Taking control, Functions to DLL injection

Author: Dr Craig S Wright GSE GSM LLM MStat

Abstract / Lead

This article is going to follow from previous articles as well as going into some of the fundamentals that you will need in order to understand the code exploitation process. In this article we look at one of the primary infection steps used to compromise a Windows host, DLL injection.

DLL injection is one of the most common methods used by malware such as a rootkit to load it into the host's privileged processes. Once injected, code can be inserted into functions being transmitted between the compromised code and a library function. This step is frequently followed with API hooking where the malicious code is used to vary the library function calls and returns.

This article is part of a monthly series designed to take the reader from a novice to being able to create and deploy their own shellcode and exploits.

Electronic copy available at: https://ssrn.com/abstract=3153492

Introduction

In previous articles, we have covered a number of topics to do with the creation of shellcode and assembly language. We continue with an introduction of one of the primary exploitation processes used against a Windows system. In subsequent articles this will be expanded into the creation of standalone exploit kits and in the deployment of a rootkit.

In this article we look at one of the primary infection steps used to compromise a Windows host, DLL injection. This process is used by attackers and is also incorporated into automated frameworks (including Metasploit) as a part of the testing and exploitation process. DLL injection is one of the more common methods used by malware such as a rootkit to load it into the host's privileged processes. Once injected, code can be inserted into functions being transmitted between the compromised code and a library function. This step is frequently followed with API hooking where the malicious code is used to vary the library function calls and returns.

In order to do this, we also need to take a step back and explain the system and the tools we will use in more detail. To achieve this, we will start with describing the various components that are used and to providing an introduction to the Python programming language. This will also extend into a simple method to analyse shellcode using GCC such that we can come to understand what the shellcode others have created is designed to do. This is a useful skill when reversing malware as well as a good way to learn from the existing code base and even to leverage some of the various tools that are freely available already.

What is a DLL?

"*A DLL is a Dynamically Linked Library of executable code*" (Shewmaker, 2006). Code libraries are important as they allow developers to reuse common functions. It is firstly inefficient and not economical to rewrite of the same section of code over and over. More importantly, when the same code is replicated in many blocks it becomes more difficult to patch or update software. In addition, standardized libraries allow developers to reuse set functions rather than having to recode them and hence reinvent the wheel each time they develop a program. It is important to note that the best programmers make mistakes. Whenever they have to recode the same material the chances of an error increase.

A DLL can reference a common function allowing the programmer to just learn the call needed rather than having to rewrite create their own. In this, the code would reference the external function using the DLL which is loaded into memory for use by the program. In this article we will be discussing DLLs are loaded into a running memory process. This is a feature of Windows and not a bug. That said, many features also lead to exploitations.

DLL Injection

First, malicious code such as a rootkit starts by seeking to replace itself in the address space of another process. There are several methods available to inject code of one of the simplest techniques involves injecting a dynamically linked library (DLL) file. This file will overwrite the address space of the process that the malicious code is attempting to subvert. There are valid uses for code injection both in standard programming as well as many security tools. PWDUMP2 by Todd Sabin is an example of a security program that uses DLL injection to run a thread with increased security privileges so that it can decrypt credential files on a Windows host. On top of that, programs such as games use DLL injection to access privileged hardware functions. So although malicious code is known to make use

Electronic copy available at: https://ssrn.com/abstract=3153492

of this process other code will also do this for valid reasons. This of course makes it difficult to restrict access to these types of functions and also simplifies the process of injecting code.

DLL injection is used in user level programs for reading files, writing data to disk or a database and to access network streams in amongst many other uses. As such, it is an activity that can be used for both valid and malicious ends and it is the intent that the code is created for that makes it malicious and not the fact that it uses injection to achieve its ends. There are several common DLL injection techniques. In this article we will be looking at the SetWindowsHookEx method initially and then follow up with the combination of using CreateRemoteThread and LoadLibrary.

When used maliciously, DLL injection allows the attacker or attacking process to force a module into memory. This is loaded through a program that would not normally request that function be loaded. One of the frequently deployed uses of shell code in Windows is to inject remote control software such as NetCat or VNC. This added power that is attributed to malicious shell code when used against the Windows system is balanced against the fact that it is generally more difficult to determine the location of a desired function in Windows than it is in Linux (due to the fact that system calls and functions are generally static between operating system versions in Linux whereas these are changing with nearly every patch and update in Microsoft software). For this reason, it can be more difficult to create reliable shellcode in Windows than it is to do so in Linux.

