

Open into Ollydbg RoomRover.exe and you should get something like below:

00473F70	60	PUSHAD
00473F71	BE 00A04400	MOV ESI,RoomRove.0044A000
00473F76	8DBE 0070FBFF	LEA EDI,DWORD PTR DS:[ESI+FFFFB7000]
00473F7C	57	PUSH EDI
00473F7D	83CD FF	OR EBP,FFFFFFFF
00473F80	EB 10	JMP SHORT RoomRove.00473F92
00473F82	90	NOP
00473F83	90	NOP
00473F84	90	NOP
00473F85	90	NOP
00473F86	90	NOP
00473F87	90	NOP
00473F88	8A06	MOV AL,BYTE PTR DS:[ESI]
00473F8A	46	INC ESI

You should know that UPX works packing the whole programs code sections, then add an .upx executable section to the PE file, which contains the unpacking routine. When you launch a program packed with UPX you land at the EP of the unpacking routine (like any packer).

This routine is quite linear and terminates with a JMP to the real OEP which it meanwhile has been written in memory. Well, keeping this in mind, scroll down Ollydbg till you see the end of the UPX unpacking routine (you should see it because after its end there's nothing, all 00):




004740BC	50	PUSH EAX
004740BD	47	INC EDI
004740BE	B9 5748F2AE	MOV ECX,AEF24857
004740C3	55	PUSH EBP
004740C4	FF96 705D0700	CALL DWORD PTR DS:[ESI+75D700]
004740CA	09C0	OR EAX,EAX
004740CC	74 07	JE SHORT RoomRove.004740D5
004740CE	8903	MOV DWORD PTR DS:[EBX],EAX
004740D0	83C3 04	ADD EBX,4
004740D3	EB D8	JMP SHORT RoomRove.004740D0
004740D5	FF96 745D0700	CALL DWORD PTR DS:[ESI+75D740]
004740DB	61	POPAD
004740DC	E9 B0ADF0FF	JMP RoomRove.0044EE91
004740E1	0000	ADD BYTE PTR DS:[EAX],AL
004740F3	0000	ADD BYTE PTR DS:[EAX],AL

The last JMP is the JMP to the OEP. Well, place a BP (using F2) at 004740DC and press F9 to continue execution till there.

Now you are an F8 key away from the OEP: press F8 you will land to the OEP.

Note that if you see something like below, you should press CTRL-A to analyze the code and you'll get a magic transformation ;-):

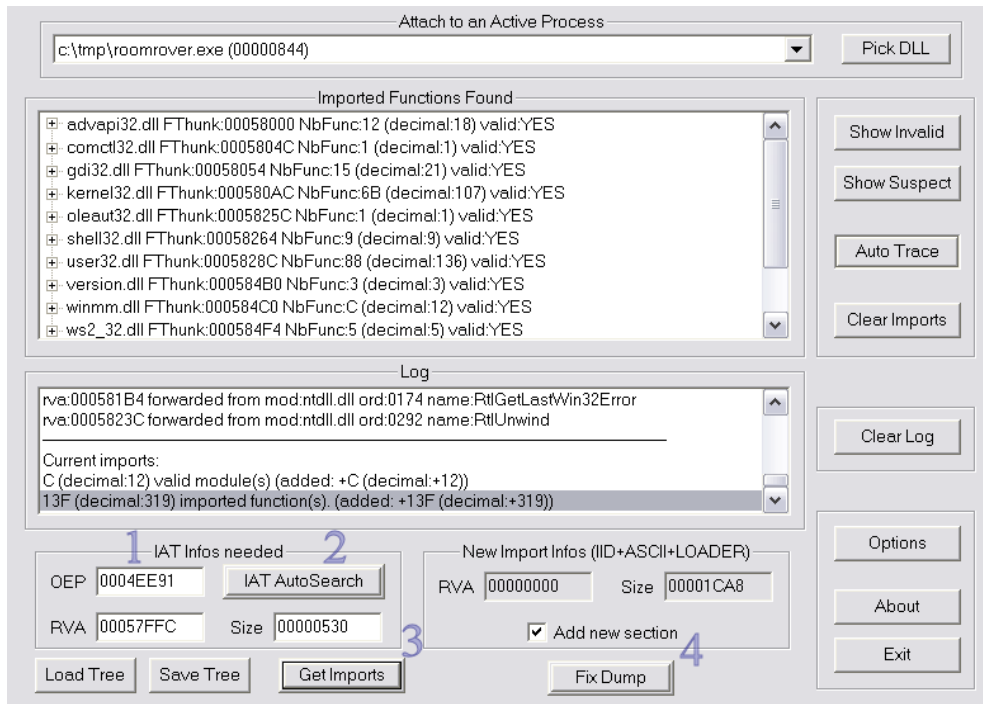
0044EE91	> 6A 60	PUSH 60
0044EE93	68	DB 68
0044EE94	C0	DB C0
0044EE95	11	DB 11
0044EE96	46	DB 46
0044EE97	00	DB 00
0044EE98	E8	DB E8
0044EE99	97	DB 97
0044EE9A	55	DB 55
0044EE9B	00	DB 00
0044EE9C	00	DB 00
0044EE9D	BF	DB BF
0044EE9E	94	DB 94
0044EE9F	00	DB 00
0044EEA0	00	DB 00



