

|  |  |
| --- | --- |
| Type | 32 bit Windows Executable File |
| File Name | Ardamax.Keylogger.zip |
| MD5Hash | 5de75a478ffb3aa01a88f4e539f3edc0 |
| URL Download | <https://github.com/ytisf/theZoo/blob/master/malwares/Binaries/Keylogger.Ardamax/Keylogger.Ardamax.zip> |
| Team Member Names | Tan Jia Shun (S10198161D)  Do Li Fang Sarah (S10194833D) |
| Date | 17/8/2020 |

**REVERSE ENGINEERING MALWARE ASSIGNMENT**

Table of Contents

[1 Introduction 4](#_Toc48486959)

[2 Lab Setup 4](#_Toc48486960)

[2.1 VMware Setup 5](#_Toc48486961)

[2.1.1 Network Diagram 8](#_Toc48486962)

[2.1.2 Network Configuration 9](#_Toc48486963)

[3 Passive Information Gathering (IDA Pro) 10](#_Toc48486964)

[3.1 Imported APIs 10](#_Toc48486965)

[3.1.1 KERNEL32.dll 10](#_Toc48486966)

[3.1.2 MSVCRT.dll 16](#_Toc48486967)

[3.1.3 USER32.dll 17](#_Toc48486968)

[4 Code Analysis (IDA Pro) 18](#_Toc48486969)

[4.1 Graph of Function Calls for Major Subroutine 18](#_Toc48486970)

[4.1.1 Full graph 18](#_Toc48486971)

[4.1.2 Graph breakdown 18](#_Toc48486972)

[4.2 Graph of Major Subroutines from Main Function 20](#_Toc48486973)

[4.3 Purpose of Major Subroutine: \_WinMain@16 21](#_Toc48486974)

[4.4 Description of Subroutine 23](#_Toc48486975)

[4.4.1 Subroutine 1: sub\_401230 23](#_Toc48486976)

[4.4.2 Subroutine 2: sub\_401000 38](#_Toc48486977)

[5 Patching 41](#_Toc48486978)

[5.1 Main Routine 41](#_Toc48486979)

[5.1.1 JE short loc\_401479 41](#_Toc48486980)

[5.1.1 CALL sub\_401000 41](#_Toc48486981)

[5.2 Subroutine: sub\_401000 42](#_Toc48486982)

[5.2.1 GetProcAddress 42](#_Toc48486983)

[6 General Analysis 42](#_Toc48486984)

[6.1 Type of Malware 42](#_Toc48486985)

[6.2 Functionality of Malware 42](#_Toc48486986)

[6.3 Interaction with the Malware 49](#_Toc48486987)

[6.4 Removal of Malware 50](#_Toc48486988)

# Introduction

The following report records the analysis of the malware, Ardamax Keylogger, using both advanced static and dynamic analysis techniques learnt from the Reverse Engineering Malware module. Ardamax Keylogger was downloaded from the following website : [https://github.com/ytisf/theZoo/blob/master/malwares/Binaries/Ardamax Keylogger/Ardamax Keylogger.zip](https://github.com/ytisf/theZoo/blob/master/malwares/Binaries/Artemis/Artemis.zip) and executed in a safe and isolated environment for analyzing purposes. The malware is protected by a password which is required for it to be extracted and analysed. The required password to be inputted is “infected”. The Ardamax Keylogger has a hash value of “5de75a478ffb3aa01a88f4e539f3edc0 “which can be used to verify the authenticity of the file. We ensure that the analysis of the malware will not bring any harm to any user and that this knowledge will be utilised only for the enhancement of our own technical skills.

# Lab Setup

The tools that were utilized to analyze the malware Ardamax Keylogger are the VMware Workstation 15 Pro which had an image of Windows 10 loaded on to it, the Interactive Disassembler Professional 5.0 (IDA Pro) as well as OllyDBG.

We used the VMware Workstation to ensure our security as we wanted to isolate the malware from our Operating Systems to prevent any damages or unauthorized changes to the data in our systems. This is a normal practice when analyzing malware and should be done so during every analysis. Since we are dealing with malicious software, it is natural that extra steps are required in assuring the safety of our systems. Hence, explaining the usage of the VMware Workstation. It allows us to work in a safe environment without worry of the malware spreading or harming our data.

To conduct the analysis of the malware, we used IDA Pro to disassemble the malware and OllyDBG to patch the file in order to prevent the file from carrying out certain commands that bring harm to the host’s system. These tools track registers, take note of procedures, API calls, switches, tables, constants and strings as well as identify the location of routines from object files and libraries.

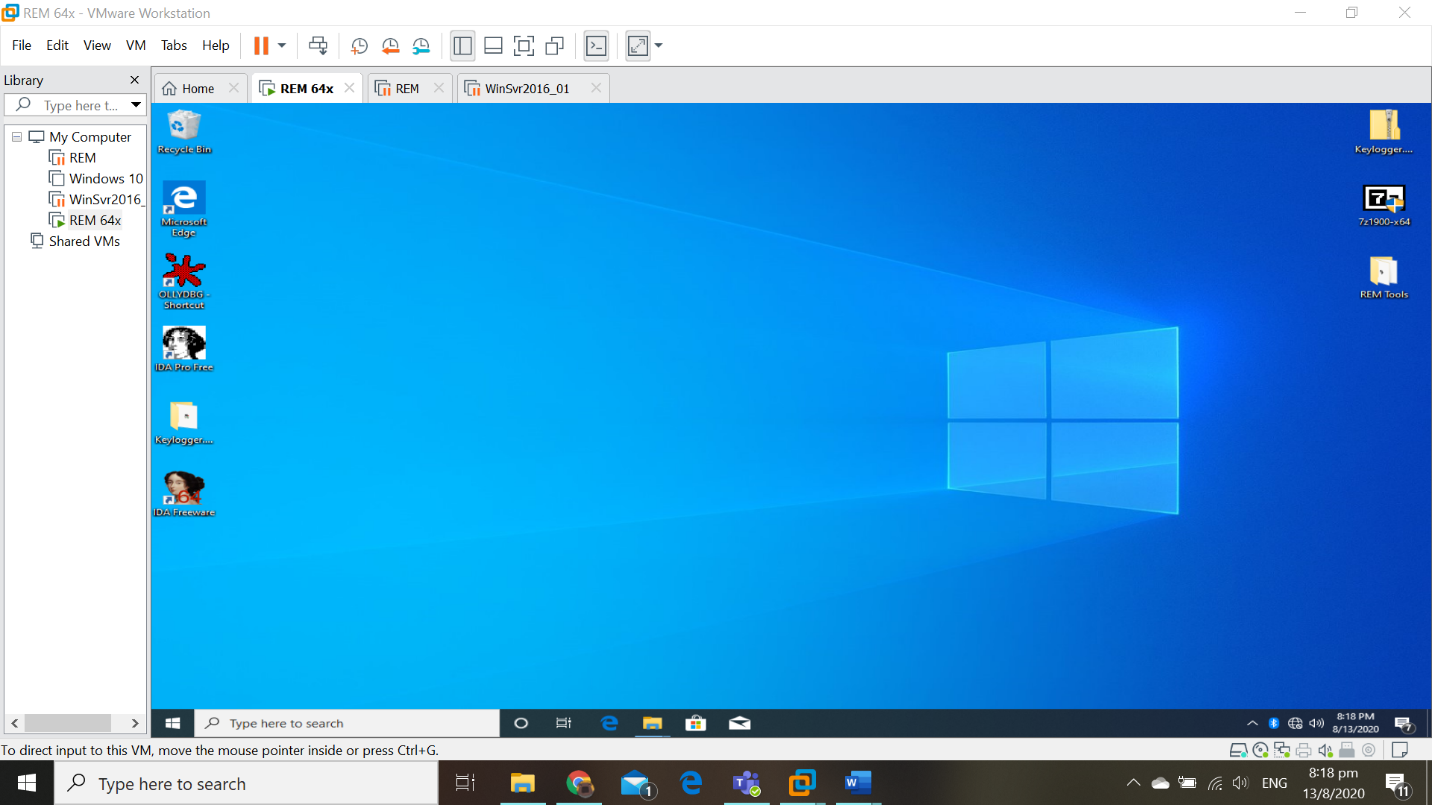
Diving deeper into OllyDBG, it is a tool that allows us to replace malicious codes with no operation codes called NOP that as stated above, causes the malicious lines of code to not be executed by the computer.

