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:

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".

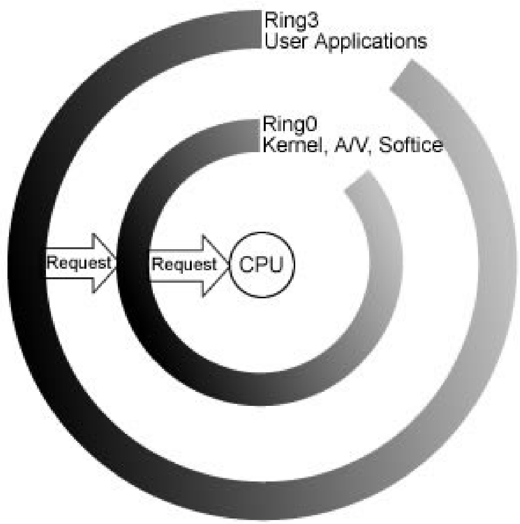

Figure 1 – The "Ring" structure of Operating Systems

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.

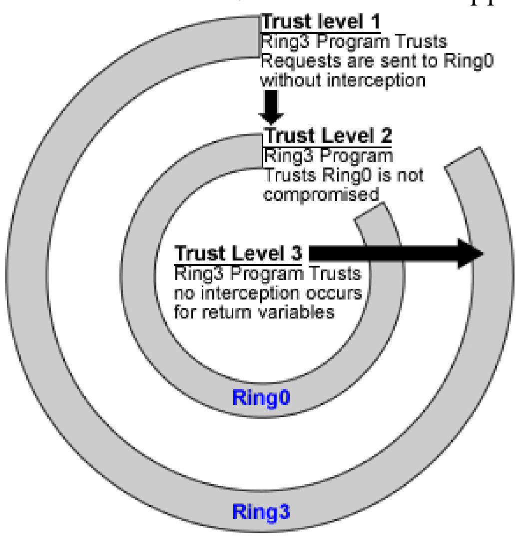

Figure 2- Graphical Representation of Windows Trust Model

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
  );
```

<u>Parameters</u>
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:

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:

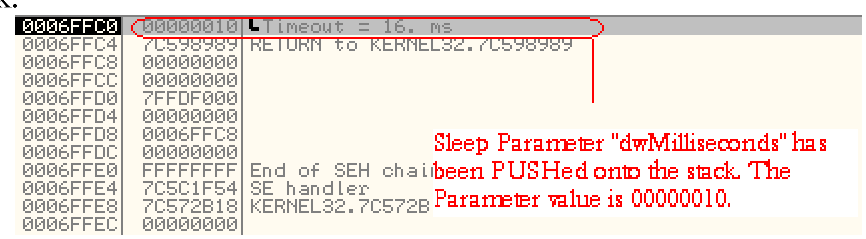

Figure 4 – The Stack just before the Sleep function is called

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:

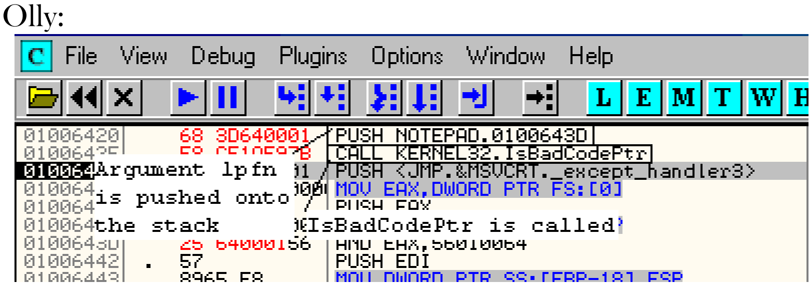

Figure 5 – IsBadCodePtr being called

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.

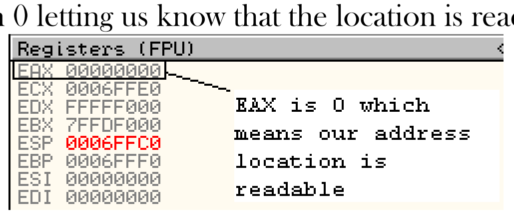

Figure 6 – The Registers after IsBadCodePtr is called

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