The Weakness of the Windows API
Part 1 in a 3 Part Series

Abusing Trust Relationships in Windows Architecture In Order To Defeat Program Protection

Gabri3l of ARTeam

Version 1.0 – October 2005

Index

1. Abstract:.......... 2

2. Windows Architecture and Trust: 2

3. Windows API: 4

4. Trust Level 1:.......... 7

4a. Internal Modification To Reroute API Call:.......... 8
4b. External Modification To Reroute API Call:.......... 13
5. Conclusion:.......... 24

6. References:.......... 24

7. Greetings:.......... 24

8. Contact:.......... 24

9. Disclaimer:.......... 24

10. Verification:.......... 24

1. Abstract:

When a program incorporates the Windows API into it's code a level of trust is assumed. The program trusts that the API will function as expected and return results that are correct. This trust relationship ends up becoming a very vulnerable target. This paper gives an overview of the current Windows API and covers the vulnerable trust locations. Simple attacks will then be demonstrated for all vulnerable locations.

The information in this paper may seem like common knowledge for the advanced reverser, but should be a good resource for those looking to learn the fundamentals of using Windows architecture against itself.

2. Windows Architecture and Trust:

Before we begin learning about the Windows API, we need to understand how Windows is structured. When using any operating system you need to understand that they operate at varying levels of privilege. What this means is that depending on what privilege level you operate at that determines how much permission you have over the operations of the computer. When talking about privilege levels we need to think in terms of "Rings".

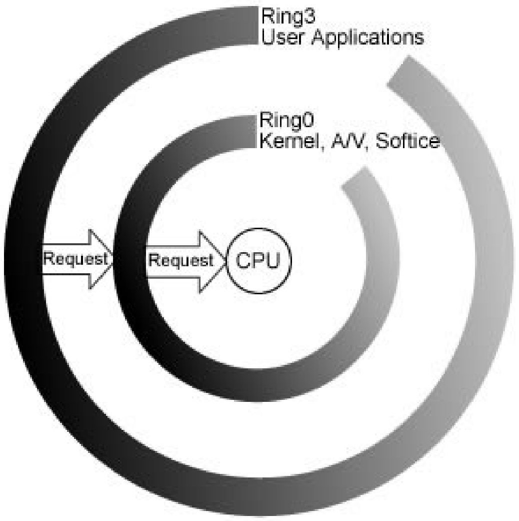


Figure 1 – The "Ring" structure of Operating Systems

The CPU is at the center, and around the CPU is Ring0. Ring0 is the Ring with the highest privilege level. Operations performed at Ring0 are in direct operation with the CPU. This is where the Windows kernel resides and often your A/V will run with Ring0 privileges. Ring3 is where all other Windows Applications run. Ring3 is also often called "User Mode". Programs that run in Ring3 have much less privileges than programs operating in Ring0. Ring3 applications cannot directly interact with the CPU. Instead they must submit a request to the kernel running in Ring0. The kernel then requests the operation to be performed by the CPU.

Communication between Ring3 and Ring0 architecture is separated into three trust levels. The **first level of trust** for the Ring3 program is the assumption that the request intended for the Ring0 system is actually received by the Ring0 system. This is the first weak link in the trust relationship between the Ring3 and Ring0 systems. Once the first level of trust is assumed the **second level of trust** begins. Requests sent by Ring3 programs are sent under the assumption that the Ring0 system is secure and has not been compromised. The Ring3 program relies upon the Ring0 system to perform the intended operation, and perform it correctly. By relying upon a secure Ring0 system a second weak link appears in the trust relationship. The **third trust level** is an extension of the second level and exists more in the Ring3 system than communication between Ring0 and Ring3. When the Ring0 operation completes, execution is returned to the Ring3 program. Often, when Ring0 operations are completed a variable is returned to the Ring3 program informing it of important information. The Ring3 program trusts that the variables have not been intercepted, and this is where the third weak link in the Windows architecture appears.

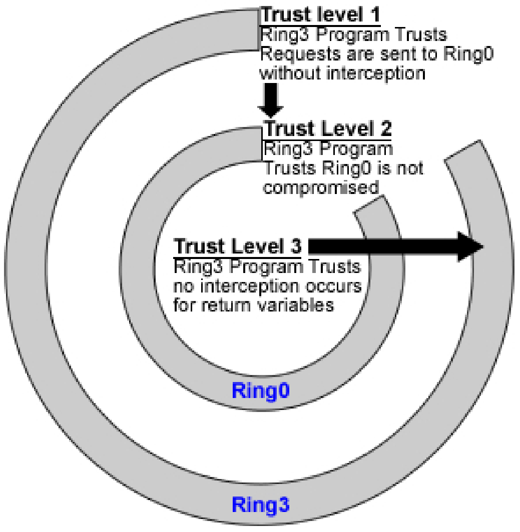

Trust level 1
Ring3 Program Trusts Requests are sent to Ring0 without interception
Trust Level 2
Ring3 Program Trusts Ring0 is not compromised
Trust Level 3
Ring3 Program Trusts no interception occurs for return variables
Ring0
Ring3

Figure 2- Graphical Representation of Windows Trust Model

Because of how Windows architecture is developed, these trusted relationships can be abused in many ways. To learn more about the different methods, we are going to examine the the trust relationships by running a debugger. The debugger will allow you to intercept and re-route outgoing Ring3 requests. The debugger can also allow you to modify current Ring0 operations, substituting created code for the expected operation. Finally the debugger will allow you intercept and manipulate return variables before execution is returned to the Ring3 program.

When debugging an application your debugger will run in either Ring0 or Ring3 depending on the debugger. **SoftICE by Compuware**[1] is a debugger that runs in Ring0. This means that when you activate SoftICE you can intercept and manage all of the Ring3 operation. This gives you much more power over the computers operation, but it also has drawbacks because it interrupts all windows execution and has a steep learning curve.

[1] SoftICE: http://www.compuware.com/products/devpartner/softice.htm

A debugger that operates in Ring3 means that the debugger will place itself between the running program and Ring3. The debugger intercepts any operations performed by the debugged application. This means, however, that the debugger must still then pass all requests to the Ring0 kernel. Another drawback of a Ring3 debugger is the fact that it only manages to debug one program at a time, and not all Ring3 operations as a Ring0 debugger would do. One of the most popular Ring3 debuggers is **Ollydbg by Oleh Yuschuk**[2].

3. Windows API:

Because applications running in Ring3 need to send requests to the kernel; Windows has created functions that User Applications can use to request specific operations to be performed by the kernel. These functions are called the Windows API (**A**pplication **P**rogramming **I**nterface). It is the existence of these functions that allow for program developers to easily perform low level operations without the need to run at a high privilege level. It is also the existence of these functions that provide for stability of the operating system. When a program needs to access a low level function they just call a specific API function, it would be chaos if every separate program that wanted to access a file had their own method of doing so and their own way of opening data. The Windows API ensures that every time a program opens a file it is opened the same way and it allows the kernel to manage what program has permission to open, close, or modify data.

When an API function is used, the program still needs to tell the API function exactly what needs to be done. This is achieved by passing variables to the API function when it is called. These variables are commonly called Arguments or Parameters. An example of an API function that requires Parameters is the **Sleep** command.

The Sleep function suspends the execution of the current thread for a specified interval.

```
VOID Sleep(
  DWORD dwMilliseconds          // sleep time in milliseconds
 );
```

Parameters

dwMilliseconds

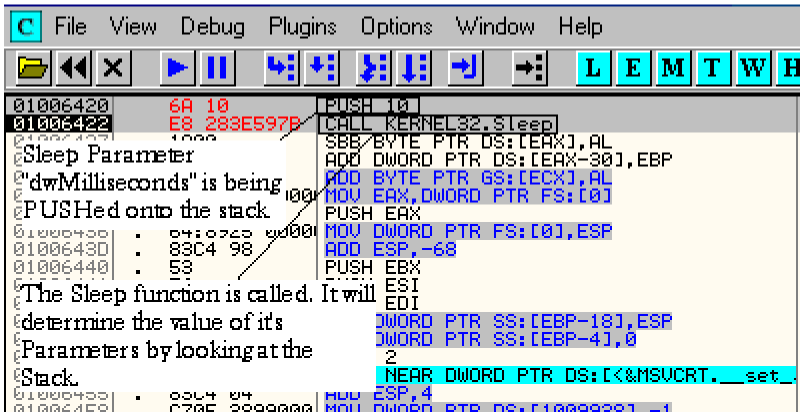
Specifies the time, in milliseconds, for which to suspend execution.

When calling the **Sleep** function the program must also pass to the Parameter "dwMilliseconds". This parameter tells the kernel exactly how long to make the current thread "sleep".