0044EE91	> 6A 60	PUSH 60
0044EE93	68 C0114600	PUSH RoomRove.004611C0
0044EE98	E8 97550000	CALL RoomRove.00454434
0044EE9D	BF 94000000	MOV EDI,94
0044EEA2	8BC7	MOV EAX,EDI
0044EEA4	E8 97E8FFFF	CALL RoomRove.0044D740
0044EEA9	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
0044EEAC	8BF4	MOV ESI,ESP
0044EEAE	893E	MOV DWORD PTR DS:[ESI],EDI
0044EEB0	56	PUSH ESI
0044EEB1	FF15 BC814500	CALL DWORD PTR DS:[4581BC]
0044EEB7	8B4E 10	MOV ECX,DWORD PTR DS:[ESI+10]

Now dump the process and fix the IAT of the dumped program using ImportRec: use OllyDump plugin and ImportRec in the following ways.

Start Address:	400000	Size:	77000	Dump	
Entry Point:	73F70	-> Modify:	4EE91	Get EIP as OEP	Cancel
Base of Code:	4A000	Base of Data:	75000		
<input checked="" type="checkbox"/> Fix Raw Size & Offset of Dump Image					
Sect...	Virtual Si...	Virtual O...	Raw Size	Raw Off...	Charactari...
UPX0	00049000	00001000	00049000	00001000	E0000080
UPX1	0002B000	0004A000	0002B000	0004A000	E0000040
.rsrc	00002000	00075000	00002000	00075000	C0000040
<input checked="" type="checkbox"/> Rebuild Import					
<input checked="" type="radio"/> Method1 : Search JMP[API] CALL[API] in memory image					
<input type="radio"/> Method2 : Search DLL & API name string in dumped file					



Now execute the dumped program. You should get an error reporting “Could not open the script inside the EXE”.

8.3 Patching the self-checking “protection”

The problem is that the program has some script stored inside the packed exe file, and before executing it, this script is read from the executable file, opening the original packed executable as a storage file. It’s not really a protection, is a trick the developers found to distribute the whole program as a single file instead having a separate script file, but hurts our activity ☺

The only way to open a file on disk is to use the CreateFile API.

See the MSDN help about this API

```
HANDLE CreateFile(  
    LPCTSTR lpFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile  
);
```

I’ll not explain all the parameters, but what we need here is that if the lpFileName is equal to NULL the API opens with the desired access dwDesiredAccess itself (of course only for reading).