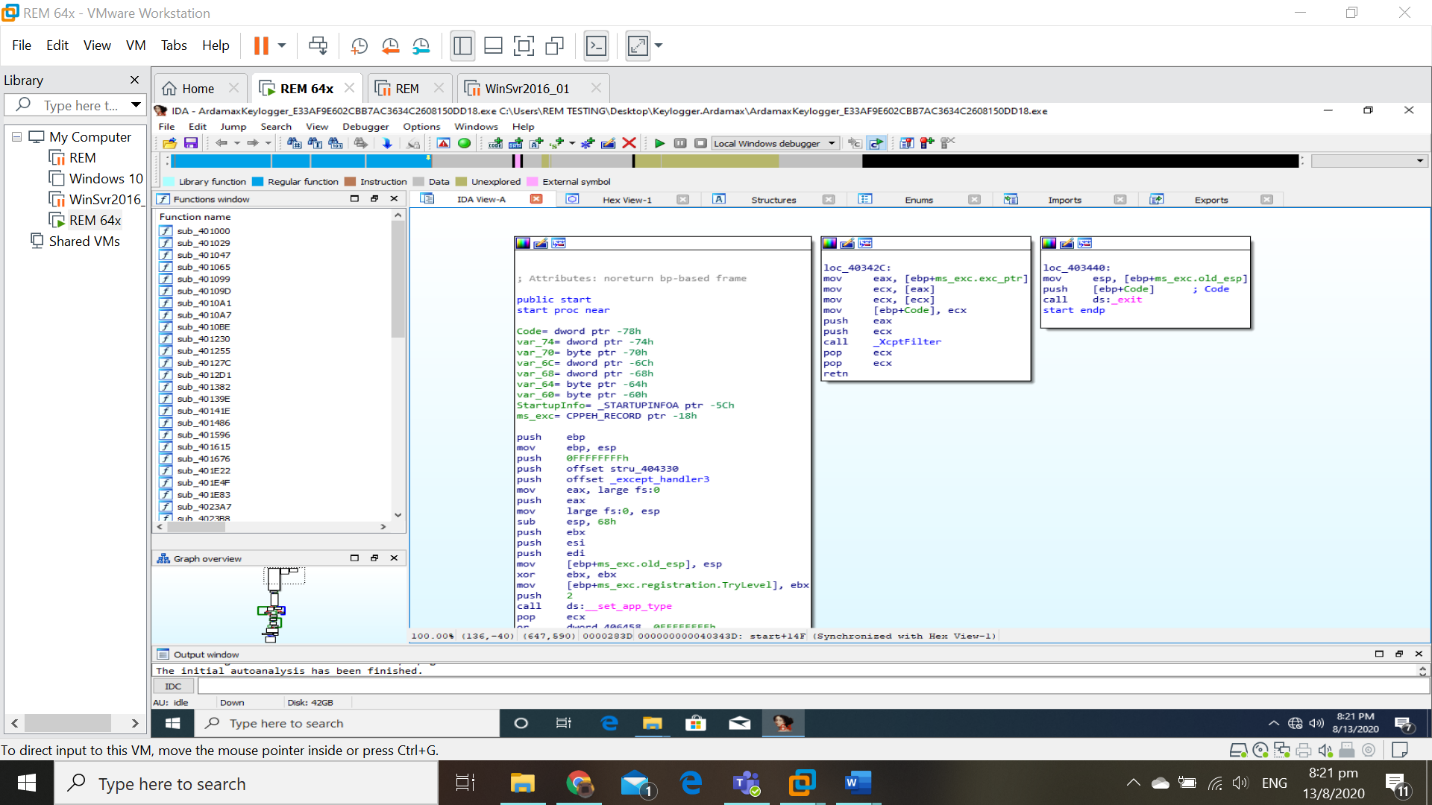
## VMware Setup

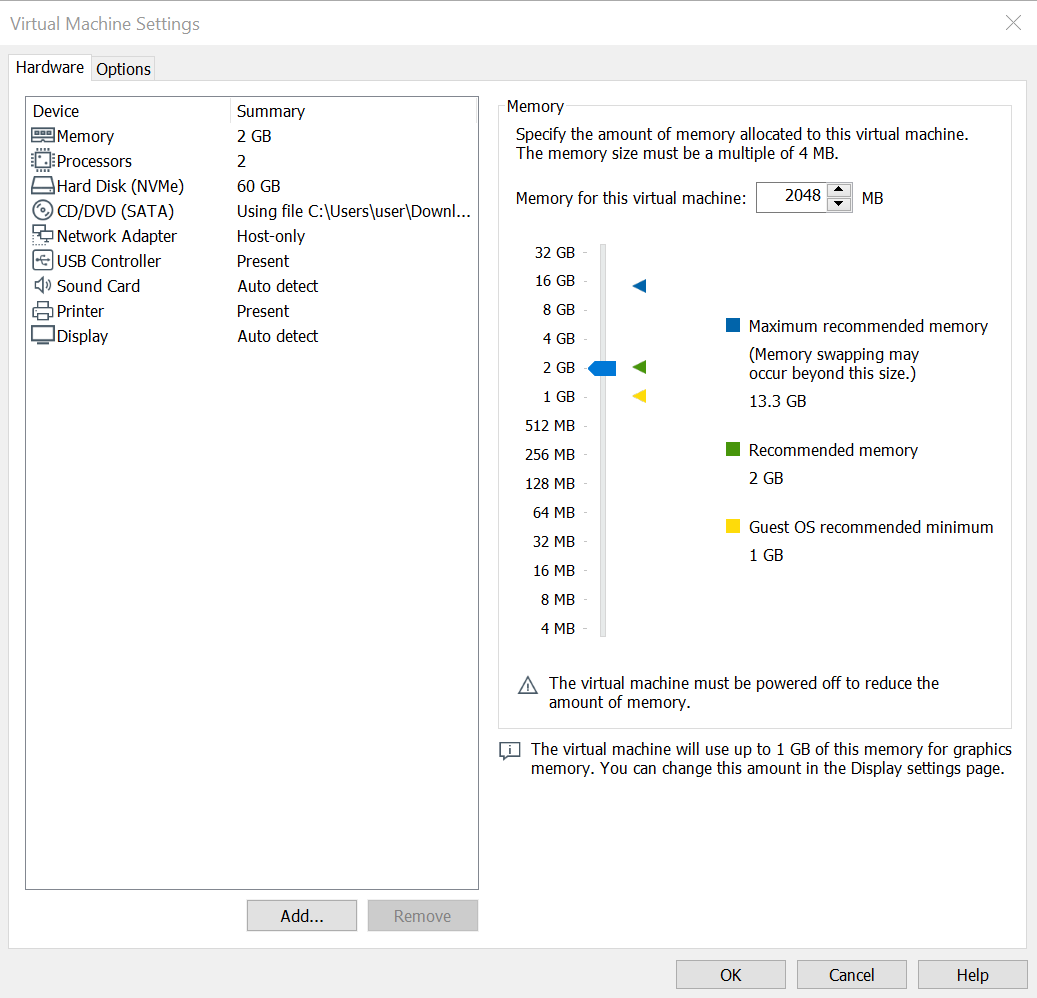
Host Operating System: Windows 10

Virtual Guest Operating System: Windows 10

Version of VMware: VMware Workstation 15 Pro

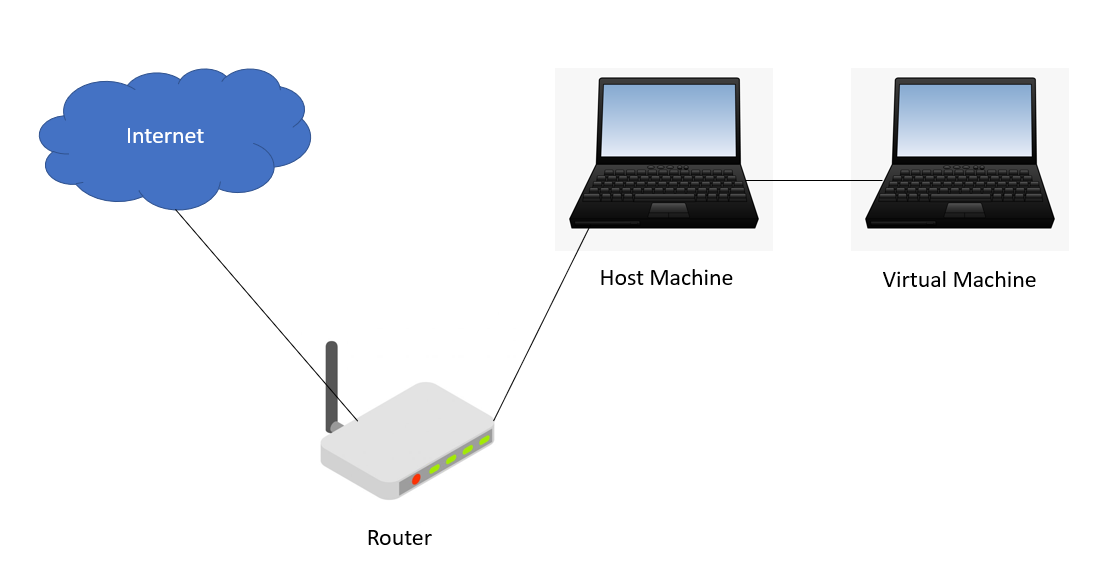






VMware Workstation 15 Pro is installed and will be running on the Windows 10 host machine. The guest machine which is the system that we are running on the VMware Workstation 15 Pro operates a Windows 10 operating system with a capacity of 2GB memory and 60GB of hard disk space. The virtual machine’s network adapter is also set to the network address translation (Host-only) network adapter mode. Host-only networking provides a network connection between the virtual machine and the host computer. It makes use of a virtual Ethernet adapter that is visible to the host operating system. This creates an isolated virtual network which is perfect for the analysis of the malicious software.

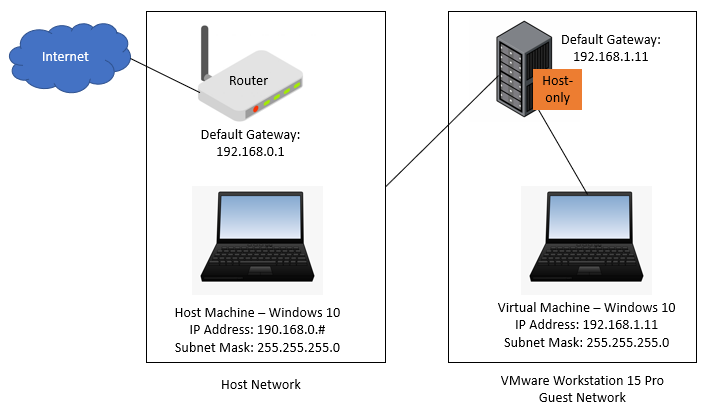
### Network Diagram



To provide a safe and isolated environment, we have set up the Windows XP virtual machine on the host’s system. This prevents any malicious codes to be executed or carried out by the malware during the analysis of Ardamax Keylogger. The purpose of using a virtual machine is because of its ability to take and revert back to snapshots. A VMware snapshot is a copy of the virtual machine’s disk file (VMDK) which can be taken at any point in time when the virtual machine is running. These snapshots are used to restore the VMware to a previous state when the snapshot was taken. Hence, it is known as “reverting back” to a snapshot. We typically revert back to a snapshot only when a failure or system error occurs. These are the steps we would take when the VMware has been infected by the malware. This allows us to freely analyze Ardamax Keylogger without the fear of destroying our data.

In the figure above, we can see that the host machine is connected to the router which has internet access. Since the virtual machine is running in the host-only network connection mode, the malware is unable to gain access to the internet and obtain the host system’s information. Host-only networking provides a network connection between host and the virtual machine. It utilizes a virtual Ethernet adapter that is visible to the host operating system. It creates an isolated virtual network. Hence, by creating the isolated virtual network, we are able to prevent the malware from getting more information and data about ourselves which further enhances our security while analyzing it.

### Network Configuration



This diagram provides a clearer visualization of the connection and configuration of the virtual guest machine’s network which is running on the host machine. The IP addresses, default gateways, subnet masks as well as the hardware are clearly drawn up. This diagram shows how the Host-only network adapter mode creates an isolated network which prevents the virtual guest machine from accessing the Internet even though it is running on the host machine which does have access.

# Passive Information Gathering (IDA Pro)

Employing the usage of IDAPro, this section of the report will document the information that we have gathered without editing or going into deep analysis of the malware. We obtained this information by looking through IDAPro’s graph view and names window to search for the imported APIs. This would allow us to construct a general idea of how the malware functions before diving in deeper to find out the nooks and crannies of the malware’s functioning.

## Imported APIs

API is known as and is the acronym for Application Programming Interface (API) which acts as a software intermediary that provides a platform for two applications to converse with one another. These APIs will give us a better picture of what the malware is looking for without establishing much contact. There is a total of 41 imported API in the Ardamax Keylogger which serve different purposes. They have been imported from different DLL files, namely KERNEL32.dll, MSVCRT.dll and USER32.dll. However, since we will only be analyzing the main function of the malware, we will only cover the important APIs that have been called in the relevant subroutines. This reduces the APIs that we will go through to a grand total of 13. In the following sections, we will be going through these APIs and their functions.

### KERNEL32.dll



#### GetTempPathW



**Purpose:** Retrieves the path of the directory designated for temporary files.

**Parameters:**

1. nBufferLength
   1. Size of string buffer identified by lpBuffer
2. lpBuffer
   1. Pointer to a string buffer that receives the null-terminated string specifying the temporary file path

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-gettemppathw>

#### GetModuleHandleW



**Purpose:** Retrieves a module handle for the specified module. The module must have been loaded by the calling process.

**Parameters:**

1. lpModuleName
   1. Name of loaded module

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlew>

#### GetModuleFileNameW



**Purpose:** Retrieves the fully qualified path for the file that contains the specified module. The module must have been loaded by the current process.

**Parameters:**

1. hModule
   1. A handle to the loaded module whose path is being requested
2. lpFileName
   1. A pointer to a buffer that receives the fully qualified path of the module
3. nSize
   1. Size of the lpFilename buffer

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulefilenamew>

#### CreateFileW



**Purpose:** Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

**Parameters:**

1. lpFileName
   1. Name of the file or device to be created or opened
2. dwDesiredAccess
   1. The requested access to the file or device which can be summarized as read, write, both or none
3. dwShareMode
   1. The requested sharing mode of the file or device which can be read, write, both, delete, all of the above or none
4. lpSecurityAttributes
   1. Pointer to a SECURITY\_ATTRIBUTES structure that contains two separate but related data members: an optional security descriptor and a Boolean value that determines whether the returned handle can be inherited by child processes
5. dwCreationDisposition
   1. An action to take on a file or device that exists or does not exist
6. dwFlagsAndAttributes
   1. The file or device attributes and flags, FILE\_ATTRIBUTE\_NORMAL being the most common default value for files
7. hTemplateFile
   1. A valid handle to a template file with the GENERIC\_READ access rights

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew>

#### SetFilePointer



**Purpose:** Moves the file pointer of the specified file.

**Parameters:**

1. hFile
   1. A handle to the file
2. lDistanceToMove
   1. The low order 32-bits of a signed value that specifies the number of bytes to move the file pointer
3. lpDistanceToMoveHigh
   1. A pointer to the high order 32-bits of the signed 64-bit distance to move
4. dwMoveMethod
   1. The starting point for the file pointer moves

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-setfilepointer>

#### CloseHandle



**Purpose:** Closes an open object handle.

**Parameters:**

1. hObject
   1. A valid handle to an open object

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle>

#### GetTempFileNameW



**Purpose:** Creates a name for a temporary file. If a unique file name is generated, an empty file is created and the handle to it is released; otherwise, only a file name is generated.

