WEEK 1 – Chapter 0: Malware Analysis Primer from ‘Practical Malware Analysis’

The Goal of Malware Analysis:

To provide the information you need to respond to a network intrusion. Your goals will typically be to determine exactly what happened, and to ensure that you’ve located all infected machines and files. When analysing suspected malware, your goal will typically be to determine exactly what a particular suspect binary can do, how to detect it on your network, and how to measure and contain its damage.

Once you identify which files require full analysis, it’s time to develop signatures to detect malware infections on your network. As you’ll learn throughout this book, malware analysis can be used to develop **host-based** and **network signatures**.

***Host-based Signatures***

Host-based signatures/indicators are used to **detect** **malicious** **code** on victim computers. These indicators identify files created or modified by the malware or specific changes that it makes to the registry. Unlike antivirus signatures, **malware indicators focus on what the malware does to a system, not on the characteristics of the malware itself**, which makes them **more** **effective** in **detecting** **malware** that **changes** **form** or that has been **deleted** from the **hard** **disk**.

***Network Signatures***

Network signatures are used to **detect** **malicious** **code** by **monitoring** **network** **traffic**. Network signatures can be created without malware analysis, but signatures created with the help of malware analysis are usually far more effective, offering a **higher detection rate** and **fewer false positives**. After obtaining the signatures, the final objective is to figure out exactly **how** the **malware** **works**. This is often the most asked question by senior management, who want a full explanation of a major intrusion. The in-depth techniques you’ll learn in this book will allow you to determine the purpose and capabilities of malicious programs.

Malware Analysis Techniques

Most often, when performing malware analysis, you’ll have **only the malware executable**, which **won’t** be **human-readable**. In order to make sense of it, you’ll use a variety of tools and tricks, each revealing a small amount of information. You’ll need to use a variety of tools in order to see the full picture. There are two fundamental approaches to malware analysis: **static** and **dynamic**.

**Static analysis:** Involves **examining** the **malware** **without** **running** it.

**Dynamic analysis:** Involves **running** the **malware**.

Both techniques are further categorized as **basic** or **advanced**.

***Basic Static Analysis***

Basic static analysis consists of **examining** the executable file **without** **viewing** the **actual** **instructions**.

✓ Pros

Basic static analysis **can** **confirm** whether a file is **malicious**, provide **information** **about** its **functionality**, and sometimes **provide** **information** that will **allow** you to produce **simple** **network** **signatures**. Basic static analysis is **straightforward** and can be **quick**.

🗶 Cons

It’s largely **ineffective** **against** **sophisticated** **malware**, and it **can** **miss** **important** **behaviours**.

***Advanced Static Analysis***

Advanced static analysis consists of **reverse-engineering the malware’s internals** by **loading the executable into a disassembler** and **looking at the program instructions** in order to **discover** what the **program** **does**.

✓ Pros

The **instructions** are **executed** **by** the **CPU**, so advanced static analysis tells you **exactly** **what** the **program** **does**.

🗶 Cons

However, advanced static analysis has a **steeper learning curve** than basic static analysis and **requires specialized knowledge of disassembly**, **code** **constructs**, and **Windows operating system concepts**.

***Advanced Dynamic Analysis***

Advanced dynamic analysis **uses** a **debugger** to **examine** the **internal** **state** of a **running** **malicious** **executable**. Advanced dynamic analysis techniques **provide** **another** **way** to **extract** detailed **information** from an **executable**.

✓ Pros

These techniques are **most** **useful** when you’re trying to **obtain** **information** that is **difficult** to **gather** with the **other** **techniques**.

***Basic Dynamic Analysis***

Basic dynamic analysis techniques involve **running** the **malware** and **observing** its **behaviour** on the system in order **to** **remove** the **infection**, **produce** effective **signatures**, or both.

However, before you can run **malware** safely, you must **set** **up** an **environment** that will **allow** you to **study** the **running** **malware** **without** risk of **damage** to your system or network.

✓ Pros

Like basic static analysis techniques, basic dynamic analysis techniques can be **used** **by most people** **without** **deep** **programming** **knowledge**.

