The Wayback Machine - https://web.archive.org/web/20030210154000/http://www.internals.com:80/articles/a…

API Spying Techniques for Windows 9x, NT and 2000

Yariv Kaplan

API spying utilities are among the most powerful tools for exploring the inner structure of applications and operating systems. Unfortunately, neither the SDK nor the DDK provide any documentation or examples demonstrating a way for implementing such a utility. This article will attempt to shed some light on this subject by presenting several techniques for hooking API calls issued by Windows applications.

There are several things you'll need to consider before sitting down to write your own API spying utility. First, you'll need to decide whether you want to spy upon a single application or set up a system-wide API interceptor. Each approach can be useful in different situations. For example, assume that you need to write an application that blocks the execution of specific processes according to rules set by an administrator. Obviously, you will need a way to monitor the execution of new processes and terminate the ones, which are marked as restricted. One way to accomplish that would be to establish a system-wide API interceptor that monitors calls made to the CreateProcess function (actually, to both the Ansi and Unicode versions of this function). Whenever an application issues a call to one of these functions in order to create a new process, your interceptor will gain control and perform whatever processing is necessary.

Other types of applications might require a simpler API interceptor capable of monitoring just one application at a time. A good example is BoundsChecker from NuMega - a tool capable of analyzing API calls in order to detect memory leaks and other bugs lurking inside Windows applications.

Whether you decide to employ a system-wide solution or opt for a simpler one, you'll still have to choose one among several API hooking techniques. The following sections will explore the various ways available for implementing an API interceptor for Windows, focusing on the advantages and disadvantages of each technique.

Proxy DLL

This is by far the easiest technique for hooking API calls under Windows. As an example for a use of this technique, consider an anti-virus application that scans incoming email messages for viruses. An obvious requirement for such an application is the ability to hook Winsock's I/O functions in order to analyze data transfers between email clients and remote mail servers.

This can be easily accomplished by creating a proxy DLL, which contains a stub for each of the functions exported from the Winsock library. If the proxy DLL is named exactly like the original Winsock library (i.e. wsock32.dll) and placed in the same directory where the target email application resides, then the interception occurs automatically. When the target application attempts to load the original Winsock library, the proxy DLL is loaded instead. All the calls made to Winsock's functions are routed to the exported stubs in the proxy. After performing the necessary processing, the proxy DLL simply routes the calls to Winsock and returns control back to its caller.

Lest you be afraid that the above method of API interception is too simple, there is a catch. Although simple to implement, this technique has one major drawback - hooking a single function located in a DLL that exports 200 functions would require creating a stub for each of these functions in your proxy DLL. This could be rather tedious and also impossible at times, when some of the functions are not fully documented.

If you wish to see a working example of this technique, consult Matt Pietrek's WinInet utility, which was published on the Dec 94 issue of MSJ.

Patch those calls

When thinking about ways for intercepting an API call, there are two locations where you can intervene - either at the source of the call (the application code) or at the destination (the target function). This technique relies on the first option.

Trend Micro, Inc.
EX1018-1

For each API function that you wish to intercept, you patch all the locations in the target application where calls to this function are issued. The modification can be done either on disk (to the executable file itself) or in memory (after the executable is already loaded). The tough part is pin pointing the exact locations where patching is necessary. In order to accomplish that, you'll need to implement a disassembler capable of analyzing assembly instructions. Obviously, writing a disassembler is far from being a trivial task, making this API interception technique one of the least popular among the group.

IAT Patching

If you have ever looked into API hooking before, then you probably heard about Import Address Table patching. The numerous advantages of this technique make it the most elegant and common way of hooking API functions under Windows.

The foundation of this technique relies on the fact that 32-bit Windows executable files and DLLs are built upon the Portable Executable (PE) file format. Files based on this specification are composed of several logical chunks known as sections. Each section contains a specific type of content. For example, the .text section holds the compiled code of the application while the .rsrc section serves as a repository for resources such as dialog boxes, bitmaps and toolbars.

Among all of the sections present in a Windows executable file, the .idata section is particularly useful for those who wish to implement an API interceptor. A special table located in this section (known as the Import Address Table) holds file-relative offsets to the names of imported functions referenced by the executable's code. Whenever Windows loads an executable into memory, it patches these offsets with the correct addresses of the imported functions.

