Journal of Mobile, Embedded and Distributed Systems, vol. VII, no. 2, 2015
ISSN 2067 – 4074

Reverse Engineering Malicious Applications

Ioan Cristian IACOB
IT&C Security Master
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
ROMANIA
cristian.iiacob@gmail.com

Abstract: Detecting new and unknown malware is a major challenge in today's software. Security profession. A lot of approaches for the detection of malware using data mining techniques have already been proposed. Majority of the works used static features of malware. However, static detection methods fall short of detecting present day complex malware. Although some researchers proposed dynamic detection methods, the methods did not use all the malware features. In this work, an approach for the detection of new and unknown malware was proposed and implemented. Each sample was reverse engineered for analyzing its effect on the operating environment and to extract the static and behavioral features.

Key-Words: Reverse Engineering, Applications, Malicious, Security, Malware

1. Malware History

Malware is the short term used for malicious software. Malware can be any unwanted software used to gain unauthorized access, disrupt computer operation, or gather sensitive information. Malware is defined by its malicious intent, acting against the requirements of the computer user.
Before internet access became affordable to most computer users, viruses spread by infecting floppy disks. The malware inserted a copy of itself into the machine code instructions of executables. Viruses depended on users exchanging software on floppies or thumb drives so they can spread in computer environments. Because of the lack of computer networks, viruses were written mostly for fun, not for information theft. This means that they were very "loud", and computer users immediately knew they were infected.
The first malware which spread using computer networks originated on multitasking Unix systems. SunOS and VAX BSD systems were the targets of the first well-known worm created in 1988. The method used for propagation was to exploit security holes (vulnerabilities) in network server programs and ran itself as a separate process. Since then, malware has evolved and started gathering information. This meant they needed to be present for longer periods of time in order to gather as much information/data as they can. So the writing techniques changed from a "loud" behavior and notorious to a stealthier and obscure approach.
Today, because of its popularity, Microsoft's Windows OS is the preferred target for malware writers, although a few are also written for Linux and Unix systems. [1]

2. Malware Classification

A *virus* is a type of malware that replicates by inserting a copy of itself into other executables. The majority of viruses are attached to executables, this means the virus may reside in a system but will not be active until a user runs that program. When the software is executed, the virus will pass execution to the legitimate process after it has done its job.
The virus spreads when the infected file is transferred between different computers, networks, e-mails, etc. [2]
In contrast to viruses, *worms* propagate by exploiting vulnerabilities of network services, which make them human independent. After an infection has been

achieved, the virus can pivot its action to the new infected host to enable it to access other network segments that would otherwise be inaccessible from the original starting point.
The *Trojan* was named after the horse used by the Greeks to infiltrate Troy. It is designed to be similar with legitimate software and trick users into launching it. After the system has been compromised, the Trojan can gather and steal intellectual property, change or delete files, create backdoors to attackers etc.
This kind of malware spreads using human interaction, and not by infecting other files or propagating through the network.
The term *bot* comes from the word "robot", which is an autonomous mechanism/software that is able to interact with the nearby environment providing services that would otherwise be conducted by a human being. Typically, in information technology, bots are used to create a mesh of compromised computers that are controlled from a central point called a command and control center using different communication protocols such as HTTP/S. These infrastructures are usually used to generate money for the bot master.
Malware that deploys techniques to maintain persistence and undetected are called *rootkits*. Rootkits became popular on Linux operating system and from their name has been derived as a software kit that run with root privileges, as root is the administrative user inside Linux. Rootkits can be used for both malicious activities and also for legitimate ones such as those used by antivirus companies to detect malware. More on this subject will be discussed in chapter III.

3. Malware Propagation Techniques

3.1 Web browsing

The easiest way of getting infected is through drive-by-download. Malware often spreads through unwanted software downloads, malicious PDF documents, word documents, or fake software. Using this technique, malware authors have no target other than to infect as many computers as possible.
Modern browsers like Chromium (the open source project on which Google developed Chrome) include two mechanisms that are designed with security in mind. One component is the browser kernel that interacts with the operating system and the other is the rendering engine that runs inside a sandbox with restricted privileges. This design help to improve browser security and mitigate attacks from malicious websites. [26]

3.2 USB thumb drives

Thumb drives are also used to spread malicious software. This method uses the AutoRun feature to launch malware when the storage device is mounted by the operating system. A common attack scenario is performed by intentionally dropping USB drives in front of targeted organizations.

3.3 Email Spear Phishing

Spear phishing is an e-mail spoofing fraud attempt that targets a specific organization, seeking unauthorized access to confidential data. Spear phishing attempts are not initiated by random attackers, but are more likely to be conducted by perpetrators out for financial gain, trade secrets or military information.
Similar to e-mail messages used in regular phishing expeditions, spear phishing messages appear to come from a trusted source. Phishing messages usually appear to come from a large and well-known company or Web site with a broad membership base, such as eBay or PayPal. In the case of spear phishing, however, the apparent source of the e-mail is likely to be an individual within the recipient's own company and generally someone in a position of authority.

3.4 Watering Hole

Watering hole is a computer attack strategy used to compromise a targeted group. Attackers first observe frequently visited websites that they visit and trust, afterwards they infect these websites with malware in the hope that a person from the targeted group will get infected.

3.5 Zero Day Exploits (java, office docs, flash)

A zero-day exploit is a previously unknown vulnerability in a computer application or operating system, one that developers have not had time to address and patch. It is called a "zero-day" because the programmer has had zero days to fix the flow and therefore a patch is not available. Once a patch is available, it is no longer a "zero-day exploit".
After these vulnerabilities have been found by legitimate users, they are documented and reported to software developers to be patched. Known vulnerabilities are accounted and publicly available at *cve.mitre.org*. "CVE is a dictionary of publicly known information security vulnerabilities and exposures." [3]

4. Malware Goals

Monetizing Malware
In the recent years, an extensive diversification has been perceived of the underground economy associated with malware and the subversion of Internet-connected systems. [4]

Credentials Theft (CCs & Bank Accounts, email credentials)
In the early days of malware, the main purpose was notoriety, but since the internet has grown, malware writers concentrated their efforts on making money. One of their strategies, after infecting computer environments, is to steal user credentials from banking websites, systems accounts, ftp, email, etc. These credentials are sent through the internet to the attacker so he can sell them on the black market or use them for their own good. [5]
Some examples include: Citadel, SpyEye, Pony, Zeus, Carberp and Dyre.

Rogue Software (fake AV, Battery Boosters)
Rogue software is a misleading type of software that simulates the user interface of a legitimate application with the intent of dropping malware onto the computer or to persuade the user into paying money for it.

BlackSEO and SPAM
Black SEO is a practice that increases a page's rank in search engines through means that violate the search engines' terms of service. [6] This is accomplished by renting botnet infrastructures to entities that desire higher page rank. The bot master commands his bots to search and access webpages, thus resulting rank increase.
SPAM is the process of sending unsolicited emails containing advertisements or attachments to users. By clicking the links from the message body, users can be redirected to phishing websites or site that are hosting malware.
This type of activity also has its own black-market and bot masters can rent their infrastructure to other individuals. Payments are made using cryptocurrencies to preserve the anonymity of both entities.

Pay per Install (install other malware)
Pay-per-install is a technique used by bot masters to make money by renting or selling their infrastructure of compromised hosts to other entities. These entities continue by installing new malware on the hosts, and by doing so, he is further expanding his current botnet.

Ransomware (CryptoLocker)
Ransomware is the type of malware that restricts users' access to their data by encrypting critical parts and afterwards demanding a ransom to be paid by the victim to the malware author in order to re-enable access. [7]

Espionage and Sabotage
"Advanced Persistent Threat (APT) is a set of stealthy and continuous hacking processes often orchestrated by human targeting a specific entity. APT usually targets organizations and/or nations for business or political motives. APT

processes require high degree of covertness over a long period of time. As the name implies, APT consists of three major components/processes: advanced, persistent, and threat. The advanced process signifies sophisticated techniques used by malware to exploit vulnerabilities in systems. The persistent process suggests that an external command and control server is continuously monitoring and extracting data from a specific target. The threat process indicates human involvement in orchestrating the attack" (Musa, n.d.)

Stuxnet (sabotage)

Stuxnet is an advanced worm which was discovered in July 2010, and it was designed to infect industrial programmable logic controllers (PLCs). It infected at least 22 manufacturing sites and had a major impact on Iran's nuclear enrichment programs, but also infected an U.S. manufacturing plant. Stuxnet is the first malware to target industrial processes and the costs of eliminating it are not unneglectable.