**Parameters:**

1. lpPathName
   1. The directory path for the file name
2. lpPrefixString
   1. The null-terminated prefix string
3. uUnique
   1. An unsigned integer to be used in creating the temporary file name
4. lpTempFileName
   1. A pointer to the buffer that receives the temporary file name

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-gettempfilenamew>

#### ReadFile



**Purpose:** Reads data from the specified file or input/output (I/O) device. Reads occur at the position specified by the file pointer if supported by the device.

**Parameters:**

1. hFile
   1. A handle to the device
2. lpBuffer
   1. A pointer to the buffer that receives the data read from a file or device
3. nNumberOfBytesToRead
   1. The maximum number of bytes to be read
4. lpNumberOfBytesRead
   1. A pointer to the variable that receives the number of bytes read when using a synchronous hFile parameter
5. lpOverlapped
   1. A handle to an OVERLAPPED structure

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile>

#### LoadLibraryW



**Purpose:** Loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded.

**Parameters:**

1. lpLibFileName
   1. The name of the module

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryw>

#### GetProcAddress

#### 

**Purpose:** Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).

**Parameters:**

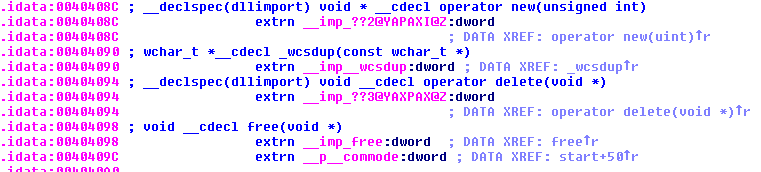
1. hModule
   1. A handle to the DLL module that contains the function or variable
2. lpProcName
   1. The function or variable name, or the function’s ordinal value

**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>

### MSVCRT.dll



#### \_imp\_??\_2@YAPAXI@Z [operator new(uint)]



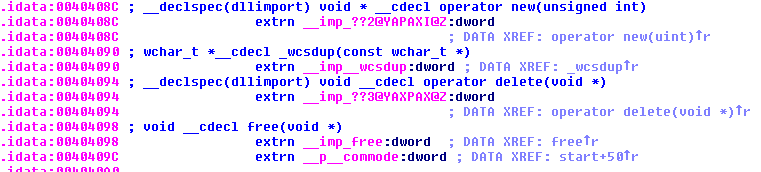
**Purpose:** Allocate storage space

**Parameters:**

1. size\_t
   1. Size in bytes of the requested memory block

**MDSN Reference:** <http://www.cplusplus.com/reference/new/operator%20new/>

#### \_imp\_wcsdup



**Purpose:** Duplicates strings.

**Parameters:**

1. **strSource**
   1. Null-terminated source string

**MDSN Reference:** <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strdup-wcsdup-mbsdup?view=vs-2019>

### USER32.dll



#### MessageBoxW



**Purpose:** Displays a modal dialog box that contains a system icon, a set of buttons, and a brief application-specific message, such as status or error information. The message box returns an integer value that indicates which button the user clicked.

**Parameters:**

1. hWnd
   1. A handle to the owner window of the message box to be created
2. lpText
   1. The message to be displayed
3. lpCaption
   1. The dialogue box title
4. uType
   1. The contents and behavior of the dialogue box

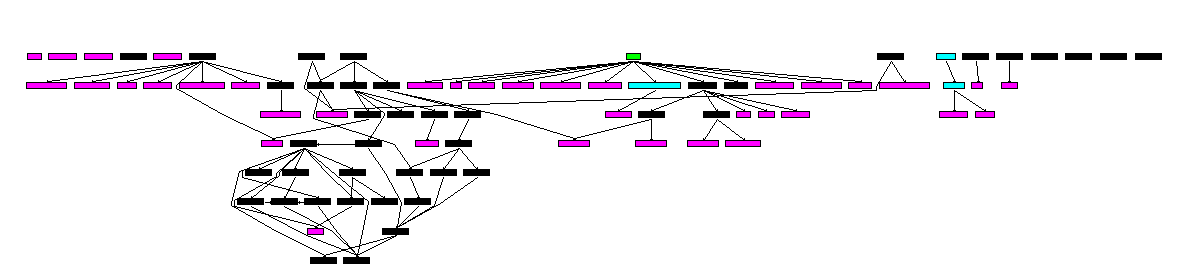
**MDSN Reference:** <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxw>

# Code Analysis (IDA Pro)

The following section in the report will explain the purposes of the subroutines in the malware, specifically the ones directly connected to the main subroutine.

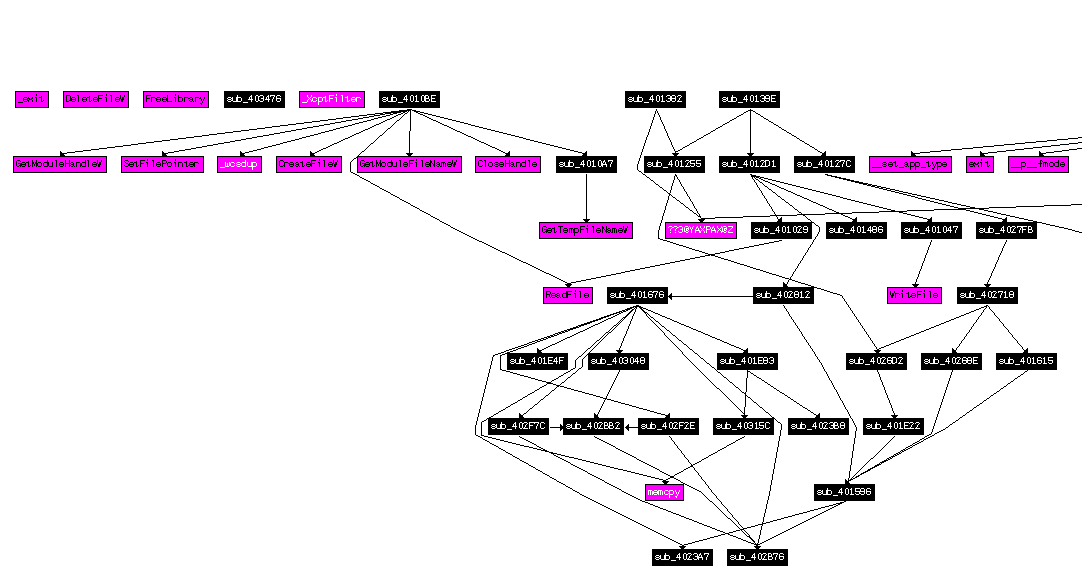
## Graph of Function Calls for Major Subroutine

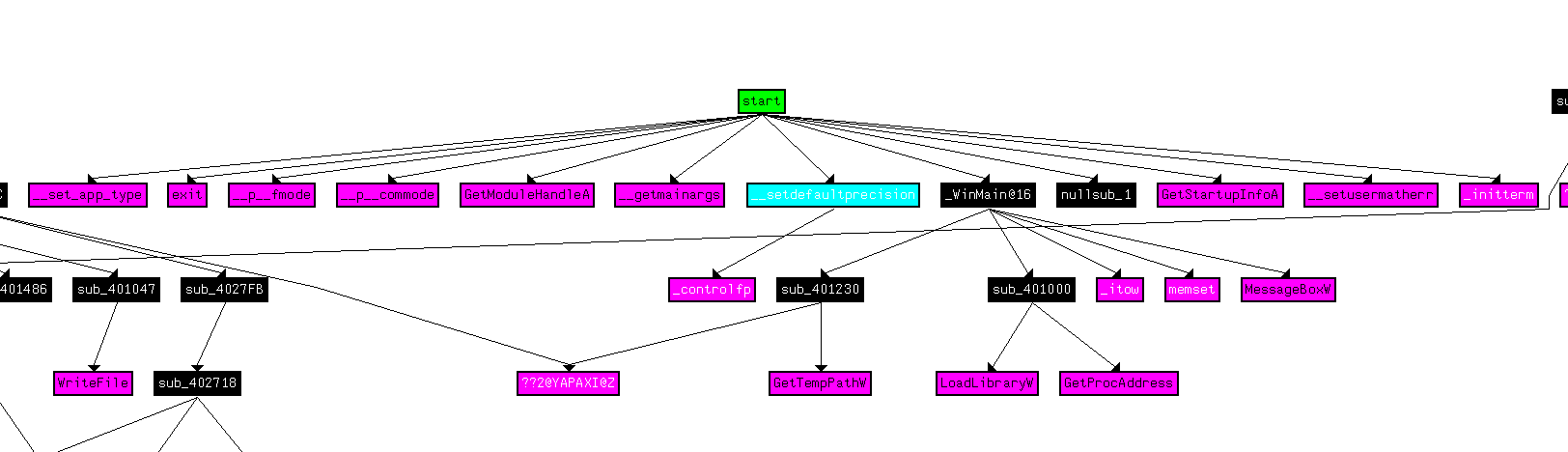
### Full graph

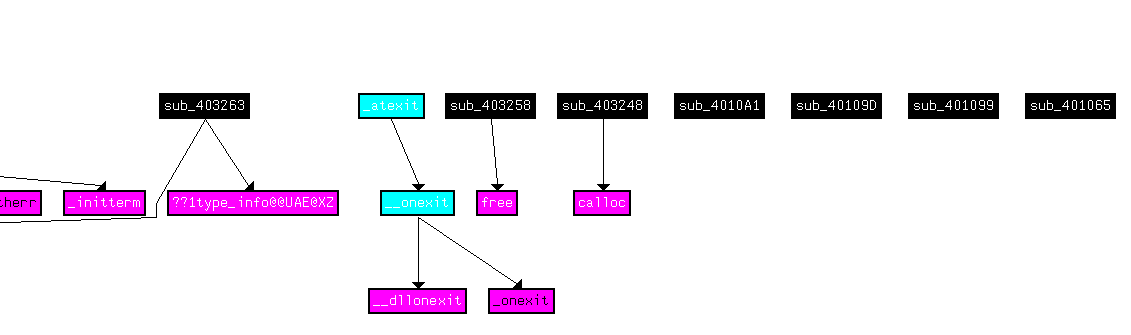


### Graph breakdown

The screenshots that follow are close-ups of the full graph. They are taken from left to right and are arranged as so. The most important and crucial part of the graph would be the second screenshot in the middle as that is where we are able to examine our major subroutine and the other subroutines and API that it is calling.