The Windows API serves as an abstraction layer sitting between user mode and kernel mode on the operating system (that is it handles requests between Ring 3 and Ring 0). For this reason, it is necessary to interface with the Windows API whenever any system-level interaction is required. Windows contains a simple resolution process linked to the API that accounts for the changing function address locations.

In this way, Windows can restrict direct access to sockets and network ports unlike Linux that allows direct system calls. As such, to open up a TCP network port in Windows and place it in to listen mode it is necessary for the developer to code an application that communicates through the API using a function such as WSAsocket().

RemoteDLL

One of the best Windows tools for experimenting with DLL injection is RemoteDLL (http://www.novell.com/coolsolutions/tools/17354.html). The still has been around for more than six years now but still works well.

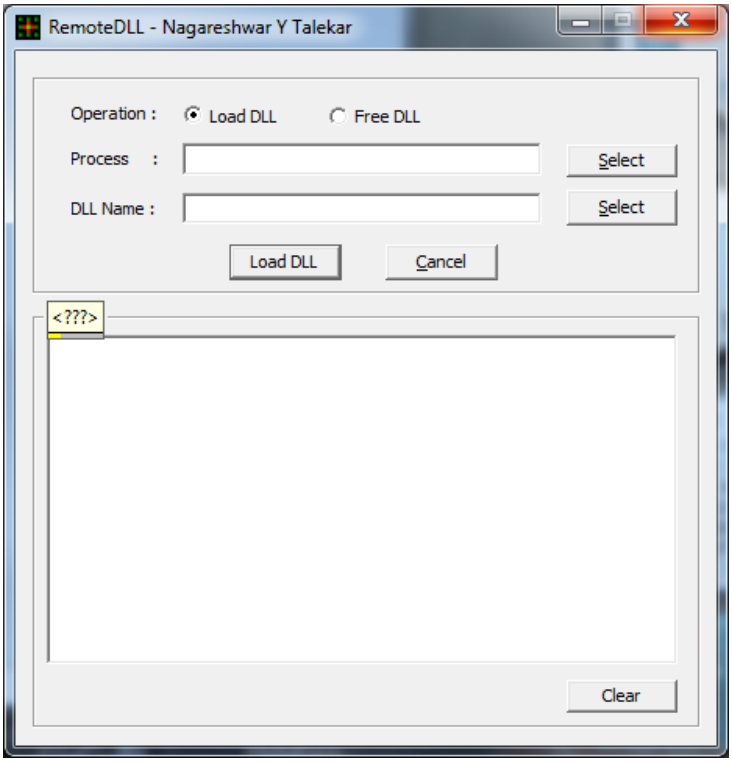

Figure 1: Starting RemoteDLL.

On start-up (Figure 1), this tool allows us to select a running process. This is done by clicking the "Select" tab on the right of the process box. Once you're clicked this button, a "Process List" will be displayed containing a list of the Process IDs (Figure 2) running on the system and the process name.

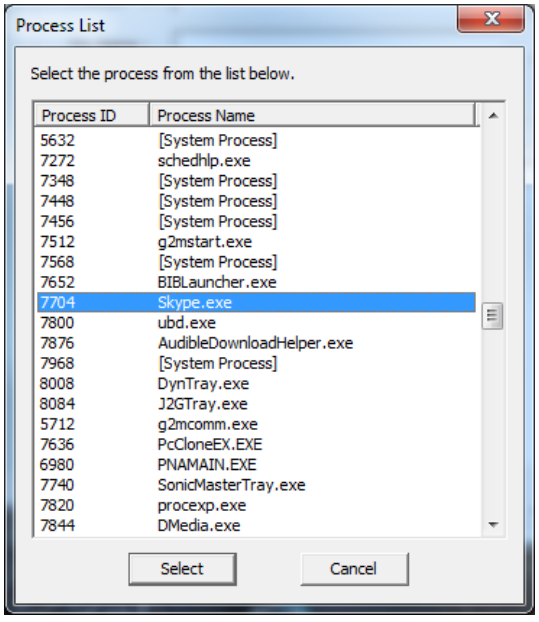

Figure 2: A Process List displayed in RemoteDLL.

