

Creating the Invisible Oraclecum

Program Registration Through Memory Manipulation

Gabri3l of ARTeam

Version 1.0 - July 2005

1. Abstract:

This tutorial will cover the development and testing of a modified oraclecum. The goal is to make the oraclecum virtually invisible to the end-user. To do so the oraclecum will be designed to function with little user input and no output. And the design of the oraclecum will be changed to allow for direct memory manipulation.

The oraclecum will be developed in such a way that it will function as a framework, allowing it to be adapted to any target by changing only a few variables.

Finally the oraclecum will be tested on two separate targets. This will demonstrate the adaptability of the framework and the overall flexibility of an oraclecum.

2. Problem:

Probably the most common program registration scheme is the serial registration. A serial is either unique or non-unique depending on the registration method. When working with non-unique serials it is simple enough to find and distribute those serials allowing for multiple registrations. A unique serial is a harder task. A unique serial is calculated based on one, or more, unique identifiers. These identifiers can be the registration name, computer name, or even hardware id. There are a few options that can be taken when a program uses a unique serial registration.

The first option is to calculate a serial based on a unique identifier. This identifier&serial combination can then be used to register multiple copies of the program. This attack fails when the unique identifier is either unable to be modified, complicated to modify, or even unknown. This attack also fails when the identifier&serial combination becomes known to the programmer. The programmer can then add either the serial or identifier to a blacklist, disallowing any registration with that identifier&serial for any future program versions.

The next option is to patch the registration scheme allowing for any serial to be used to register the program. This requires the analysis of the registration scheme, and modification of the original executable. This attack will fail if the executable saves the identifier&invalid-serial to a specific location. If the program is updated the registration scheme is not modified within the new executable, the program will then read the invalid serial from a file or the registry. This will cause the program to revert back to unregistered. Another flaw in this attack is that programs that update frequently often check for the presence of a valid non-modified executable before allowing an update. Even replacing the modified executable with a non-modified version may not keep the program registered. For every update there will be the need to re-analyze and re-patch the executable.

Another attack method is the development of a keygen. This will calculate a specific key based on submitted unique identifiers. This is the most effective method of defeating a registration scheme because there is no modification of the program, there is no single identifier and/or serial that can be blacklisted, and it will work for future updates until the

registration scheme is changed. This method, however, also requires the most work. The keygen developer needs to first understand the registration scheme, find out how to effectively reverse the scheme to allow for creating unique keys, and then they need to code the keygen. This can often be a time consuming project.

3. Solution:

Often times when a program is protected by a serial registration scheme the unique serial is compared against the invalid serial. When that takes place, many times the serial is fully calculated and resident in memory. This is a weakness that we can, and will, take advantage of. If we find out where in memory the serial is located we can read it out and register the program. These are often the steps taken when serial fishing a program. But as we know, the serial fishing method has its limitations. We want to develop a method to easily read that serial from memory. **Shub-Nigurrath[ARTeam]** has written a paper that outlines the development of what is know as an oracle. This a program that loads an executable, stops execution at a determined spot, and then reads the valid serial out of memory. The unique serial is displayed to the user. The program then continues as normal allowing the user to enter in the valid unique serial. The paper included both the use of, and framework for, the oracle. Developing an oracle is a easy method to allow the end-user to generate a valid unique serial. There is no modification of the original executable and no risk of blacklisting.