Why does Windows go into the trouble of patching the IAT in the first place? Well, as you probably know, Windows executable files and DLLs are often relocated (due to collisions) after they are loaded into memory. This makes it impossible to set in advance the target addresses of calls made to imported functions in the executable code. In order to ensure that these calls successfully reach their destination, it would have been necessary for Windows to locate and patch every single call made to an imported function after an executable image is loaded into memory. Obviously, such a large amount of processing during initializing of new processes and DLLs would have slowed down the system, giving the user the notion as if Windows is a slow and unresponsive operating system ;-).

Fortunately, the designers of Windows were quite resourceful when they addressed this issue. In the current implementation of Windows executables and DLLs, calls made to imported functions are routed through the IAT using an indirect JMP instruction. The fact that imported function calls are "drained" through one location saves Windows the trouble of traversing the executable image in memory, looking for call instructions that are destined for patching.

What all of this got to do with API spying? Well, it seems that Windows IAT redirection mechanism offers a perfect way for intercepting API calls. By overwriting a specific IAT entry with the address of a logging routine, an API interceptor can gain control before the original function gets a chance to be executed by the processor.

Obviously, there are other issues involved in the implementation of this technique, such as the requirement for the logging code to be executed in the memory context of the intercepted application. These issues are discussed in the following resources:

- Matt Pietrek's excellent book: "Windows 95 System Programming Secrets" contains the source code of an API interception utility called APISpy32. This utility was initially published as part of an article written by Matt for the Dec 1995 issue of MSJ. A newer version of APISpy32 is available and can be ordered through Matt's web site at http://www.wheaty.net.
- Spying on COM Objects by Dmitri Leman, WDJ, July 1999.

Patch the API

This is my favorite technique for hooking API functions. It has the inherent advantage of being able to trace API calls issued from different parts of an application while requiring a modification only to a single location - the API function itself.

There are several approaches that can be used here. One option is to replace the first byte of target API with a breakpoint instruction (Int 3). Any call issued to that function would generate an exception, which would be

Trend Micro, Inc.
EX1018-2

reported to your API interceptor in case it serves as a debugger of the target process. Unfortunately, there are several problems with this approach. First, the poor performance of Windows exception handling mechanism would considerably slow down the system. A second problem is related to the implementation of Windows debugging API. As soon as a debugger shuts down, it terminates all the applications that were under its control. Obviously, such a behavior is completely unacceptable in case you're implementing a system-wide interceptor, which must be able to terminate itself before its target applications cease executing.

Another possibility is to patch the target function with one of the control-transfer instructions of the CPU (i.e. either a CALL or a JMP). Once again, there are several problems with this approach. First, it is possible that the patching would overrun the end of the intercepted function. This can occur in cases where the target API is shorter than 5 bytes (CALL and JMP are each 5 bytes long). Another issue is the need to constantly switch between the patched and "unpatched" versions of the intercepted function. This means that once your logging routine receives control from the CPU, it must restore the intercepted function to its previous unhooked state. This is required to allow the API interceptor to route the call to the original function without generating an infinite loop of calls back to the logging routine. Note that during the time the CPU is executing the original function, other calls to that function might be issued from different parts of the system. Since the function is in the unhooked state at that stage, the API interceptor will miss those calls. A more sophisticated API interceptor might utilize a better technique in order to overcome this limitation. Take a look at the following figure to get a better idea how this could be accomplished.