🗶 Cons

**Won’t** be **effective** with **all** **malware** and can **miss** **important** **functionality**.

Types of Malware:

BACKDOOR Malicious code that **installs** **itself** **onto** a **computer** to **allow** the attacker **access**. Backdoors usually **let** the **attacker** **connect** to the **computer** with **little** or **no** **authentication** and **execute** **commands** on the local **system**.

BOTNET Similar to a backdoor, in that it **allows** the **attacker** **access** to the **system**, but **all** **computers** **infected** with the **same** **botnet** **receive** the **same** **instructions** from a **single** **command**-**and**-**control** **server**.

DOWNLOADER **Malicious** **code** that **exists** **only** to **download** **other** **malicious** **code**. Downloaders are commonly **installed** **by** **attackers** when they **first gain access** to a **system**. The **downloader program** will **download** and **install** **additional** **malicious** **code**.

INFORMATION-STEALING MALWAREMalware that **collects** **information** from a **victim’s** **computer** and usually **sends** it **to** the **attacker**. Examples include **sniffers**, **password** **hash** **grabbers**, and **keyloggers**. This malware is typically **used** to gain **access** to **online** **accounts** such as **email** or **online** **banking**.

LAUNCHER Malicious program **used** to **launch** **other** **malicious** **programs**. Usually, launchers use **non**-**traditional** **techniques** to **launch** other **malicious** **programs** in order **to** **ensure** **stealth** or **greater** **access** to a **system**.

ROOTKIT Malicious code **designed** to **conceal** the **existence** of **other** **code**. Rootkits are usually **paired** with other **malware**, such as a **backdoor**, to **allow** **remote** **access** to the attacker and **make** the **code** **difficult** for the **victim** to **detect**.

SCAREWARE Malware designed to **frighten** an infected **user** **into** **buying** **something**. It usually has a **user** **interface** that makes it **look** **like** an **antivirus** or other **security** **program**. It **informs** **users** that there is **malicious** **code** on their **system** and that the only way to get rid of it is to **buy** **their** “**software**,” when in reality, the **software** it’s **selling** **does** **nothing** **more** than **remove** the **scareware**.

SPAM-SENDING MALWARE Malware that **infects** a user’s **machine** and then **uses** that **machine** to **send** **spam**. This malware **generates** **income** for **attackers** by **allowing** them to **sell** **spam**-**sending** **services**.

WORM OR VIRUS Malicious **code** that can **copy** **itself** and **infect** **additional** **computers**.

*Malware often spans to multiple categories:*

A program might have a **keylogger** that **collects** **passwords** and a **worm** **component** that **sends** **spam**.

Malware can also be classified based on whether the **attacker’s** **objective** is **mass** or **targeted**. **Mass malware**, such as **scareware**, takes the **shotgun** **approach** and is **designed** to **affect** as **many** **machines** as possible. One of the two objectives, the **most** **common** and **usually** the **less** **sophisticated** and **easier** to **detect** and **defend** **against** **because** **security** **software** **targets** **it**.

Targeted malware, like a **one-of-a-kind backdoor**, is **tailored** **to** a **specific** **organisation**. **Targeted** **malware** is a **bigger** **threat** to **networks** than **mass** **malware**, it is **nearly** **impossible** to **protect** your **network** against that **malware** and to **remove** **infections**. Targeted malware is **usually** **very** **sophisticated**, and analysis will often require **advanced** analysis **skills**.

WEEK 1 – Chapter 1: Basic Static Techniques from ‘Practical Malware Analysis’

Antivirus Scanning: A Useful First Step

When first analysing prospective malware, a good first step is to run it through multiple antivirus programs, which may already have identified it. But antivirus tools are certainly not perfect.

They rely mainly on a **database** of **identifiable** **pieces** of **known** **suspicious** **code** (*file signatures*), as well as **behavioural** and **pattern**-**matching** **analysis** (*heuristics*) to **identify** **suspect** **files**.

One problem is that **malware** **writers** can **easily** **modify** their **code**, thereby changing their program’s **signature** and **evading** **virus** **scanners**.

Also, rare malware often goes **undetected** by antivirus software because it’s **simply** **not** **in** the **database**. Finally, **heuristics**, while often successful in identifying unknown malicious code, can be **bypassed** **by** **new** and **unique** **malware**.