4. Goal:

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

5. Part 1:

Creating a framework:

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

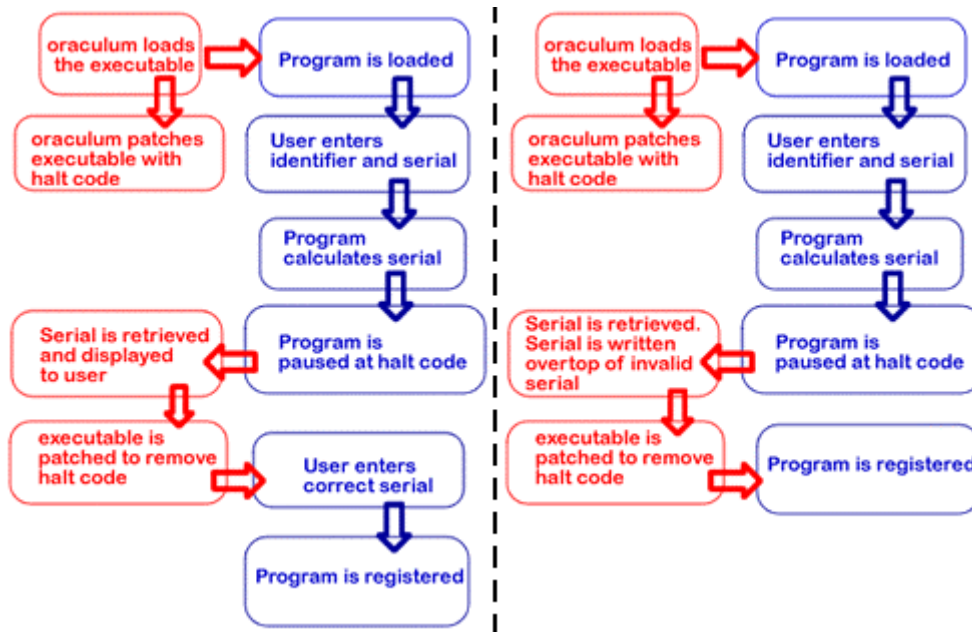


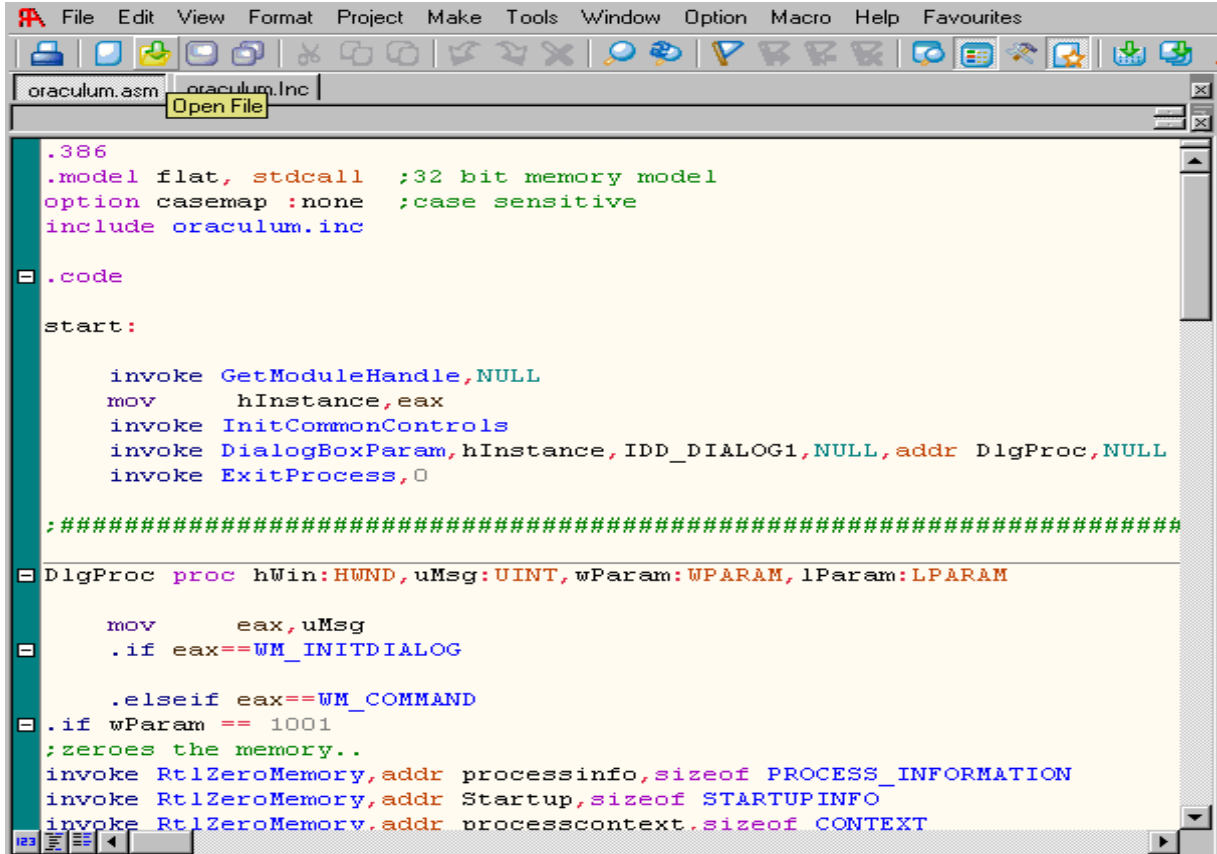
Figure 1 - Oraculum flowchart versus Invisible Oraculum flowchart