Here we see a number of processes that do not have a descriptive process name. They are listed as **[System Process]** next to the Process ID. We can match these using a tool such as Process Explorer (Figure 3) from SysInternals (http://technet.microsoft.com/en-us/sysinternals/bb896653). Here as an

example we can match Process ID 7456 as displayed in RemoteDLL (Figure 2) with the details of the process in Process Explorer. This allows us to see that the system process is "sidebar.exe"

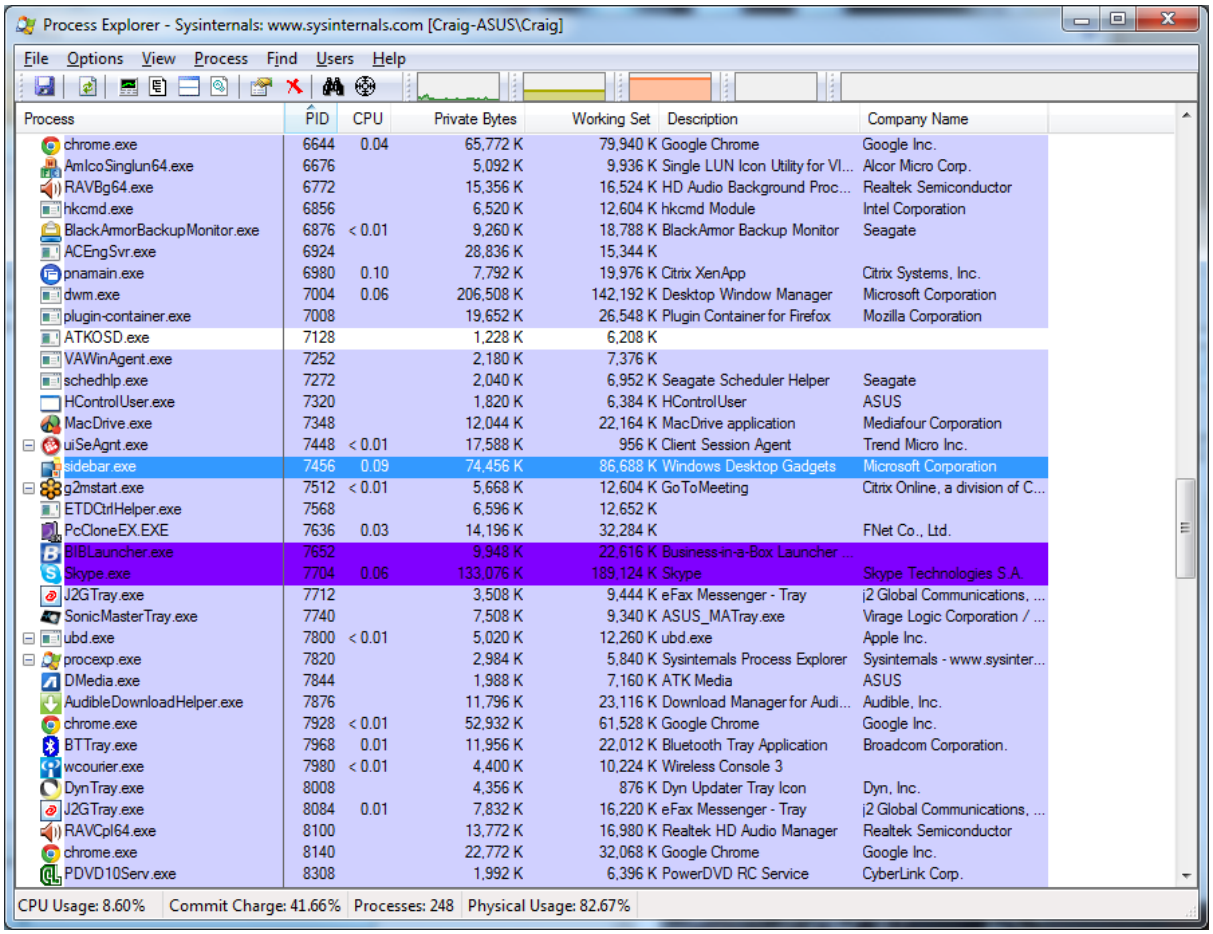

Figure 3: Process Explorer.