The Parameters of an API function are often the weakest point of a program. Because the API functions require specific information to work correctly, the program freely passes that information along. This simple exchange of information allows a debugger to read and/or modify the API arguments. Determining the function values when debugging a program is

[2] Ollydbg: http://www.ollydbg.de/

simple. All API function values are PUSHed onto the Stack prior to calling the function. When the function is called; it POPs the values off the Stack to fill in it's parameters.
For example let us look at what the Sleep API function call looks like when using Ollydbg:



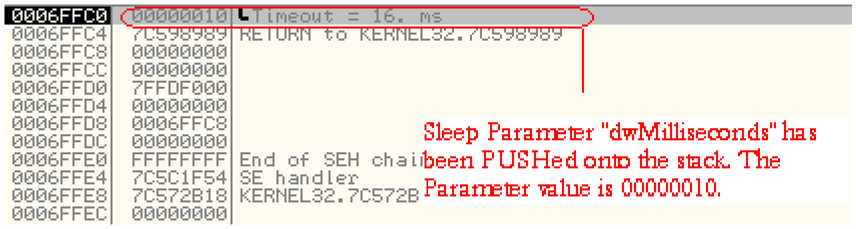
```
01006420     6A 10        PUSH 10
01006422     E8 283E597B  CALL KERNEL32.Sleep
                          SBB BYTE PTR DS:[EAX],AL
                          ADD DWORD PTR DS:[EAX-30],EBP
                          ADD BYTE PTR GS:[ECX],AL
                          MOV EAX,DWORD PTR FS:[0]
                          PUSH EAX
                          MOV DWORD PTR FS:[0],ESP
0100643D  .  83C4 98      ADD ESP,-68
01006440  .  53           PUSH EBX
                          ESI
                          EDI
                          DWORD PTR SS:[EBP-18],ESP
                          DWORD PTR SS:[EBP-4],0
                          2
                          NEAR DWORD PTR DS:[<&MSVCRT.__set_
                          ESP,4
```

Sleep Parameter "dwMilliseconds" is being PUSHed onto the stack.

The Sleep function is called. It will determine the value of it's Parameters by looking at the Stack.

Figure 3 – The Sleep function being called

Looking at the code we can see that the program first PUSHes the value 10 onto the Stack. Then the API function Sleep is called.
Looking at the Stack just before Sleep is called, we can see our Parameter value at the top of the Stack:



```
0006FFC0   00000010  └Timeout = 16. ms
0006FFC4   7C598989  RETURN to KERNEL32.7C598989
0006FFC8   00000000
0006FFCC   00000000
0006FFD0   7FFDF000
0006FFD4   00000000
0006FFD8   0006FFC8
0006FFDC   00000000
0006FFE0   FFFFFFFF  End of SEH chai
0006FFE4   7C5C1F54  SE handler
0006FFE8   7C572B18  KERNEL32.7C572B
0006FFEC   00000000
```

Sleep Parameter "dwMilliseconds" has been PUSHed onto the stack. The Parameter value is 00000010.

Figure 4 – The Stack just before the Sleep function is called

When Windows executes the sleep function it will use the value from the top of the stack to fill in it's "dwMilliseconds" Parameter. This means if we executed this specific section of code, the program would sleep for 16 milliseconds.

After Sleep has completed running, program execution is returned to the main executable. However, in many instances the API functions need to return a value to the main executable. The returned value for API functions, along with function parameters, are all defined in the MSDN Windows API Guide[3]. Another resource for Windows API definitions is the Win32.hlp[4] file.

An example of an API function that returns a value is IsBadCodePtr. This API function can be called to determine if the program can read memory from a specific location. The argument passed to the IsBadCodePtr function is **lpfn**; a memory address location. The IsBadCodePtr function then checks to see if the location in memory can be read from. If

[3]MSDN Windows API Guide: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_start_page.asp
[4]Win32 API Reference: http://spiff.tripnet.se/~iczelion/download.html

the memory location can be read by the program the function returns 0. If the memory cannot be read then the function returns a non-null value.

The **IsBadCodePtr** function determines whether the calling process has read access to the memory at the specified address.

BOOL **IsBadCodePtr**(
  FARPROC lpfn   // address of function
);

Parameters

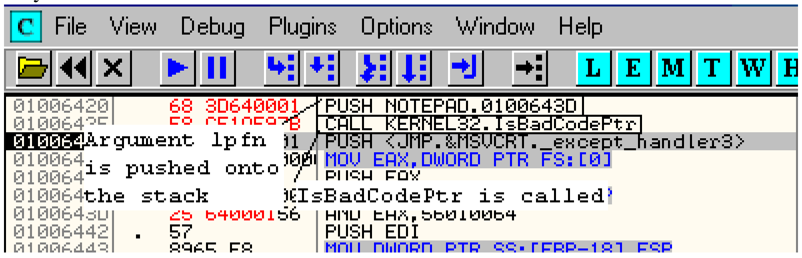
lpfn

Points to an address in memory.

Return Values

If the calling process has read access to the specified memory, the return value is zero.
If the calling process does not have read access to the specified memory, the return value is nonzero. To get extended error information, call GetLastError.

It is important to know that when a value is returned by an API function it is always returned to the EAX register. This is what the IsBadCodePtr function looks like when called within Olly:



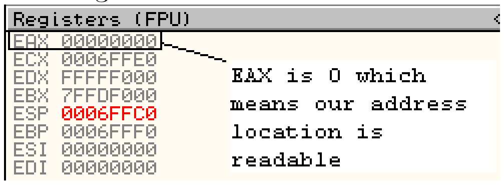
```
01006420      68 3D640001  PUSH NOTEPAD.0100643D
010064        E8           CALL KERNEL32.IsBadCodePtr
010064                     PUSH <JMP.&MSVCRT._except_handler3>
010064                     MOV EAX,DWORD PTR FS:[0]
010064                     PUSH EAX
010064
01006442   .  57           PUSH EDI
```

Argument lpfn is pushed onto the stack

IsBadCodePtr is called

Figure 5 – IsBadCodePtr being called

The argument passed is 0100643D which we can see is directly below the calling location, so the function will return 0 letting us know that the location is readable.



```
Registers (FPU)
EAX 00000000
ECX 0006FFE0
EDX FFFFF000
EBX 7FFDF000
ESP 0006FFC0
EBP 0006FFF0
ESI 00000000
EDI 00000000
```

EAX is 0 which means our address location is readable

Figure 6 – The Registers after IsBadCodePtr is called

If we had passed an argument such as FF for **lpfn** the API function would have returned a nonzero value letting us know that the location we specified is unreadable. By allowing the Windows API to communicate with the program through return values we give the Ring3 programs more power to operate as a Ring0 program would. However, because the return values are well documented, and are returned through the EAX register it we can see our weak point in the trust relationship between the Ring3 program and the the API.

In some instances an API function needs to return more detailed information than the EAX register will allow. In this instance the API function can manipulate memory allocated by the Ring3 program. The memory to be manipulated is passed to the API function along with the other arguments. An example of an API function that returns a value and manipulates memory is GetFileSize:

The **GetFileSize** function retrieves the size, in bytes, of the specified file.

```
DWORD GetFileSize(
   HANDLE hFile,     // handle of file to get size of
   LPDWORD lpFileSizeHigh          // address of high-order word for file size
  );
```

Parameters

hFile

Specifies an open handle of the file whose size is being returned. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

lpFileSizeHigh

Points to the variable where the high-order word of the file size is returned. This parameter can be NULL if the application does not require the high-order word.

Return Values

If the function succeeds, the return value is the low-order doubleword of the file size, and, if lpFileSizeHigh is non-NULL, the function puts the high-order doubleword of the file size into the variable pointed to by that parameter.
If the function fails and lpFileSizeHigh is NULL, the return value is 0xFFFFFFFF. To get extended error information, call GetLastError.
If the function fails and lpFileSizeHigh is non-NULL, the return value is 0xFFFFFFFF and GetLastError will return a value other than NO_ERROR.

When GetFileSize is called the arguments passed are **hFile:** t*he handle of the file* and **lpFileSizeHigh:** *a memory address for the program to return the high-order word.* When GetFileSize is executed the low-order dword is passed back to the program through EAX while the high-order dword is written to the address defined in lpFileSizeHigh. By knowing where an API function returns information we can specifically focus on that memory location in order to gather and manipulate data that the Ring3 program is using.

4. Trust Level 1:

The first trust level is when a Ring3 program assumes that the request sent to the Windows API has not been intercepted or modified. Because this implicit trust exists there is no verification system for API calls. We can use this lack of security to modify program behavior by intercepting specific API calls. There are different methods to intercept API calls. The first involves modifying application code internally to patch the DLL loaded in memory. The second involves using an external method, often called hooking, to reroute outgoing API calls to user created code.

4a. Internal Modification To Reroute API Call:

Our goal in this section is to write a program that will modify the call to the API function. The modified call will redirect to user code rather than executing the normal function. This modification can be easily achieved when we have direct access to the executable. To understand we will examine how an executable makes a call to an API function.

Windows API functions are not static within their DLLs. This means that the address of a DLL call may change between machines. This is especially the case between operating system versions ex: NT/XP/XP SP1/XP SP2. Because the program needs to work on multiple Windows versions, the addresses of API function calls are not hardcoded into the executable. Instead, when an executable is first loaded, the Windows loader is responsible for loading the executable image into memory. The executable lists all of the functions it will require from each dll. The Windows loader then modifies a table of function pointers in the executable so that they all point to the correct API location. This is called the **import address table.**[5] This table of pointers is accessed by the executable in two different ways. Either by directly using the pointer in the function call:

CALL DWORD PTR DS:[401025]
Where **401025** contains the address location of the API function.