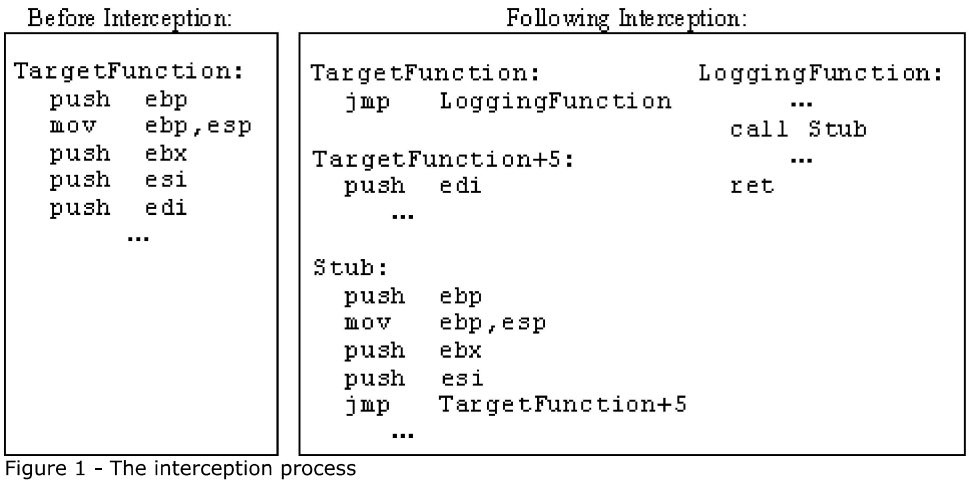


Figure 1 - The interception process

In this case, the API interceptor places a JMP instruction at the beginning of the target function, but not before saving the first 5 bytes of the function to a pre-allocated buffer in memory (a stub). The exact number of bytes to be copied to the stub may change depending on the instructions present at the head of the function. It cases where 5 bytes do not fall within instruction boundary, it is necessary to copy additional bytes until there is enough space for the JMP instruction to be inserted. Note that relative control-transfer instructions (i.e. JMPs and CALLs) need to be modified during copying to ensure that they transfer control to the right location in memory when executed from the stub. Obviously, performing such an analysis of assembly instructions requires the assistance of a disassembler, which, as I have mentioned before, is not very easy to implement.

If you are not intimidated by the complexity of this technique and wish to use it with your own applications, you might want to have a look at the source code of Detours - an API interception library, which was developed by one of the members in Microsoft's research department.

Breaking address space barriers

By now, you should have a pretty good idea on how to implement an API interceptor capable of redirecting API calls to your own logging code. However, one problem remains unsolved - how do you ensure that the logging code is executed in the right address space? Before we can answer that question, we need to better understand the internal architecture of Windows memory management unit.

As you probably know, each 32-bit Windows application gets a unique address space to toy with. During a task switch, Windows updates its page tables to reflect the new process's linear to physical memory mapping (also known as the process memory context). As a result, the page table entries that correspond to the private memory area of the process are modified to point to different physical addresses while those that correspond to shared memory regions remain untouched.

Trend Micro, Inc.
EX1018-3

Under Windows 9x, the 4GB linear address space is divided into several distinct memory regions, each with its own predefined purpose. MS-DOS and the first portion of the 16-bit global heap occupy the lowest 4MB. The next region spans from 4MB to 2GB and it is where Windows 9x loads each process's code, data and DLLs. Since each process physical addresses are unique, Windows 9x ensures that when a specific process is active, the page table entries corresponding to the 4MB-2GB region are mapped to this process's physical memory pages. The idea is for all processes to share the same linear addresses but not the same physical addresses (which is obviously impossible). Think of it as if the 4MB-2GB linear memory region is a window into the physical address space. This window slides each time a process is scheduled for execution to provide a view of the unique physical memory locations occupied by that process.

So what's the real benefit of this mechanism anyway? Well, since all processes use the same linear addresses, Windows can load them into different physical locations without performing any code fix-ups. This means that the memory representation of a process can be (almost) an exact copy of its on-disk image.

Continuing our exploration of the Windows 9x address space reveals that the region spanning from 2GB to 3GB is reserved for the upper portion of the 16-bit global heap. Also in this region are the memory-mapped files and Windows system DLLs (such as USER32, GDI32 and KERNEL32), which are shared among all running processes. This region is extremely useful for API interceptors, since it is visible in all the active address spaces. In fact, APISpy32 loads its spying DLL (APISpy9x.dll) into this area, thus ensuring that its logging code is accessible from any process issuing a call to an intercepted function.

Under Windows NT/2000, the story is a bit different. There is no documented way of loading a DLL into an area shared by all processes, thus the only way to ensure that the logging code is accessible by the target process is to inject the spying DLL into its address space. One of the ways to accomplish this is by adding the name of the DLL to the following registry key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_Dlls

This key causes Windows to load your DLL into every address space in the system. Unfortunately, this technique can only be used to inject a DLL into processes that link with user32.dll, meaning that console applications, which do not usually link with this DLL, are not included. Other methods for injecting a DLL into a process's address space are thoroughly described in the article "Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" by Jeffrey Richter, which was published on the May 1994 issue of MSJ. Additional resources are available below:

- Knowledge base articles: Q197571, Q125680, Q125691.