So, it's time to go forward: open the dumped file and place a BP on CreateFileA. You land inside kernel32:

7C801A24	8BFF	MOV EDI,EDI
7C801A26	55	PUSH EBP
7C801A27	8BEC	MOV EBP,ESP
7C801A29	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C801A2C	E8 73C80000	CALL kernel32.7C80E2A4

And the calling stack confirms our theory.

0012B4E4	00455DE6	CALL to CreateFileA from dump_.00455DE0
0012B4E8	0012B99C	FileName = "C:\Tmp\dump_.exe"
0012B4EC	80000000	Access = GENERIC_READ
0012B4F0	00000003	ShareMode = FILE_SHARE_READ FILE_SHARE_WRITE
0012B4F4	0012B510	pSecurity = 0012B510
0012B4F8	00000003	Mode = OPEN_EXISTING
0012B4FC	00000000	Attributes = NORMAL
0012B500	00000000	hTemplateFile = NULL
0012B504	0046594E	dump_.0046594E

CreateFileA is called at 00455DE0 (use ALT-K to show the calling stack).

00455DC8	.. E9 C7000000	JMP dump_.00455E94	
00455DCD	> 6A 00	PUSH 0	hTemplateFile = NULL
00455DD1	. 56	PUSH ESI	Attributes
00455DD3	. FF75 F4	PUSH DWORD PTR SS:[EBP-C]	Mode
00455DD5	. 8D45 E4	LEA EAX,DWORD PTR SS:[EBP-1C]	
00455DD7	. 50	PUSH EAX	pSecurity
00455DD9	. FF75 F8	PUSH DWORD PTR SS:[EBP-8]	ShareMode
00455DDB	. FF75 F0	PUSH DWORD PTR SS:[EBP-10]	Access
00455DDD	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	FileName
00455DE0	. FF15 F0804501	CALL DWORD PTR DS:[&kernel32.CreateFileA]	CreateFileA
00455DE6	. 8BF0	MOV ESI,EAX	
00455DE8	. 3BF7	CMP ESI,EDI	

The idea to skip this problem is to have in the RoomRover folder two files: the original executable packed file, just renamed to RoomRover.bak and the patched & dumped executable. Then we want to change the CreateFileA call passing to it a real path to the RoomRover.bak file. Doing this the unpacked program will not anymore be used to find the script, it will be used instead the unpacked original program.

Complex? I don't think, a look at the code will help.

The code just captured above will become:

00455DC8	.. E9 C7000000	JMP RoomRove.00455E94	
00455DCD	> E9 12E30100	JMP RoomRove.004740E4	
00455DD2	. 90	NOP	
00455DD3	. 8D45 E4	LEA EAX,DWORD PTR SS:[EBP-1C]	
00455DD5	. 50	PUSH EAX	pSecurity
00455DD7	. FF75 F8	PUSH DWORD PTR SS:[EBP-8]	ShareMode
00455DD9	. FF75 F0	PUSH DWORD PTR SS:[EBP-10]	Access
00455DDB	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	FileName
00455DE0	. FF15 F0804501	CALL DWORD PTR DS:[&kernel32.CreateFileA]	CreateFileA

I modified the instruction at the address 00455DCD into a "JMP RoomRove.004740E4": before there was in the same place the first "PUSH 0" of the CreateFileA parameters.



This JMP goes to what was a free location of the program (I found it manually or you can use ToPo, see other tutorials of our about ToPo) and at the address 004740E4 there's the new CreateFileA call:

004740E4	> 6A 00	PUSH 0	hTemplateFile = NULL
004740E6	. 56	PUSH ESI	Attributes
004740E7	. FF75 F4	PUSH DWORD PTR SS:[EBP-C]	Mode
004740E9	. 8D45 E4	LEA EAX, DWORD PTR SS:[EBP-1C]	pSecurity
004740ED	. 50	PUSH EAX	ShareMode
004740EE	. FF75 F8	PUSH DWORD PTR SS:[EBP-8]	Access
004740F1	. FF75 F0	PUSH DWORD PTR SS:[EBP-10]	ASCII ".\RoomRover.bak"
004740F4	. C745 08 A11E	MOV DWORD PTR SS:[EBP+8], RoomRove.00461	FileName
004740FB	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	CreateFileA
004740FE	. FF15 F080450	CALL DWORD PTR DS:[<&kernel32.CreateFile	
00474104	. 90	NOP	
00474105	. ^ E9 DC1CFE	JMP RoomRove.00455DE6	
0047410A	. 90	NOP	
0047410B	. 00	DB 00	