Or calls can be accessed indirectly through what is called a Thunk Table, this is also often called a JMP Table. In this case, the call to an API function is directed to the Thunk Table where a JMP is made to the function location:

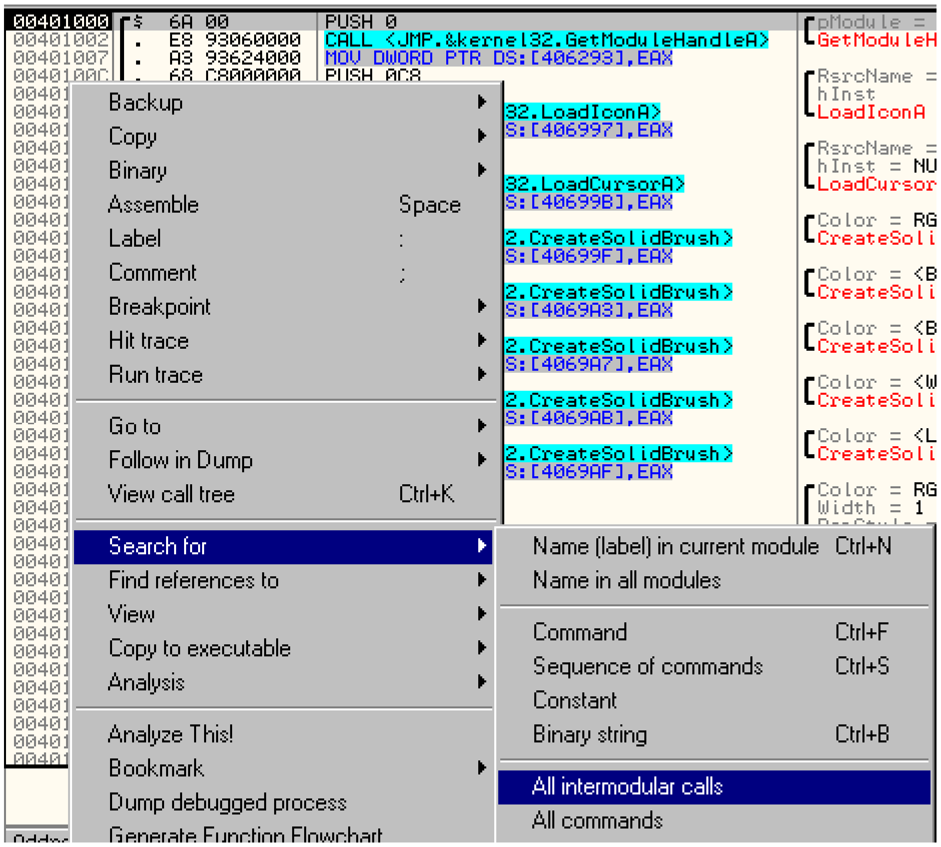
CALL 401090
401090: JMP DWORD PTR DS:[401025] ;OUR THUNK TABLE
Where **401025** contains the address location of the API function

Now that we understand how the call to a function is made, we can modify the calling convention to always call our own function. Through either redirecting the direct address or modifying the Thunk Table to jump to our own code.

The target for this exercise is called HookMe.exe and is included with the tutorial. Our goal is to modify HookMe.exe so that the call to the MessageBox function displays our own text rather than the hardcoded text of the executable.
Begin by opening Hookme.exe in Olly, you will find yourself located at the Entry Point of the executable. We are going to start by finding out whether our executable uses the Direct

[5] Understanding the Import Address Table: http://sandsprite.com/CodeStuff/Understanding_imports.html

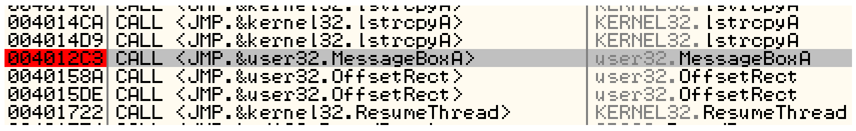
Inline method of API pointer access or the indirect method of a JMP Thunk Table. Right-Click and Search for->All Intermodular Calls:



```
00401000 $  6A 00        PUSH 0
00401002 .  E8 93060000  CALL <JMP.&kernel32.GetModuleHandleA>
00401007 .  A3 93624000  MOV DWORD PTR DS:[406293],EAX
```

Figure 7 – Entry Point of HookMe.exe

In the new window containing our called API functions, scroll down until you find the CALL to user32.MessageBoxA. Look at our CALL to MessageBoxA. The CALL is actually calling a JMP rather than the actual function location. This means that our executable uses the Thunk Table method to call API functions. This will make redirecting our MessageBox function even easier. Now, set a BP on the call to MessageBoxA so when we break we can redirect the function to our own code.



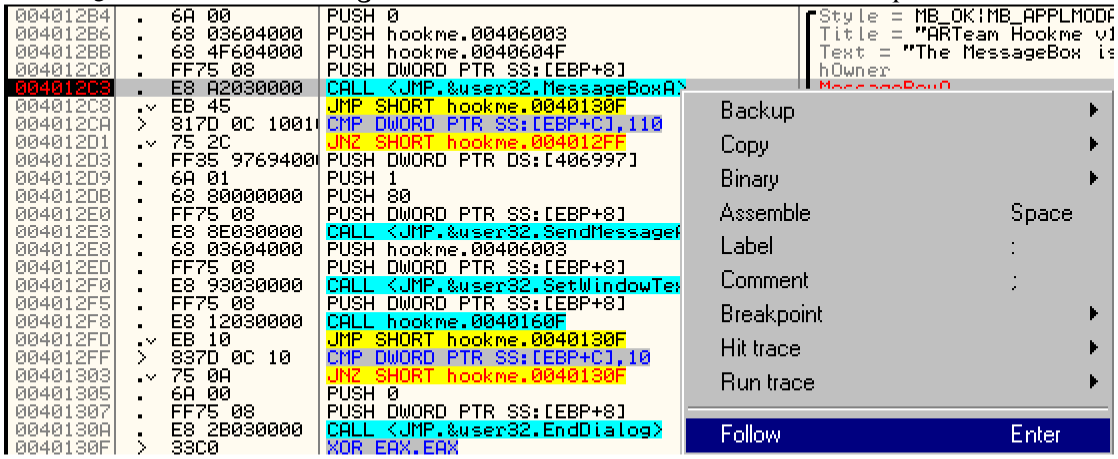
```
004014CA CALL <JMP.&kernel32.lstrcpyA>      KERNEL32.lstrcpyA
004014D9 CALL <JMP.&kernel32.lstrcpyA>      KERNEL32.lstrcpyA
004012C3 CALL <JMP.&user32.MessageBoxA>     user32.MessageBoxA
0040158A CALL <JMP.&user32.OffsetRect>      user32.OffsetRect
004015DE CALL <JMP.&user32.OffsetRect>      user32.OffsetRect
00401722 CALL <JMP.&kernel32.ResumeThread>  KERNEL32.ResumeThread
```

Figure 8 – Breakpoint on MessageBoxA function

Once the breakpoint has been set, run HookMe.exe. After HookMe.exe loads press the MessageBox button and you will break on the call to MessageBoxA.

We currently know that HookMe.exe uses the Thunk Table method for calling API functions. This is useful when we are looking to redirect functions because it allows us to modify only one location and redirect all specific function calls. Assume that we wanted to redirect every GlobalAlloc call. If the executable used the Direct Inline method of function calling we would have to edit all 12 calls in the program. However because the executable uses the Thunk Table method we can just modify one JMP and all 12 calls will then JMP to our modified location.