Injecting at the right moment

Knowing how to inject a piece of code into another process's address space is one thing, but timing is also a crucial factor when implementing an API interceptor. Inject at the wrong moment and your interceptor might miss calls issued by the target application. This problem really comes to light when implementing a system-wide interceptor. In that case, the interceptor needs to inject its spying DLL into a process's address space immediately after that process is executed, but right before it issues calls to intercepted functions. The best way to accomplish this is by monitoring calls to the CreateProcess function. When such a call is detected, the interceptor's logging routine passes control to the original CreateProcess function with a modified dwCreationFlags parameter, which contains the CREATE_SUSPENDED value. This ensures that the target process is started, but placed in a suspended state. The interceptor can then inject its spying DLL into the target process's address space and activate it using the ResumeThread API function.

Other ways for detecting the execution of processes under Windows are presented in the following section. Unfortunately, due to their asynchronous nature, they are less suited for detecting the appropriate time for injecting a spying DLL into another process's address space.

Detecting Process Execution

As you saw earlier, it is often necessary for an application to be able to detect the execution of new processes. One possibility, which was previously mentioned, is to hook the CreateProcess function and monitor calls made to that function from different parts of the system. However, implementing a system-wide API interceptor for the sole purpose of hooking a single function often does not justify the effort.

Fortunately, there is a simpler way to accomplish that under Windows 95 (OSR 2 and above), 98 and NT/2000, without requiring the support of a full-fledged system-wide API interceptor. Under Windows 9x, it is

Trend Micro, Inc.
EX1018-4

possible for a virtual device driver to respond to the CREATE_PROCESS message sent by VWIN32 whenever a new process is executed. Windows NT/2000 offers similar functionality through the use of the undocumented PsSetCreateProcessNotifyRoutine function exported from NTOSKRNL. This function allows a device driver to register a callback function that receives notifications from the operating system whenever a new process is started. If you are well versed in device driver development, you should have no trouble deciphering these interfaces yourself using the following examples as your guide:

- The Nerditorium column by Jim Finnegan, which was published on the January 1999 issue of MSJ, presented a utility that detects execution of processes under Windows NT/2000.
- The ProcSpy32 utility available on my web site detects execution of processes under Windows 9x using a combination of a device driver and an OCX component.

Winsock Hooking

Many of the programmers that look into API hooking techniques are seeking a way for monitoring network activity performed by Winsock applications. Such functionality is often required by anti-virus utilities, personal firewalls and Internet content blocking applications (e.g. CyberPatrol).

If you require such functionality in your application, you need not write a system-wide API interceptor, but rather use a mechanism, which was introduced along with Winsock 2. This mechanism is thoroughly described in the article Unraveling the Mysteries of Writing a Winsock 2 Layered Service Provider, which appeared in the May 99 issue of MSJ.

An alternative technique for monitoring network activity under Windows relies on the way NDIS drivers are layered on top of each other. By writing an intermediate driver (or alternatively by hooking NDIS interfaces), it is possible to monitor not only TCP/IP communication, but also any other data transferred through the network adapter. Take a look at the following resources for more information about these techniques:

- VToolsD available from NuMega technologies comes bundled with a sample driver (HookTDI), which demonstrates a way for monitoring network activity under Windows 9x.
- IMSAMP is a sample NDIS intermediate driver, available on Microsoft's web site.

DDE and BHO (a.k.a. Browser Helper Object)

In cases where Winsock does not provide sufficient information about the underlying network activity, a programmer can use two additional services, which are tailored specifically for the purpose of monitoring Internet browsers such as Netscape Navigator and Internet Explorer. Through these interfaces it is possible not only to monitor the data transferred through the network, but also to track the browser window, which initiated the network transaction.

Detailed information about these techniques can be found at the following locations:

- Controlling Internet Explorer 4.0 with Browser Helper Objects by Scott Roberts.
- Customizing Microsoft Internet Explorer 5.0 by Dino Esposito.
- Browser Helper Objects: The Browser The Way You Want It by Dino Esposito.
- WebSpy - A visual basic utility that uses DDE to monitor URLs entered into web browsers.
- Netscape DDE documentation
- Knowledge base articles: Q229970, Q167821, Q167826, Q191508, Q160957, Q160976.

GDI Hooking

I often get asked about ways for monitoring graphics operations under Windows. Up until now, there was no documented way to monitor these calls unless you were willing to create a system-wide API interceptor that hooks every GDI function in existence. Obviously, this is far from being an easy solution. Fortunately, the new accessibility API available under Windows 9x introduces a mechanism that allows applications to monitor graphics operations before they reach the video driver. Windows 2000 offers similar functionality through a slightly different interface. Detailed information about these interfaces can be found in the following resources:

- Documentation of the SetDDIHook function available as part of the accessibility API for Windows 9x.
- Knowledge base article Q229664.