Of course I also had to add the string ".\RoomRover.bak" somewhere in the program. I added it at the address 461EA1 because it was just after the last string of the program, but you can also place it anywhere you like, for example just after the routine you wrote at 4740E4.

Note that the last thing is to JMP back to the first instruction after the original CreateFileA, at 00455DE6, so as to take the original program flow.

Note that I used the name RoomRover.bak because the patchers automatically creates .bak files, so less work to do for who will use our patch.

8.4 Finding the serial code

Now the root is open for cracking the program..

Run the application and press register. You get an exception at 4020BB, look above this instruction and note that all the JMPs are redirecting to the same place at 402213, which is the exit point of the exception.

004020A5	74 6C	JE SHORT dump_.00402113
004020A7	3BFB	CMP EDI,EBX
004020A9	74 05	JE SHORT dump_.004020B0
004020AB	C607 00	MOV BYTE PTR DS:[EDI],0
004020AE	EB 63	JMP SHORT dump_.00402113
004020B0	8B4424 20	MOV EAX, DWORD PTR SS:[ESP+20]
004020B4	8D0C80	LEA ECX, DWORD PTR DS:[EAX+EAX*4]
004020B7	8D448E 11	LEA EAX, DWORD PTR DS:[ESI+ECX*4+11]
004020BB	8020 FB	AND BYTE PTR DS:[EAX],0FB
004020BE	EB 53	JMP SHORT dump_.00402113
004020C0	8B4424 20	MOV EAX, DWORD PTR SS:[ESP+20]
004020C4	83F8 0A	CMP EAX,0A

To skip this, modify:

004020A5 /74 6C JE SHORT RoomRove.00402113

with:

004020A5 /EB 6C JMP SHORT RoomRove.00402113

and you will be able to get to the registration dialog without other annoying exceptions.



Insert any serial you like and you will get the BadBoy message, Now press Pause in Ollydbg, go up in the stack a little, as shown:

Address	Stack	Procedure / arguments	Called from	Frame
00128760	77D193F5	Includes ntdll.KiFastSystemCallRet	USER32.77D193F3	00128794
00128764	77D3EA24	USER32.WaitMessage	USER32.77D3EA1F	00128794
00128798	77D2688A	USER32.77D3E895	USER32.77D26885	00128794
001287C0	77D3B7C5	USER32.77D267D4	USER32.77D3B7C0	001287BC
00128A80	77D3B12B	USER32.SoftModalMessageBox	USER32.77D3B126	00128A7C
00128B00	77D65FDF	USER32.77D3AFB6	USER32.77D65FDA	00128BCC
00128C28	77D66084	USER32.MessageBoxTimeoutW	USER32.77D6607F	00128C24
00128C5C	77D50598	? USER32.MessageBoxTimeoutA	USER32.77D50593	00128C58
00128C7C	77D50550	? USER32.MessageBoxExA	USER32.77D5054B	00128C78
00128C80	00000000	hOwner = NULL		
00128C84	001290B0	Text = "The Serial Number you are		
00128C88	00128CB0	Title = "Invalid Serial Number"		
00128C8C	00010010	Style = MB_OK MB_ICONHAND MB_APPL		
00128C90	00000000	LanguageID = 0 (LANG_NEUTRAL)		
00128C98	004496D5	? USER32.MessageBoxA	RoomRove.004496CF	00128C94
00128C9C	00000000	hOwner = NULL		
00128CA0	001290B0	Text = "The Serial Number you are		
00128CA4	00128CB0	Title = "Invalid Serial Number"		
00128CA8	00010010	Style = MB_OK MB_ICONHAND MB_APPL		
00128B00	0041EAD4	? RoomRove.00449530	RoomRove.0041EACF	
0012F1CC	00421BD0	RoomRove.00419D60	RoomRove.00421BDA	
0012F1D0	00000000	Arg1 = 00000000		
0012F1D4	00000000	Arg2 = 00000000		
0012F1D8	00000000	Arg3 = 00000000		
0012F234	00421B47	RoomRove.00421120	RoomRove.00421B42	
0012F2B0	00421353	RoomRove.00421120	RoomRove.0042134E	
0012F32C	0040209A	RoomRove.00421120	RoomRove.00402095	
0012F4C8	0043DF6E	? RoomRove.004016B0	RoomRove.0043DF69	
0012F4E0	0043EA24	RoomRove.0043DEB0	RoomRove.0043EA1F	
0012F930	77D18709	Includes RoomRove.0043EA24	USER32.77D18706	
0012F95C	77D187EB	? USER32.77D186E1	USER32.77D187E6	
0012F9C4	77D1B368	? USER32.77D18734	USER32.77D1B363	0012F9C0

