College of Adminstration and Economics **Tanmyat AL-Rafidain** 91(30) 2008
P.P [9-17]

Controlling APIs Function Call Using Code Overwriting Technique

Rawaa Putros Polos
Assistant Lecturer
Department of Computers Sciences
College of Computers and Mathematics Sciences
University of Mosul

ABSTRACT

Through API, Windows application can request the aid of the operating system. So, controlling API's call allows getting access and controlling other programs.

The idea of this research relies upon the technique of directly manipulating and modifying API code in memory at run time.

The research presents software that injects a DLL using Windows hook technique into processes address space. The injected DLL replaces each desired API function with a new function. So, any call made by those processes to the controlled API will cause to call the new replacement function.

This technique had been applied on a number of API functions and dealt with files in order to control file manipulation operations which uses programs in the system.

Microsoft visual C^{++} version 6.0 is used by the researcher to develop the software.

السيطرة على عملية استدعاء الدالة API باستخدام تقنية كتابة الشفرة

المستخلص

من الممكن لتطبيقات نظام التشغيل ويندوز طلب مساعدة النظام من خلال استدعاء دوال API وبهذا فإن التحكم باستدعاء تلك الدوال تمنح إمكانية الوصول والتحكم ببقية البرامج.

تعتمد فكرة هذا البحث على تقنية معالجة وتغيير شفرة دوال API مباشرة في الذاكرة أثناء وقت التنفيذ.

يقدم البحث برنامجاً يقوم بحقن مكتبة ربط ديناميكية باستخدام تقنية خطاف الويندوز في فراغ عناوين المعالجات. حيث تقوم مكتبة الربط بتبديل جميع دوال API المختارة بدوال جديدة. وبهذا فإن أي استدعاء من قبل تلك المعالجة لدالة API سيؤدي إلى استدعاء الدالة الجديدة البديلة.

تم تطبيق هذه التقنية على عدد من دوال API التي تتعامل مع الملفات لغرض السيطرة على العمليات المتعلقة بالملفات والمستخدمة من قبل بقية البرامج في النظام.

استخدمت لغة مايكروسوفت فيجوال سي ++ الإصدار 6.0 من قبل الباحث لتطوير البرنامج.

تأريخ استلام البحث ٢٠٠٧/٢/١٩ **تأريخ قبول النشر ٢٠٠٧/٨/٢٦**

1. Introduction

Several modern systems are attracting the attention to their ability to utilize existing Windows application by employing hooking techniques. Also, the development for system monitoring and the analysis tools depends heavily upon API hooking techniques.

Code overwriting is one of these techniques. It represents a fundamental approach of getting control over a particular piece of code execution. It provides a straight forward mechanism that can easily alter the execute flow of these programs, and eventually affecting the operating system behavior without having their source code available [Chien, 2005, 3; Pietrek, 1994, 2].

A key motivation for code overwriting approach is to inject user - supplied code to offer an easy way to use interface and ability to capture different APIs, which are called by other programs. This approach provides many advantages [Rauen, 2006]:

- The ability to control APIs functions called which are extremely helpful and enables developers to track down specific invisible action that occur during the API call.
- It is quite useful technique for knowing OS in depth.
- It can provide an easy way to change and extend existing module functionality.
- Monitoring calls to functions using code overwriting can be applied to identify the action of the programs as a whole. It enables the recognition of, when and how the functions are called. This will bring a better understanding of the programs.

The power of this technique is that it is a convenient way to modify program's behavior without the need to program source code.

2. Windows API Functions:

Windows applications use a rich set of system defined functions called Application Program Interface (API).

Microsoft Windows comprises a well documented set of APIs, through these APIs, a process can request the aid of the operating system, and the operating system is defined by its APIs.

They encompasses all the functions call that the application program can make of the OS and provide all system services, as well as definitions of associative data types and structures. In Windows, the APIs also implies particular program architecture [Petzold, 1998, 25].

These functions are contained in Dynamic Link Libraries (DLL), in which each program can be accessed it when it is executed. These functions are added only when the application is loaded into memory for execution [M. Johnson, 2004, 120].

Therefore, the attempt to monitor API functions will give a chance for controlling Windows applications that called these functions.

