Hooking Methodology for the Windows Operating System

Jang, Seung-Ju¹
¹Department of Computer Engineering, Dongeui University

Abstract - This paper describes hooking technique in Windows operating system. There are various Windows operating system hooking techniques. The hooking technique in the Windows operating system is usually used as a convenient function, but it is also used for malicious purposes. This paper aims to be the foundation of the technology development for protecting malicious program related to hooking in Windows operating system by looking at hooking technique used in Windows operating system.

Keywords – Windows O.S, Hooking, Hooking Method, SSDT, IAT, Detours

I. INTRODUCTION

In computer programming, hooking covers a range of techniques used to alter the behavior of an operating system, of applications, or of other software components by intercepting function calls, messages or events between software components. The hook is that handles such intercepted function calls, events or messages. Due to the emergence of a variety of cyberattacks and malicious code continues to increase, the damage is becoming increasingly. So far, most of the known malware is being made to target the Windows OS. The biggest reason is that the Windows OS is easy to use and familiar to users.

According to statistics, the current OS that is most susceptible to malware as Windows O.S. Although Windows O.S has a security policy, there are various ways to hook the Windows O.S. This paper examines the hooking techniques that occurs in Windows operating system. By looking into the window hooking techniques, we can prepare for various problems that may occur in window because of hooking. The structure of the Windows operating system is shown in Fig. 1 [9].

![Figure 1. Windows operating system architecture](image-url)

In Fig. 1, the kernel provides the base OS functionality for any Windows devices. This functionality includes process, thread, and memory management. The kernel also provides some file management functionality. Kernel services enable applications to use this core functionality.
NK.exe is the OEM application layer (OAL) process. Kernel.dll, is loaded during boot. Nk.exe is the core of the Windows OS. Use the kernel process and thread functions to create, terminate, and synchronize processes. Processes, which represent single instances of running applications enable users to work on more than one application at a time. Threads enable an application to perform more than one task at a time [1, 2, 10].

This paper researches the development trends of Windows hooking technology. In chapter 2, I will explain Windows O.S hooking technology trend. In chapter 3, I will explain the windows O.S hooking technology. And finally, I will conclude in chapter 4.

II. WINDOWS O.S HOOKING TECHNOLOGY TREND

A hook is a mechanism by which an application can intercept events, such as messages, mouse actions, and keystrokes. A function that intercepts a particular type of event is known as a hook procedure. The following is some example for hooks:

- Messages hooking
- Function call hooking
- Event hooking
- Keyboard hooking
- Kernel hooking

The system supports many different types of hooks. Each type provides access to a different aspect of its message-handling mechanism. The system maintains a separate hook chain for each type of hook. A hook chain is a list of pointers to special, application-defined callback functions. When a message that is associated with a particular type of hook occurs the system passes the message to each hook procedure referenced in the hook chain. The action a hook procedure can take depends on the type of hook involved.

To take advantage of a particular type of hook, the developer provides a hook procedure and uses the SetWindowsHookEx function to install it into the chain associated with the hook.

```cpp
#include <windows.h>

// variable to store the HANDLE to the hook.
HHOOK _hook;

// This struct contains the data received by the hook callback.
KBDLLHOOKSTRUCT kbdStruct;

// This is the callback function
LRESULT __stdcall HookCallback(int nCode, WPARAM wParam, LPARAM lParam)
{
    if(nCode >=0)
    {
        // the action is valid: HC_ACTION.
        if(wParam == WM_KEYDOWN)
        {
            // lParam is the pointer to the struct containing the data needed.
            kbdStruct =*((KBDLLHOOKSTRUCT*)lParam);
            // a key (non-system) is pressed.
        }
    }
    return DEF_hookCallback(nCode, wParam, lParam);
}
```
if(kbdStruct.vkCode == VK_F1)
{
    // F1 is pressed!
    MessageBox(NULL,"F1 is pressed!","key pressed",
    MB_ICONINFORMATION);
}

// call the next hook in the hook chain.
Return CallNextHookEx(_hook, nCode, wParam, lParam);
}

Void SetHook()
{
    // Set the hook and set it to use the callback function above
    if(!(_hook =SetWindowsHookEx(WH_KEYBOARD_LL,HookCallback, NULL,0)))
    {
        MessageBox(NULL,"Failed to install hook!","Error", MB_ICONERROR);
    }
}

Void ReleaseHook()
{
    UnhookWindowsHookEx(_hook);
}

void main()
{
    // Set the hook
    SetHook();

    MSG msg;
    while(GetMessage(&msg, NULL,0,0))
    {
        ...... 
    }

    

**Figure 2. Hook example for Windows O.S**

In Fig. 2, the value of the hook code depends on the type of the hook. Each type has its own characteristic set of hook codes [3]. The values of the wParam and lParam parameters depend on the hook code, but they typically contain information about a message that was sent or posted. The SetWindowsHookEx function always installs a hook procedure at the beginning of a hook chain. When an event occurs that is monitored by a particular type of hook, the system calls the procedure at the beginning of the hook chain associated with the hook. Each hook procedure in the chain determines whether to pass the event to the next procedure. A hook procedure passes an event to the next procedure by calling the CallNextHookEx function [7].
III. WINDOWS O.S HOOKING TECHNOLOGY

The process of calling an API in the user mode of the Windows operating system is as follows. We assume that you call the CreateFile() function, which is commonly used in the Windows operating system.

When calling the CreateFile() function in a normal application, it passes the string corresponding to the file name as the first argument. The CreateFileA() or CreateFileW() function is called depending on whether the file name is in ASCII or UNICODE format.