Trend Micro, Inc.
EX1018-5

File System Monitoring

For information about file system monitoring techniques, check out the following resources:

- Inside the Windows 95 File System by Stan Mitchell.
- Windows NT File System Internals by Rajeev Nagar.
- FileMon - A file system monitoring utility for Windows 9x, NT and 2000.
- E4M - An on-the-fly disk encryption utility that uses a file system filter driver to monitor disk activity.
- "Examining the Windows 95 Layered File System," by Mark Russinovich and Bryce Cogswell, Dr. Dobb's Journal, December 1995.
- "Examining The Windows NT File System," by Mark Russinovich and Bryce Cogswell, Dr. Dobb's Journal, February 1997.

Interrupt Hooking

In the old days of DOS, interrupt hooking was widely used by TSR (Terminate and Stay Resident) utilities and other applications to extend the operating system functionality and monitor its behavior. Under Windows, interrupts still play a major role, and are mainly used as a portal that connects user-mode (a.k.a. ring 3) code to the operating system's kernel. If you wish to hook interrupts under Windows, you should have a look at the following resources:

- Undocumented Windows Nt by Prasad Dabak, Sandeep Phadke and Milind Borate.
- Monitoring NT Debug Services by Jose Flores, Windows Developer Journal, February 2000.
- NTSpy - A utility that monitors NT's system calls by hooking interrupt 2Eh.

COM Hooking

API spying applications are great for monitoring Windows APIs, but their lack of support for COM interfaces makes them useless when trying to monitor OCXs and other OLE components. Fortunately, there is a way for monitoring COM interfaces under Windows. This technique was presented in the article Spying on COM Objects by Dmitri Leman, which was published on the July 1999 issue of WDJ.

16-bit API Interception

Interception of 16-bit code is not common anymore, but some programmers still require such capabilities in their applications. If you are unfortunate enough to be still working on 16-bit code, you might want to have a look at the article: "Hook and Monitor Any 16-bit Windows Function With Our ProcHook DLL" by James Finnegan, which was published on the January 1994 issue of MSJ.

NT System Calls

If you have ever examined ntdll.dll with QuickView, you might have noticed that it exports a set of functions that begin with the Nt prefix. These functions are actually small stubs of code that pass control to the Windows NT kernel (NTOSKRNL) using interrupt 2E. Many of the functions exported from kernel32.dll are nothing more than control transfer routines to the stubs located in ntdll. For example, when a Windows application issues a call to CreateFile located in kernel32.dll, the call is redirected to NtCreateFile, which passes it on to NT's kernel for further processing. The special design of this mechanism allows a device driver to hook these interfaces, thus providing a way for monitoring activities performed by Windows NT/2000 applications. A thorough description of this mechanism is presented in the following resources:

- Undocumented Windows Nt by Prasad Dabak, Sandeep Phadke and Milind Borate.
- Windows NT/2000 Native API Reference by Gary Nebbet.
- Regmon - A utility that monitors access to the registry by using system call hooking techniques.
- NTSpy - A utility that monitors NT's system calls by hooking interrupt 2Eh.
- Tracing NT Kernel-Mode Calls by Dmitri Leman, WDJ, April 2000.

Resources

- Undocumented Windows Nt by Prasad Dabak, Sandeep Phadke and Milind Borate.
- Windows NT/2000 Native API Reference by Gary Nebbet.
- Advanced Windows (3rd Ed) by Jeffrey Richter.

Trend Micro, Inc.
EX1018-6

- Inside the Windows 95 File System by Stan Mitchell.
- Windows NT File System Internals by Rajeev Nagar.
- Windows NT Device Driver Development by Peter Viscarola and Anthony Mason.
- Developing Windows NT Device Drivers by Edward N. Dekker and Joseph M. Newcomer.
- Writing Windows Wdm Device Drivers by Chris Cant.
- Programming the Microsoft Windows Driver Model by Walter Oney and Forrest Foltz.
- Windows Undocumented File Formats by Pete Davis and Mike Wallace.
- "Windows 95 System Programming Secrets" by Matt Pietrek.



Copyright © 1999, 2000 Yariv Kaplan
yariv@internals.com

Trend Micro, Inc.
EX1018-7