By right clicking on the process in Process Explorer we can select properties and bring up detailed information about the process (Figure 4).

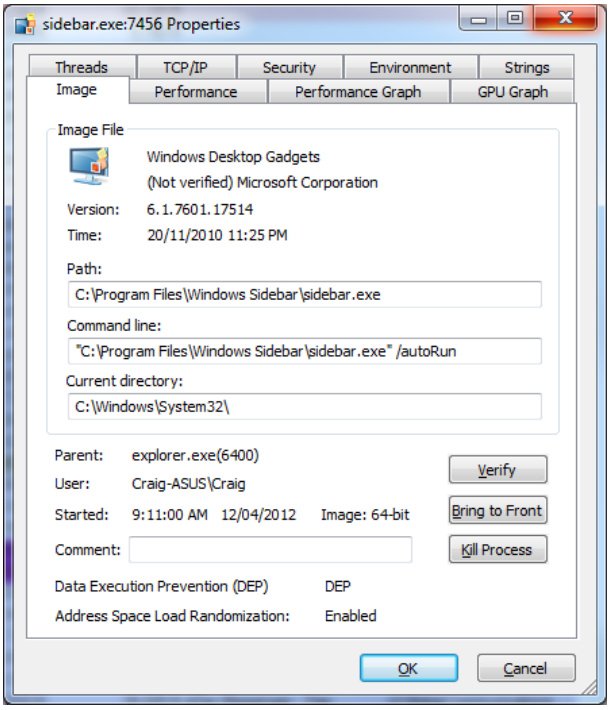

Figure 4: The details and information on "sidebar.exe" in Process Explorer.

When investigating processes that we might want to hook into on a system that we control this tool provides us with a lot of information about the security controls that we may need to overcome. In this example we see that Data Execution Prevention (DEP) and Address Space Load Randomisation are both enabled. We also see that the parent process with PID 6400 is explorer.exe.

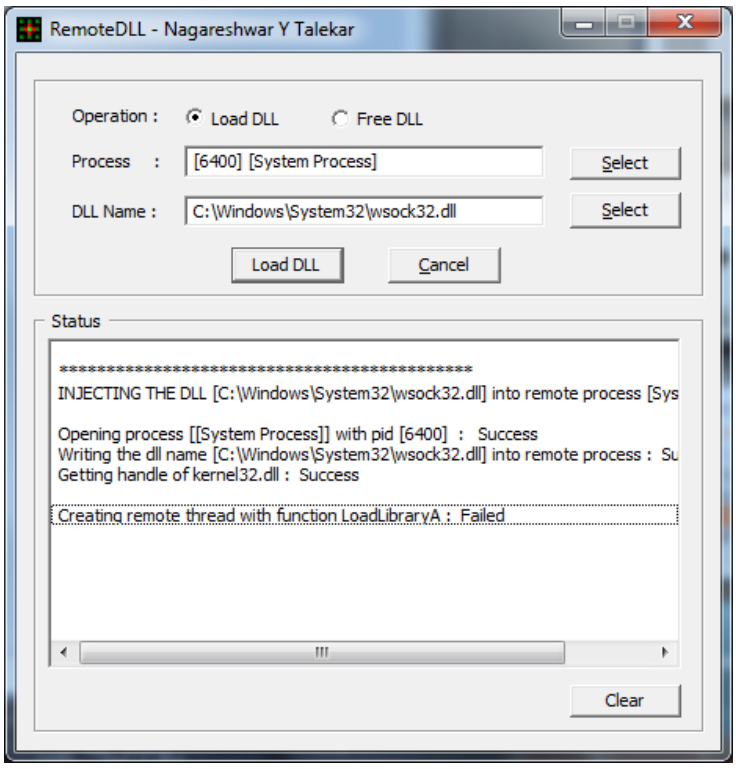

Figure 5: Loading a DLL in RemoteDLL.

Once we have selected the process that we want to inject our DLL into (in this case PID 6400 all as we saw explorer.exe) we have to select the DLL to inject into it. Once we have done this we click on the "Load DLL" button (Figure 5) and the DLL will be injected if we have everything set up correctly. Explorer.exe is a good test program as it automatically respawns if it crashes.