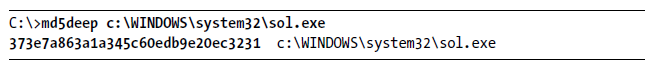
Hashing: A Fingerprint for Malware

***Hashing***is a common **method** used **to** **uniquely identify malware**. The malicious software is run **through** a **hashing** **program** that **produces** a **unique** **hash** that **identifies** that **malware** (*a sort of fingerprint*).

The **Message-Digest Algorithm (MD5)** hash function is the one **most** **commonly** **used** for **malware** **analysis**, though the **Secure Hash Algorithm 1 (SHA-1)** is also popular.

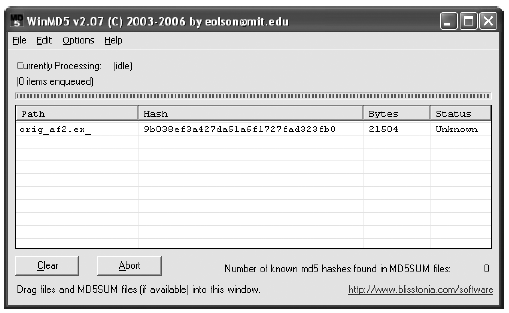
md5deep

We can use the freely available md5deep program to calculate the hash of the Solitaire program that comes with Windows:



The hash is the numerical number above.

The GUI-based WinMD5 calculator, shown in Figure 1-1, can calculate and display hashes for several files at a time:



Once you have a unique hash for a piece of malware, you can use it as follows:

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

Finding Strings

A *string* in a program is a **sequence** of **characters** such as “*the*”. A program **contains** **strings** **if** it **prints** a **message**, **connects** to a **URL**, or **copies** a **file** to a **specific** **location**.

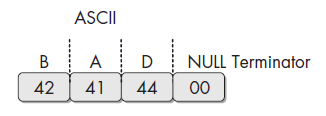
Searching through the strings can be a simple way to get hints about the functionality of a program. For example, if the program accesses a URL, then you will see the URL accessed **stored** as a **string** in the **program**. You can use the Strings program (http://bit.ly/ic4plL), to search an executable for strings, which are **typically** **stored** in either **ASCII** or **Unicode** **format**.

Both **ASCII** and **Unicode** **formats** **store** **characters** **in** **sequences** that **end** with a **NULL** **terminator** to **indicate** that the **string** is **complete**.

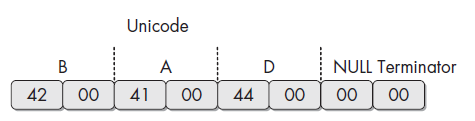
**ASCII** strings use **1 byte per character**, whereas **Unicode** uses **2 bytes per character**.

The image beneath shows the string ‘*BAD*’ stored as ASCII.

The ASCII string is stored as the bytes: **0x42**, **0x41**, **0x44** & **0x00**.



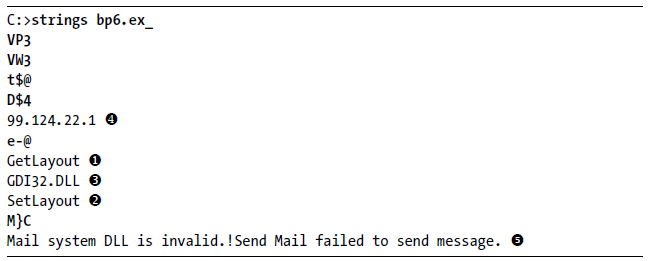
The Unicode string is stored as the bytes **0x42**, **0x00**, **0x41**, **0x00**, and so on as showed beneath.



When Strings searches an executable for ASCII and Unicode strings, it **ignores** **context** and **formatting**, so that it can **analyse** **any** **file** **type** and **detect** **strings** **across** an **entire** **file** (*though this also means that it may identify bytes of characters as strings when they are not*). Strings searches for a **three**-**letter** **or** **greater** **sequence** of **ASCII** and **Unicode** **characters**, followed by a **string** **termination** character.

Sometimes the strings detected by the Strings program are **not** actual **strings**. For example, if Strings finds the sequence of bytes **0x56**, **0x50**, **0x33**, **0x00**, it will **interpret** that as the **string** **VP3**. But those bytes may **not** actually **represent** that **string**; they could be a **memory** **address**, **CPU** **instructions**, or **data** used by the **program**. Strings leaves it **up** **to** the **user** to **filter** out the **invalid** **strings**.