Double click on it then you land here:

0041EAB8	77 05	JR SHORT RoomRove.0041EAC2
0041EABD	B9 30854500	MOV ECX,RoomRove.00458530
0041EAC2	DD4424 20	FLD QWORD PTR SS:[ESP+20]
0041EAC6	83EC 08	SUB ESP,8
0041EAC9	DD1C24	FSTP QWORD PTR SS:[ESP]
0041EACC	56	PUSH ESI
0041EACD	50	PUSH EAX
0041EACE	51	PUSH ECX
0041EACF	E8 5CAA0200	CALL RoomRove.00449530

And the beginning of the checking routine is a little above:

0041E9A2	C2 0C00	RETN 0C
0041E9A5	8A45 02	MOV AL,BYTE PTR SS:[EBP+2]
0041E9A8	84C0	TEST AL,AL
0041E9AA	75 10	JNZ SHORT RoomRove.0041E9C9
0041E9AC	DD05 88864500	FLD QWORD PTR DS:[458688]
0041E9B2	83EC 08	SUB ESP,8
0041E9B5	DD1C24	FSTP QWORD PTR SS:[ESP]
0041E9B8	6A 00	PUSH 0

```
0041E9A5      8A45 02      MOV AL,BYTE PTR SS:[EBP+2]
```

Place a BP here and restart the program, on the data stack window you should see the real registration code (scroll down a little to see it, anyway it's at the address [ESP+DC])! In my case it was something like "errv-ayeva-xyta"..



0012B180	00000005	
0012B184	009FB4E8	ASCII "--ayeve-xyta"
0012B188	009F9A64	
0012B18C	004477B2	RETURN to RoomRove.004477B2 from RoomRove.004010
0012B190	009FB5A4	ASCII "ayeve"
0012B194	009FB4E8	ASCII "--ayeve-xyta"
0012B198	00000006	
0012B19C	00000005	
0012B1A0	00000005	
0012B1A4	009FB4E4	ASCII "erru-ayeve-xyta"
0012B1A8	009F9A04	
0012B1AC	009F9B12	
0012B1B0	0041C441	RETURN to RoomRove.0041C441 from RoomRove.0042C5
0012B1B4	009F9AC4	RETURN to 009F9AC4
0012B1B8	009E0048	ASCII "The Serial Number you are trying to enter
0012B1BC	00458530	RoomRove.00458530
0012B1C0	00000000	
0012B1C4	009F9B12	
0012B1C8	00000000	
0012B1CC	00000000	
0012B1D0	74000384	

et voilà the program is registered and fully operational!

Note that the registration number is then stored in the Window registry here:

HKEY_CURRENT_USER\Software\Boltex\Basic01

If you delete this key the program returns not registered. Remember when doing tests..

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>