If a file name is in UNICODE format, it does not go through the CreateFileA() function. When passing a file name in ASCII format, it converts the ASCII string to a wide character for UNICODE format in the CreateFileA() function, and then calls the CreateFileW() function (In Figure 3 (1), (2)).

After passing to the CreateFileW() function, it goes through several complicated processes internally. After completing the internal complexity, a code called CALL __imp__NtCreateFile appears, which calls the __imp__NtCreateFile() function in ntdll.dll. Ntdll.dll is a module that acts as a kind of intermediary between kernel32.dll and the kernel, and is mainly responsible for switching to user mode and kernel mode.

An index value which determines which of the native API functions of the kernel (In Figure 3 (3)). One of the kernel hooking techniques is hooking this native API (In Figure 3 (3)). Most API functions in the Windows operating system go through native API functions. If you hook up these native APIs, you can hook up the entire Windows operating system. If you hook up NtCreateProcess(), you can control the creation of all processes, and hooking NtWriteVirtualMemory() can control the writing attempts of all processes to memory. Figure 4 is the windows CreateFile() function hooking procedure.

**Figure 3. Windows O.S hooking procedure**

**Figure 4. Windows CreateFile() function hooking procedure**
The followings are the hooking technologies related to the Windows operating system.

• Native API intercepts by SSDT (System Service Dispatch Table) alteration
• An intercept that change the IAT (Import Address Table)
• An intercept that change the head address of functions (Detours)
• An intercept that replace DLL (DLL Injection)

This paper describes techniques related to these hooking technology.

3.1 SSDT (System Service Dispatch Table) Hooking
There are basically 2 address spaces in Windows O.S, user address space and kernel address space. This means an application is either designed to run in user mode or in kernel mode. Usually, high level programs run in user mode, while low level programs run in kernel mode. User mode process address space is private (and virtual), all the processes see the same address range. When some unexpected thing happens in user mode[9].

The SSDT (System Service Dispatch Table) is a table for managing the native APIs existing in the kernel. The window structure related to SSDT hooking is shown in Figure 5. SSDT hooking is to replace the SSDT space with the address of the desired native API into the address space. SSDT is located in memory space accessible from kernel mode. Therefore, access to the SDDT is not possible in general user mode applications. To access the SSDT, you must use the device driver.

You can access the SSDT by creating an arbitrary device driver in the Windows operating system and replacing the value of the table with the address of the function to be hooked in SSDT with the address. But if you change the value, it will display the blue screen, because SSDT is read only. This is because Windows O.S has removed the write permission of the memory using a technique called write protection [3, 4, 9].

3.2 IAT (Import Address Table) Hooking
Your code dynamically links to an external DLL and references external functions via a table of function pointers. When the windows loader loads the executable image it will also load up all DLLs that the application requires. The exe says its needs, and place the addresses to those functions into the IAT (Import Address Table).

When the windows loader goes to initialize DLL, it will also give us the opportunity to execute some code before the main application gets to run. In “dllmain” code, we will change one of these function pointers stealing it away from the real function and pointing it instead into our replacement code [8].
IAT (Import Address Table) is an array of links representing the various DLLs imported by the PE (Portable Executable) loader during process initiation. IAT hooking is a technique of modifying the address of a particular DLL in the IAT with address of hook function. Before performing IAT hooking, we must make sure that we are able to put the hook function in the user’s address space through any of the DLL injection. IAT hooking will not be useful to us if the target program performs dynamic linking through LoadLibrary and GetProcAddress APIs [5, 6, 11].

3.3 Detours
Typical reasons to intercept functions are to add functionality, modify returned results, or insert function for debugging or profiling. With access to appropriate source code, it is often trivial to insert new function or extensions by rebuilding the OS or application. However, researchers seldom have access to all relevant source code.

Detours is a library for intercepting Win32 binary functions. Interception code is applied dynamically at runtime. Instructions from the target function are placed. The detour function can either replace the target function, or extend its semantics by invoking the target function as a subroutine. Detours is a general-purpose package.

Figure 7 shows the detours technique. It is normal to be called from the source function to the target function. However, it goes to the target function through the detour function.

3.4 DLL Injection
DLL injection code which is a series of assembled language instructions is inserted into some available space in the running process and alters the registers to point at the offset. DLL injection provides a platform for manipulating the execution of a running process. It's very commonly used for logging information while reverse engineering.
Figure 8 shows the DLL injection procedure. The attacker process that wishes to hook opens the handle of the target process. It allocates memory in the target process and writes the name of the DLL to be injected into the allocated memory. In this way, it can force the memory mapping to penetrate the (LoadLibrary) target process. Alternatively, it can perform DLL injection using a window hooking function such as SetWindowsHookEx.

To inject the DLL into the process’s address space, we’ll use the SetWindowsHookEx function. We’ll hook, which is our malicious DLL, we’ll inject into the target process space (In Figure 8 (4)). The target process must get a handle first to our DLL and then get address of one of the exported functions inside that DLL.

IV. CONCLUSION

This paper describes the hooking techniques in windows operating systems. We looked into the structure of the windows operating system in relation to the hooking of the windows operating system, and discussed the techniques related to hooking in the windows operating system structure. In this paper, we explained the structure of windows operating system in relation to window hooking. And we looked into windows hooking technology. We also explained the SetWindowsHookEx function, which is commonly used for hooking windows operating systems.

The windows operating system allows hooking in a variety of ways. In this paper, we explained various hooking methods available in windows operating system. Among them, we explained how to hook through changing the SSDT (System Service Dispatch Table), through the IAT (Import Address Table) interception, through hooking the header address of the function, and through the DLL injection.

The various hooking methods provided by the windows operating system described in this paper should not be used for malicious purposes. It is helpful to utilize the hooking technology available in the windows operating system to develop techniques for dealing with malicious hooking users.

REFERENCES