3. Windows System-Wide Hooking:

Windows message hooking can be considered one of the most powerful features of windows. A hook is a mechanism by which a function can traps events (messages, mouse actions, keystrokes) before they reach an application. By hooking, windows will be informed about a callback function named hook procedure; that will be called every time an event (message) to be interested to occurs, i.e. it receives events. There are two types of message hooking depending on the affected scope [Marsh, 1994; Iczelion, 2002]:

1. Local hook: It traps events, this will occur in user own process only.
2. Global hook: It traps events, this will occur in other process either known as thread hook when affect only one process or as System - Wide hook which affect all processes in the system.

In system - wide hooking, all related events will routed through user supported hook procedure. The procedure will be called whenever the event associated with that type of hook occurs.

When the hook is created by a program, Windows creates data structure in memory, containing information on the hook. When an event occurs, if global hook type had been installed, the system must inject the Code for the hook procedure into the address space (s) of the other process (s) in the system. The system can perform the injection process if the hook procedure resides in a DLL [Iczelion, 2002; MSDN, 2005].

Windows provides functions and declarations to install and manipulate different type of hooking depending on message type associated with the hook.

There are 14 different types of hooks each of which related and manipulated different type of messages:

1. WH_CALLWNDPROC
2. WH_CBT
3. WH_DEBUG
4. WH_FOREGROUNDIDLE
5. WH_GETMESSAGE
6. WH_JOURNALPLAYBACK
7. WH_JOURNALRECORD
8. WH_KEYBOARD
9. WH_MOUSE
10. WH_MSGFILTER
11. WH_SHELL
12. WH_SYSMSGFILTER
13. WH_MOUSE_LL

14. WH_KEYBOARD_LL

In the research, WH_GetMessage hook type is used. Windows calls this hook when the GetMessage() or the PeekMessage() function is about to return a message. Therefore, all messages processed by windows in the system can be monitored using this type of hook.

4. DLL Injection into other Processes :

A Dynamic Link Library (DLL) is simply a set of source code modules with each module containing a set of functions that an application (executable) or another DLL will call.

A dynamic link library is a kind of common pool of functions. The code for the functions are dynamically linked will be stored in a DLL file, which is separate from the rest of the program. Windows will not load several copies of a DLL into memory so even if there are many instances of program running at the same time. There will be only one copy of the DLL that program uses in memory. Therefore, in physical memory, there is only one copy of DLL code and the program links to a DLL at run - time [Malware, 2006, 15-18; Iczelion, 2002].

Certainly a very popular technique for injecting DLL into a target process relies on one provided by Windows hook. Notice that, Microsoft was forced to devise a mechanism that allows a DLL to be injected into the address space of another process [Ritcher J., 2000, 215]

The basic concept as described in previous section is that the hook callback procedure must be a part of the DLL. Once the application installs a system-wide hook, the operating system maps the DLL into address space in each of its client process [Chien, 2005, 6-7; Maleware, 2006, 20].

If a process installs WH_GetMessage hook, injection scenario will be as follows [Ritcher, 1994, 6]:

1. Other processes prepare to dispatch a message to windows.
2. The system checks to see whether a WH_GetMessage hook is install on these processes.
3. The system checks to see whether the DLL containing the hook procedure is mapped into these processes address space.
4. If the DLL has not been mapped, the system forces the DLL to be mapped into processes address space.
5. The system calls the hook procedure in the processes address space.

Note that all variable that containing sharing data among the processes that have loaded the DLL must be placed in a shared data section within the DLL.

The diagram in figure (1) shows an example of a hook installed in the system and injected into the address spaces named "Application one" and "Application two".

Therefore, briefly the operating system injects the DLL automatically into all processes that meet requirements for this particular hook. A system-wide hook is loaded by multiple processes that don't share the same address space. So, when the DLL is executed, it is executed in the context of the process whose events are being caught. This means that the address it sees even for its own variable is the address of the target process [Richer, 1994, 8].