Developing a framework for the invisible oraculum will be easy since we have much of the source already available to us. The source was originally developed by **Shub-Nigurath** in C and was later adapted to ASM by **Hackerman**. For our framework we will use the ASM code written by **Hackerman**.

To begin you will need the [MASM32 package](#) and [RadASM](#). Once you have those installed go to <http://tutorials.accessroot.com> and get **Oraculum Tutorial With Framework Src V12 By Shub-nigurath.rar**

Extract the tutorial archive to your computer. Navigate to: Sources/oraculum_asm/. Within that folder you will see the assembly sources for an oraculum.

Begin by opening RadASM. From the File menu choose “Open Project”. Navigate to the `oraculum_asm/` folder and select **oraculum.rap**. You should now see the following:



```
.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
```

Figure 2 - Oraculum.asm code

Within the project you will see 4 files:

Oraculum.asm: Our main body of code. Contains all the functions that we will use to open, modify, and register the program

Oraculum.inc: This is the include file. This holds our defined variables and our undefined variables that will be used in oraculum.asm. This file is imported into oraculum.asm with the line: `include oraculum.inc`

Oraculum.dlg: This is our dialog menu. This is the menu you will see when you run the program.

Oraculum.rc: This file contains the resources to be included in your dialog menu.

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

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

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

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

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

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

My oraculum.inc file now looks like this:

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

Figure 3 - Oraculum.inc file after modification

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

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

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

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

Becomes:

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

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

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

Becomes:

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

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

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

And we replace the direct breakpoint reference:

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

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

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

Becomes:

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

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

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

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

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

```
;Fills with zero again to make our job clean  
invoke RtlZeroMemory,addr buffer,SERIAL_SIZE  
;#####COMMENTED OUT#####  
;Read the memory pointed by ECX and places into a buffer.  
;invoke ReadProcessMemory,processinfo.hProcess,processcontext.regEcx,addr buffer, SERIAL_SIZE, NULL  
;Show the real serial  
;invoke MessageBox,hWin,addr buffer,addr realserial,MB_ICONINFORMATION  
;#####  
  
;Read the good serial from memory location: goodlocation and store it into buffer  
invoke ReadProcessMemory,processinfo.hProcess,goodlocation,addr buffer, SERIAL_SIZE, NULL  
;Write the good serial from buffer onto of bad serial located at: badlocation.  
invoke WriteProcessMemory,processinfo.hProcess, badlocation, addr buffer, SERIAL_SIZE, NULL  
  
;Restore the original applications bytes so the application can continue running.  
invoke WriteProcessMemory,processinfo.hProcess, Breakpoint, addr OriginalCode,HALT_SIZE, byteswritten
```