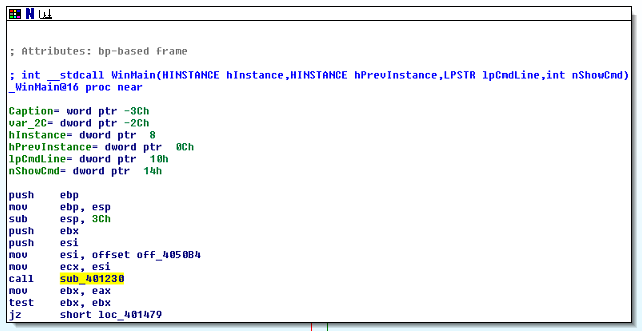
After analysing the screenshots, we can conclude that the major subroutine, **\_WinMain@16**, in Ardamax Keylogger calls out a total of 2 subroutines, namely **sub\_401230** and **sub\_401000.**

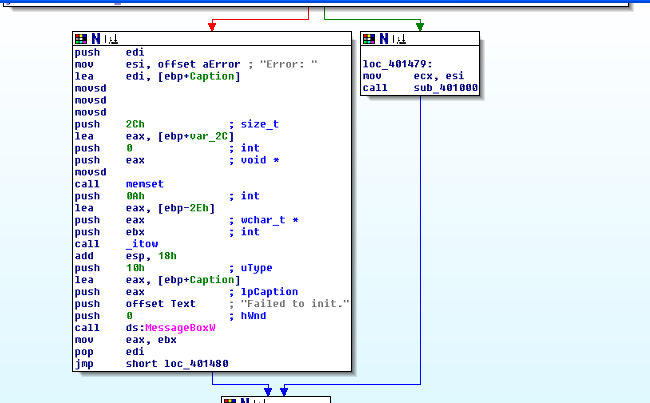
## Graph of Major Subroutines from Main Function

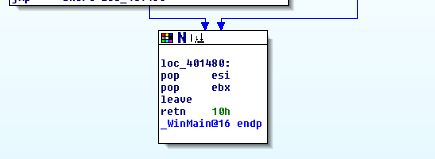
The Ardamax Keylogger’s major subroutine is **\_WinMain@16**. ‘WinMain’ is conventionally utilized in programs to label the application’s entry point. The **\_WinMain@16** subroutine is called by the **start** subroutine. The assembly code that shows this is as shown below.



In the following segment, we have included an image that visualizes the malware’s function calls. The graph below has allowed us to understand how the malware works and gives us an early understanding of how the subroutines are connected to one another. By using these graphs in our analysis, we can find out what other subroutines our major subroutine, **\_WinMain@16**, is utilizing. Using this knowledge, we are able to understand the malware at a deeper level.







## Purpose of Major Subroutine: \_WinMain@16

The major subroutine **\_WinMain@16** calls the **subroutine sub\_401230** [] at the beginning which will be explained in further detail in the other sections of the report. The **sub\_401230** subroutine would then store a value into one of the registers for later usage by the malware. It then performs a **test** instruction on the same register, ebx. [] The test instruction performs a **non-destructive AND operation** between the bits in the registers. Only the **Zero flag** can be affected through this instruction. If the result of this function causes the **Zero flag** to be set to a value of **1**, the malware will jump and proceed onto **loc\_401479**. [] The green arrow shows that the conditional jump was taken. Hence, confirming that the program would move on to **loc\_401479**.

**loc\_401479** would move the data from esi, which is most likely storing data that acts as a pointer, to ecx. The location would then call the subroutine **sub\_401000** [] which purpose would be explained and elaborated on in the following parts of the report. Following this, the program will take an unconditional jump, symbolised by the blue arrow, to proceed to **loc\_4014800** that ends the main **\_WinMain@16** subroutine.

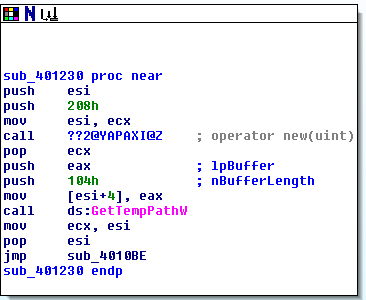
If, however, the **Zero Flag** is set to **0** in the earlier stage, it would not take the conditional jump and move towards the red arrow. At first glance, this location most likely displays an Error Message to the end user before halting the program. It calls **memset** [] which sets the first number of bytes of the block of memory pointed by the ptr to the specified value. It takes in the parameters **ptr**, **value** and **num** which values would be in **eax**, **0** and **2Ch** respectively. It then calls **\_itow** [] which converts an integer to a string. It takes in the parameters **value**, **size** and **radix** and their values would be **0Ah**, **eax** and **ebx** respectively. Additionally, it also receives a parameter called **buffer**. This function returns a pointer to the parameter **buffer** which is stored in **esp** which is the stack pointer for the system stack. This string may be storing the error message to be displayed through the **MessageBoxW** function. [] The **MessageBoxW** function that was called later on in the location displays a modal dialogue box that contains a system icon, a set of buttons and a brief application-specific message. When looking at this location’s context, it is likely that this **MessageBoxW** will be displaying a dialogue box to convey an error message. The program will then execute a jump instruction [] to location **loc\_401480** which is where the endpoint of the main function is and hence ends the **\_WinMain@16** subroutine.

At the endpoint of the main function, **loc\_401480**, the registers **esi** and **ebx** are restored using the **pop** instruction**.**

We have concluded that this subroutine’s purpose is to call other subroutines to execute the other functionalities of the malware as it mainly calls other subroutines and prints out a dialogue box with an error message if the program were to encounter an error. Thus, the **\_WinMain@16** subroutine deserves its title as the main function of the malware.

## Description of Subroutine

### Subroutine 1: sub\_401230



The subroutine **sub\_401230** uses the **mov** instruction to move the data in **ecx** back into the **esi** register and follows this by calling **??2@YAPAXI@Z**. [] After doing some research, we found that **??2@YAPAXI@Z** acts like the C++ function **operator new(uint)** allocates storage space while receiving the parameters **size\_t** that determines the size of the allocated space**.** The value of **size\_t** is presumably **208h** which was pushed right before **??2@YAPAXI@Z** was called. Looking at the flow of the program, the value that the API returns is stored in **eax** which would be utilised for the next API call. The malware then calls **GetTempPathW** which receives the parameters **nBufferLength** and **lpBuffer**. [] **GetTempPathW** retrieves the path of the directory designated for temporary files. As expected, the value in **eax**, which we have assumed to be the pointer to the string buffer, is stored in **[esi+4]**. The brackets [ ] around **esi+4** represents its memory contents. 4 is added to the value of **esi** and the resulting value is taken as an address. Hence, the values of **nBufferLength** and **lpBuffer** are **104h** and **[esi+4]** respectively. After moving the pointer value to the register and restoring the **esi** register, the location uses the **jmp** instruction to jump to the subroutine **sub\_4010BE.** This is where the endpoint of the subroutine is reached.

In summary, this subroutine is likely to be preparing for the creation of the files by reserving and allocating storage space in the computer through the APIs **??2@YAPAXI@Z** and **GetTempPathW** so that the malware be downloaded and executed in the user’s computer.

##### Sub\_401230 PARAMETER VALUES

We ran the malware in OllyDBG and analysed the values that were inputted into each function’s parameters in Sub\_401230. Note that some values may vary when the malware runs again in OllyDBG. Additional parameters may show up in the table as the value is not being pushed by the program and instead is automatically derived by the software.

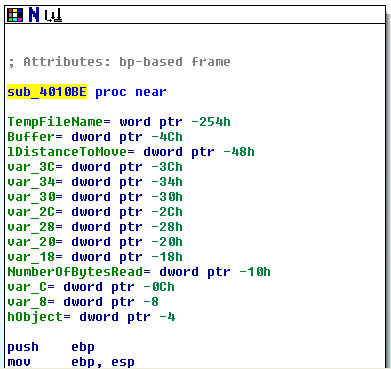
**??2@YAPAXI@Z:**

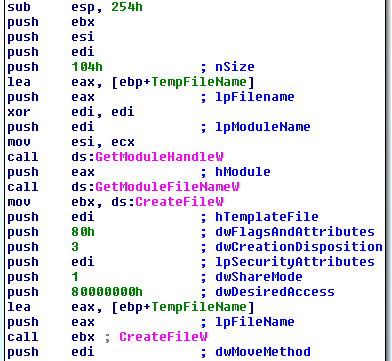
|  |  |
| --- | --- |
| **Parameter** | **Value** |
| size\_t | 208 |

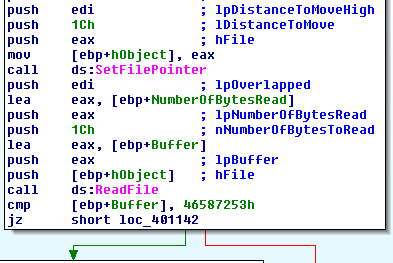
**GetTempPathW:**

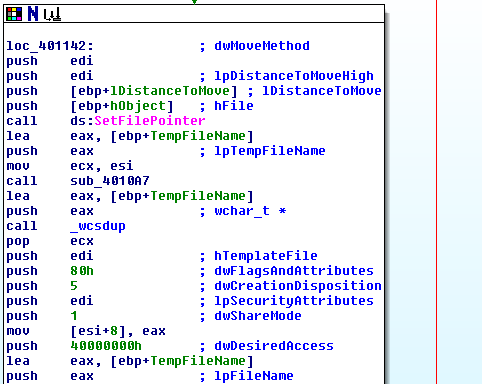
|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpBuffer | 008C0DC8 |
| nBufferLength | 104 |

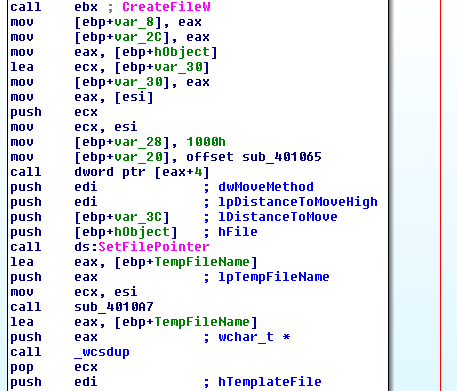
#### Subroutine 1.1: sub\_4010BE

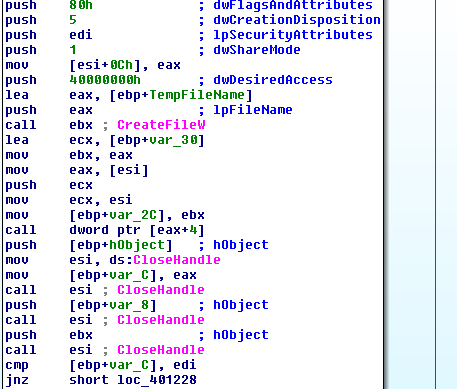


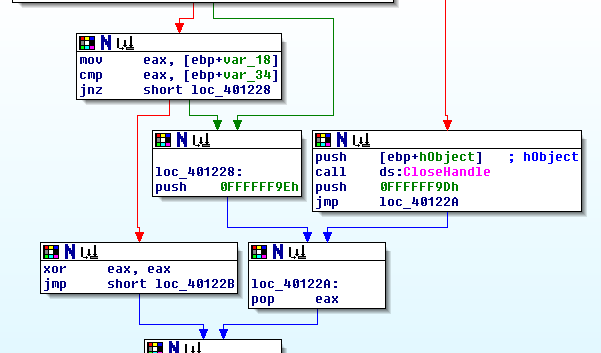


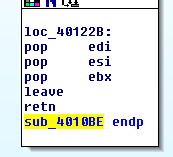






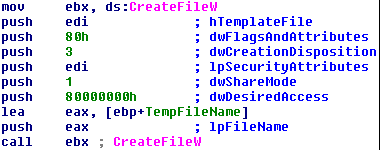


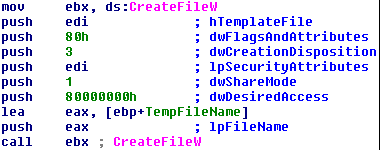




The subroutine \_sub4010BE calls the API **GetModuleHandleW** [] which retrieves a module handle for the specified module that is stored in the register **edi**. We know this because there is a comment beside the line “**push** **edi**” that states “**lpModuleName**”. [] **lpModuleName** is the parameter that **GetModuleHandleW** takes in. Hence, this parameter’s value is stored in the **edi** register. The module handle retrieved by **GetModuleHandleW** is then stored in **eax**. Next, the program proceeds to call **GetModuleFileNameW** [] that retrieves the fully qualified path for the file that contains the module that was taken. **GetModuleFileNameW** has the parameters **hModule**, **lpFilename** and **nSize**. We can conclude that the values of these parameters are stored in **eax**, **edi** and **104h** respectively. We reuse the value stored in the **edi** register in both APIs.