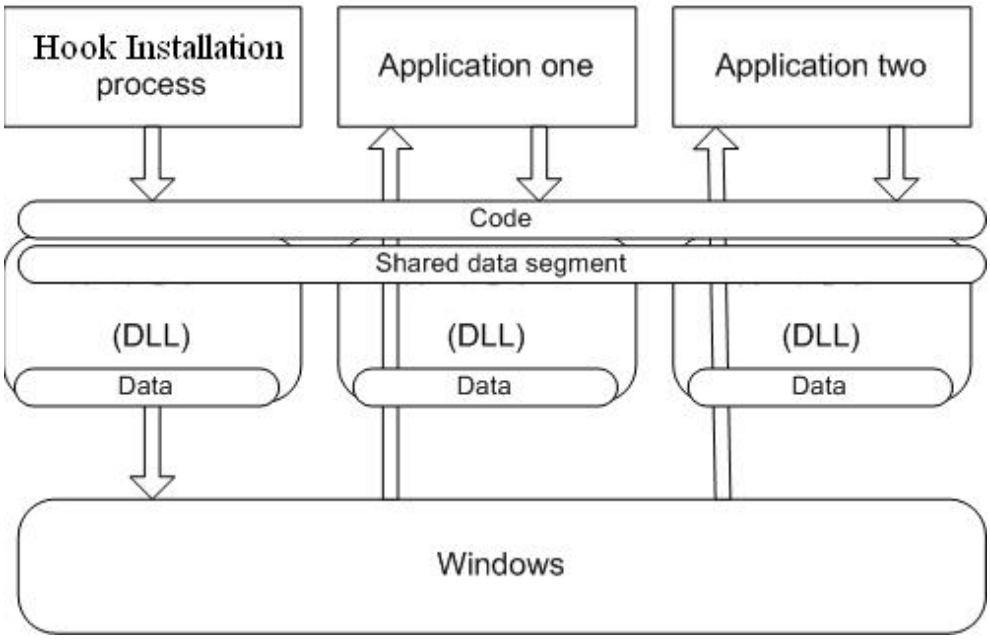

Figure 1
DLL injection using Windows Hook

5. API Code Overwriting:

Briefly, the concept behind this approach is to modify the API itself by allocating the address of a desired API function in memory. A control transfer instruction is inserted at the beginning of the API function by changing the first few bytes of that function with the call instruction that redirect the call to a custom supplied function [Hunt, 2003, 3; C. Stevens, 2005, 12].

Code overwriting is applied dynamically at run-time. It benefits from the fact that processes in Windows has the ability to access the memory space of other running processes. A process can read and write to the memory space of other processes as well as to execute the code in those

processes. So, after applying the overwriting operation, when the execution reaches the changed API function, control jumps directly to the user supplied function. In some cases user function can return control to the original API [Rauen, 2006; Chien, 2005, 17].

Typically, the implementation of this technique is accomplished by saving the first five bytes of the API function that must be overwritten in order to preserve the same functional behavior. After that, an immediate call usually placed in these bytes. The call leads to the user provided function. The replacement function can call the original API using the saving bytes [Bltur, 2005; Pietrek, 1994, 4].

The call instruction on the x86 architecture will usually required five bytes. The first byte is allocated for instruction opcode and the remaining four bytes are for the address of the user function.

6. API Code Overwriting Workflow :

The overall workflow of API code overwriting implemented in a DLL is described in the following steps:

- Install system-wide hook using WH_GetMessage hook type, which its hook procedure exists in a DLL.
- DLL injection, when the other processes call either GetMessage or PeekMessage, it loads the DLL that contains the replacement steps in their address spaces.
- Target function modification: when the DLL attaches to the process, it modifies the API function in the target process space so it directly jumps to the replacement function in the DLL.
- Calling the new function every time the changed API has been called.
- New function either calls the original API or discards the calling request.

7. Software implementation:

Visual C^{++} had been used for developing different parts of the software.

The software consists of two parts: main part that install system-wide hook, and the DLL that contains the essential steps for overwriting selected API functions.

System wide hook installed using SetWindowsHookEx() API function which has the following declaration:

HHook SetWindowsHookEx (Hooktype, pHookProc, hInstance, ThreadID)

-HookType: is an integer code describing the hook to which to attach hook procedure, WH-GetMessage used which Windows calls it

when the GetMessage() or PeekMessage() function is about to return a meesage.

-pHookProc: is the address of hook procedure that will be called to process the message for the specified hook.

-hInstance: is the instance handle of the DLL in which the hook procedure resides.

-ThreadID: is the ID of the thread to be hooked its message. For system wide hook, this parameter must be NULL.

The hook procedure has the following form:

LRESULT CALLBACK HookProc(nCode, wParam, lParam)

-nCode: known as hook code is an integer code that informs the hook procedure of any additional data it should know.

-wParam & lParam: pass information needed by the hook procedure.