There many other reasons to want to be able to inject or free DLLs from processes. Using the "Free DLL" option allows you to remove DLLs from functions loaded with in memory. An example where this is necessary would be testing the winlogon.exe process. Here the underlying DLL may not be replaced or delete it from the disk but can be freed from the secured process and reinjected. This process is also useful when testing patches.

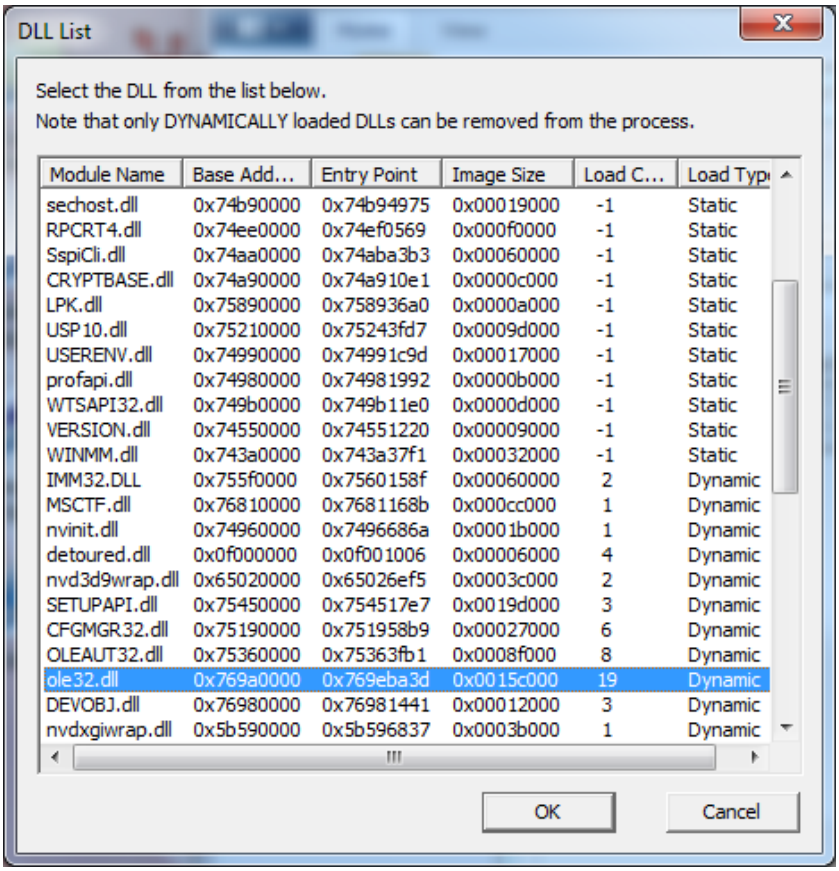

Figure 6: Removing a DLL in RemoteDLL.

We can also use RemoteDLL to remove an injected DLL. When we have selected the "Free DLL" operation and the process that we want to remove an injected DLL from clicking the "Select" button next to "DLL Name" will bring up a list of DLLs loaded into the process (Figure 6). Only dynamically loaded DLLs can be removed (that is we cannot remove static libraries).

In this list we also have a set of base addresses, entry points and the image size of the injected library. Using this information we can also carve individual libraries from a memory dump. This type of procedure is done in incident response and forensics analysis work. One reason to do this would be comparing the library loaded into memory with that which is stored on the disk. This type of process is useful when searching for malicious code on a potentially compromised system. It also allows us to extract and analyse more advanced malicious code that loads into memory and may not be resident on disk in an unencrypted format. This is the case as when code loads itself into memory it generally unpacks itself allowing us to reverse engineer the code more easily.