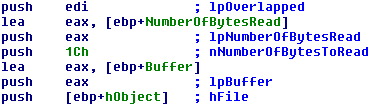
After executing the code and obtaining the module file name, the program proceeds to store the **CreateFileW** [] address to the **ebx** register. Before officially calling the function, the program pushes its respective parameters for it to be fed to the function. It takes in a total of 7 parameters, namely **lpFileName**, **dwDesiredAccess**, **dwShareMode**, **lpSecurityAttributes**, **dwCreationDisposition**, **dwFlagsAndAttributes** and **hTemplateFile**. Thankfully, the program has commented the different parameter values, giving us an easier time identifying which value belongs to which parameter. The screenshot for the parameters is shown below.



From the snippet above, we can conclude that the values for **hTemplateFile**, **dwFlagsAndAttributes**, **dwCreationDisposition**, **lpSecurityAttributes**, **dwShareMode**, **dwDesiredAccess** and **lpFileName** are stored in **edi**, **80h**, **3**, **edi**, **1**, **80000000h** and **eax**. The program calls the register **ebx**, which executes the **CreateFileW** function. [] The **CreateFileW** API creates or opens a file or I/O device. In this case, we can assume that this function was called to open the malware that has been installed in the computer. If the function is successful, it returns an open handle to the created file. However, if it fails, the value returned would be an **INVALID\_HANDLE\_VALUE**. The program proceeds to call the **SetFilePointer** function [] to move the file pointer of the recently created file. The parameters that it takes are **hFile**, **lDistanceToMove**, **lpDistanceToMoveHigh**, **dwMoveMethod.** Similarly, the program has labelled the respective parameters for us. The parameter values are as shown below.



Hence the values for **hFile**, **lpDistanceToMoveHigh**, **lDistanceToMove** and **dwMoveMethod** are stored in **edi, edi, 1Ch** and **eax** respectively**.** The **hFile** parameter that is stored in the **eax** register was moved to the address **[ebp+hObject].** []If the **SetFilePointer** function succeeds and **lpDistanceToMoveHigh** is NULL, the return value of the API is the low-order DWORD of the new file pointer. When the function fails, however, the return value would be set to **INVALID\_SET\_FILE\_POINTER**. In the event that the function succeeds, by setting the file pointer, the subroutine would be able to proceed to read the file by calling the **ReadFile** function. [] According to MDSN, **ReadFile**, like what the name suggests, reads data from the specified file and occurs at the position specified by the **SetFilePointer** function. Taking a look at what values are being pushed, we are easily able to identify the **ReadFile** parameters and its values.

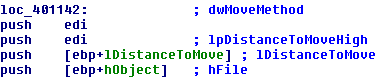


The parameters the function takes in are as follows: **lpOverlapped**, **lpNumberOfBytesRead**, **nNumberOfBytesToRead**, **lpBuffer** and **hFile**. The **lpOverlapped** parameter’s value is stored in **edi** which points to an OVERLAPPED structure if the **hFile** parameter was opened with FILE\_FLAG\_OVERLAPPED. Since this is not the case, the value stored in **edi** would be **NULL**. The **lpNumberOfBytesRead** parameter would have the value stored in **eax**, which stores the address location **[ebp+NumberOfBytesRead].** The **nNumberOfBytesToRead**, on the other hand, represents the maximum number of bytes to be read which is explicitly stated to be **1Ch**. The **lpBuffer** is similar to **lpNumberOfBytesRead** where the value that it stores is an address, as the program pushes **eax** which has the address location **[ebp+Buffer]** stored in it. The **hFile** parameter that was used before in **SetFilePointer** is reused again but pushes a different value, **[ebp+hObject]**, as the register **eax** is being utilised for the storing of other parameters.

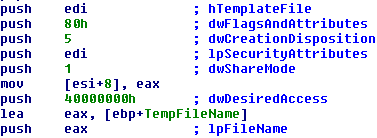
The program then carries out a **cmp** instruction that compares the values **46587253h** and **[ebp+Buffer]**. [] The **cmp** function is often used and is a nondestructive subtraction of source, **46587253h**, and destination, **[ebp+Buffer]**. It can affect the **Zero**, **Overflow**, **Carry** and **Sign** flags.

In this program, if the **Zero** **flag** is set to **0**, it would not take the conditional jump and go straight towards a location where it calls the **CloseHandle** function that closes an open object handle. This function is called to close the handle to the file that has just been opened which we would recognize as being stored in the **hFile** parameter. After closing the file’s handle, It jumps to the location **loc\_40122A** using the **jmp** instruction [] and restores the **eax** register using the **pop** instruction [], where the **hFile** parameter is stored. It proceeds to **loc\_40122B** where it restores the other registers **ebi**, **esi** and **ebx** before ending the subroutine.

If, however, the **Zero** **flag** is set to **1**, it would take the conditional jump and use the **jz** instruction to jump to the location **loc\_401142.** It then calls the **SetFilePointer** function a second time [] to move the file pointer of the malware’s file again. It receives the following parameters.



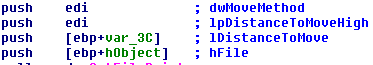
We know that the function moves the file pointer of the same file as the **hFile** parameter has the same value of the **hFile** parameter in the **ReadFile** function. This means that it is likely that the **hFile** parameter is pointing to a directory and the files needed for the malware’s installation are created in this directory. It then calls the subroutine **sub\_4010A7** to create a name for a temporary file. This subroutine would be explained in greater detail in the following section. The program then tries to duplicate a string, presumably the filename, by calling **\_wcsdup.** It receives the parameter **strSource** that is a null-terminated source string that returns a pointer to the duplicated string. The program then calls the **CreateFileW** function [] that creates another file inside the computer. It accepts the following parameters, where some have been reused from previous functions.



After creating the file with the temporary name gotten from the subroutine **sub\_4010A7**, it moves the **eax** register with the newly created file name in two separate memory addresses, **[ebp+var\_8]** and **[ebp+var\_2C]**. After storing the file names, the address **[ebp+hObject]** which holds the **hFile** parameter for the **SetFilePointer** function is stored in the **eax** register using the **mov** instruction.

The program then does a series of **mov** instructions that move different addresses into the different registers and pointers. The purpose of this is likely to move the location of the pointer so that the next file that is created would be in a different place inside the same directory.

The **SetFilePointer** function is then called for the second time []. The same process is repeated, similar to the way the program called the first **SetFilePointer** function earlier. The second **SetFilePointer** receives the following parameters with a different **lDistanceToMove** value.

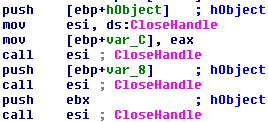


This **SetFilePointer** moves the file pointer to a different location inside the same directory. It calls the subroutine **sub\_4010A7** again to call out the **GetTempFileNameW** function which would allow the program to obtain another name for its second temporary file. This subroutine would store the temporary file name into the **eax** register and be used for the next function**, \_wcsdup**. It duplicates the filename again using the function **\_wcsdup** []. This function receives the parameter **strSource** that value is stored in the **eax** register as mentioned earlier.

Everything is now prepared for the creation of the next file. The program calls the **CreateFileW** API [] and the parameters and its respective values are shown below in the screenshot provided.

****

The program proceeds to go to the memory locations of the open object handles and calls the **CloseHandle** function a total of four times in order to close of them.



The **CloseHandle** function receives the parameter **hObject** that is a handle to an open object. The program moves the function to the **esi** register so that the program would only have to import the function once and continuously reference the same location instead of calling the function four different times. Once all the open handles are closed, the program carries out the **cmp** instruction that compares between the **edi** register and the value in the address **[ebp+var\_C]**.

If the Zero Flag is not set, meaning that it has a value of 0, it uses the **jnz** instruction to jump to the location **loc\_401228**. The **jnz** operand takes the conditional jump if the Zero flag has a value of 0 and does not take the jump when it is set to 1. This is the opposite of the **jz** operand that we have used previously. The location **loc\_401228** pushes a value of 0FFFFFF9Eh and proceeds to **loc\_40122A** that restores the **eax** register by executing a pop instruction. It then reaches the endpoint of the subroutine, **loc\_40122B**, and pops the **edi**, **esi** and **ebx** register before completing the subroutine **sub\_4010BE**.

On the other hand, if the Zero flag is set to 1, the program does a **mov** instruction that carries the **[ebp+var\_18]** address location to the eax register. It executes another **cmp** operand that compares between another address location, **[ebp+var\_34],** and the **eax** register. It carries out another **jnz** instruction that checks if the Zero flag is set or not.

If the Zero flag is not set, it jumps to the location **loc\_401228**, and pushes the same value 0FFFFFF9Eh. It then takes the same route as mentioned earlier where it restores the **eax**, **edi**, **esi** and **ebx** registers in order by going through the locations loc\_40122A and **loc\_40122B** before stopping the subroutine completely.

If the Zero flag is set, it performs the XOR instruction between the same register, eax and eax. [] The XOR instruction performs a Boolean exclusive-OR operation between each pair of matching bits in two operands. In this case, it is performing the XOR instruction on the same value. This would be mean that the program is setting the **eax** register to have a value 0. The program then jumps to the same location, **loc\_40122B**, where the registers **edi**, **esi** and **ebx** are restored. This ends the subroutine.

In conclusion, after the analysis of this malware, I have concluded that the subroutine **sub\_4010BE** was meant to create the temporary files that the malware installer requires in order to install the keylogger into the system. It opens the original Ardamax Keylogger file and reads it, obtaining the information of what files are needed to be created in order to successfully execute the installer. After **sub\_4010BE** finishes, it goes back to the subroutine **sub\_401230** and ends it. It then goes back to the main subroutine **\_WinMain@16** to execute the remaining half of that subroutine.

##### Sub\_4010BE PARAMETER VALUES

We ran the malware in OllyDBG and analysed the values that were inputted into each function’s parameters in Sub\_4010BE. Note that some values may vary when the malware runs again in OllyDBG such as the filename that was created. Additional parameters may show up in the table as the value is not being pushed by the program and instead is automatically derived by the software.

