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**.

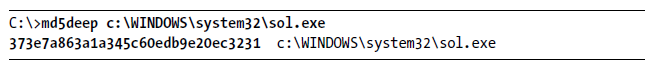
Hashing: A Fingerprint for Malware

***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.

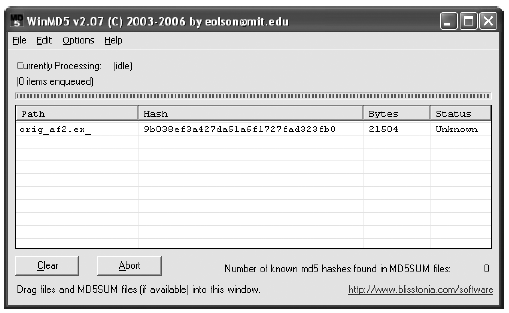
md5deep

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**.

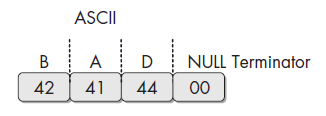
Searching through the strings can be a simple way to get hints about the functionality of a program. For example, if the program accesses a URL, then you will see the URL accessed **stored** as a **string** in the **program**. You can use the Strings program (http://bit.ly/ic4plL), to search an executable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

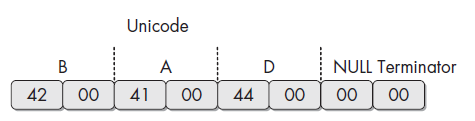
**ASCII** strings use **1 byte per character**, whereas **Unicode** uses **2 bytes per character**.

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

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



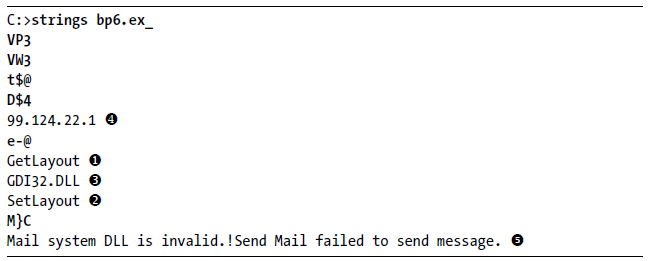
The Unicode string is stored as the bytes **0x42**, **0x00**, **0x41**, **0x00**, and so on as showed beneath.



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.*