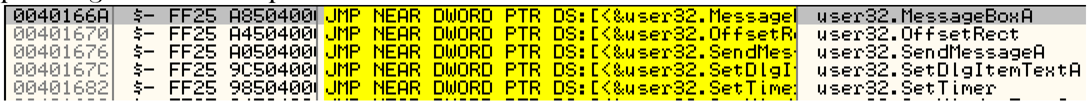
Let's continue by following our MessageBox CALL to our JMP Thunk Table. Right-Click on **Call <JMP.&user32.MessageBoxA>** and choose Follow from the drop down menu.



```
004012B4  .   6A 00          PUSH 0                                   Style = MB_OK|MB_APPLMODA
004012B6  .   68 03604000    PUSH hookme.00406003                     Title = "ARTeam Hookme v1
004012BB  .   68 4F604000    PUSH hookme.0040604F                     Text = "The MessageBox is
004012C0  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]                hOwner
004012C3  .   E8 A2030000    CALL <JMP.&user32.MessageBoxA>
004012C8  .v  EB 45          JMP SHORT hookme.0040130F
004012CA  >   817D 0C 1001   CMP DWORD PTR SS:[EBP+C],110
004012D1  .v  75 2C          JNZ SHORT hookme.004012FF
004012D3  .   FF35 9769400   PUSH DWORD PTR DS:[406997]
004012D9  .   6A 01          PUSH 1
004012DB  .   68 80000000    PUSH 80
004012E0  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]
004012E3  .   E8 8E030000    CALL <JMP.&user32.SendMessageA
004012E8  .   68 03604000    PUSH hookme.00406003
004012ED  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]
004012F0  .   E8 93030000    CALL <JMP.&user32.SetWindowTex
004012F5  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]
004012F8  .   E8 12030000    CALL hookme.0040160F
004012FD  .v  EB 10          JMP SHORT hookme.0040130F
004012FF  >   837D 0C 10     CMP DWORD PTR SS:[EBP+C],10
00401303  .v  75 0A          JNZ SHORT hookme.0040130F
00401305  .   6A 00          PUSH 0
00401307  .   FF75 08        PUSH DWORD PTR SS:[EBP+8]
0040130A  .   E8 2B030000    CALL <JMP.&user32.EndDialog>
0040130F  >   33C0           XOR EAX,EAX
```

Figure 9 – Follow the Call to the Thunk Table

We are now located at one of the programs JMP table locations. Keep in mind that not all JMP tables are located together in an executable. Many times they may be spread out depending on how complex the executable is.



```
0040166A  $-  FF25 A850400  JMP NEAR DWORD PTR DS:[<&user32.Message   user32.MessageBoxA
00401670  $-  FF25 A450400  JMP NEAR DWORD PTR DS:[<&user32.OffsetR   user32.OffsetRect
00401676  $-  FF25 A050400  JMP NEAR DWORD PTR DS:[<&user32.SendMes   user32.SendMessageA
0040167C  $-  FF25 9C50400  JMP NEAR DWORD PTR DS:[<&user32.SetDlgI   user32.SetDlgItemTextA
00401682  $-  FF25 9850400  JMP NEAR DWORD PTR DS:[<&user32.SetTime   user32.SetTimer
```

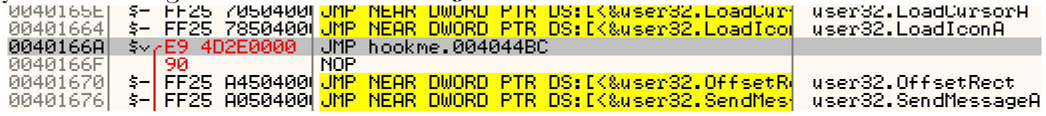
Figure 10 – Jump Table

We know that when the executable wants to call the MessageBox function it will actually CALL this JMP location. So if we were to change the JMP so that it redirected to somewhere else, every time the program calls MessageBoxA it will JMP to our own code rather than MessageBoxA.

To continue we need to find some space in the executable to write our own code. We will then change the JMP to direct to this code-cave.[6] With this program there is a good bit of extra space near the end of the executable: 4044A6 and up. In my case I chose to begin my code at 4044BC. So that is where we will redirect the JMP to. Now press SPACE to Assemble the current line. Before changing any code, select the **JMP NEAR DWORD PTR DS:[4050A8]**, or whatever JMP instruction appears in the Assemble Box. Right-Click and choose Copy. This will save our JMP code to the clipboard so we can use it later to jump directly to the MessageBox function. In the new box that opens up, change the code

[6]For more information about code-caves and how to locate them read Beginner Olly Tutorial Part5: http://tutorials.accessroot.com

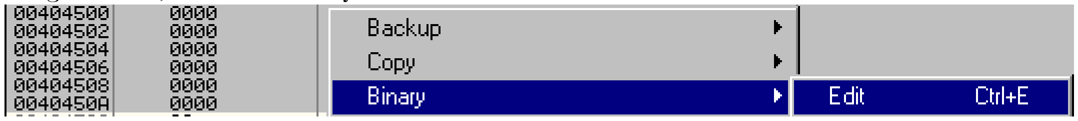
to JMP 4044BC, also check "Fill with NOP's, and press Assemble to modify the line. Now every time MessageBox is called we will JMP to our code-cave instead.



```
0040165E  $-  FF25 7050400 JMP NEAR DWORD PTR DS:[<&user32.LoadCur  user32.LoadCursorA
00401664  $-  FF25 7850400 JMP NEAR DWORD PTR DS:[<&user32.LoadIco  user32.LoadIconA
0040166A  $v  E9 4D2E0000  JMP hookme.004044BC
0040166F      90           NOP
00401670  $-  FF25 A450400 JMP NEAR DWORD PTR DS:[<&user32.OffsetR  user32.OffsetRect
00401676  $-  FF25 A050400 JMP NEAR DWORD PTR DS:[<&user32.SendMes  user32.SendMessageA
```

Figure 11 – Modified JMP Table

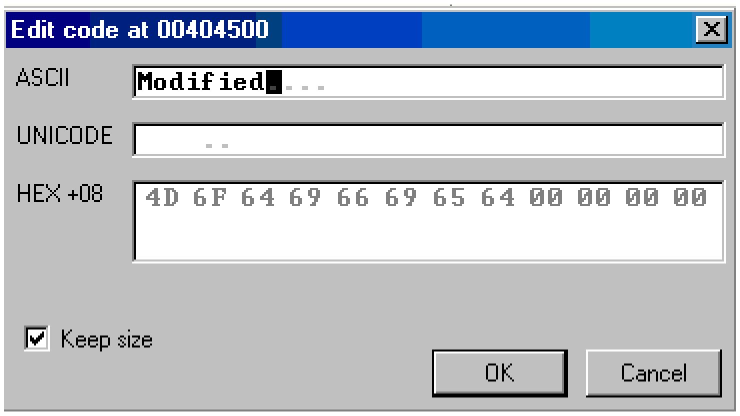
Now the object of our new code is to replace the hardcoded message with our own modified message. This means that we will also need to find a location to store our own text. Again, space for the modified text can be easily found by scrolling down in Ollydbg. I put my text at 404500. When you find the location you wish to use, select the bytes in Olly and Right-Click, choose Binary->Edit:



```
00404500    0000
00404502    0000
00404504    0000
00404506    0000
00404508    0000
0040450A    0000
```

Figure 12 – Choosing to Binary Edit Memory

In the new box, choose the ASCII input box and type whatever text you desire. Press OK when you are finished. Be sure to note the exact location that you are editing, since we will be using it in our code.



Edit code at 00404500

ASCII: Modified...

UNICODE:

HEX +08: 4D 6F 64 69 66 69 65 64 00 00 00 00

[✓] Keep size

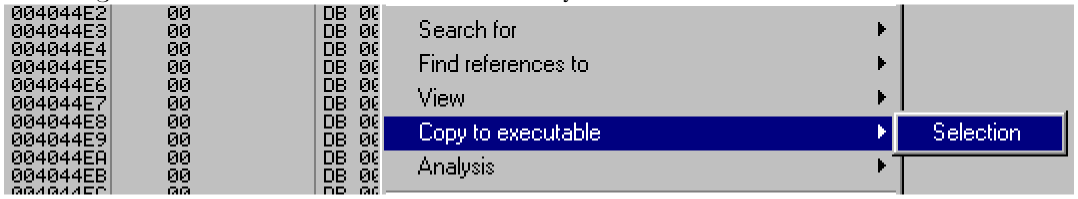
Figure 13 – The Binary Edit Box

Once you have edited the code to contain the replacement text, return to the code-cave location. We are now going to write some code to replace the MessageBox "Text" argument on the stack with our own substitute text. Here is the code I wrote to do so:

```
PUSH ECX                              ;PUSH ECX TO PRESERVE REGISTER INTEGRITY
MOV EAX,ESP                           ;MOVE THE TOP OF THE STACK INTO EAX
ADD EAX,0C                            ;ADD 12 (0C) TO EAX TO POINT TO OUR "TEXT" ARGUMENT
MOV ECX,hookme.00404500               ;MOVE THE ADDRESS OF OUR OWN TEXT INTO ECX
MOV DWORD PTR DS:[EAX],ECX            ;MOVE OUR SUBSTITUTE TEXT OVERTOP THE ORIGINAL
POP ECX                               ;RESTORE ECX
JMP NEAR DWORD PTR DS:[<&user32.MessageB>   ;JMP TO MESSAGEBOXA AND CONTINUE EXECUTION
```

This code will make EAX point to the "Text" argument on the stack, it will then overwrite that argument with the address of our newly created substitute text. Once the replacement has occurred and ECX has been restored. Choose the next empty line down and Assemble