PLCs allow the automation of electromechanical processes such as centrifuges for separating nuclear material. Stuxnet spread by exploiting four zero-day vulnerabilities and was able to compromise Iranian PLCs, collecting information on industrial systems and disrupting the enrichment process of uranium. Stuxnet reportedly ruined almost 20% of Iran's nuclear centrifuges.

Stuxnet has three modules: a worm that executes all routines related to the main payload of the attack; a link file that automatically executes the propagated copies of the worm; and a rootkit component responsible for hiding all malicious files and processes, preventing detection of the presence of Stuxnet.

The Stuxnet first infected a computer via an infected USB thumb drive and then propagates through the network and scanning for Siemens Step7 Software. In absence of both, the malware remained dormant. When it arrived on a targeted computer, it introduced a rootkit onto the Siemens software what modified the parameters given to the machinery and reported to the user normal operation parameters.

Red October (espionage)

Red October was a cyberespionage malware program discovered in October 2012 and uncovered in January 2013 by Kaspersky Lab. The malware was reportedly operating worldwide for up to five years prior to discovery, transmitting information ranging from diplomatic secrets to personal information, including from mobile devices. The primary vectors used to install the malware were emails containing attached documents that exploited vulnerabilities in Microsoft Word and Excel. Later, a webpage was found that exploited a known vulnerability in the Java browser plugin. Red October was termed an advanced cyberespionage campaign intended to target diplomatic, governmental and scientific research organizations worldwide.

After being revealed, domain registrars and hosting companies shut down as many as 60 domains used by the virus creators to receive information. The attackers shut down their operation as well. [8]

Regin (espionage)

According to popular antivirus companies, Regin has one of the most technical competence which is rarely seen and it has been used to spy on governments, infrastructure operators, businesses, researchers, and private individuals.

Regin was revealed by several antivirus companies in November 2014. Figure 1 illustrates the main countries targeted by this campaign. [9]

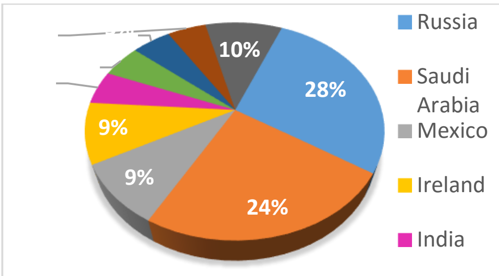

Figure 1. Countries affected by Regin

Antivirus companies have been unable to determine the attack vector used. Regin has been compared to Stuxnet and is thought to have been developed by well-founded teams of developers, possibly a government, as a targeted multi-purpose data collection tool. [10]

The attack is comprised of several stages, each being encrypted or hidden except the first stage which represents the initial dropper of the malware. This modular approach enables the attacker to customize the attack scenario, and the multitude of stages is used to elude reverse engineers by making them reconstruct the puzzle in a bigger amount of time and with fewer clues. Only by acquiring all five stages it is then possible to conduct a thorough analysis on the attack. Also some of the components were not written to disk, these would only reside in memory to escape disk forensics.

This first stage begins a chain of events, downloading, decrypting and executing the next level up until the fifth which is the final one. Figure 2 displays these different stages in a cascading order. [11]

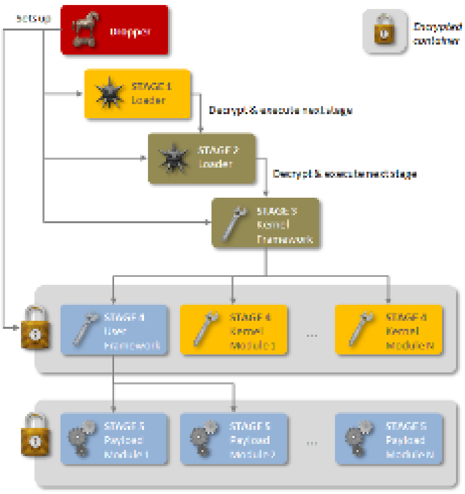

Figure 2. Symantec: Regin Top-tier Espionage. Source [9]

Flame (espionage)

Flame is a modular computer malware discovered in 2012. The malware is being used for targeted cyber espionage in Middle Eastern countries.

Its discovery was announced on 28 May 2012 by MAHER Centre of Iranian National, Computer Emergency Response Team (CERT), Kaspersky Lab and CrySyS Lab of the Budapest University of Technology and Economics.

Flame can spread to other systems over a local network or via USB thumb drives. It can record audio, screenshots, keyboard activity and network traffic. The program also records Skype conversations and can turn infected computers into Bluetooth beacons which attempt to download contact information from nearby Bluetooth-enabled devices. This data, along with locally stored documents, is sent on to one of several command and control servers that are scattered around the world. The program then awaits further instructions from these servers. [12]

5. The Goal of Malware Analysis

There are several reasons why someone should invest resources to dissect the inner workings of a malware. There is a series of questions that can be answered only by conducting an analysis over a piece of malware. Some of the most common reasons why you might analyses a malicious program include:

- to reveal indicators of compromise that can be used to further create signatures and find infected hosts;
- to assess the damage taken after an intrusion;
- to identify and understand the vulnerabilities that were exploited in order gain control and to further mitigate them;
- to identify the responsible one for installing the malware;
- to reveal the purpose of the code;
- to find possible solutions to perform data recovery.

Malware analysis is the action of taking apart the executable code and study its behavior. While the reverse engineering takes place, the analyst must focus and find answers to questions above.

The reverse engineer must create a safe and isolated environment in which he can conduct the malware analysis. Isolated environments are mandatory in order to prevent accidental damage to production sites. One solution to this is to create a physically isolated network from the corporate one, with its own network services, hosts, software and also isolated from the Internet. Several tools can be used to simulate the Internet and other communication protocols. More techniques used in this scope will be detailed further in this thesis.

6. Types of Malware Analysis and Tools

There are two approaches to disassembling and analyzing software. Both are important, none is better than the other and both are used to compare results and confirm observed behavior. The techniques used in software reverse engineering are comprised into dynamic analysis and static analysis.
The objective of this paper is to describe how a malware sample must be analyzed, how to approach heavily obfuscated code, route the code flow to exhibit malware behavior that may occur under certain conditions, understand detection mechanisms etc.

6.1 Dynamic Malware Analysis

Dynamic analysis involves running the code and understanding its behavior by viewing system API calls, network connections and traffic, used hardware resources, files and processes created, registry interaction hooking etc. Dynamic analysis is a good first approach to discovering functionality and also to confirm behavior seed in static analysis. But there are instances where dynamic analysis cannot achieve complete analysis due to anti-debugging techniques and thus static analysis need to be performed by the analyst in order to understand the complete picture. Further some of the most popular tools will be displayed in order to understand each.

Sysinternals

Windows Sysinternals is a suite of software utilities used to diagnose, monitor and troubleshot the Windows operating system environment. Originally developed by a third party entity and now under the umbrella of Microsoft, this suite is a must have, and must know by every advanced system user. The tools that are provided for free by this suite include a process manager, process monitor and API calls, Autorun manager, rootkit detection, network connection handler, memory tools, memory mapper etc.

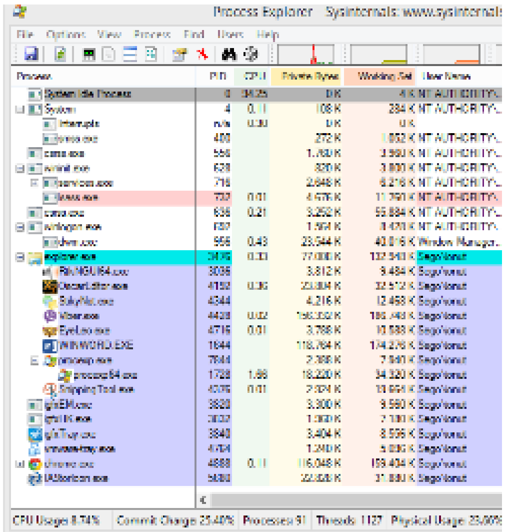

Figure 3. Sysinternals Process Explorer

Malware analysts use the suite to launch malware into execution and rapidly get an idea of its activity within the system.
Figure 3 illustrated a screen shot of one of the most used tool, Process Explorer is a process manager which includes a rich set of features for collecting information about the operating system and running process. It is able to display network connections, strings, file handles, processor usage associated with each process.