Figure 4 - Oracleum.asm after modification

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

The final changed code for `oracle.asm` should be:

```
#####COMMENTED OUT#####  
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A REGISTRY  
  
;Read the memory pointed by ECX and places into a buffer.  
;invoke  
ReadProcessMemory,processinfo.hProcess,processcontext.regEcx,addr  
buffer, SERIAL_SIZE, NULL  
  
;Uncomment to show the real serial in a messagebox  
;invoke MessageBox,hWin,addr buffer,addr  
realserial,MB_ICONINFORMATION  
  
;Write the serial stored in buffer to memory pointed by EAX.  
;invoke  
WriteProcessMemory,processinfo.hProcess,processcontext.regEax,addr  
buffer, SERIAL_SIZE, NULL  
#####  
  
#####  
;USE THIS CODE IF YOUR SERIAL IS LOCATED IN A MEMORY LOCATION  
  
;Read the good serial from memory location: goodlocation and  
store it into buffer  
invoke ReadProcessMemory,processinfo.hProcess,Goodlocation,addr  
buffer, SERIAL_SIZE, NULL  
  
;Write the good serial from buffer ontop of bad serial located  
at: badlocation.  
invoke WriteProcessMemory,processinfo.hProcess, Badlocation, addr  
buffer, SERIAL_SIZE, NULL  
  
#####
```

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

6. Part 2:

Proof of Concept (Part 1):

We have developed our invisible oraculum and it is now time to put it into practice. Our first target will be a very simple serial fish: [deibiz_xxl's Learn The First Few Tricks #1](#).

- To follow along you will need the program: [Ollydbg](#).

Open Ollydbg and then open LTFFT.exe in Olly. You will be here:

Address	Disassembly	Comment
00401220	PUSH EBP	
00401221	MOV EBP, ESP	
00401223	SUB ESP, 8	
00401226	MOV DWORD PTR SS:[ESP], 1	
0040122D	CALL NEAR DWORD PTR DS:[<&msvcrt.__set_app_type]	msvcrt.__set_app_type
00401233	CALL LTFFT.00401100	
00401238	NOP	
00401239	LEA ESI, DWORD PTR DS:[ESI]	
00401240	PUSH EBP	
00401241	MOV EBP, ESP	
00401243	SUB ESP, 8	
00401246	MOV DWORD PTR SS:[ESP], 2	
0040124D	CALL NEAR DWORD PTR DS:[<&msvcrt.__set_app_type]	msvcrt.__set_app_type
00401253	CALL LTFFT.00401100	
00401258	NOP	
00401259	LEA ESI, DWORD PTR DS:[ESI]	
00401260	PUSH EBP	
00401261	MOV ECX, DWORD PTR DS:[<&msvcrt.atexit>]	msvcrt.atexit
00401267	MOV EBP, ESP	
00401269	POP EBP	
0040126A	JMP NEAR ECX	
0040126C	LEA ESI, DWORD PTR DS:[ESI]	
00401270	ASCII "UI:3Pe", 0	
00401277	MOV EBP, ESP	
00401279	POP EBP	
0040127A	JMP NEAR ECX	
0040127C	NOP	
0040127D	NOP	
0040127E	NOP	
0040127F	NOP	
00401280	PUSH EBP	

Figure 5 - LTFFT.exe loaded in Ollydbg

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

Type your password: ARTeam

Bad... you should work harder.

Press any key to continue . . .

If you are reading this paper I will assume you know how to search for this "BadBoy" message in Olly.