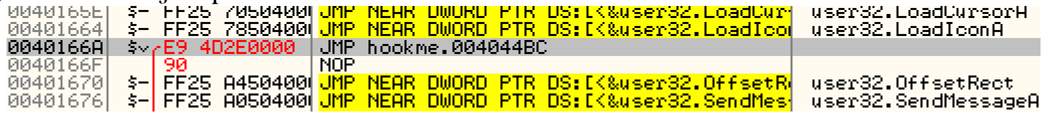
the line. In the Assembly box Right-Click and Paste our JMP we copied earlier. Now it will JMP back to MessageBoxA to continue normal program execution.
We can now save all the modifications we made to the executable file. Highlight the code from 4044BC to the end of the text we added at 404500. Once all the code has been selected Righ-Click and choose Copy to Executable and then choose Selection. In the new window Right-Click and Save the file. I saved my modified file as Hookme.Modified.exe.



```
004044E2    00    DB 0
004044E3    00    DB 0
004044E4    00    DB 0
004044E5    00    DB 0
004044E6    00    DB 0
004044E7    00    DB 0
004044E8    00    DB 0
004044E9    00    DB 0
004044EA    00    DB 0
004044EB    00    DB 0
```

Figure 14 – Copy Modifications to Executable

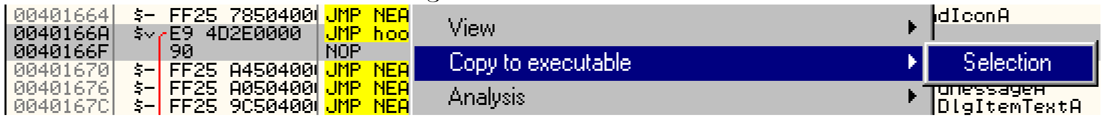
Now with the file saved we need to re-modify the JMP table. Olly will only either save changes inside the code section or save a selection of modified code. Because we can only save one or the other we chose to save the code we added in the code cave since re-writing it would take more time than modifying a JMP. This means our modification to the JMP table was not saved. We can easily fix this by opening Hookme.Modified.exe in Olly and following the same steps as we took before to find the JMP to MessageBoxA . Re-modify the JMP table to jump to our code cave as before:



```
00401664  $-  FF25 7850400  JMP NEAR DWORD PTR DS:[<&user32.LoadIco   user32.LoadIconA
0040166A  $v  E9 4D2E0000   JMP hookme.004044BC
0040166F      90            NOP
00401670  $-  FF25 A450400  JMP NEAR DWORD PTR DS:[<&user32.OffsetR   user32.OffsetRect
00401676  $-  FF25 A050400  JMP NEAR DWORD PTR DS:[<&user32.SendMes   user32.SendMessageA
```

Figure 15 – Modified JMP Table

Once the changes have been made, select both lines 40166A and 40166F and Right-Click choose Copy to Executable and then choose Selection. In the new window Right-Click and Save the file. You can save the file again as Hookme.Modified.exe.



```
00401664  $-  FF25 7850400  JMP NEA
0040166A  $v  E9 4D2E0000   JMP hoo
0040166F      90            NOP
00401670  $-  FF25 A450400  JMP NEA
00401676  $-  FF25 A050400  JMP NEA
0040167C  $-  FF25 9C50400  JMP NEA
```

Figure 16 – Copy Modified JMP to Executable

Once our new executable has been saved we can close Olly. It is now time to see if the modifications we made the the executable work. Find Hookme.Modified.exe and Run it. Once it has loaded press the MessageBox button.
The final result:

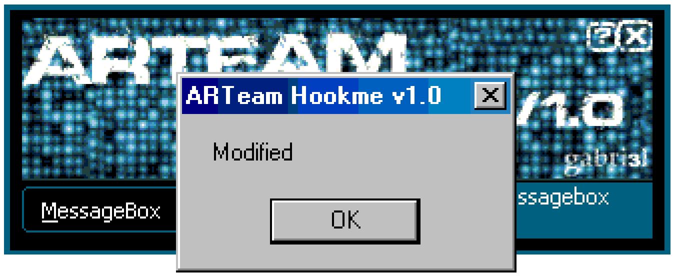


Figure 17 – Hookme Running with an Internal Modification

This was a very simple method of internal modification. By modifying the JMP table we are able to take advantage of the trust between the Ring3 program and the Ring0 system. This provides you the opportunity to modify and and adapt what you have just learned. With the ability to internally redirect specific API calls you can change how the program interacts with the kernel. This attack can easily be used against simple program protection schemes. For example, assume that you have a program that performs an integrity check on itself. Rather than trying to find out exactly how to bypass that integrity check, you could redirect to a code cave and change the arguments for CreateFileA to point to the original file renamed with a .BAK extension. And it would not be much more complex than simply replacing text:

```
PUSHAD                            ;PUSH REGISTERS
MOV ECX,DWORD PTR DS:[ESP+24]     ;MOVE THE LOCATION OF "FILENAME" ARGUMENT INTO ECX

MOV DL,BYTE PTR DS:[ECX]          ;**ROUTINE LOOP** MOVE FIRST STRING BYTE INTO DL
CMP DL,0                          ;CHECK TO SEE IF DL IS NULL
JE SHORT END                      ;IF DL IS NULL JUMP TO END
CMP DL,2E                         ;COMPARE DL TO "."
JE SHORT EXE COMPARE              ;IF DL IS "." JUMP TO EXE COMPARE
INC ECX                           ;ELSE INCREMENT ECX
JMP SHORT ROUTINE LOOP            ;JUMP BACK TO TOP OF ROUTINE LOOP

INC ECX                           ;**EXE COMPARE** INCREMENT ECX
MOV DL,BYTE PTR DS:[ECX]          ;MOVE NEXT CHARACTER INTO DL
CMP DL,65                         ;COMPARE DL AGAINST "e"
JNZ SHORT ROUTINE LOOP            ;JUMP IF NOT "e" BACK TO ROUTINE LOOP
INC ECX                           ;INCREMENT ECX
MOV DL,BYTE PTR DS:[ECX]          ;MOVE NEXT CHARACTER INTO DL
CMP DL,78                         ;COMPARE DL AGAINST "x"
JNZ SHORT ROUTINE LOOP            ;IF DL IS NOT "x" JUMP BACK TO ROUTINE LOOP
INC ECX                           ;INCREMENT ECX
MOV DL,BYTE PTR DS:[ECX]          ;MOVE NEXT CHARACTER INTO DL
CMP DL,65                         ;COMPARE DL AGAINST "e"
JNZ SHORT ROUTINE LOOP            ;JUMP IF NOT "e" BACK TO ROUTINE LOOP
DEC ECX                           ;DECREMENT ECX TO MOVE BACK TO BEGINNING OF "exe"
DEC ECX                           ;DECREMENT ECX TO MOVE BACK TO BEGINNING OF "exe"
MOV DWORD PTR DS:[ECX],4B4142     ;MOVE "BAK" over "exe"

POPAD                             ;**END** RESTORE REGISTERS
JMP DWORD PTR DS:[CREATEFILEA]    ;JUMP BACK TO CREATEFILEA
```

This code is not perfect and is just a quick example of how you can apply this knowledge to a real life protection situation. Because these examples are simple they are not always encountered in program protection schemes. If the executable was packed or protected you would not have access to the JMP table without unpacking it. This makes modifying a function's arguments harder. Since we are unable to modify the packed source we need to move onto external methods of modifying function calls.

4b. External Modification To Reroute API Call:

When the option of internally modifying an executable becomes to hard or complicated, we need to develop a way to externally reroute an API call. This method will still take advantage of Trust Level 1 and bridge the gap to Trust Level 2. Our goal is still to modify the function arguments of an API call. In this section we are going to focus on the same program, hookme, but for this example I have packed hookme.exe with a commercial protector. The packed file is now named **hookme.packed.exe**. Because the program is packed we do not have access to the code section making an internal patch much more difficult. So instead of focusing on modifying the API call internally we are going to develop a way to modify an API function's arguments externally. To achieve this goal externally we are going to write a program that edits the API function after it has been loaded into memory.

Before we begin developing our program we need to understand exactly how Windows loads executable files. Windows, when it first emerged, heralded the fact that it was a multitasking environment. This meant that more than one program could be running at a time. This also means that more than one program will be accessing the Windows API. Before Windows 2000 the operating system DLL's were all loaded into a single memory space, called high memory. All programs operated in the same memory space, software was loaded in low memory and accessed the high memory during API calls. This meant that all the programs running on Windows 98 and lower operated in a single memory space. If an application corrupted memory outside it's range it could lead to disastrous results for all the other applications. This also means that if an application were to somehow modify a system DLL in high memory it would affect every single running application that relied on that DLL. Windows 2000, however, changed how program memory was assigned and accessed. With the newer Windows operating systems each running program is assigned a specific self contained memory location. Each programs memory is then addressed in the same way it was in 98, essentially providing the program a memory space as if it was the only program running on the computer. Because of this every individual programs memory is re-addressed into high and low memory, so Windows loads a copy of each dll the program needs into that high memory address space for every program. This means that each program has its own copy of the DLL's it needs. Now if a program was to corrupt high memory it would only affect that specific program, and not any of the others. This ends up making Windows XP much more stable than 98 and below, and it also provides us a better playground. We have the ability to access and modify any addressed memory for our running program without fear of corrupting other programs, or the stability of the OS.

With the newer versions of Windows we have the ability to modify all the files in a programs address space without fear of affecting any other programs. This extends to all the Window's DLL files loaded as well. What we are going to do is develop a solution that will edit the Windows DLL in memory so that when a specific function is called the DLL will redirect to our own code. In our code we will edit the function's arguments and then return execution to the Windows API. All this will take place without modifying or alerting the packed program.