**GetModuleHandleW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpModuleName | NULL |

**GetModuleFileHandleW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpFilename | 00400000 |
| hModule | NULL |
| nSize | 104 |
| pathBuffer | 0019FC34 |

**CreateFileW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hTemplateFile | NULL |
| dwFlagsAndAttributes | Normal |
| dwCreationDisposition | OPEN\_EXISTING |
| lpSecurityAttributes | NULL |
| dwShareMode | FILE\_SHARE\_READ |
| dwDesiredAccess | GENERIC\_READ |
| lpFileName | C:\Users\Alan\Desktop\ArdamaxKeylogger\_E33AF9E602CBB7AC3634C2608150DD18.exe |

**CreateFileW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hTemplateFile | NULL |
| dwFlagsAndAttributes | Normal |
| dwCreationDisposition | TRUNCATE\_EXISTING |
| lpSecurityAttributes | NULL |
| dwShareMode | FILE\_SHARE\_READ |
| dwDesiredAccess | GENERIC\_WRITE |
| lpFileName | C:\Users\Alan\AppData\Local\Temp\@B080.tmp |

**CreateFileW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hTemplateFile | NULL |
| dwFlagsAndAttributes | Normal |
| dwCreationDisposition | TRUNCATE\_EXISTING |
| lpSecurityAttributes | NULL |
| dwShareMode | FILE\_SHARE\_READ |
| dwDesiredAccess | GENERIC\_WRITE |
| lpFileName | C:\Users\Alan\AppData\Local\Temp\@2626.tmp |

**SetFilePointer:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| dwMoveMethod | FILE\_BEGIN |
| lpDistanceToMoveHigh | NULL |
| lDistanceToMove | 1C |
| hFile | 0000010C |

**SetFilePointer:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| dwMoveMethod | FILE\_BEGIN |
| lpDistanceToMoveHigh | NULL |
| lDistanceToMove | 3A00 |
| hFile | 000000B8 |

**SetFilePointer:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| dwMoveMethod | FILE\_BEGIN |
| lpDistanceToMoveHigh | NULL |
| lDistanceToMove | 3ED4 |
| hFile | 000000B8 |

**ReadFile:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpOverlapped | NULL |
| lpNumberOfBytesRead | 0019FE78 |
| nNumberOfBytesToRead | 1C |
| lpBuffer | 0019FE3C |
| hFile | 0000010C |

**CloseHandle:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hObject | 0000010C |

**SetFilePointer:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| dwMoveMethod | FILE\_BEGIN |
| lpDistanceToMoveHigh | NULL |
| lDistanceToMove | 3A00 |
| hFile | 0000010C |

**\_wcsdup:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| strSource | C:\Users\Alan\AppData\Local\Temp\@6261.tmp |

**CloseHandle:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hObject | 000000E4 |

**CloseHandle:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hObject | 000000E4 |

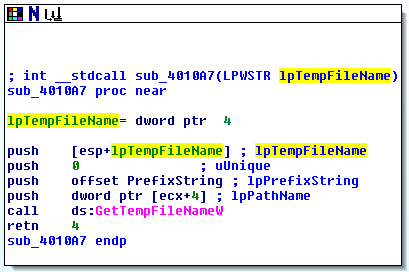
**CloseHandle:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hObject | 000000E8 |

**CloseHandle:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| hObject | 000000EC |

#### Subroutine 1.2: sub\_4010A7



This subroutine’s main purpose is to call out the **GetTempFileNameW** function. It is likely that this function is called out multiple times with the exact same parameters in other code segments, hence the need for this to belong to a single subroutine. The function **GetTempFileNameW** receives the parameters **lpTempFileName**, **uUnique**, **lpPrefixString** and **lpPathName**. The values would be stored in **[esp+lpTempFileName], 0, offset PrefixString and dword ptr [ecx+4]** respectively. If this function is executed successfully, the return value specifies the unique numeric value used in the temporary file name. The “retn 4” instruction [] carries out the **return** instruction and moves the stack pointer 4 more bytes. We found that this is typically used by methods with calling conventions where it is the callee’s to remove passed arguments from the stack. It then reaches the endpoint of the subroutine and carries over to the subroutine **sub\_4010BE**.

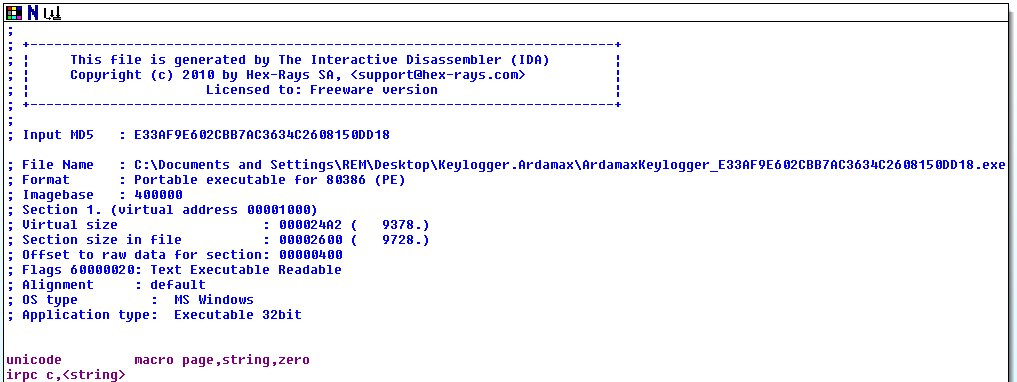
##### Sub\_4010A7 Parameter values

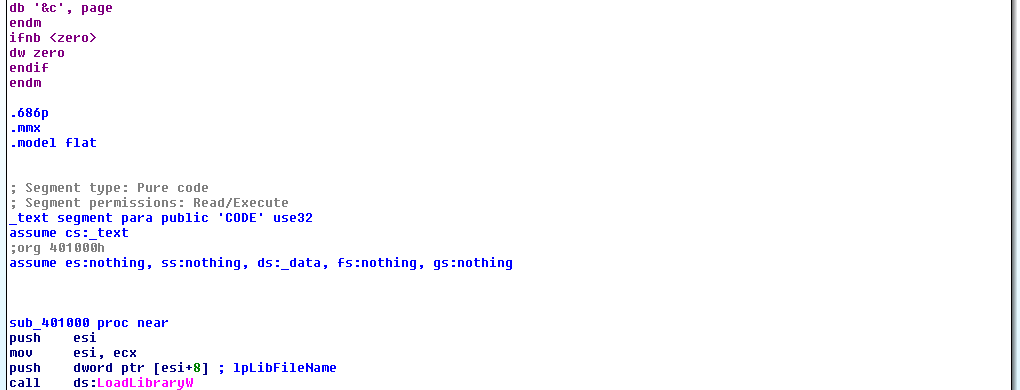
We ran the malware in OllyDBG and analysed the values that were inputted into each function’s parameters in Sub\_4010A7. Note that some values may vary when the malware runs again in OllyDBG (such as the filename of a deleted file). Additional parameters may show up in the table as the value is not being pushed by the program and instead is automatically derived by the software.

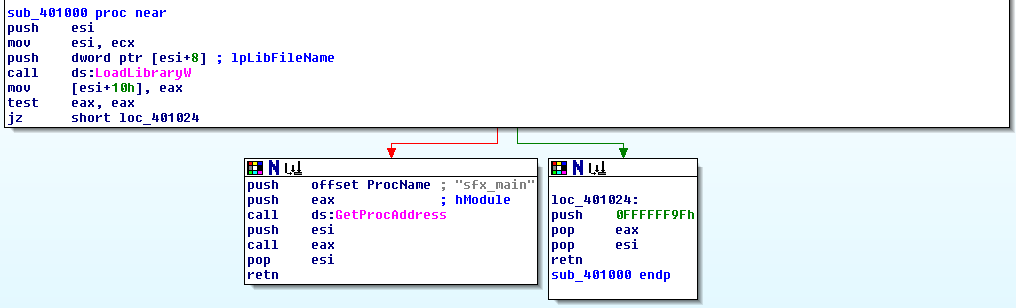
**GetTempFileNameW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpTempFileName | 0019FC34 |
| uUnique | 0 |
| lpPrefixString | “@” |
| lpPathName | C:\Users\Alan\AppData\Local\Temp\ |

### Subroutine 2: sub\_401000







After the setting up of the keylogger on the user’s computer, the subroutine sub\_401000 is called. This subroutine calls out the **LoadLibraryW** function [] that loads the specified module into the address space of the calling process. The return value of this function is a handle to the specified module. It is likely that it loads the handle into the **eax** register as the program does a **mov** instruction, moving the contents from **eax** to the address **[esi+10h]**. [] The **LoadLibraryW** function takes in only one parameter, **lbLibFileName** that has a value of **dword ptr [esi+8].** The parameter is the name of the module which would be loaded by the function.

The program then carries out the test instruction between the same register, **eax**, which can set the Zero flag. This sets a condition for the next jump which utilises the “**jz**” instruction to decide where the data flows.

If the **Zero flag** is set to **1**, the program would jump to the location **loc\_40124** and push a value of 0FFFFFF9Fh before restoring the registers **eax** and **esi**. This location represents the endpoint of the subroutine and the subroutine would stop at this point.

However, if the **Zero flag** is set to **0**, the function **GetProcAddress** is called which retrieves the address of an exported function or variable from the specified dynamic-link library (DLL). It takes the parameters **hModule** and **lpProcName**. **hModule** is the handle to the DLL module that contains the function or variable while the **lpProcName** is the function or variable name. We can infer that the parameters are trying to reference a function as there is a comment beside the pushing of the **ProcName** parameter, stating “sfx\_main”. This could be a main function, hence the conclusion. The values of these parameters are stored in the **eax** register which contained the handle generated by the **LoadLibraryW** function and the **offset** of **ProcName** respectively. The return value of **GetProcAddress** is the address of the exported function. We can assume that the return value is stored in the **eax** register as it is called later on in the location. This calls the exported function and allows it to be executed. Once the execution has been completed, the subroutine reaches its endpoint.

The purpose of this subroutine would be to load and execute the exported **sfx\_main** function from the DLL file which is loaded and obtained by the **LoadLibraryW** and **GetProcAddress** functions.

##### Sub\_401000 parameter values

We ran the malware in OllyDBG and analysed the values that were inputted into each function’s parameters in Sub\_401000. Note that some values may vary when the malware runs again in OllyDBG. Additional parameters may show up in the table as the value is not being pushed by the program and instead is automatically derived by the software.

**LoadLibraryW:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpLibFileName | C:\Users\Alan\AppData\Local\Temp\@DAF.tmp |

**GetProcAddress:**

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| lpProcName | sfx\_main |
| hModule | 10000000 |

# Patching

We utilized OllyDBG to patch the malware, Ardamax Keylogger. Our main objective in this patching process is to compile and edit the malware in such a way that it prevents the file Ardamax Keylogger, which is the malware’s installer, from installing the keylogger into the computer.