Windows Hooks

It is noted that "*Windows hooks can be considered one of the most powerful features of Windows*" (Iczelion, 2002) and also offers one path to inject code. Iczelion (2002) lists 14 types of hooks in the tutorial:

- ***WH_CALLWNDPROC*** *called when SendMessage is called*
- ***WH_CALLWNDPROCRET*** *called when SendMessage returns*
- ***WH_GETMESSAGE*** *called when GetMessage or PeekMessage is called*
- ***WH_KEYBOARD*** *called when GetMessage or PeekMessage retrieves WM_KEYUP or WM_KEYDOWN from the message queue*
- ***WH_MOUSE*** *called when GetMessage or PeekMessage retrieves a mouse message from the message queue*
- ***WH_HARDWARE*** *called when GetMessage or PeekMessage retrieves some hardware message that is not related to keyboard or mouse.*
- ***WH_MSGFILTER*** *called when a dialog box, menu or scrollbar is about to process a message. This hook is local. It's specifically for those objects which have their own internal message loops.*
- ***WH_SYSMSGFILTER*** *same as WH_MSGFILTER but system-wide*
- ***WH_JOURNALRECORD*** *called when Windows retrieves message from the hardware input queue*
- ***WH_JOURNALPLAYBACK*** *called when an event is requested from the system's hardware input queue.*
- ***WH_SHELL*** *called when something interesting about the shell occurs such as when the task bar needs to redraw its button.*
- ***WH_CBT*** *used specifically for computer-based training (CBT).*
- ***WH_FOREGROUNDIDLE*** *used internally by Windows. Little use for general applications*
- ***WH_DEBUG*** *used to debug the hooking procedure*

Hooks are particularly useful in a graphical environment. When a user interacts with a window in the GUI (such as making a selection, resizing windows or otherwise interacting) an event is created that needs to be sent to the application. Windows uses hooks as a means of allowing developers to utilize these events. The application will read a message from its queue and process the message using a series of hook filters which are registered against the application. These hook filters detail which messages the application will accept. Multiple hook filters and generally registered to any application. These are changed together to create a chain.

On receipt of a message by the filter function the filter will evaluate the event can then execute arbitrary code. The processes then run include those used by malicious code such as monitoring keystrokes. Foon (2002) used this feature of Windows in the development of the "*Shatter Attack*" for privilege escalation.

In the "*Shatter Attack*", the message created by the system is delivered on the execution of an event (such as a mouse click). This message is sent first to the User Process (Figure 7, A) where the GetMessage() / DispatchMessage() functions evaluate the message and select the appropriate hook chain (Figure 7, B).

The Hook Chain consists of several (one or more) Filter Functions (Figure 7, C) which process the event in turn and can then execute arbitrary code based on the expected response and input. This can include Hooking API functions and is commonly used by malware to insert a keystroke monitor and other less desired functions.