There are different ways to infiltrate a programs address space. We could write a loader that would run the executable and modify the API function.[7] We also have the option of injecting our code directly into the address space.[8] For this example we are going to use a

[7]Cracking With Loaders: Theory, General Approach, And A Framework: http://tutorials.accessroot.com
[8]Three Ways to Inject Your Code into Another Process: http://www.codeproject.com/threads/winspy.asp

dynamic of code injection, we will introduce our own DLL into the executable's address space. Our DLL will then modify the Windows DLL and redirect function calls back to itself. The DLL will then modify the API arguments and return execution back to the Windows DLL.

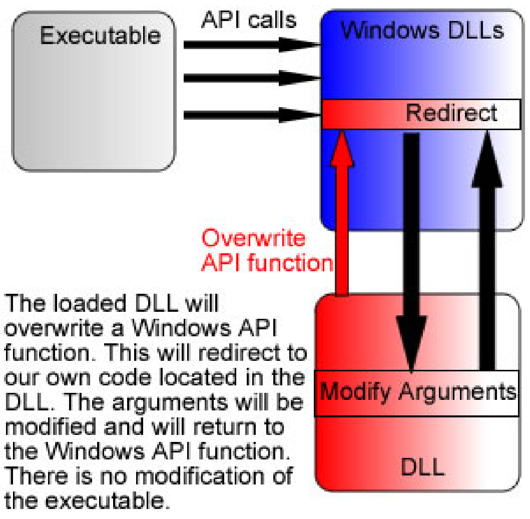

Executable
API calls
Windows DLLs
Redirect
Overwrite API function
The loaded DLL will overwrite a Windows API function. This will redirect to our own code located in the DLL. The arguments will be modified and will return to the Windows API function. There is no modification of the executable.
Modify Arguments
DLL

Figure 18 – Visual Representation of API Modification

To begin we will examine the steps we need to take to modify a dll in memory.

- Find the address location of the DLL
- Find the address of the function we want to redirect
- Find the address of our own DLL
- Find the address of the function in our DLL we will redirect to
- Redirect the code from the first function to our modified function

All of the above steps can be achieved by using functions from our notorious Windows API.

- The **LoadLibrary** function will map a specified executable module into the address space of the calling process and then return the base address of the module. If the module is already loaded then the function will return the base address location.
- The **GetProcAddress** function will return the address of a specified exported DLL function.
- **LoadLibrary** can also be used to find the base address of our own DLL
- **GetProcAddress** will return the address of our own exported function
- Having all the above information we can then redirect the function located at the Windows DLL address to the address of our own exported DLL function.

The DLL will be developed in MASM, however it will be using API calls that are universal to Windows, so it will be simple to port to C++ or another development language. The tools needed for this section are RadASM[9] and MASM32[10].

[9]RadASM Win32 assembly IDE: http://radasm.visualassembler.com/
[10]MASM32 version 8.2: http://www.masm32.com/

Install both RadASM and MASM32 to your C: directory. This will help eliminate the need to modify library and include paths in RadASM. If you have issues refer to the RadASM help file. It can be downloaded from the website.

The basics of writing a DLL in MASM are simple.[11] Since we are coding our DLL using the RadASM IDE (Interactive Development Environment), it will remove some of the work when coding our assembly programs.

1. Open RadASM and select File->New Project.
2. Choose **MASM** for the Assembler and check **DLL Project**. Enter in both the Project Name and what the Project file will be called. I named my project SpyFunc. Press Next to continue.
3. Do not choose a Template, just press Next.
4. For File Creation only choose **ASM** and **DEF**, and choose **BAK** for the Folder Creation. Press Next to continue.
5. In the next window press Finish to begin your project.

You will see in the right panel that you have two files: **SpyFunc.asm** and **SpyFunc.def**. **SpyFunc.asm** is our main assembly file and it will contain all the code for the DLL. **SpyFunc.def** is the definitions file for the DLL, in there we will define what functions we want to export from SpyFunc.dll.

The SpyFunc DLL will function as such:

During the initialization phase of the DLL (this is when it is loaded into memory) it will find the the address of both the Windows API function MessageBoxA, and to a function in the SpyFunc DLL I called **SpyFuncReplace**. The DLL will archive the original bytes of the MessageBoxA function and then overwrite the beginning of the function with a JMP to SpyFuncReplace. Now when MessageBoxA is called it will redirect to SpyFuncReplace.

When execution is redirected to SpyFuncReplace, the dll will function in almost the same way that our internally modified code functioned earlier. It will find the top of the stack, add a variable amount to that number to find the address of the "Text" argument. That argument will then be overwritten with the address of our own replacement text. The stack now contains our modified argument. The MessageBoxA function is then restored and we JMP to the beginning of that function. Execution continues as normal but the API function will now process our substitute text instead of the original. This is the end of our SpyFuncReplace function.

[11]How to build a DLL in MASM32: http://www.website.masmforum.com/tutorials/dlltute/masmdll.htm

Included below is the code for **SpyFunc.asm:**

```
;SpyFunc v1.0 by Gabri3l [ARTeam]
;Supplement to Weakness of Windows API Tutorial
;
;------------------------------------------------------------------------------------
-----
;When the DLL is loaded the specified function is overwritten by an infinite loop
;Can be used by calling LoadLibrary or defining the DLL in APP_INIT Registry Key
;************************************************************************************
*****
;**This is very beta code, it can screw up your system if you are not careful, so be
careful**
;**
**
;**             BTW: Thanks to Human and rukiddin for their help!
**
;************************************************************************************
*****
;
;Any questions or comments: Visit us at http://cracking.accessroot.com
;------------------------------------------------------------------------------------
-----


.586                                                 ;for 586 processor or better
.model flat, stdcall                                 ;32-bit memory and standard call

;Function Includes
;-------------------------------------
include    \masm32\include\windows.inc
include    \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
;--------------------------------------
SpyReplace  PROTO                                   ;Prototype for our replace function

.data                                               ;Defined Data
nDll         db "user32.dll",0                      ;DLL to Hook
nProc        db "MessageBoxA",0                     ;Function to Hook
nRedirect    db "Spyfunc.dll",0                     ;DLL to Redirect to
nFunc        db "SpyReplace",0                      ;Function to Redirect to
nSize        dd 069000h                             ;Size of the DLL
nIncrement   dd 0Ch                                 ;Amount to increment the stack to point
to modifiable argument
nReplaceWith db "The function has been modified",0;replacement text for modified function
argument
nTest1       db "This is a test"                    ;Known argument to test out our
modified function
nTest2       db "Test"                              ;Known argument to test out our
modified function


.data?                                              ;Undefined Data
nAddress     dd ?                                   ;Address of the DLL when loaded into
memory
nLoc         dd ?                                   ;Location of Function to Hook
nOldProt     dd ?                                   ;Old Protection of DLL
nRedirectAddress dd ?                               ;Address of the DLL to redirect to
nRedirectLoc     dd ?                               ;Location of Function to redirect to
nOldBytesOne     dd ?                               ;Old Bytes of overwritten function
nOldBytesTwo     dd ?                               ;Another buffer to hold overwritten
bytes
nTopOfStack      dd ?                               ;Hold the value at the top of the
stack.

.code                                               ;the beginning of the code section
LibMain proc h:DWORD, r:DWORD, u:DWORD              ;the dll entry point
```

```
INVOKE LoadLibrary, ADDR nDll             ;Load the DLL to be hooked
.if eax!=0
mov nAddress, eax                         ;Move Location of DLL into nAddress
INVOKE GetProcAddress,nAddress,ADDR nProc ;Get the Address of the function to be hooked
mov nLoc, eax                             ;Move Function location into nLoc
.if eax!=0
INVOKE VirtualProtect,nAddress,nSize,PAGE_EXECUTE_READWRITE,OFFSET nOldProt ;Grant
Read/Write access to DLL
INVOKE LoadLibrary, ADDR nRedirect        ;Load the DLL to be hooked
.if eax!=0
mov nRedirectAddress, eax                 ;Move Location of DLL into nRedirectAddress
INVOKE GetProcAddress,nRedirectAddress,ADDR nFunc ;Get the Address of the function to be
redirected to
.if eax!=0
mov nRedirectLoc, eax                     ;Move Function location into nRedirectLoc
MOV EAX, DWORD PTR DS:[nLoc]              ;Move address of function into EAX
PUSH ECX                                  ;Push ECX to keep Register integrity
mov ECX, DWORD PTR DS:[EAX]               ;Move Old bytes of the function into ECX
MOV nOldBytesOne, ECX                     ;Move Old bytes of the function into buffer
ADD EAX, 4                                ;Increment by 4 to read next bytes
mov ECX, DWORD PTR DS:[EAX]               ;Move Old bytes of the function into ECX
MOV nOldBytesTwo, ECX                     ;Move Old bytes of the function into ECX
SUB EAX, 4                                ;Subtract 4 to restore EAX
mov BYTE PTR DS:[EAX], 0E9h               ;Write the JMP command into the function
MOV ECX, nRedirectLoc                     ;Move address of Redirected function into ECX
ADD EAX, 5                                ;Increment EAX to move to end of operands
Sub ECX, EAX                              ;Subtract Calling Address from Address to be
called
SUB EAX, 4                                ;Increment EAX to point to BYTES directly after
JMP Command
mov DWORD PTR DS:[EAX], ECX               ;Write new function location after CALL command
POP ECX                                   ;Restore ECX
mov eax, 1                                ;Set EAX to 1 so the DLL will initialize
.endif
.endif
.endif
.endif
ret                                       ;return
LibMain Endp                                      ;end of the dll entry

SpyReplace proc                                   ;The function to replace the MessageBox
text with our own
PUSH ECX                                          ;Push ECX to preserve integrity
MOV nTopOfStack,ESP                               ;Push the address of the top of the
stack into nTopOfStack
MOV EAX, nIncrement                               ;Add 12 (0Ch) to EAX to move it to the
old argument of the stack
ADD EAX, nTopOfStack                              ;Add the value of the top of the stack
to EAX
MOV ECX, OFFSET nReplaceWith                      ;Move the address location of our
replacement text to ECX
MOV DWORD PTR DS:[EAX], ECX                       ;Move replacement text address into the
stack overwrite old arguement
MOV EAX, DWORD PTR DS:[nLoc]                      ;Move address of function into EAX
MOV ECX, nOldBytesOne                             ;Move the old function bytes into ECX
MOV Dword PTR DS:[EAX],ECX                        ;Restore the first bytes of the
modified function
ADD EAX,4                                         ;Increment ECX by 4 to move to the next
4 bytes
MOV ECX, nOldBytesTwo                             ;Move the next old function bytes into
ECX
MOV Dword PTR DS:[EAX],ECX                        ;Restore the final bytes of our
modified function
Sub EAX,4                                         ;Decrement ECX by 4 so it contains the
Address of the beginning of our restored function
POP ECX                                           ;Restore ECX
```

```
JMP EAX                                                ;Jump to the beginning of our restored
function
ret                                                    ;Return (not used)
SpyReplace endp                                        ;End of replacement function

End LibMain                                            ;end of the dll
```