Fortunately, most invalid strings are obvious, because they do not represent legitimate text. For example, the following excerpt shows the result of running Strings against the file *bp6.ex\_*:



In this example, the bold strings can be ignored. Typically, if a string is **short** and doesn’t **correspond** to **words**, it’s probably **meaningless**.

On the other hand, the strings ***GetLayout*** at 1 and ***SetLayout*** at 2 are **Windows** **functions** used by the Windows graphics library. We can easily identify these as **meaningful** **strings** because **Windows** **function** **names** normally **begin** with a **capital** **letter** and **subsequent** **words** also begin with a capital letter.

**GDI32.DLL** at 3 is **meaningful** because it’s the name of a **common** **Windows** **dynamic** **link** **library** (**DLL**) used by **graphics** **programs**. (*DLL files contain executable code that is shared among multiple applications*.)

As you might imagine, the number **99.124.22.1** at 4 is an **IP** address—most likely one that the **malware** will use in some fashion.

Finally, at 5, “*Mail system DLL is invalid.! Send Mail failed to send message*.” is an **error** **message**. Often, the most useful information obtained by running Strings is found in error messages. This particular message **reveals** **two** **things**: The **subject** **malware** **sends** **messages** (*probably through email),* and it **depends** on a **mail** **system** **DLL**. This information suggests that we might want to check email **logs** for **suspicious** **traffic**, and that **another** **DLL** (*Mail system DLL*) might be **associated** with this particular **malware**. Note that the missing DLL itself is not necessarily malicious; malware often uses legitimate libraries and DLLs to further its goals.

Packet and Obfuscated Malware

**Obfuscated programs**:

Ones whose **execution** the **malware** author has **attempted** to **hide**.

**Packed programs:**

A **subset** of **obfuscated** **programs** in which the malicious program is **compressed** and **cannot** be **analysed**. Both techniques will severely limit your attempts to statically analyse the malware.

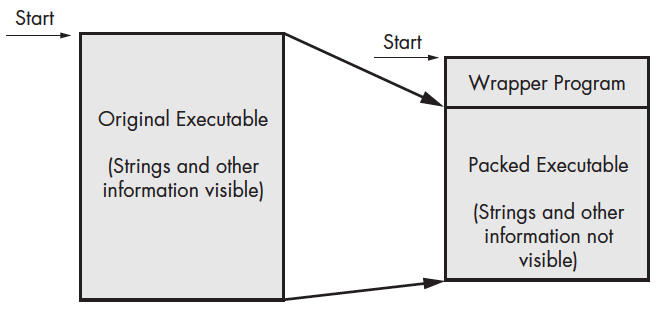
Obfuscation = *the action of making something obscure, unclear, or unintelligible.*

**Legitimate** **programs** almost always **include many strings**. Malware that is packed or obfuscated contains very **few** **strings**. If upon searching a program with Strings, you find that it has only a few strings, it is **probably** either **obfuscated** or **packed**, **suggesting** that it may be **malicious**. You’ll likely need to throw more than **static analysis** at it in order to investigate further.

**NOTE**: *Packed and obfuscated code will often include at least the functions* ***LoadLibrary*** *and* ***GetProcAddress****, which are used to load and gain access to additional functions.*

Packing files

When a packed program is run, a **small** **wrapper** **program** is also executed to **decompress** the packet file and then **runs** the **unpacked** **version** of the file. When a packed program is **analysed** **statically**, **only** the **small** **wrapper** **program** can be dissected.



Left is the original executable, right shows the packed executable, information is invisible to most static analysis tools.

Detecting packers with PEiD

The PEiD program allows for detection on which type of packer or compiler has been used to build an application, making analysing the packed file easier.



The image above shows information on a file called *orig\_af2.ex\_*.

PEiD development has discontinued but remains the best tool available for packer and compiler detection. When a program is packed, you must unpack it in order to be able to perform any analysis.

PEiD can be subject to vulnerabilities. For example, PEiD version 0.92 contained a buffer overflow that allowed an attacker to execute arbitrary code. This would have allowed a clever malware writer to write a program to exploit the malware analyst’s machine. Be sure to use the latest version of PEiD.