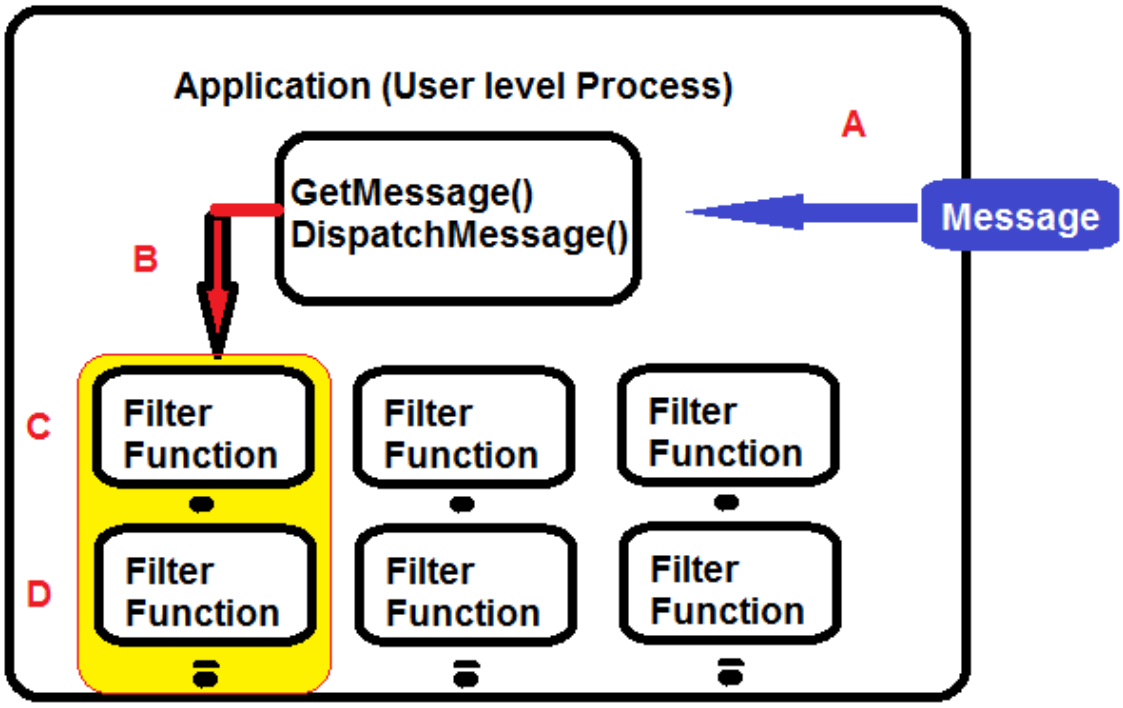

Figure 7: The Windows Hooking Process.

In the next article, we will step through a number of these attacks and look at the shell code used to implement them.

System Calls

In an x86/64 system, code that is running at a lower privilege level (this is a numerically higher ring such as the user level, Ring 3) is restricted from calling into code that is running at a higher privilege level (that is a numerically lower ring such as the Kernel level, Ring 0). In the event that code attempts to jump levels in this manner, a general protection (GP) exception is automatically generated by the CPU and the Operating System will generate a general protection exception handler to (hopefully) enact a suitable response (kill the application).

Windows system calls are generated using the "INT 0x2E" software interrupt. This is used as a signal that the system should switch into kernel-mode (Shanley, 1996). In the next article, we will extend this to looking at how the user-mode code tell the kernel-mode code what system function to execute. We will examine how an index is inserted into the EAX register before the "INT 0x2E" instruction is executed and how we can examine this step using a debugger (such as Olly) allowing us to set a breakpoint and watch this process as it occurs.

The kernel-mode Interrupt Service Routine (ISR) monitors the EAX register. When a request is loaded and the parameters are correct, this data is copied from the user-mode stack to the indicated kernel-mode function. We will investigate how this can be used install a hook and be used by our shellcode in the next article.

Conclusion

In the next instalment in the series of articles we will continue with DLL injection before starting on API hooking. At this point we have learnt the basics of DLL injection and are ready to move onto applying it. The next article will include a section on functions and calls, extend DLL injection and then move to the actual API hooking process in coming articles. When we then put all of this together, we will have the foundations for creating shellcode for exploits and hence an understanding of the process that penetration testers and hackers use in exploiting systems. With these skills, you will see how it is possible to either create your own exploit code from scratch or even to modify existing exploit code to either add functionality or in order to bypass signature based IDS/IPS filters.

With this knowledge, you will learn just how easy it is for sophisticated attackers to create code that can bypass many security tools. More, armed with this knowledge you will have the ability to reverse engineer attack code and even malware allowing you to determine what the attacker was intending to launch against your system. In this way, you can improve your forensic and incident response skills.

Author's bio

About the Author:

Dr Craig Wright is a lecturer and researcher at Charles Sturt University and executive vice – president (strategy) of CSCSS (Centre for Strategic Cyberspace+ Security Science) with a focus on collaborating government bodies in securing cyber systems. With over 20 years of IT related experience, he is a sought-after public speaker both locally and internationally, training Australian and international government departments in Cyber Warfare and Cyber Defence, while also presenting his latest research findings at academic conferences.

In addition to his security engagements Craig continues to author IT security related articles and books. Dr Wright holds the following industry certifications, GSE, CISSP, CISA, CISM, CCE, GCFA, GLEG, GREM and GSPA. He has numerous degrees in various fields including a Master's degree in Statistics, and a Master's Degree in Law specialising in International Commercial Law. Craig is working on his second doctorate, a PhD on the Quantification of Information Systems Risk.

References

Foon. (2002). Exploiting design flaws in the Win32 API for privilege escalation - Shatter Attacks - How to break Windows, from http://www.net-security.org/article.php?id=162

Iczelion, A. (2002). Tutorial 24: Windows Hooks. *Iczelion's Win32 Assembly Homepage* Retrieved 17 Apr 2012, from http://win32assembly.online.fr/tut24.html

Shanley, T. (1996). *Protected Mode Software Architecture*: Mindshare Inc.

Shewmaker, J. (2006). *Analyzing DLL Injection.* Paper presented at the NS2006, GSM Presentation.