I need to note that the code above is in a beta form. It does not account very well for any errors that may result from any of the API functions failing. For the current situation the code is sufficient to demonstrate how API calls can be modified. However, It may cause problems when you load a program in Olly that uses this DLL. The code will be improved and refined later in the tutorial when the objective is to intercept and modify API return values.

Our definition file needs to include two lines. We need to define the name of our dynamic library and we also need to declare any functions we want to export from the DLL. When a function is exported that means it is made available to any module in the address space that wants to call it. Exporting a function will allow us to find the address of that function by using the GetProcAddress API feature.

Below are the definitions to be included in **SpyFunc.def**:

```
LIBRARY    SpyFunc                                   ;The name of our library
EXPORTS    SpyReplace                                ;The name of the exported function
```

After all the code has been entered for both SpyFunc.asm and SpyFunc.def we can build our DLL.

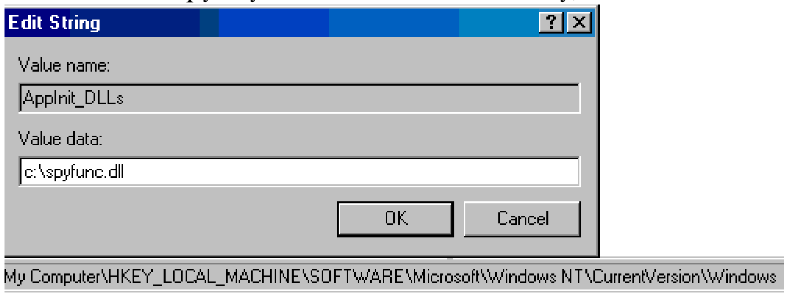
1. In RadASM choose Make->Build to compile the DLL
2. Output results will be displayed in a window at the bottom of the program. Your DLL will be located in the ...RadASM/MASM/Projects/Spyfunc/ folder

Now that we have a DLL we need to find a way to insert that DLL into a programs memory space. To do so we are going to use a Registry Key that Windows has provided for us called AppInit_DLLs.[12] DLLs that are specified in this key are loaded by each and every Windows executable. This makes it very easy for us to insert our DLL into any programs address space. This is not an elegant solution though, because the DLLs are loaded for every file it can cause problems for programs that we are not targeting. There are other ways to execute files when programs are loaded rather than the AppInit_DLLs key, such as a system wide message hook[13], and the Image File Execution Options key.[14]

Open RegEdit.exe and browse to:
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows
In the right hand pane you should see a value named AppInit_DLLs. If you do not see a value with that name Right-Click in the right hand pane and choose NEW->String Value.

[12]Working with the AppInit_DLLs registry value: http://support.microsoft.com/default.aspx?scid=kb;en-us;197571
[13]WepMetering: http://www.viksoe.dk/code/wepmetering.htm
[14]Execution redirection thru 'Image File Execution Options' key, 29A Zine, Issue #8, Article 17: http://vx.netlux.org/29a/main.html

Name the new value AppInit_DLLs.
Double-Click the value to bring up the edit box.
Now enter in the location of the spyfunc DLL. (because the value does not work well with spaces I found it easiest to copy my DLL to the C: directory)


Edit String

Value name:
AppInit_DLLs

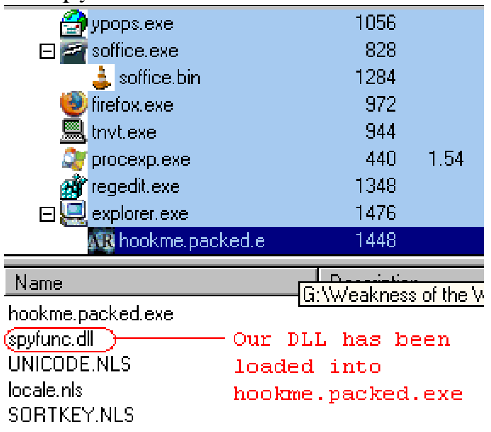
Value data:
c:\spyfunc.dll

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows

Figure 19 – Setting the Value Data for AppInit_DLL's

Once you press OK all files opened from now on will load SpyFunc.dll in addition to all the normal Windows DLLs. Press Okay to confirm the changes but leave RegEdit.exe open so we can restore AppInit_DLLs after testing.

Now that the DLL is built and is being inserted in all running processes we can begin testing it. Open HookMe.Packed.exe but do not press the MessageBox button. First we are going to verify that the DLL is being loaded. We can do this by using a program called Process Explorer.[15] Process Explorer functions almost as Windows task manager does except is much more advanced. It allows you to view extended data on all the processes you have running. It can return information including what DLLs that a program currently has open. Once you have downloaded and installed Process Explorer open it up. A list will be populated containing all the current processes. Scroll through the list and select our Hookme.Packed.exe. You can see in the lower pane all the loaded DLLs, look to the top of the list and we can find our spyfunc.dll:


ypops.exe	1056	
soffice.exe	828	
soffice.bin	1284	
firefox.exe	972	
tnvt.exe	944	
procexp.exe	440	1.54
regedit.exe	1348	
explorer.exe	1476	
hookme.packed.e	1448	

Name	Description
hookme.packed.exe	G:\Weakness of the W
spyfunc.dll	
UNICODE.NLS	
locale.nls	
SORTKEY.NLS	

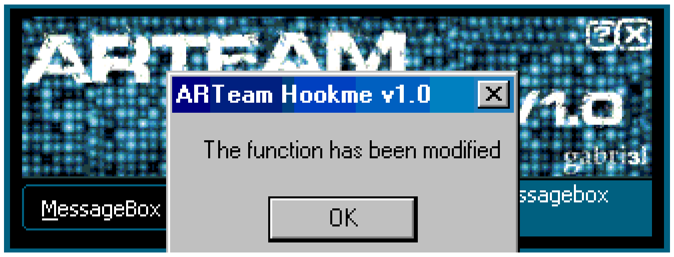
Figure 20 – Viewing Hookme.packed.exe in Process Explorer

We can see from the image that our DLL has been successfully loaded into the packed hookme executable. This means that it has also been initialized. When a DLL is loaded into address space the first thing that is executed is the initialization code. So just by being

[15]SysInternals Process Explorer: http://www.sysinternals.com/Utilities/ProcessExplorer.html

loaded into memory our DLL has executed the code we inserted into the initialization section. The MessageBoxA function has been edited so it will redirect back to the DLL. Now switch back to RegEdit.exe and remove the data for the AppInit_DLLs value to prevent spyfunc.dll from entering other address spaces. The current copy of spyfunc.dll will still remain in the Hookme.packed address space. All we have to do is execute the MessageBoxA function.

Press the MessageBox button on the hookme executable. The result is:



ARTeam Hookme v1.0

The function has been modified

OK

Figure 21 – Final Result of Using a DLL to Redirect API Call

The DLL has worked correctly. We developed a way to infiltrate the address space of an executable in an effort to modify the arguments of specific API calls. This method was more advanced and desirable than an internal modification because it functions on both a packed and unpacked executable. The external method enabled us to target programs that we may not have the time or ability to obtain the unpacked code.