After the system wide hooking installed by the main part, each process that call GetMessage() or PeekMessage() will load the DLL in its address space then the DLL will be executed in the context of that process. This is the injection or mapping operation.

When the DLL's injection done, the whole DLL is mapped, not just the hook procedure. DLL contains necessary steps for performing overwrites operation on a number of selected API functions; it contains new replacement function for each of the selected API. These steps are described below:

1. A new function that has the same declaration as the API function is prepared. It will be called instead of the original API. For example, the new replacement function of DeleteFile API function is:

   BOOL CALLBACK New_DeleteFile(LPCTSTR lpFileName)

   Instead of the original API:

   BOOL CALLBACK DeleteFile(LPCTSTR lpFileName).
2. Locating the address of the API function in the memory using GetProcAddress() and LoadLibrary() API functions:
   - LoadLibrary() used to obtain the handle of the system DLL that contains the API.
   - GetProcAddress() used to get the actual address of the API. This function accept as parameters the DLL handle obtained from the LoadLibrary() and API function name:

     typedef BOOL (CALLBACK *Temp_DeleteFile) (LPCTSTR);

     Temp_Delete File Original_Delete File = GetProcAddress (LoadLibrary("kernel.dll"),"DeleteFile");
3. Prepare A structure to be used for holding the first bytes of the original API to be overwritten & the new five bytes.
4. Save the first five bytes of the API using ReadProcessMemory API function that accept as parameters the address of that API obtained

from step 2 and a variable of structure type defined in step 3 to store these bytes.

5. Overwrite the bytes with CALL instruction to the new function defined in step 1. This step is accomplished using WriteProcessMemory API function that accepts as parameters the address of the same API obtained from step 2 and a variable of structure type defined in step 3 contains the values of new bytes.

When the execution reaches the new function, it displays a message box asking the user to choose either calling the original API functions via the saved bytes or discards the call.

The software applies code overwriting technique upon the following API functions:

- DeleteFile Lib "kernel32"
- CopyFile Lib "kernel32"
- MoveFile Lib "kernel32"
- OpenFile Lib "kernel32"
- ReadFile Lib "kernel32"
- WriteFile Lib "kernel32"

So, when other running applications program attempt to call any of the above functions while the code overwriting software is running, the decision of continue API call or stopping it will be under user control.

8. Conclusion

Code overwriting technique for controlling APIs call is a suitable and strongest method for modifying program logic, because the source code for the target application is not available most of times, and it is relatively simple to implement by building a DLL that contains the replacement function, separating it from the rest of the software. Also, injection a DLL into a process's address space is a good way to determine what's going on within a process.

This method is quite effective, but it has disadvantage. It degrades the entire performance of the system because it effects the processing of the whole system.

References:

1. Charles Petzold, "Programming Windows", 1998, Microsoft Press, 5th edition.
2. Eric Chien, "Techniques of Adware and Spyware", 2005, Semantic Security response.
3. Galen Hunt and Doug Brubacher, "Detours: Binary Interception of Win32 Functions", 2003, Microsoft Research.
4. Iczelion, "Iczelion's Tutorials for Win32ASM, Tutorial 24: Windows Hooks", 2002, http://win32assembly.online.fr/tut24.html.
5. James Bulter and Sherri Sparks, "Windows RootKits of 2005 part one", 2005, http://www.securityfocus.com/infocus/850.

6. Jeffery Richter, "Load your 32 DLL into Another Process's address space using INJLIB", 1994, Microsoft systems Journal/9 No. 5.
7. Jeffery Richter, "Programming Application for Microsoft Windows", Microsoft press, 2000, 4th edition.
8. Johnson M. Hart, "Windows System Programming", 2004, Addison Wesley Professional, 3rd edition.
9. Kyle Marsh, "Win32 Hooks", 1994, Microsoft Developer Network Technology Group, MSDN 2005.
10. Malware-Test, "Rootkit Internals Workshop", 2006, www.malware-test.com/smf/.
11. Mathias Rauen, "API Hooking Methods", 2006, http://www.madshi.net/apihooking.htm.
12. Matt Pietrek, "Learn System-Level win32 Coding techniques by writing an API Spy Program", 1994, Microsoft Systems Journal Vol. 9, No. 12.
13. MSDN 2005, "Monitoring System events", 1999, Microsoft corporation SDK.
14. Stevens C., "Windows Rootkit Overview", 2005, Symantec Security Response.