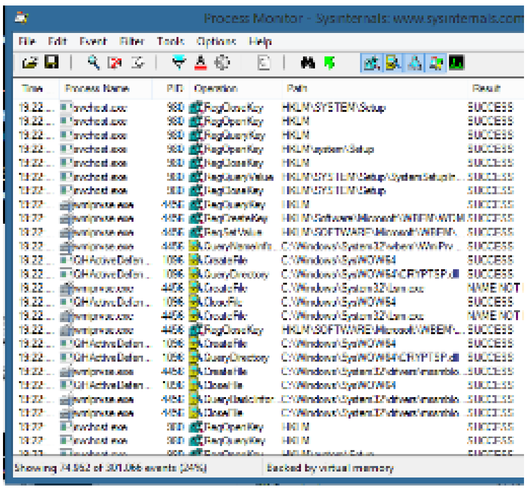

Figure 4. Sysinternals Process Monitor

Figure 4 displays a screenshot of Process Monitor, a great tool to monitor process behavior and its interaction with the operating system in real time. It can observe registry activity, created and terminated process or files along with timestamps for each action. It is able to filter contents based on user defined rules and highlight different types of action.

The last tool referred to in this paper is TCP Viewer. This utility is illustrated Figure 5, it displays connections created and maintained by the processes and can also manage them along with their process. This tool is often used to view the network activity and network destinations of certain malware. Results from TCP viewer can be combined with network traffic from tools like Wireshark or TCPDump to get a more detailed picture of the malware's network activity.

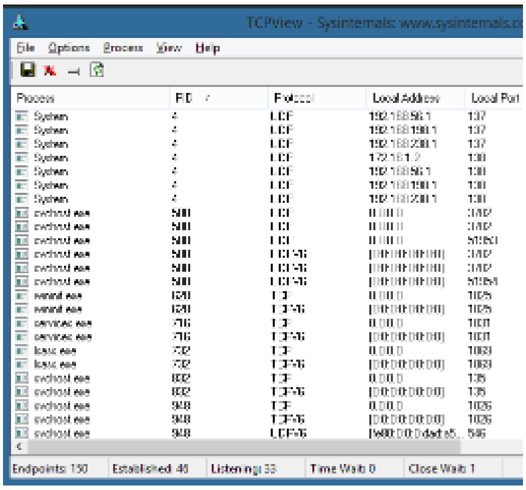

Figure 5. Sysinternals TCP Viewer

OllyDbg, WinDbg and ImmDbg

Olly Debugger (illustrated in Figure 6) is one of the most frequently used x86 debugger, it is able to disassemble executables, recognize processor registers, API calls, strings, constants and locate functions or structures. Functionality can be extended with third-party plugins and it is free of cost. Windows debugger and Immunity debugger have the same functionality, with similar graphical interfaces but are not as popular even though Windows Debugger has more powerful functionalities and can achieve kernel debugging which the others do not.

Debuggers enable reverse engineers to follow each line of code and also modify execution flow by interacting with the code and processor registers. This is often helpful to enable hidden functionality of malware otherwise inaccessible due to a variety of reasons (i.e. logic bombs).

Sandboxing

In computer science, a sandbox is an isolated and controlled environment in which untrusted code is ran to observe its behavior and visualize network activity. A sandbox has an isolated environment to prohibit and prevent possible damage of malicious software to the host operating system. After the software has finished execution and the logs have been acquired the sandbox automatically reverts back to a clean state.

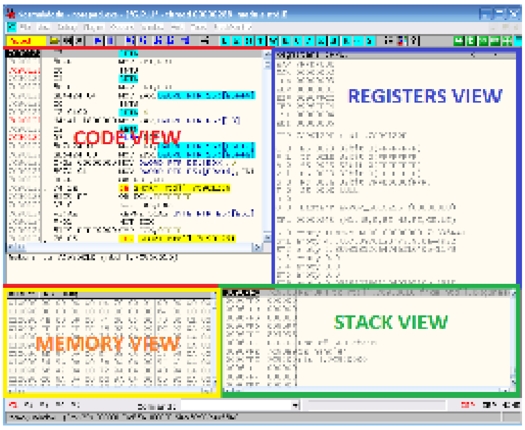

Figure 6 - Olly Debugger

Sandboxes are used to automate malware analysis and designed to easily scale up to enable fast triage of hundreds of malware samples. These tools must implement a series of techniques to evade detection, simulate human activity, and to execute as much as possible from the malware sample. Modern malware include routines that detect the presence of debuggers, sandboxed environments, virtual machines and other monitoring tools installed onto the system, also malware can include logic bombs to evade dynamic analysis of malicious code. Login Bombs are algorithms that do enable check if a condition has been met and after malicious routines are ran. These checks can include calendaristic dates, certain key strokes, elapsed time since install etc.

There are two types of sandboxes, emulated and virtualized.
A software emulator is a program that simulates the functionality of another applications or hardware. Emulators can collect detailed information about execution of a particular application and emit a report based on them and also emulator can be developed to run software written for different CPU architectures such as Android which runs on ARM processors. A big disadvantage to emulator is lack of performance because of the stacked software layers.
With virtualization, a program runs on the underlying system hardware. The hypervisor manages accesses of different applications to the underlying hardware. Such that different virtual machines are isolated and independent from each other. However, when programs run inside a virtual machine, it is being executed on the actual hardware and thus detailed data collection is difficult to achieve. An advantage of this approach is that programs run at the native speed of the hardware host and also virtualizing software provides means by which the user can create a restore point in time and revert back to it when needed. It is recommended that malware analysts should deploy physically isolated environments and also they must emulate the Internet to enable optimal analysis using dynamic techniques.

6.2 Static Malware Analysis

Static analysis involves disassembling the code and understanding its behavior reading instructions without running them. Static analysis helps to discover hidden functionally of software which cannot be observed by dynamic analysis, it is a more tedious and intensive process but also has its limitations and malware can deploy techniques to harden analysis. The main idea here to remember is that static analysis does not run the code and can achieve in-depth understanding of executable code.
Further, some of the most used applications and techniques will be presented and demonstrated.

IDA

The Interactive Disassembler, as its name implies, it is a disassembler for computer software and it is able to generate assembly language from Microsoft Windows executable, Linux executables, Apple OS X. Commonly known as IDA, the tool is a commercial software but for there is also a free version of it available on the official web site, and as expected, it lacks the features added between version 5 and 6.8. [24]
IDA performs analysis of assembly code and can distinguish functions, API imports, construct a diagram (graph) containing code blocks with their connections and it can also produce a pseudocode representation of assembly code similar to C/C++. IDA permits analysts to develop their own Python scripts to extend capabilities of the disassembler and some are also available for download from the Internet.
This tool is one of the most powerful weapons used by malware reverse engineers against malware. It has multiple tabs each with its disting features and capabilties such as: code view, hexadecimal editor, structures editor, Import table, export table, string view, pesudocode view, Python console etc.

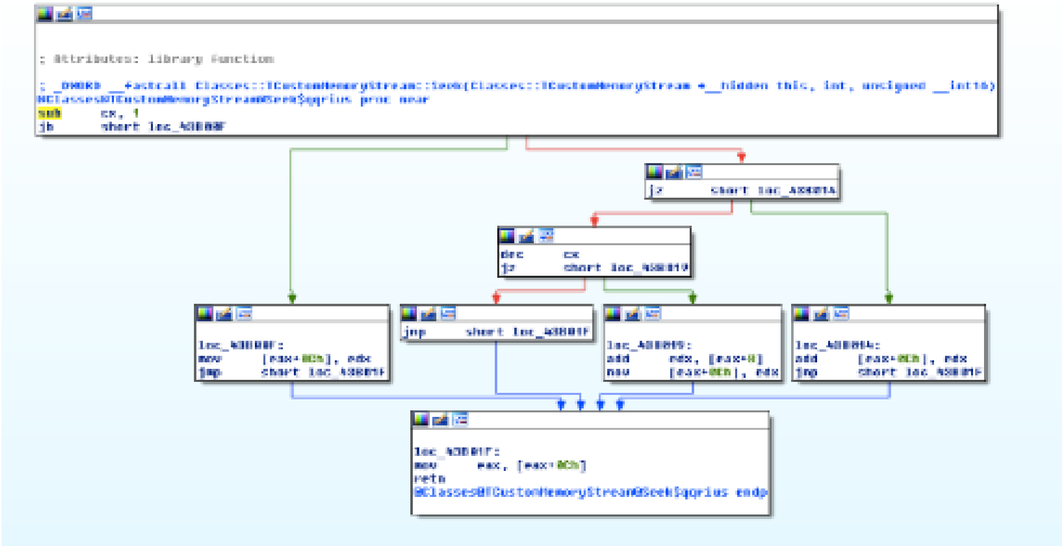

Figure 7 - IDA Screenshot

7. Malware Strategy and Defense

Anti-Reverse Techniques and Anti-Disassembly Techniques

Anti-disassembly techniques involve using specially designed code or data in a software to cause disassembly analysis tools to improper list the program's code. This method is crafted by malware authors manually, with a separate tool in the build and deployment process or interwoven into their malware's source code. [13]

Malware authors often exceed the basic functionality of their code to implement a series of techniques to hide from computer users or to thwart analysis and detection. These techniques are uses to prevent a malware analyst completing his job, or to increase the skill level required to reverse engineer their malware. Automating the process of scrambling the code helps malware developers' gain more time in the detriment of the analyst. Anti-disassembly techniques are also good at preventing automated analysis techniques. Modern antivirus software and automated tools employ disassembly analysis to classify malware into families.