## Main Routine

### JE short loc\_401479

**Before:**



**Patched:**

**This make the malware installer show an error message that allow user to know the installation failed**



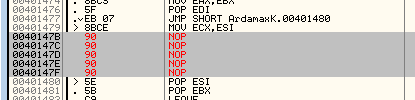
### CALL sub\_401000

**Before:**



**Patched:**

**This disable the malware installer from installing the malware to the infected computer**



## Subroutine: sub\_401000

### GetProcAddress

**Before:**



**Patched:**

**This disable the malware installer from getting GetProcAddress**



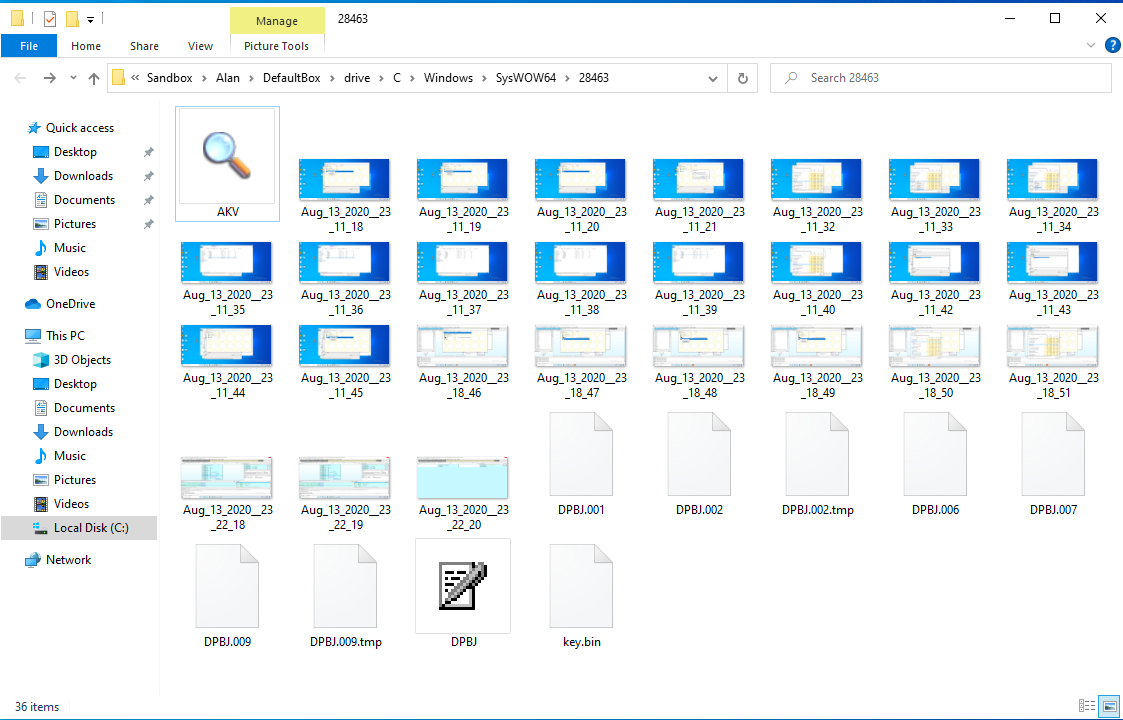
# General Analysis

## Type of Malware

Through our analysis and research, we have come to the conclusion that the malware is a type of keylogger. Keyloggers are a type of monitoring software that are specifically designed to record keystrokes made by the user. In the malware’s context, if the malware infects your computer, the keylogger would be able to keep track of your keystrokes when typing in important information like the user’s banking account details which can be used to sell or use for profit.

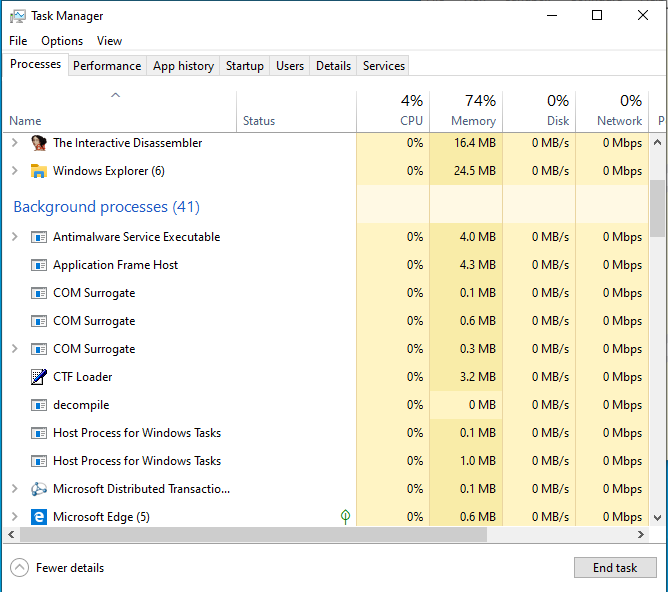
## Functionality of Malware

This malware has the ability to open, write, delete and copy files in the infected computer. The malware in the zip file, Ardamax Keylogger, is the malware installer to install the keylogger into the system. The main malware file is installed into the computer using the program in the zip file.



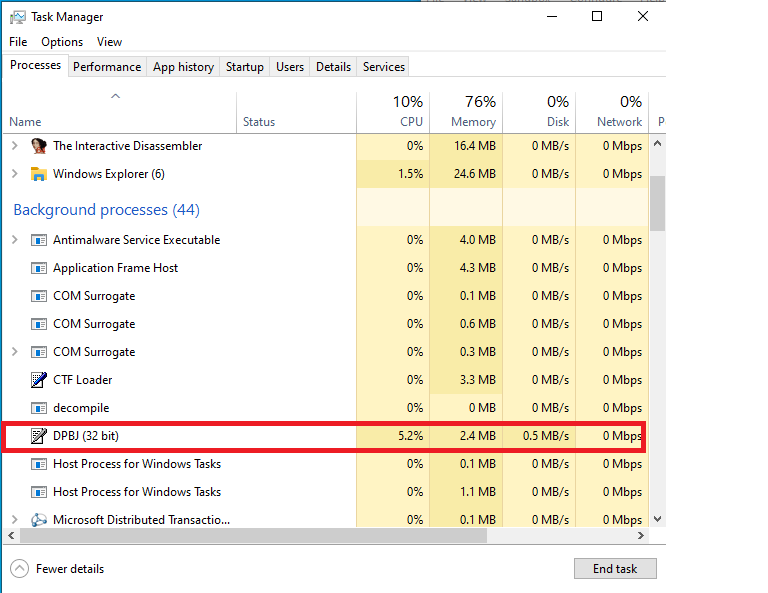
When we ran the malware for the very first time, there was no evidence of an application being opened. This means that the malware, when executed, does not give the user any visual stimuli to indicate that it is running. Hence, we can infer that it is an underlying process.

Before we ran the malware, the task manager looked like the screenshot below.

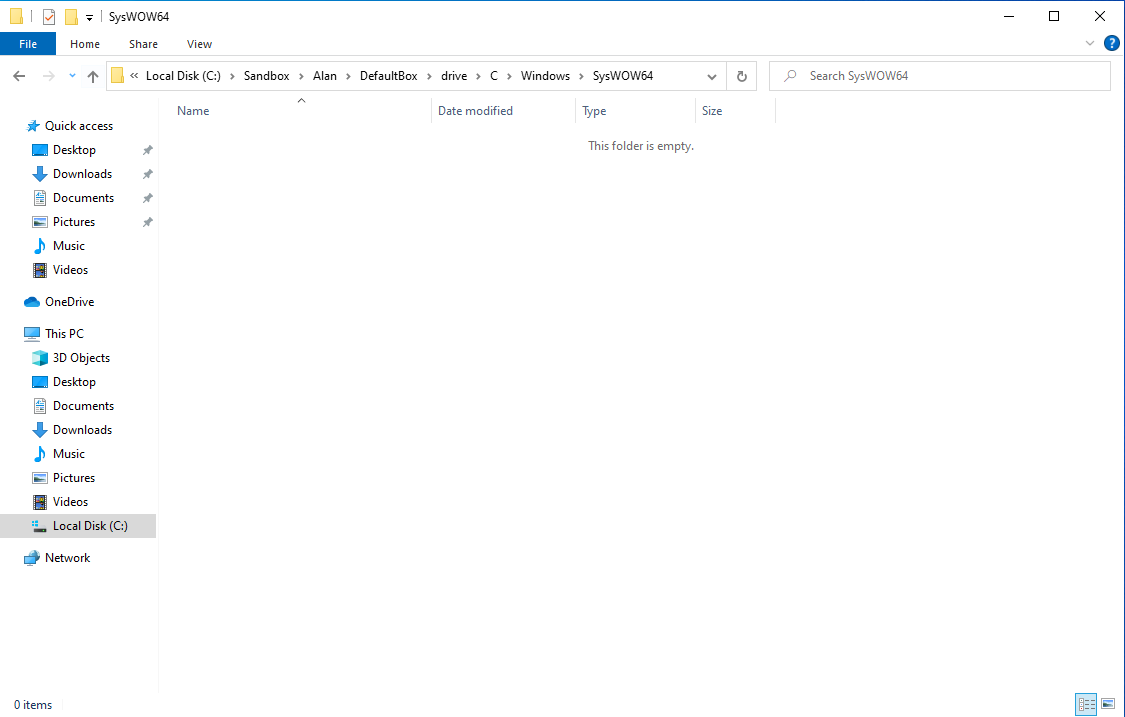


When looking further into the task manager to look at the hidden processes, we took notice of the unknown application that was running in the background. The process that was running was “DPBJ.exe”.

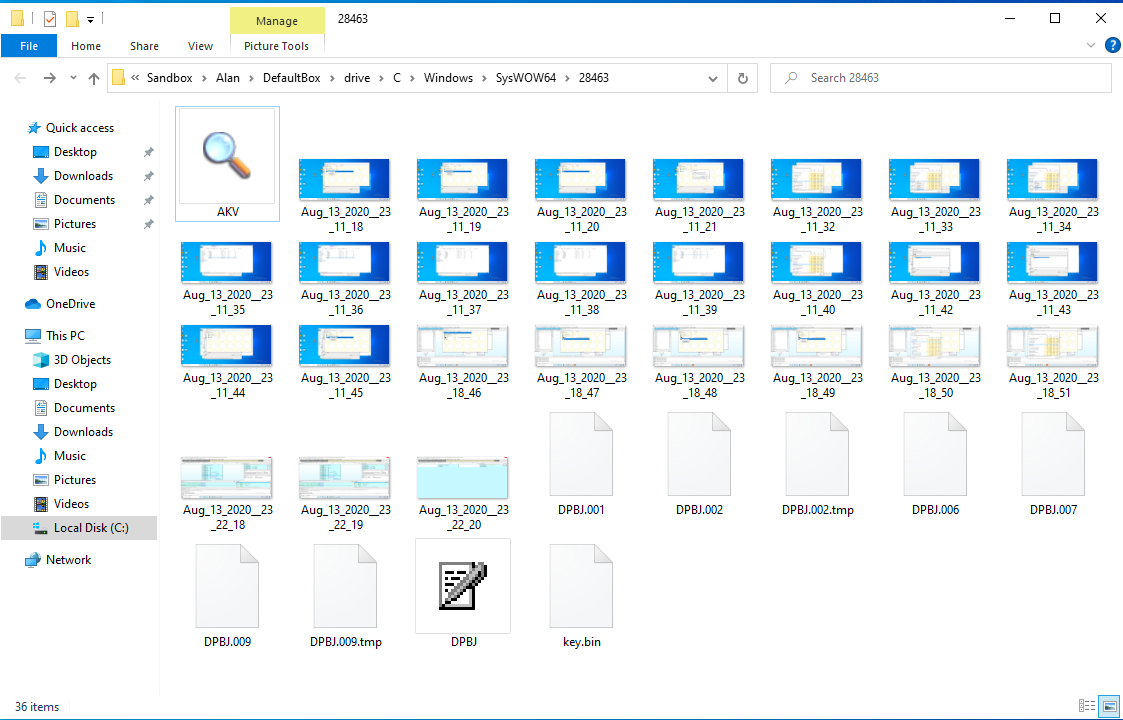
We can see this in the following screenshot which has been displayed below.



In order to check where this process was being stored in the computer, we right clicked the process and selected the “Open File Location” option. This directed us to a hidden folder that was created in ‘C:/Windows/sysWOW64”. We did another round to check what was inside this folder before we ran the malware. The screenshot below shows what was inside this file before we executed the Ardamax Keylogger installer.