Portable executable file format

The format of a file can reveal a lot about the programs 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. Nearly **every** **file** with **executable** **code** that is loaded by **Windows** is in the **PE format**, though some legacy file formats do appear on rare occasion in malware.

PE files begin with a header that includes information about:

* *the code, the type of application, required library functions and space requirements.*

The information in the PE header is of **great value** to the **malware analyst**.

Linked Libraries and Functions

One of the **most useful** pieces of information we can gather about an **executable** is the **list of functions that it imports**.

Imports are **functions** used by one program that are actually **stored** in a **different** **program**, such as **code** **libraries** that contain functionality **common** to **many** **programs**. Code libraries can be connected to the main executable by *linking*.

Programmers link imports so they don’t need to re-implement certain functionality in multiple programs. Code libraries can be linked; **statically**, **at runtime**, or **dynamically**.

Understanding how the library code is lined is **critical** to our **understanding** of **malware** because the information we can find in the **PE** file **header** depends on **how** the **library** **code** has been **linked**.

Static, Runtime and Dynamic Linking

Static

Static linking is the **least commonly used method** of **linking libraries**.

Static linking is common in **UNIX** and **Linux** **programs**.

When a library is statically linked to an executable, **all code** from that **library** is **copied** into the **executable**, which makes the executable **grow** **in** **size**.

When analysing code, it’s **difficult** to **differentiate** between statically linked code and the **executable’s** **own** **code**, because **nothing** in the PE file header **indicates** that the file contains **linked** **code**.

runtime

Runtime linking is unpopular in friendly programs - **commonly** **used** in **malware** especially when its **packed** or **obfuscated**.

Executables that use runtime linking connect to libraries **only** **when** that function is **needed**, **not** **at** program **start**, as with **dynamically** **linked** **programs**.

Several Microsoft Windows functions allow programmers to import linked functions not listed in a program’s file header. Of these, the two most commonly used are **LoadLibrary** and **GetProcAddress**. **LdrGetProcAddress** and **LdrLoadDll** are also used. **LoadLibrary** and **GetProcAddress** allow a program to **access** **any** **function** in any **library** on the **system**, which means that when these functions are used, you **can’t** **tell** **statically** which **functions** are being **linked** to by the **suspect** **program**.

Dynamic (most common & most interesting for malware analysis)

When libraries are dynamically linked, the host **OS** **searches** for the **necessary** **libraries** when the **program** is **loaded**.

When the program **calls** the **linked** **library** **function**, that **function** **executes** **within** the **library**.

The **PE file header** store information about **every** **library** that will be **loaded** and **every** **function** that will be **used** **by** the **program**.

The **libraries** **used** and **functions** **called** are often the **most** **important** **parts** of a **program**, and **identifying** them is particularly important, because it allows us to **guess** at **what** the **program** **does**.

Exploring Dynamically Linked Functions with Dependency Walker

The **Dependency Walker program**, distributed with some versions of **Microsoft Visual Studio** and other **Microsoft** **development** **packages** lists **only dynamically linked functions** in an **executable**.



This image above shows the **Dependency Walker’s analysis** of *SERVICES.EX\_*.

|  |  |
| --- | --- |
| DLL | Description |
| *Kernel32.dll* | This is a very **common** DLL that **contains core functionality**, such as **access** and **manipulation** of memory, files and hardware. |
| *Advapi32.dll* | This DLL provides access to **advanced core Windows components** such as the Service Manager and Registry. |
| *User32.dll* | This DLL contains **all the** **user-interface components**, such as buttons, scroll bars and components for controlling and responding to **user actions**. |
| *Gdi32.dll* | This DLL contains functions for displaying and manipulating graphics. |
| *Ntdll.dll* | This DLL is the **interface to the Windows kernel**. Executables generally **do** **not** **import** this file **directly**, although it is always **imported** **indirectly** by **Kernel32.dll**. If an executable imports this file, it means that the **author intended to use this functionality** is **not normally available** to **Windows** **programs**. Some tasks, such as hiding functionality or manipulating processes will **use** **this** **interface**. |
| *WSock32.dll & Ws2\_32.dll* | These are **networking** **DLLs**. A program that accesses either of these most likely connects to a network or **performs** network-related **tasks**. |
| *Wininet.dll* | This DLL contains higher-level networking functions that implement **protocols** such as FTP, HTTP & NTP. |

Windows Function Naming Conventions

you will often encounter function names with an Ex suffix, such as **CreateWindowEx**.