If you are unsure of how to proceed, visit:

<http://tutorials.accessroot.com> and start from **Beginner Olly Tutorial Part 1**

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

<pre> 0040132B . 85C0 TEST EAX,EAX 0040132D . 75 0E JNZ SHORT LTFFT.00401330 0040132F . C70424 EC3040 MOV DWORD PTR SS:[ESP],LTFFT.004030EC 00401336 . E8 B5050000 CALL <JMP.&msvcrt.printf> 0040133B . EB 0C JMP SHORT LTFFT.00401349 0040133D > C70424 183140 MOV DWORD PTR SS:[ESP],LTFFT.00403118 00401344 . E8 A7050000 CALL <JMP.&msvcrt.printf> 00401349 > C70424 393140 MOV DWORD PTR SS:[ESP],LTFFT.00403139 00401350 . E8 6B050000 CALL <JMP.&msvcrt.system> 00401357 . 75 0E JNZ SHORT LTFFT.00401363 </pre>	<pre> ASCII "Well done, have you patched your n printf ASCII "Bad... you should work harder.00" printf ASCII "PAUSE" system </pre>
--	--

Figure 6 - Location of “BadBoy” message in LTFFT.exe

Looking at the information window, we find that our BadBoy was called by 40132D. So we will set a breakpoint on that location.

Run the program again. When prompted; enter any serial. When you break at location 40132D look at the code in Olly:

<pre> 0040130D . E8 DE050000 CALL <JMP.&msvcrt.printf> 00401312 . E8 40000000 CALL LTFFT.00401357 00401317 . C74424 04 6041 MOV DWORD PTR SS:[ESP+4],LTFFT.00404060 0040131F . C70424 704040 MOV DWORD PTR SS:[ESP],LTFFT.00404070 00401326 . E8 A5050000 CALL <JMP.&msvcrt strcmp> 0040132B . 85C0 TEST EAX,EAX 0040132D . 75 0E JNZ SHORT LTFFT.00401330 0040132F . C70424 EC3040 MOV DWORD PTR SS:[ESP],LTFFT.004030EC 00401336 . E8 B5050000 CALL <JMP.&msvcrt.printf> 0040133B . EB 0C JMP SHORT LTFFT.00401349 0040133D > C70424 183140 MOV DWORD PTR SS:[ESP],LTFFT.00403118 00401344 . E8 A7050000 CALL <JMP.&msvcrt.printf> </pre>	<pre> printf ASCII "[DEIBIZ]" ASCII "ARTeam" ASCII "Well done, printf ASCII "Bad... you printf </pre>
--	---

Figure 7 - Serial comparison in LTFFT.exe

I have circled the code that we find interesting. The program takes the correct serial stored in location 00404060 and writes it to [ESP+4], it then takes the serial we entered stored at 00404070 and writes it to [ESP].

The program then does a string compare and jumps to GoodBoy or BadBoy accordingly.

This little bit of code is all the information we need to write our invisible oraculum. We will set our BP on 00401317. We know the Goodlocation: 00404060 and we know the Badlocation: 00404070. We also know the length of our serial: 9 bytes (8 bytes + NULL termination byte)

In RadASM we will modify our oraculum.inc file to:

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

Figure 8 - Oraculum.inc modified for LTFFT.exe

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

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

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

Type your password: ARTeam

Well Done, have you patched your name?

Press any key to continue . . .

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

Proof of Concept (Part 2):