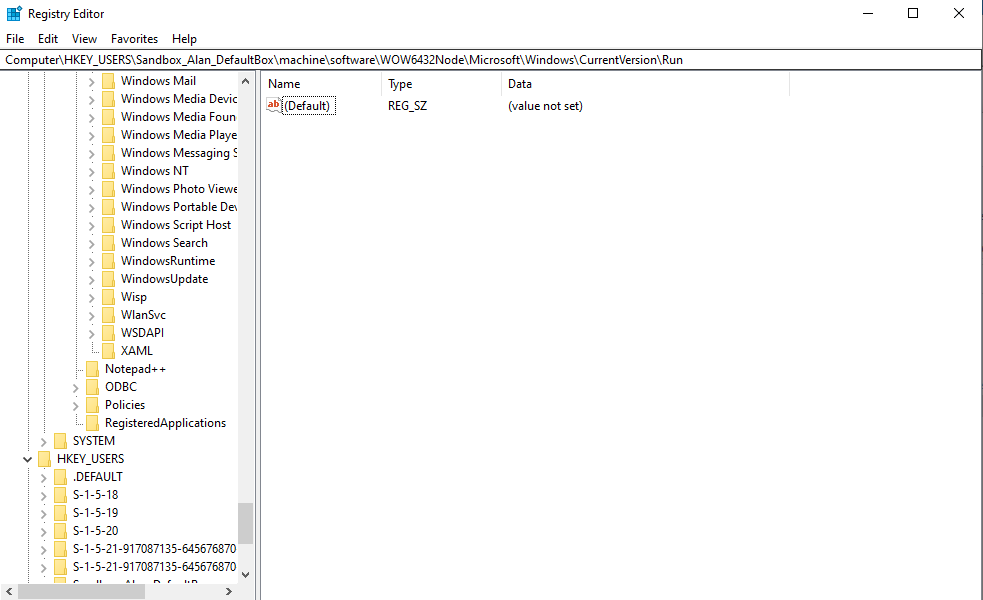


As we can see, there is nothing inside the folder. However, after the malware is executed, a directory ‘28643’ was created and contents were added into the folder. The folder would contain the files in the screenshot.

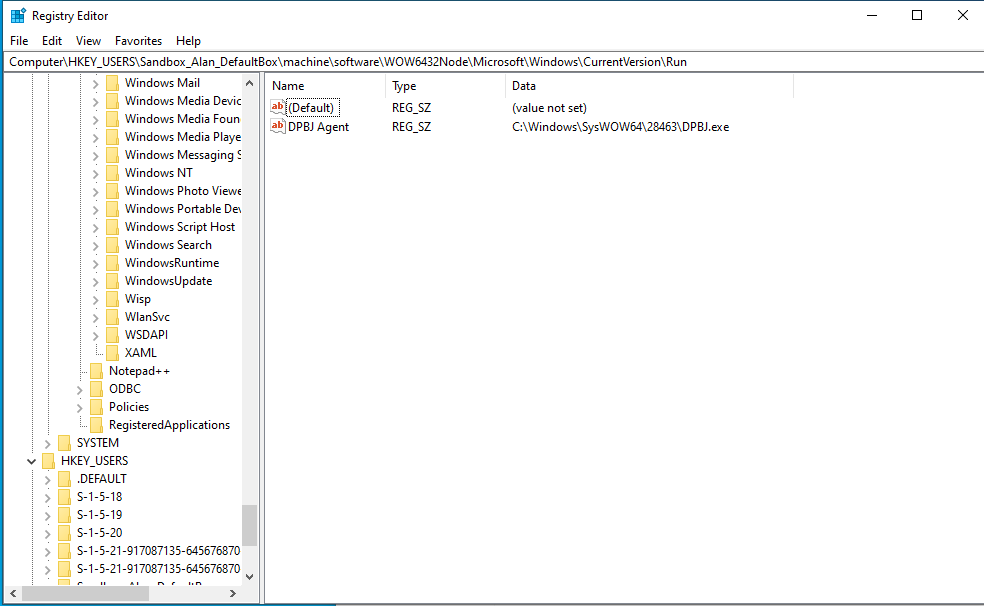


This included 2 separate executed files, inside including many temp files as well as screenshots of the system that we did not know were being captured. We can hence conclude that the malware is able to take screenshots of the system without the user’s knowledge. Furthermore, the screenshots are taken quite frequently. This malware would then be able to trace what the user is doing by analysing the screenshots.

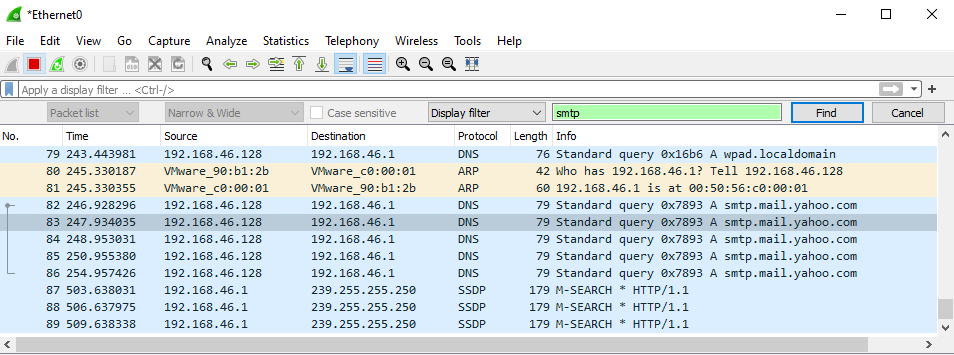
Additionally, the malware creates a registry key in the HKEY\LOCAL\_MACHINE\SOFTWARE registry. Before the malware was created, the registry looked like the screenshot below.



However, after the malware was executed, the registry looked like the screenshot below.



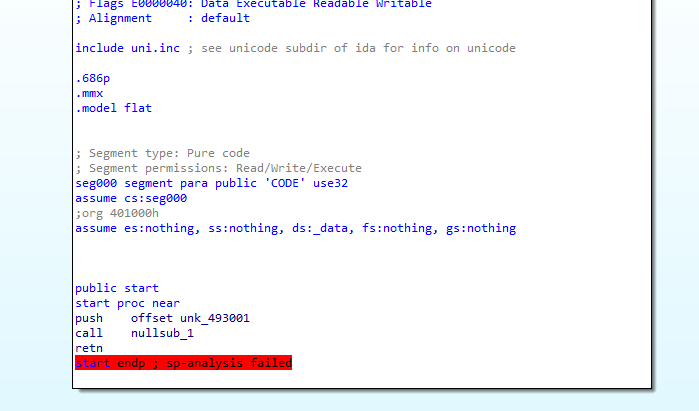
This registry key allows the malware to be executed when the user starts up the system without needing to meet any pre-existing conditions such as the user providing the malware with permission to execute by clicking on a given icon.



Utilising Wireshark, a network packet analyser, we were able to deduce that the malware sends the taken screenshots to a Yahoo email address every 2 minutes. Once all the screenshots have been sent, the malware deletes all the screenshots in the folder and sleeps for a minute.

## Interaction with the Malware

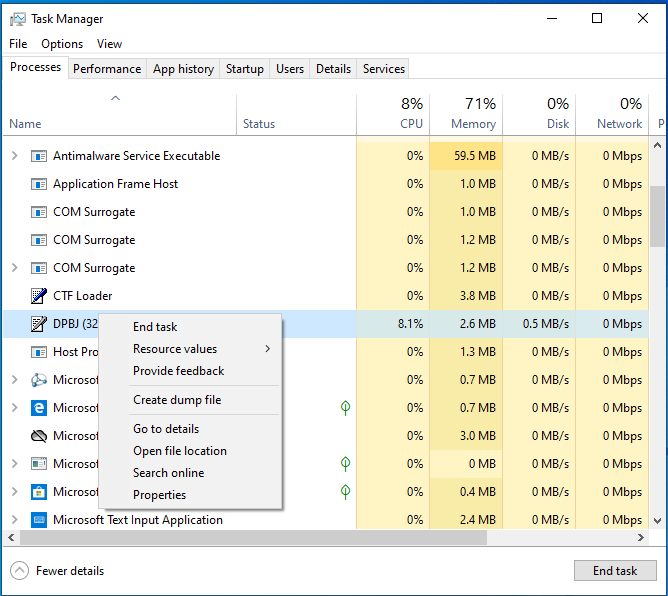
Unfortunately, we were not able to interact with the malware. The malware does not provide any sort of user interface for the user to interact with the program. This is to be expected of a keylogger as in order for the malware to successfully achieve its goal of stealing user information, it is imperative that the user does not know of its existence. Since there was no interface, we ran the malware through OllyDBG and made full use of the step over feature. The tool allowed us to discover the contents and inner workings of the malware, the Ardamax Keylogger. However, we were only able to analyse the malware installer which was included in the Ardamax.Keylogger.zip but not the main malware file that was created, DBPJ.exe. The main malware file, DBPJ.exe, that was created by the malware installer was encrypted.



Hence, IDAPro and OllyDBG was not able to access and analyse the file. We were however, able to use Sandboxie. Sandboxie is an open-source sandboxing program for Microsoft Windows. Its main goal is to create an isolated operating environment that can run or install applications without modifying the host system. This allows us to analyse what exactly the DBPJ executable file is doing to the system when it is executed. Hence, allowing us to discover the location of the keylogger and obtain the registry key set by the malware.

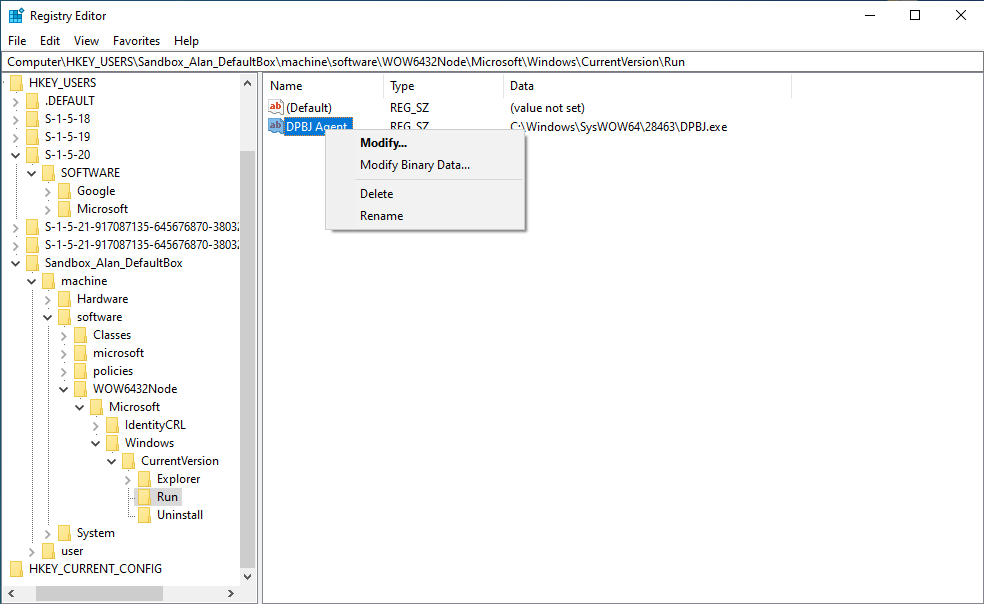
## Removal of Malware

Eliminating this malware from the infected computer can be easy with pre-existing knowledge of the software. To remove the malware from the infected computer, we can open the task manager to find the malware file, DPBJ.exe. We would then right click on the DPBJ.exe process and select the option “Open File Location”. This redirects us to the directory which contains the malware file.



Once we are able to access the directory that the malware is located in, we return to the Task Manager window and right click DBPJ.exe. This allows us to select “End task” which stops the malware from running. This will prevent the malware from performing any keylogging, taking screenshots of the victim’s computer, and allows us to delete all the files that were created by the malware in the file explorer. We will proceed by deleting all the files in the previously opened folder.

We will then open the registry editor in order to delete the created registry key. Go to HKEY\_LOCAL\_MACHINE\software\WOW643332Node\microsoft\Windows\CurrentVersion\Run. Delete the DPBJ Agent by right clicking it and selecting the “Delete” option. An example of this is in the screenshot below.



Once these steps have been completed, the malware has been successfully eliminated from the computer in its entirety.