When Microsoft updates a function and the **new** function is **incompatible** with the **old** **one**, **Microsoft** continues to **support** the **old** **function**. The new function is given the same name as the old function, with an **added** **Ex suffix**.

Functions that have been significantly updated twice have two Ex suffixes in their names.

Many functions that take **strings as parameters** include an A or a W at the **end of their names**, such as **CreateDirectoryW**. This letter does not appear in the documentation for the function; it simply indicates that the function accepts a string parameter and that there are two different versions of the function:

* One for **ASCII strings (A)**.
* One for **wide character strings (W)**.

Imported Functions

* The **PE file header** also includes **information** about **specific functions used** by an **executable**.
* The **names** of these **Windows** **functions** can give you a good idea about **what** the **executable** **does**.
* Microsoft does an excellent job of **documenting** the **Windows** **API** through the **Microsoft** **Developer** **Network (MSDN) library**.

Exported Functions

***DLLs*** and ***EXEs*** **export** **functions** to **interact** with **other** **programs** and **code**. Typically, a DLL **implements one or more functions** and **exports** them **for use by an executable** that can then **import** and **use** them.

The **PE** **file** contains information on which **functions** a file **exports**. **Exported** **functions** are **most common in** DLLs as the functions from them are **typically** **used** in **EXEs**.

* **EXEs** **are not designed** to provide **functionality** for other **EXEs** and exported functions are rare.
* If you discover exports in a executable, they often will provide useful information.

One common convention is to use the name used in the Microsoft documentation. For example, in order to **run a program as a service**, you must first define a **ServiceMain** function. The presence of an exported function called **ServiceMain** tells you that the **malware** **runs** **as part of a service**.

Unfortunately, while the **Microsoft documentation** calls this function **ServiceMain**, and it’s common for programmers to do the same, the function can have any name. Therefore, the names of exported functions are actually of limited use against sophisticated malware. **If** **malware** uses **exports**, it will often either **omit names entirely** or **use unclear or misleading names**.

Static Analysis in Practice

Potentialkeylogger.exe - an unpacked executable

Beneath is an list of functions imported using the **Dependency Walker** on **Potentialkeylogger.exe**. Because we see so many **imports**, we can conclude that this file is **not packed**.



Only a small amount of these functions are interesting for malware analysis:

1. The imports from Kernel32.dll tell us that this software can **open** and **manipulate** **processes** (*such as OpenProcess, GetCurrentProcess, and GetProcessHeap) and files (such as ReadFile, CreateFile, and WriteFile*). The functions **FindFirstFile** and **FindNextFile** are particularly interesting ones that we can use to search through directories.
2. The imports from User32.dll are even more interesting. The large number of GUI manipulation functions (*such as RegisterClassEx, SetWindowText, and ShowWindow*) indicates a **high** **likelihood** that this **program** **has a GUI** (*though the GUI is not necessarily displayed to the user*).
3. The function **SetWindowsHookEx** is commonly used in **spyware** and is the most popular way that **keyloggers receive keyboard inputs**. This function has some legitimate uses, but if you suspect **malware** and you see this function, you are probably looking at **keylogging functionality**.
4. The function **RegisterHotKey** is also interesting. It registers a **hotkey** (*such as* ***CTRL-SHIFT-P***) so that **whenever** the **user** **presses** that **hotkey** **combination**, the **application** is **notified**. No matter which application is currently active, a hotkey will bring the user to this application.
5. The imports from GDI32.dll are **graphics-related** and simply **confirm** that the **program** **probably** **has** a **GUI**. The imports from Shell32.dll tell us that this program can **launch other programs**—a **feature** common to both **malware** and **legitimate** **programs**.
6. The imports from Advapi32.dll tell us that this program uses the **registry**, which in turn tells us that we should **search** **for** **strings** **that look like registry keys**. **Registry** **strings** **look** a lot **like** **directories**. In this case, we found the string *Software\Microsoft\Windows\CurrentVersion\Run*, which is a **registry key** (*commonly used by malware*) that controls which programs are automatically run when Windows starts up.
7. This executable also has several **exports**: **LowLevelKeyboardProc** and **LowLevelMouseProc**. Microsoft’s documentation says, “*The LowLevelKeyboardProc hook procedure is an application-defined or library-defined call-back function used with the SetWindowsHookEx function.*” In other words, this function is used with **SetWindowsHookEx** to specify which **function** will be **called** when a **specified event occurs**—in this case, the low-level keyboard event. The documentation for **SetWindowsHookEx** further explains that this **function** will be **called** when certain **low-level keyboard events occur**.