Sequences of code can have multiple representations, some of them may be obscure and hind the real functionality of the code. A good example of this is that disassemblers interpret each byte of code as part of only at a time, and therefor disassemblers can be tricked into disassembling the wrong code offset and the valid instruction been skipped. Popular methods take advantage of the presumptions that the disassembly software does when performing code analysis thus obscurity is accomplished.

Overlapping Code (code branching)

For example, the code fragment from Figure 8 has been disassembled by an automated tool.

Figure 8 - Wrong interpretation of assembly code

This fragment has been disassembled using a method called *linear-disassembly* and it resulted an inaccurate interpretation. This is a result of overlapping code. The *jump* instruction from address 0x00401005 makes and a conditional jump to address 0x0040100E and the disassembler does not process both paths because they overlap. To be more precise, the **XOR EAX, EAX** instruction will reset EAX and the **JZ** (jump if zero) instruction from address 0x401005 will always be met, thus the incorrect branch of an if-statement has been disassembled. This misinterpretation can be mitigated by manually specifying the disassembler to parse code from a specific address, which in our case is 0x0040100E.

Figure 9 displays the correct pattern of the code flow.

```
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 xor     eax, eax
.text:00401003 ; ---------------------------------------
.text:00401005                 db  74h ; t
.text:00401006                 db    7
.text:00401007                 db  0Fh
.text:00401008                 db  85h ; à
.text:00401009                 db    4
.text:0040100A                 db    0
.text:0040100B                 db    0
.text:0040100C                 db    0
.text:0040100D                 db 0E9h ; T
.text:0040100E ; ---------------------------------------
.text:0040100E                 push    3
.text:00401010                 push    offset unk_4020D0
.text:00401015                 mov     eax, [ebp+8]
.text:00401018                 push    eax
.text:00401019                 call    ds:_strnicmp
.text:0040101F                 add     esp, 0Ch
.text:00401022                 test    eax, eax
.text:00401024                 jnz     short loc_40102F
.text:00401026                 mov     eax, 1
.text:0040102B                 jmp     short loc_401031
```

Figure 9. Reassembled binary from address 0x0040100E

After the code logic has been corrected, the instructions now have a precise meaning and can be easily read by the analyst. A call to the API function *stricmp()* can now be observed including its parameters. Further adjustments can made to sanitize the code. Because the code branching was there only to mislead the analysis and had no effect on the programs behavior, in Figure 10 the instructions between addresses 0x00401005 and 0x0040100D have been replaced with **NOP** instructions.

After this final modification the code flow is easier to read and also helps future analysts whom may also review this project.

```
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 xor     eax, eax
.text:00401005                 nop
.text:00401006                 nop
.text:00401007                 nop
.text:00401008                 nop
.text:00401009                 nop
.text:0040100A                 nop
.text:0040100B                 nop
.text:0040100C                 nop
.text:0040100D                 nop
.text:0040100E                 push    3
.text:00401010                 push    offset unk_4020D0
.text:00401015                 mov     eax, [ebp+8]
.text:00401018                 push    eax
.text:00401019                 call    ds:_strnicmp
.text:0040101F                 add     esp, 0Ch
.text:00401022                 test    eax, eax
.text:00401024                 jnz     short loc_40102F
.text:00401026                 mov     eax, 1
.text:0040102B                 jmp     short loc_401031
```

Figure 10. NOP-ed out false instructions

Code Obfuscation (Packing and Encryption)

Code obfuscation is the action deliberately performed by the author of a software product to create difficult to understand code flow. Programmers use this technique to prevent competitors reverse engineer their legitimate software and then create a similar product, thus preserving intellectual property. Malware writers use this technique to elude analysis of malicious code, conceal its purpose and bypass antivirus scans. The same principle has also been applied to hardware devices with the same scopes.

Obfuscation is performed in all the programming languages including, C/C++, Java, .NET, JavaScript, PHP, Python, Assembly etc. This technique is most frequently applied using automated software that scramblers the code, but there are also manual and labor intensive approaches to achieve a more complex output in limited scenarios. Obfuscation does not alter the original code pattern of the software.

Multiple form of obfuscations can be performed on software. These include obfuscating source code or executable binary. The some of the most popular methods used to obfuscate include:

- Keyword substitution;
- Code packing;
- Code encryption;
- Junk Code.

Keyword substitution is a technique used to scramble variables' and functions' names into random names that have no meaning or resemblance to the scope of the object. Figure 11 illustrates this technique.

```
char*M,A,Z,E=40,J[40],T[40];main(C){for(*J=A=scanf(M="%d",&C);
--            E;             J[              E]             =T
[E   ]=  E)   printf("._");  for(;(A-=Z=!Z)  ||  (printf("\n|"
)    ,   A    =              39              ,C             --
)    ;   Z    ||    printf   (M   ))M[Z]=Z[A-(E   =A[J-Z])&&!C
&    A   ==             T[                                  A]
|6<<27<rand()||!C&!Z?J[T[E]=T[A]]=E,J[T[A]=A-Z]=A,"_.":" |"];}
```

Figure 11. Obfuscated source code

Code packing is another method to compress and obfuscate the executable file. Code encryption uses the same technique but adding symmetric keys to the algorithm. This is achieved by applying mathematical operations on the source code using a special software called a "packer". The newly generated executable includes a routine (usually at the beginning of the code) that decompresses the full binary inside the memory and then the

execution is passed to the newly decompressed code. Free packers can be downloaded from the Internet, but also some of them require a license which also have a better obfuscation algorithm or even multiple algorithms. Depending on the complexity of the packing algorithm, reverse engineers can identify the unpacking routine, take notice of the memory addresses passed as parameters, and then find the call/jump to the newly generated code and from there the unpacked code can be dumped from memory to disk.
There are multiple known packers, like UPX, PECompact, ASPack, Themida, FSG. In the following paragraphs UPX will be used as an example. Ultimate Packer for eXecutables (UPX) [14] is one of the most popular packers used because of its ease of use and free to download. The executable code, data, Import Address Table (IAT) and Import Descriptor Table (IDT) is compressed into a single section called UPX1 as Figure 12 illustrates. UPX0 is a placeholder for the unpacked code, UPX1 is the container of the packed code and UPX2 contains the unpacking routine.

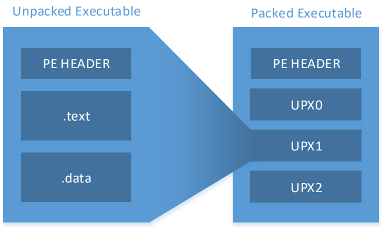

Figure 12 . UPX Compression example

As expected, when the packed executable is launched, the unpacking routine does the exact opposite. Figure 13 displays that the entry point of the packed executable is inside UPX2, and after the unpacking routine has finished the execution must jump back to the original entry point of the application. Also during the unpacking routine, the Import Address Table (IAT) and Import Descriptor Table (IDT) of the original executable are rebuilt.
Because UPX uses simple packing/unpacking routines, when using IDA to disassemble the code of the packed executable, it is easy to spot the jump to the unpacked executable. Figure 14 displays the code flow of the unpacking stub, and also highlights to jump to memory address that the disassembler sees empty.