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

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

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

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

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

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

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

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

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

#####
```

We begin by loading keygenme.exe into Olly.
Run the program and enter any name and serial combination.
You will get the message: "Bad Serial"

We search for that message in Olly and we end up here:

0040249E	. 56	PUSH ESI	
0040249F	. E8 C4470000	CALL keygenme.00406C68	
004024A4	. 83C4 08	ADD ESP,8	
004024A7	. 85C0	TEST EAX,EAX	
004024A9	. 75 15	JNZ SHORT keygenme.004024C0	
004024AB	. 6A 40	PUSH 40	
004024AD	. 68 0E114100	PUSH keygenme.0041110E	
004024B2	. 68 13114100	PUSH keygenme.00411113	
004024B7	. 6A 00	PUSH 0	
004024B9	. E8 B2DC0000	CALL <JMP.&USER32.MessageBoxA>	Style = MB_OK!MB_ICONASTERISK!MB_APPLMODAL
004024BE	. EB 5A	JMP SHORT keygenme.0040251A	Title = "Good"
004024C0	. 6A 40	PUSH 40	Text = "Good, Now Make a Keygen!"
004024C2	. 68 2C114100	PUSH keygenme.0041112C	hOwner = NULL
004024C7	. 68 31114100	PUSH keygenme.00411131	MessageBoxA
004024CC	. 6A 00	PUSH 0	
004024CE	. E8 9DDC0000	CALL <JMP.&USER32.MessageBoxA>	Style = MB_OK!MB_ICONASTERISK!MB_APPLMODAL
004024D3	. EB 45	JMP SHORT keygenme.0040251A	Title = "Nope"
			Text = "Bad Serial"
			hOwner = NULL
			MessageBoxA

Figure 9 - Location of "BadBoy" reference in Keygenme.exe

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

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

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

We see that the ESI register contains the correct serial, while the EAX register contains our entered serial.

With our new framework we can easily use our oracle to overwrite the EAX register with the value from ESI.

We must now decide where to put our Halt_code?

We will not put it at the same place we put our breakpoint: 0040249F because looking directly above our breakpoint we see that our serials are being PUSHed onto the stack:

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

So instead we will put our Halt_code on 0040249D just after the serial is calculated but before the serials are pushed onto the stack.

Back in RadASM modify these two invokes in your oraculum.asm file to the following:

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

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

Now modify the oraculum.inc file to:

```
#####

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

.data?
byteswritten      dd ?
hInstance         dd ?

#####
```

Figure 10 - Oraculum.inc modified for keygenme.exe

With the modified code, build oraculum.exe as you did before and place it in the same folder as keygenme.exe. When you are ready, run the oraculum. Keygenme.exe will load. You can now enter any user name and serial and press Okay. The result: You get the GoodBoy message!

In less than 10 minutes you have created a file that can defeat this programs protection and will work for every user. There was no modification of the program and no need to analyze the serial generation routine.

7. Conclusion:

Future of the oraculum

You have been provided a very simple framework to create an invisible oraculum. The potential for improvement is vast. With a little work you can add multiple breakpoints to accommodate serial routines that test portions of a serial at a time. There is also the ability to modify the code to store the correct serial until another breakpoint is reached. This would allow you to overwrite the incorrect serial at another point in the program. You can also remove the dialog resource completely and allow the oraculum to operate completely invisible to the user.

Faced with multiple protection schemes it is important to adapt. It is important to take many different and new approaches to the same problem. The oraculum is one of those approaches. A simple method now made simpler. I hope you enjoyed this tutorial and I sincerely hope that you learned something new. The tools have been given to you now it is up to you to create.

8. References:

- [1] "Oraculum Tutorial With Framework Src v1.2", Shub-nigurrath of ARTeam, <http://tutorials.accessroot.com>
- [2] "RadASM", KetilO, <http://www.radasm.com>
- [3] "MASM32", MASM32 Project, <http://www.masm32.com>
- [4] "Ollydbg", Oleh Yuschuk, <http://www.ollydbg.de>
- [5] "Learn the first few tricks #1", Debiz, <http://www.crackmes.de>
- [6] "D Keygenme", DaXXor 101, <http://www.crackmes.de>

9. Greetings:

Thanks to all the ARTeam members and ARTeam forum members.

Thanks to all the people who take time to write tutorials.

Thanks to all the people who continue to develop better tools.

Thanks to all the people at Exetools and Woodmann for providing great places of learning.

Thanks also to The Codebreakers Journal, and the Anticrack forum.

Thanks to all the great teams: SND, TSRH, MP2K, ICU, REA, and all the others.

And finally, thanks to you for taking the time and interest to read this tutorial.

10. Contact:

If you have any questions, comments, or complaints feel more than free to email me at:

Gabri3l2003[at]yahoo.com or stop by the ARTeam forum at: <http://forum.accessroot.com>

11. Disclaimer:

All code included with this tutorial is free to use and modify, we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form.