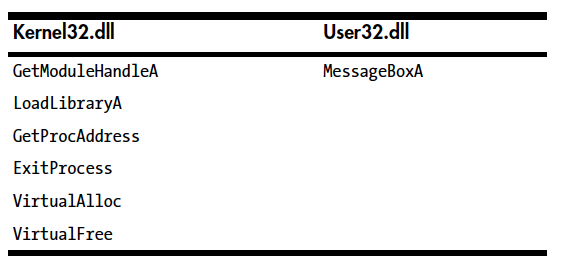
**Conclusion:**

Using the information gleaned from a static analysis of these imports and exports, we can draw some significant conclusions or formulate some hypotheses about this malware:

For one, it seems likely that this is a local keylogger that uses **SetWindowsHookEx** to **record** **keystrokes**. We can also surmise that it has a **GUI** that is **displayed** only to a **specific** **user**, and that the **hotkey** **registered** with **RegisterHotKey** specifies the **hotkey** that the **malicious** **user** **enters** to see the **keylogger** **GUI** and **access** **recorded** **keystrokes**. We can further speculate from the registry function and the existence of *Software\Microsoft\Windows\CurrentVersion\Run* that this **program** sets itself to **load** at **system** **start**-**up**.

An example of a packed program - a dead end

Beneath is a list of the functions imported by a second piece of unknown **malware**. The **brevity** of this list tells us that this **program** is **packed** or **obfuscated** - which is further confirmed by that fact that this program has **no readable strings**. Even a *Hello, World* program would have more imports than this. Knowing that this is **packed** is a valuable piece of information for malware analysis.



The PE File Headers and Sections

**PE** **headers** can provide **considerably more information** than **just** **imports**.

The PE file format contains a **header followed by a series of sections**. The header contains **metadata** **about the file itself**. **Following** the **header** are the **actual sections** of the **file**, each of which contains **useful information**.

*The following are the most* ***common*** *and* ***interesting******sections*** *in a PE file:*

**.text** The *.text* section contains **the instructions that the CPU executes**. All other sections store **data** and **supporting** **information**.

**.rdata** The*.rdata* section contains the **import** and **export** **information**, which is the same information available from both **dependency walker** and **PEview**. Sometimes a file will contain an *.****idata*** and ***.edata*** section, which store both the **import** and **export** **information**.

**.data** The *.data* section contains the **program’s global data**, which is **accessible** from **anywhere** **in** the **program**. Local data is not stored in this section or elsewhere in the PE file.

**.rsrc** The *.rsrc* section includes the **resources** **used** by the **executable** that are not **considered** **part** of the **executable**, such as ***icons***, ***images***, ***menus*** and ***strings***. Strings can be stored either in the *.rsrc* section or in the **main** **program**, but they are often stored in the *.rsrc* section for **multi-language support**.

Sections of a PE File for a Windows Executable

|  |  |
| --- | --- |
| Executable | Description |
| .text | Contains the **executable** **code**. |
| .rdata | Holds **read-only data** that is **globally accessible within** the **program**. |
| .data | Stores **global data access** **throughout** the **program**. |
| .idata | **Sometimes** **present** and **stores** the **import** **function** **information**; **if** this section is **not** **present**, the **import** **function** **information** is **stored** in the **.rdata section**. |
| .edata | **Sometimes present** and **stores** the **export function information**; if this section is **not present**, the **export function information** is **stored** in the **.rdata section**. |
| .pdata | **Present** only in **64-bit** **executables**, **stores exception-handling information**. |
| .rsrc | Stores **resources** **needed** **by** the **executable**. |
| .reloc | Contains **information** for **relocation** of **library files**. |

Examining Files with PEview