We can apply this external method to some common program protection. For example, we can modify our DLL to hook IsDebuggerPresent. In the HookMe folder I included a packed executable that checks to see if it is open in a debugger, it is named **HookMe.IsDebuggerPresent.exe**.. If the executable is open inside of a debugger it will not run. Instead it will display a messagebox informing you of that fact.

Before we continue, there are already techniques to bypass the IsDebuggerPresent API function and Olly already has plugins that will do just that.[16] However, we are going to modify it ourselves as an example to the applications for external modification.
Let's examine the IsDebuggerPresent API call:

The **IsDebuggerPresent** function indicates whether the calling process is running under the context of a debugger.

BOOL **IsDebuggerPresent**(VOID)

Parameters
This function has no parameters.

Return Value
If the current process is running in the context of a debugger, the return value is nonzero.
If the current process is not running in the context of a debugger, the return value is zero.

[16]Hide Debugger Plugin: http://ollydbg.win32asmcommunity.net/stuph/

In the case of IsDebuggerPresent we do not have any API arguments to modify. What we can modify is the Return Value. When a program is being debugged IsDebuggerPresent will return a non-zero value. A program can then check to see if EAX is zero, if it is not zero then the executable knows that it is operating inside a debugger. What we can do is modify our DLL to redirect the IsDebuggerPresent API function to our own function location and change EAX to 0.

We can make some minor changes to our current SpyFunc DLL code to achieve this. All we have to do is add another function to the DLL. First add a new prototype to the beginning of the ASM file:

```
SpyReplace  PROTO                          ;Prototype for our replace function
SpyDebug    PROTO                          ;Prototype for our IsDebuggerPresent function
```

Now we add the SpyDebug Function:

```
SpyDebug proc                                      ;Beginning of the SpyDebug Function
       XOR EAX,EAX                                 ;Set EAX equal to 0
       ret                                         ;RETURN to regular code execution
SpyDebug endp                                      ;End of SpyDebug function
End LibMain                                        ;end of the dll
```

Now we can modify our .data section to point to kernel32.dll, IsDebuggerPresent, and our SpyDebug function:

```
.data                                              ;Defined Data
nDll         db "kernel32.dll",0                   ;DLL to Hook
nProc        db "IsDebuggerPresent",0              ;Function to Hook
nRedirect    db "Spyfunc.dll",0                    ;DLL to Redirect to
nFunc        db "SpyDebug",0                       ;Function to Redirect to
```

Finally we need to change the SpyFunc.def file to contain SpyDebug in it's export list:

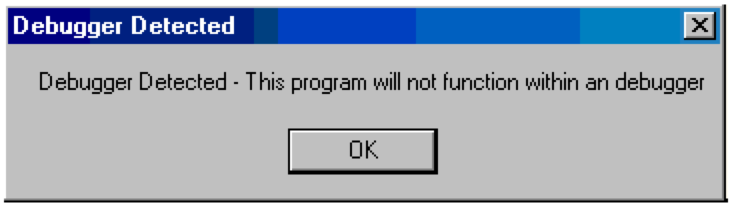
```
EXPORTS   SpyReplace
EXPORTS   SpyDebug
```

Now we can re-build our DLL.

1. In RadASM choose Make->Build to compile the DLL
2. Replace the old SpyFunc.dll in your C: drive with the newly created one.

Open Ollydbg but do not load the HookMe.IsDebuggerPresent executable. Instead disable any hide debugger plugins you may have. (*Disabling some debugger hiding plugins require Ollydbg to be reopened. Comply and restart Olly if that is required of you*)

Load the HookMe.IsDebuggerPresent executable in Ollydbg and run it once to determine the outcome without the spyfunc DLL loaded:


Debugger Detected

Debugger Detected - This program will not function within an debugger

Figure 22 - HookMe.IsDebuggerPresent Can Not be Run Within a Debugger

The executable uses the IsDebuggerPresent API function to determine whether or not you are operating within a debugger. We can now use our DLL to fool the program into thinking that it is operating outside of a debugger. Press ALT+F2 in Olly to unload the executable.

As we did before, we need to add SpyFunc.dll to the AppInit_DLLs value located at:
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows

We can now open HookMe.IsDebuggerPresent in Ollydbg. Since it is being reloaded it will import and initialize spyfunc.dll when it is running. With the Hookme program loaded Right-Click in Olly and choose Go-To->Expression. In the new box type IsDebuggerPresent:

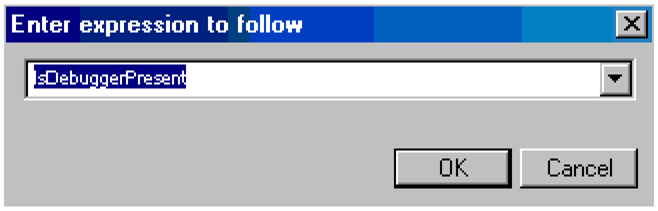
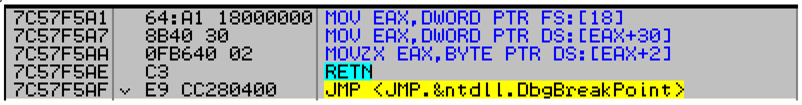


Figure 23 – Go-To IsDebuggerPresent

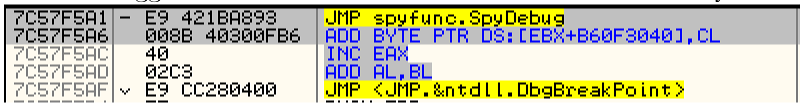
Press OK and we will be at the beginning of the IsDebuggerPresent function. As we can see it has not yet been modified:



```
7C57F5A1     64:A1 18000000  MOV EAX,DWORD PTR FS:[18]
7C57F5A7     8B40 30         MOV EAX,DWORD PTR DS:[EAX+30]
7C57F5AA     0FB640 02       MOVZX EAX,BYTE PTR DS:[EAX+2]
7C57F5AE     C3              RETN
7C57F5AF  v  E9 CC280400     JMP <JMP.&ntdll.DbgBreakPoint>
```

Figure 24 – Unmodified IsDebuggerPresent Function

So let's press RUN to execute Hookme and load SpyFunc.dll. The first thing you will notice is that the IsDebuggerPresent API function has been successfully modified:



```
7C57F5A1  -  E9 421BA893     JMP spyfunc.SpyDebug
7C57F5A6     008B 40300FB6   ADD BYTE PTR DS:[EBX+B60F3040],CL
7C57F5AC     40              INC EAX
7C57F5AD     02C3            ADD AL,BL
7C57F5AF  v  E9 CC280400     JMP <JMP.&ntdll.DbgBreakPoint>
```

Figure 25 – Modified IsDebuggerPresent Function

The second thing you will notice is that HookMe has successfully loaded without giving the debugger detected error! Our DLL has modified the IsDebuggerPresent API to always return a 0 so any program with SpyFunc.dll loaded into the address space will believe that it has not been loaded within a debugger.

Because the DLL we created operates in Ring3 it is still only taking advantage of the first trust level. It is a relatively simple attack and some protections have been developed that check for such modifications to the API functions rendering both the internal and external methods useless. If we were to build a program that would operate in Ring0 we could attack the second trust level by actually modifying how the Windows API handles specific calls, rather than redirecting calls in the Ring3 address space.

5. Conclusion:

In this first part we have covered many different methods to abuse the First Trust Level inherent in the Windows API. The knowledge and code has been given to you. With these tools to expand upon, you can quickly and intelligently adapt to whatever protection scheme you may come across.
In the second part of this series we will move into exploiting the Second Trust Level. Part 2 will examine how we can create a program to operate in Ring0. It will cover some of the more powerful protection methods, and how modifying the kernel will help to better explore these protections.

6. References:

1. "The Evolution of 32-Bit Windows Viruses" Windows 2000 Magazine Online Peter Szor, Eugene Kaspersky, July 1, 2000[17]
2. "Windows NT 2000 Native API Reference" Gary Nebbett
3. "API Hooking Revealed" Ivo Ivanov[18]
4. "Exploiting Software: How to Break Code" Greg Hoglund, Gary McGraw
5. "Hooking Windows API - Technics of hooking API functions on Windows" Holy_Father, CodeBreakers Journal Volume 1 No. 2[19]
6. "Writing DLL in Assembler for External Calling in Maple" Alec Mihailovs, Ph.D.[20]

7. 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.

8. 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

9. Disclaimer:

10. Verification:

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

[17]The Evolution of 32-Bit Windows Viruses: http://vx.netlux.org/lib/aek06.html
[18]API hooking revealed: http://www.codeproject.com/system/hooksys.asp
[19]Hooking Windows API: http://www.codebreakers-journal.com/viewarticle.php?id=40&layout=abstract
[20]Writing DLL in Assembler for External Calling in Maple:
http://webpages.shepherd.edu/amihailo/assembler1.html