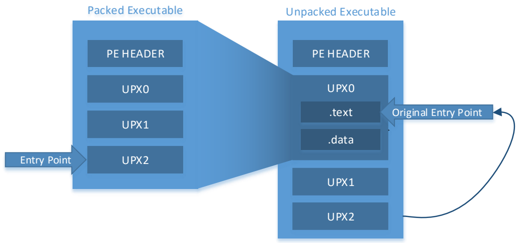

Figure 13. UPX decompression example

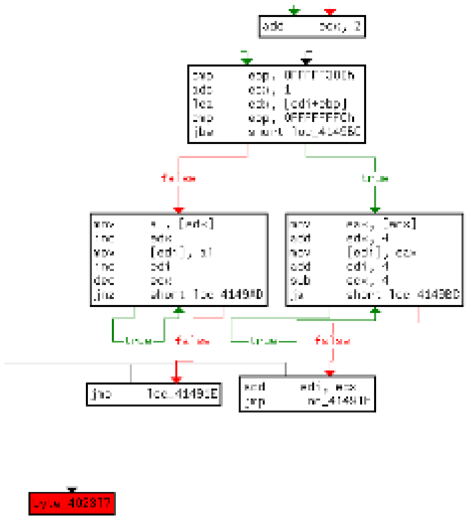

Figure 14. IDA Pro Graph of the unpacking routine

After the jump has been identified, the packed executable can now be opened in a debugger (like OllyDbg or WinDBG), execute the code until the jump, and dump the process from memory to continue analysis on the unpacked code.
There are cases when the "jump tail" cannot be identified by IDA, and the reverse engineer must follow code paths to find this jump. Stronger packing algorithms use multiple stubs to pack different parts of the code and also in insert fake code to deviate the analysis.

Anti-Debugging Techniques

This subchapter aims to classify and present some of the most popular anti-debugging techniques used on Windows NT-based operating systems.
Anti-debugging techniques are methods for an application to detect if it executes

under the control of a debugger. These techniques are used by commercial executable protectors, packers and malicious software, to prevent or slow-down the process of reverse-engineering.

IsDebuggerPresent() and PEB BeingDebugged Flag

The easiest way of detecting the presence of an attached debugger is by calling a function from Windows API named *IsDebuggerPresent.* This function determines whether the calling process is being debugged by a user-mode debugger. Figure 15 illustrates how the programmer can write his code to do a simple debugger check. This check can be easily circumvented by the analyst by patching the code where the check is made. Figure 16 displays a disassembled part of a code that uses this function to alter the code functionality. It can be observed that if a debugger is not detected, the executable will make a call to *GetCurrentProcessId()* and it will push it as an argument to the next function *sub_401794,* if the debugger is present, it will call another function and pass it 3 as argument.

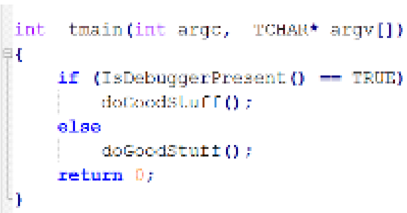

Figure 15 – IsDebuggerPresent

Figure 16. IsDebuggerPresent Code View

There are also scenarios when the programmers writes large amounts of checks in his code, and this implies that the reverse engineer should patch each check in the code. A simpler way to patch all of these debugger checks is by modifying a flag present in the Process Environment Block specific for that running process. The underlying mechanism of *IsDebuggerPresent*() checks a flag named *BeingDebugged* (**Error! Reference source not found.**) from the Process Environment Block which relies inside the user space of the operating system. If the flag is *TRUE*, the function will also return *TRUE*, thus by attaching a debugger to the application the flag can be set to FALSE and a global patch is applied (Figure 17).

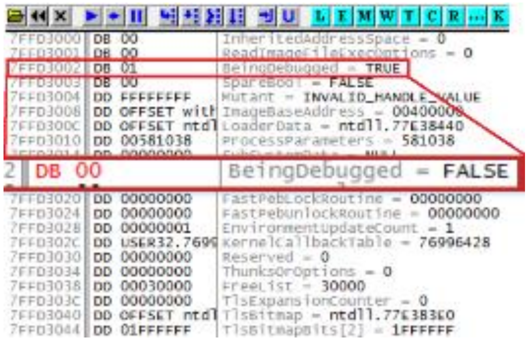

Figure 17. PEB Flags

Timing Checks

The theory behind timing checks is that an executing section of code, especially a small section, would only require a small amount of time. Therefore, if a timed section of code takes a greater amount of time than a certain set limit, then there is most likely a debugger attached, and someone is stepping through the code.

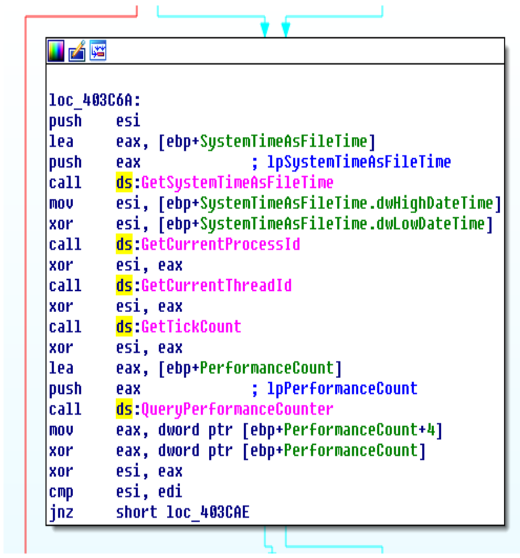

Figure 18. GetTickCount code view

This type of attacks has many small variations, and the most common example uses the IA-32 *RDTSC* instruction. Other methods utilize different timing methods such as *GetTime*, *GetTickCount,* in Figure

ISSN 2067 – 4074

18, and *QueryPerformanceCounter* (also illustrated in Figure 18). [15]

Read Time-Stamp Counter

RDTSC is an IA-32 instruction that stands for Read Time-Stamp Counter, which is pretty self-explanatory in itself. Processors since the Intel Pentium have had a counter attached to the processor that is incremented every clock cycle, and reset to 0 when the processor is reset. As you can see, this is a very powerful timing technique; however, Intel doesn't serialize the instruction; therefore, it is not guaranteed to be 100% accurate. This is why Microsoft encourages the use of its Win32 timing APIs since they're supposed to be as accurate as Windows can guarantee. The great thing about timing attacks, in general, though is that implementing the technique is rather simple; all a developer needs to do is decide which functions he or she would like to protect using a timing attack, and then he or she can simply surround the blocks of code in a timing block and can compare that to a programmer set limit, and can exit the program if the timed section takes too much time to execute. Figure 19 illustrates this concept: [15]

```
#define SERIAL_THRESHOLD 0x10000 // 10,000h ticks
DWORD GenerateSerial(TCHAR* pName)
{
    DWORD LocalSerial = 0;
    DWORD RdtscLow = 0; // TSC Low
      asm
    {
        rdtsc
        mov RdtscLow, eax
    }
    size_t strlen = _tcslen(pName);
    // Generate serial
    for(unsigned int i = 0; i < strlen; i++)
    {
        LocalSerial += (DWORD) pName[i];
        LocalSerial ^= 0xDEADBEEF;
    }
      asm
    {
        rdtsc
        sub eax, RdtscLow
        cmp eax, SERIAL_THRESHOLD
        jbe NotDebugged
        push 0
        call ExitProcess
        NotDebugged:
    }
    return LocalSerial;
}
```

Figure 19 – RDTSC source code example

Win32 Timing Functions

The concepts are exactly the same in this variation except that this has a different approach of timing the function. In Figure 20, *GetTickCount()* is used, but as commented, could be replaced with *GetTime()* or *QueryPerformanceCounter()*.

```
#define SERIAL_THRESHOLD 0x10000 // 10,000h ticks
GenerateSerialWin32Attack(TCHAR* pName)
{
    DWORD LocalSerial = 0;
    size_t strlen = _tcslen(pName);
    // Could be replaced with timeGetTime()
    DWORD Counter = GetTickCount();
    // Generate serial
    for(unsigned int i = 0; i < strlen; i++)
    {
        LocalSerial += (DWORD) pName[i];
        LocalSerial ^= 0xDEADBEEF;
    }
    // Could be replaced with timeGetTime()
    Counter = GetTickCount() - Counter;
    if(Counter >= SERIAL_THRESHOLD)
        ExitProcess(0);

    return LocalSerial;
}
```

Figure 20. Source Code Timing Checks

Malware Behaviour

This subchapter aims to classify and present some of the most popular techniques used by malware authors to hide malicious activity from the user, interpose between windows API functions, modify code on execution etc.

Process Hollowing

Process hollowing is another mechanism of those that seek to hide the presence of a malicious process. A bootstrap application generates a seemingly legitimate process in a suspended state (i.e. svchost.exe), then the legitimate process is then unmapped and replaced with the code that is to be hidden from the user. If the preferred image base of the new image does not match that of the old image, the new image must be rebased. Once the new image is loaded in memory the EAX register of the suspended thread is set to the entry point. The process is then resumed and the entry point of the new image is executed.

To successfully perform process hollowing the source image must meet a few requirements:

1. To increase compatibility, the subsystem of the source image should be set to windows.
2. The compiler should use the static version of the run-time library to remove dependence to the Visual C++ runtime DLL. This can be achieved by using the /MT or /MTd compiler options.
3. Either the preferred base address (assuming it has one) of the source image must match that of the destination image, or the source must

contain a relocation table and the image needs to be rebased to the address of the destination. For compatibility reasons the rebasing route is preferred. The /DYNAMICBASE or /FIXED:NO linker options can be used to generate a relocation table.

Figure 21 exemplifies the steps taken to achieve a successful hollowing. First the target process must be launched in a suspended state by passing the *CREATE_SUSPENDED* flag to the *CreateProcess()*. Once the process is run, its memory space can be modified. Next, the base address of the destination image must be located by querying the process with *NtQueryProcessInformation()* to acquire the address of the process environment block (PEB). Next, a new block of memory is allocated for the source image. The size of the block is determined by the *SizeOfImage()* member of the source images optional header. Usually to simplify the code, the author will flag the entire block as *PAGE_EXECUTE_READWRITE*. After the memory has been allocated, the new image must be copied to the destination process memory by using *WriteProcessMemory()* starting with its portable executable headers. Following that, the data of each section is copied. By applying the proper memory protection options to the different sections would make the hollowing tougher to detect. Finally, the thread is resumed, executing the entry point of the source image. [17]

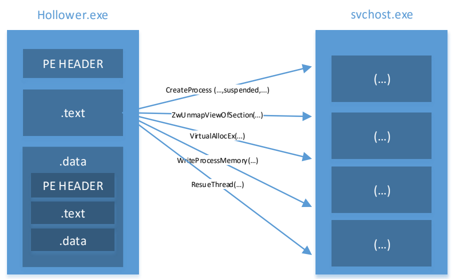

Figure 21. Process Hollowing Technique

The most common use of this technique is seen using svchost.exe as target, but it can be also used on other windows processes. Svchost.exe is used often because it has multiple instances which can be seen in the task manager and it easy to escape from user detection. Other windows processes have only one instance and it would be very easy for an experienced user to spot two identical processes when only one should be present. **Error! Reference source not found.** represents a screenshot from the pseudocode view of a malware sample that uses process hollowing to hide the malicious process.

Figure 22. Process Hollowing Pseudocode view

ISSN 2067 – 4074

Process/DLL Injection

Code injection is a technique used by programmers to run code within the address space of another process. There are multiple techniques used to achieve the injection. These techniques are used to influence the behaviour of the targeted program by adding new components to the running process. Often this technique is used add malicious behaviour to processes or they are used to apply Windows hot patch updates which do not require operating system reboot.

The injected code could hook system function calls, steal identifiable information, or do other malicious activity in the name of a legitimate process. Code injection is a way of hiding from automated or human detection of malicious code, usually incident response personnel search for malicious processes with odd disk paths. This impersonation is beneficial for bypassing restrictions enforced by the operating system on a particular process. It's important to note that appropriate level of privileges are required on the system to start manipulating with other program's memory. [18]

Code injection can be performed in both user mode processes and also into kernel mode processes. The most popular method used to achieve user mode injection include Windows API functions, AppInit_DLL and Detours also known as Function Hooking. [18]

The functions provided by the Windows API used to achieve process injection are:

- *OpenProcess()* - used to attach to the running process;
- *VirtualAllocEx()* – used to allocate memory inside the process;
- *WriteProcessMemory()* – used to copy the code into the process memory and also determine the appropriate address in memory;
- CreateRemoteThread() - used to instruct the targeted process to execute the injected code;
- SetWindowsHookEx() - used to create a hook between Windows API functions.

These functions can be used in different techniques and are summarised in Figure 23. [19]

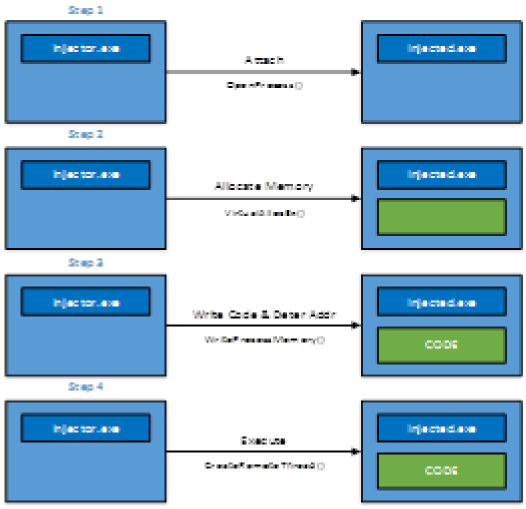

Figure 23 - Code Injection using APIs

In the first step a handle to the target process is acquired so that *injector.exe* can interact with *injected.exe.* This is achieved by calling *OpenProcess()* function and then requesting access rights in order to perform the next steps. The second step is responsible with allocating memory in the targeted process to be able to copy code in it. *VirtualAllocEx()* takes the amount of memory to allocate as one of its parameters and creates new space with the desired length. Step three is responsible with copying the new code into the process by using *WriteProcessMemory()* function. Most execution functions take a memory address to start from and that address must be identified. The starting address can be searched in memory by using *LoadLibraryA()*. And finally, the last step is to execute the new code into a separate thread. The *CreateRemoteThread()* is probably the most used function, it is very reliable but others can be used to avoid detection.

Function Hooking (IDT, SSDT, IAT, IRP)

In the Windows operating system, a hook is a mechanism that enables a function to capture events before they reach the application. The routine can perform different tasks on events and modify or discard them. Functions that obtain events are named filter functions and are classified according to the type of event they capture. In order for the operating system to call a filter function, the function must be installed and attached to an operating system hook. Attaching one or more filter functions to a hook is known as setting a hook. If a hook has more than one filter function attached, the operating system upholds a chain of filter functions. When a hook has multiple filter functions attached and an event activates the hook, the operating system calls the first filter function in the filter function chain.

The term kernel space refers to any function that resides in the kernel space of the operating system, and thus for a user application to call one of these, it must enter the kernel space via SYSENTER. The first three functions can be hooked from user mode, the others require a kernel mode driver to enable hooking. By hooking at any point in the flow chart, the function is able to intercept and tamper data that passed through, as it can be seen in Figure 24.

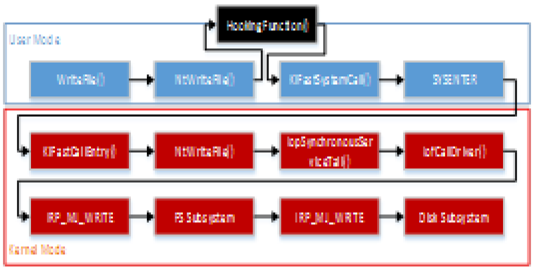

Figure 24. NtWriteFile() Hook

There are multiple types of hooks, they are categorized by the type of functions that the hook is applied to. Further, three types of hooks will be summarized.

IAT Hooks - The Import Address Table (IAT) is a table specific to each application, it contains a series of jumps to certain API functions. Because functions in DLLs change addresses, functions are called using a jump within their own jump table. When the application is executed the loader will place a pointer to each required DLL function at the appropriate address in the IAT. If an application injects inside another, it can modify the addresses in the IAT and then be able to receive control every time a function is called.

Inline hooking is a method of gaining control when a function is called, but before the called function completed execution. The flow of execution is redirected by adjusting the first few bytes of a target function. A method of achieving this is to overwrite the first five bytes of the function with a jump to malicious code, the malicious function can then read the original function arguments and do whatever it desires. If the malicious function needs results from the original function, it may call the function by executing the five bytes that were replaced then jump five bytes into the original function, which will miss the malicious call/jump to escape infinite loops. The concept is exemplified in

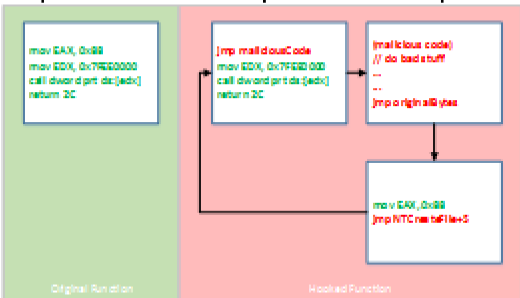

Figure 25

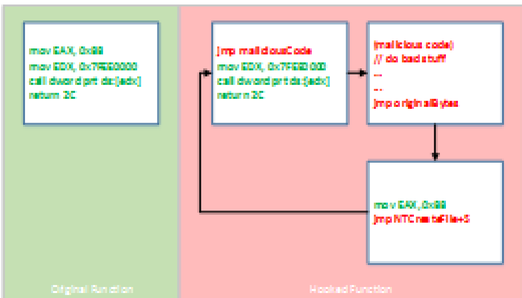

Figure 25. Inline Hook NtCreateFile()

In user mode, inline hooks are usually located inside functions that are exported by a DLL. The most effective way of detecting and bypassing these hooks is to compare each DLL against the original code. An application would need to get a list of loaded DLLs, find the original files, align and load the sections into memory. Since now the DLL has copy in memory, the application can parse the export address table and relate each function

ISSN 2067 – 4074

with the original one from the copied DLL. In order to bypass hooks, an application can then either replace the overwritten code using the code from the newly loaded DLL, alternatively, it could resolve imports in the newly loaded DLL and use it instead. This technique of bypassing DLL hooks practically involves writing a custom implementation of *LoadLibrary()*.

In kernel mode, inter-modular jumps are not frequently implemented. Hooks in ntoskrnl can usually be detected by disassembling each instruction in each function, then looking for jumps or calls that point outside of ntoskrnl and into other modules. Also, the method described in user mode can be applied here.

SSDT Hooks - The System Service Dispatch Table (SSDT) is a table of pointers for various Zw/Nt functions, which are callable from user mode. A malicious application can replace pointers in the SSDT with pointers to its own code. All pointers from the SSDT should point inside of ntoskrnl, if a pointer relates outside of ntsokrnl, then it is a strong indicator that it has been hooked. It is possible for a rootkit to modify ntoskrnl.exe in memory and slip some code into an empty space, in this case the pointer would still point to within ntoskrnl.

IRP Hooks – Each loaded driver contains a table of 28 function pointer, these pointer are can be called by other drivers via *IoCallDriver()*, the pointers correspond to operations such as read/write (IRP_MJ_READ/IRP_MJ_WRITE). These pointers can easily be replaced by other drivers. Generally all IRP major function pointers for a driver should point to code within the driver's address space, but there are also scenarios when this "rule" can be broken, but nevertheless it is a good step towards detecting malicious drivers which have redirected the IRP major functions of legitimate drivers to their own code.

Rootkits and Bootkits

A rootkit is an application used to hide its activity from the user by modifying structures within the operating. The term rootkit is composed of the word "root" which refers to the privileged user from UNIX operating systems and the word "kit" which refers to the software component.

Rootkit detection is difficult because it is able cu circumvent the software tools that intend to discover it. For a detection tool to be effective, it must interrogate the operating system by using default methods and alternative ones and then compare the two results to distinguish differences.

There are two types of rootkits, those that reside inside the user space among other applications, these are the least advanced types and can be mitigated easily and those that reside inside the kernel space of the operating system and apply more advanced techniques to evade detection. The kernel side rootkit must use drivers to be able to interact with the operating systems. Windows x64 operating system require drivers to be digitally signed before loading them into the kernel space. This check was introduced to confirm that the code has not been altered or corrupted, to confirm the software author thus strengthening operating system security. There have been cases when this mechanism has been bypassed, FLAME exploited an MD5 collision and managed to use an existing legitimate certificate and also others like Stuxnet and Mediyes used stolen singing certificates. [20]

Rootkits that reside in kernel space and can infect the workstation early in the booting phase of the operating system is called a Bootkit. It accomplishes this by injecting code into the Master Boot Record (MBR) to point execution toy larger chunks of code inside unused space on the disk. Usually the bootkit loader persists through the transition from real mode to protected mode when the kernel has loaded, and is thus able to subvert the kernel.

Some popular rootkits throughout history include: [20]

- He4Hook, was discovered in 2003, installed SSDT hooks and dispatch functions;
- FU was discovered in 2004 and installed EProcess hooks;

- Mebroot was one of the first Bootkits and it was discovered in 2007;
- TDL3 and Kobcka were discovered in 2009 and used drivers to infect the operating system.

A rootkit can implement different techniques to accomplish its goal:

- IAT hooks in user-mode;
- patching user-mode DLLs;
- Sysenter hooks, alter CPU MSR registers;
- IDT (int 2Eh – sysenter equivalent);
- SSDT hooks;
- Patching kernel routines;
- Alter kernel objects;
- Install filter drivers;
- Hook dispatch routines of drivers;
- Different drivers can be patched in the driver stack, ex: ntfs.sys (partition level), atapi.sys (sector level – lower).

Direct Kernel Object Manipulation

DKOM is a common technique used by rootkits to hide malicious processes, drivers, files, network connections from the task manager. It accomplishes this by modifying a doubly linked list of all active processes. This is possible due to the fact that every component from the kernel space has access to any other. Because the kernel assigns processor resources to threads and not processes, the EPROCESS list can be modified without having consequences on the stability of the operating system. Figure 26 illustrates this concept of handing a process by modifying links within this list.

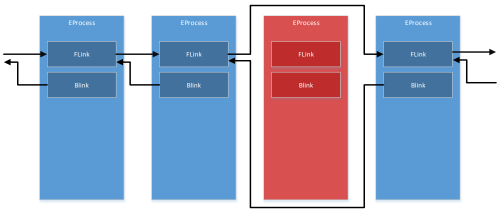

Figure 26. DKOM

The best approach of finding a hidden process is to create a tool that can read and parse the kernel space and search for EPROCESS structures that have no other structures pointing to it. This technique can also be applied to dumping the RAM contents and parsing the dump offline (onto an uninfected machine). After a hidden process has been discovered, the analyst can continue with dumping the process from memory and continue analysis by disassembling the code.

Driver dispatch routines hooking

The DRIER_OBJECT structure of a driver must be obtained in order to accomplish driver dispatch. Some of the most common functions are replaces with the rootkit functions, like IoGetDeviceObjectPointer, IoGetLowerDeviceObject, IoGetDeviceAttachmentBaseRef. Typically the disk drivers are hooked to filter access to files and sector which contain sensitive code of the malware.

Domain Generation Algorithm

Domain Generation Algorithms are used by malware authors to generate domains based on a seed which is derived from a calendaristic value. These domains are generated for malware to connect to a command and control center. The malware author also uses this algorithm to register domains ahead of time and after the time period has expired, the domain is then deleted. The large number of domains generated by this technique makes difficult for law enforcement to effectively shut down botnets because bots will attempt to contact only those domains that should be active in that period of time. This technique was popularized by the Conficker.
Typically, these algorithms use large array of words and generate a domain by choosing two or three word from the array and concatenating them and appending a top level main at the end. (i.e. [word1][word2].com) There are also scenarios where the DGA will generate domains that are composed of numbers and letters with a certain length and do not compose a meaningful word and look very random.
Third parties (Law enforcements or hackers can) can use these algorithms in their advantage. They can reverse engineer the malware and find the DGA function, replicate its functionality, generate and register domains before the

bot master. By doing this, law enforcements can replicate a command and control center to send commands to bots to uninstall themselves, or other hacker can use this technique to install other malware and steal the botnet from the original bot master. This operation of generating in advance the domains to capture bots is called DNS Sinkhole.

Over the years, malware authors have learned how to circumvent the DNS sinkhole and introduced into their malware a routine that is responsible with receiving digitally signed commands. If the received command cannot be verified and authenticated, the bot will drop the command and continue its activity.

The diagram from Figure 27 displays how the Dyreza banking malware generates its domains based on the current date and a hardcoded number range between [0,333). The example illustrates the DGA for the date July 4th 2015, and uses as input the number 16. It is very easy to understand that this algorithm can generate 333 domains every day.

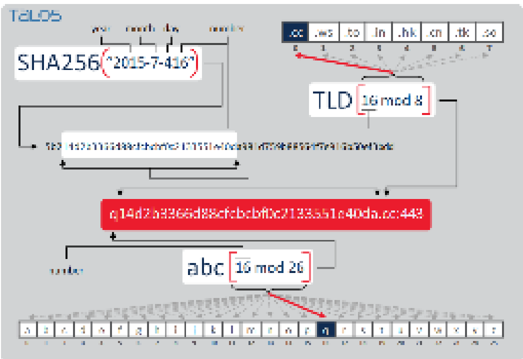

Figure 27. DGA used in Dyre Malware [21]

Polymorphism, Metamorphism and Self Modifying Code

Polymorphic code refers to code that uses a polymorphic engine to mutate the application while maintaining the code flow intact. The code is able to change itself each time it is ran but the functions of the code remain unchanged. This technique is often seen in shellcode malware and worms to avoid antivirus detection that is performed statically on the executable by doing string comparisons and polymorphism is not a mechanism used to protect itself against debugging. The most common way of achieving this is to encrypt or to pack certain parts of the code with different seeds which are also hardcoded in the application.

A limitation of polymorphic code is that decryption routine is basically the same in each sample, thus memory based signatures detections are possible to create and also hashing the code by dividing it into separate block and comparing it to a database can also help detection.

Metamorphic malware automatically adds various sequences of code each time it propagates. Some of the simplest techniques include register permutation, adding NOP instructions to create different offsets inside the code, adding useless instructions that have no impact on the code flow, thus only by applying these methods, signatures are harder to create. More advanced techniques used in metamorphic malware include reordering of functions and changing code flow.

Self-modifying code is a used to describe an executable that is able to modify it self at runtime. By writing self-modifying code, it is then harder to reverse engineer mainly because the actual code may differ between executions and fallowed execution paths. Early operating systems did not implement any form of mechanism to protect memory, applications were able to write all over the memory. Modern operating systems have divided memory into blocks that have associated different permissions. These permissions are enforced by the operating system with the help of the hardware implementation within the microprocessor. Memory blocks can be assigned different flags, like readable, executable, writeable and combinations between. Self-modifying code must have the memory pages in which it resides marked as executable (to enable code to be run from them) and writeable (to allow memory modification). Microsoft Windows operating system generally allocates non-executable pages for data and for code sections it allocates read-only pages. These are implemented to prevent buffer overflow attacks and execute unauthorized code. As depicted previously, self-modifying code must change their memory page permissions and add the Write permission in order to make modifications of the code as Figure 28 exemplifies. [22]

```
/*
 * H: call _VirtualProtect@16() to modify the page permissions by
 * adding WRITE
 * %edx points to start of MEMORY_BASIC_INFORMATION structure
 *      .Protect needs to change from 0x20 (PAGE_EXECUTE_READ) to
 *      0x40 (PAGE_EXECUTE_READWRITE)
 */
pushl        %eax                /* save value of eax register */

subl $0x4,%esp      /* leave space on stack for lpflOldProtect */
pushl        %esp                      /* addr of lpflOldProtect */
pushl        $0x40                     /* flNewProtect */
pushl        $0x3fff                   /* dwSize */
pushl        (%edx)                    /* lpAddress */
call _VirtualProtect@16
```

Figure 28. VirtualProtect Call

Now, that the memory page has been set as writeable, the code can start changing its code. Assuming that upper in the code there is present a NOP instruction (Figure 29), the code exemplified by Figure 30 allowed it to be changed into **inc %ebx.**

```
/*
 * J: The following 'nop' instruction is a place holder
 * to replace it with an 'inc %ebx' instruction.
 */
nop
```

Figure 29. NOP placeholder

```
cmpl     $0x0,%eax
je       skippl

/*
 * W: If eax is not zero, then modify the 'nop' instruction to
 * 'inc %ebx' and loop back to the address in eax.
 */

/* Sit tight -- this may hurt a little */
movb     $0x43,0x1(%eax)
```

Figure 30. Change NOP to INC EBX

This is a very simple example, with no major implication on the executed code, but this technique can be scaled to much higher capabilities. Combining it with debugging checks, a malware author can instruct its malware to dramatically change code execution or simply delete the malicious parts of it and thus making the reversing of software much more difficult.

Persistence

After the infection of a computer has taken place, the malware has to maintain control on the machine by implementing some techniques to guarantee activity after a reboot. The most common used techniques used in Windows operating system involve the use of registries and start-up folder. Investigating malware persistence locations in the Windows Registry and start-up locations is a frequent technique employed by forensic investigators to identify malicious software installed on a host. Besides these common and easy to use techniques, there is also some other that does not leave any forensic trail behind by simply placing a DLL in some specific directories with a specific name and the operating system will load it without any further checks. There are a total of 1032 paths and name combinations in which a 32 bit DLL can be placed and automatically be loaded at boot. 64 bit DLLs have a much more diverse range of paths and names because Windows OS has more 64 bit running processes running at boot time.

DLL Search Order Hijacking

When an application requires a DLL to be loaded either by statically importing it into the executable or by using *LoadLibrary(),* windows operating system will search that DLL in some predefined sequence of locations. Figure 31 displays the order in which the search is accomplished in the windows operating system. An important information to keep in mind from this figure is that the first place the application looks for a DLL is in the root directory of the executable itself. If the requested DLL name is listed in the "\\.\KnownDlls" object then it will be loaded from the System32 folder. This object is populated at boot-time using data from the following registry key HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs. [23]

ISSN 2067 – 4074

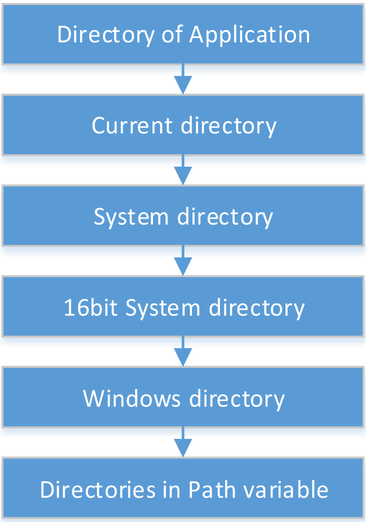

Figure 31. DLL Search Order

KnownDlls will reveal a list of about 30 of the most used DLLs, ws2_32.dll is the DLL used for networking and it is present within that list, thus the windows operating system will always load it from system32 regardless of the path from which the application is launched. This can is a security feature applied to the most important system DLLs which prohibits an attacker to load its ws2_32.dll instead of the original one. There are also DLLs important to the operating system that are not present in KnownDlls and DLL Search Order can be hijacked. Two of these DLLs are iphlpapi.dll and mswsock.dll. [23]
Executables inside system32 are not susceptible to this attack and malware authors must rely on other techniques to infect a host. Explorer.exe is a critical executable that resides in the Windows directory and it requires DLLs that are not inside KnownDlls thus permitting attackers to place ntshrui.dll along with explorer.exe instead of it being in system32. This is more of a forensic scenario but it is important to be known also by the malware analyst. [23]
This problem long existed in windows operating system and may also persist in the future due to compatibility with older software.

8. Conclusion and Future Work

As technology evolves and money shifts into the digital era, so do modern perpetrators migrate criminal activity from the physical world to the digital space. Hackers will always find ways to circumvent detection by developing means to infect digital devices, hide against forensic tools and take benefit of infected hosts. Botnets are a strong example of modern crime, and popular ones such as Zeus, SpyEye or Conficker are created to initiate distributed denial of service attacks, email spamming, cryptographic currency mining, stealing personal identifiable information etc. Advanced Persistent Threats are the finest example of what a politically motivated group can achieve. Campaigns such as Stuxnet, Flame and Regin are some of the most popular attacks known today.
As depicted in this paper, the digital space is the new place in which criminals and governmental entities migrated to make money, steal or sabotage other entities. These are strong reasons why resources should be invested for malware analysis. The paper exemplifies some techniques used by malware authors to write code, how the malware code can be analyzed and understood to further mitigate infection and assess damage of computer infrastructures.
With smartphones becoming more accessible, in the past few years they gained popularity and various companies now develop high-end terminals and several operating systems to choose from like Android, iOS, Windows etc. Basically a smartphone is a pocket sized computer and is treated as so by implementing functionalities seen on the personal computer. Because they store more personal data about their owner than PCs do, hackers started to take interest in them and developed ways of infecting mobile operating systems. Today, Android OS is the most targeted mobile platform because its majority in the smartphone market. Mobile malware is still into its early ages but it is thought to gain popularity as the years pass by. [20]
Because of the rising interest of malware developers in mobile devices, companies started to develop antivirus engines, imaging tools, virtual environments, decompilers and memory analysis tools to also improve security in this niche.

A team of developers published malware written for Graphical Processing Units (GPUs), this is a proof-of-concept that includes a key logger and a rootkit. The main advantage of this kind of malware is the lack of forensic tools that can assess these threats. GPUs benefit of faster processing power for mathematical calculations which are used for encryption, has more processing cores than the microprocessor and it can also interact with the host's memory by using the Direct Memory Access protocols. [25]
Considering that modern microprocessors include graphical units inside their die, it is likely that this concept could become a reality and the forensic industry will have develop tools to assess this niche.
An important aspect of this paper was the opportunity to work with live malware, gain capabilities into malware research, understand and apply reverse engineering skills using tools used in the computer security sector. Malware analysis is a science with ever evolving competitors, new techniques are applied to malicious software and the reverse engineering industry needs to compete and provide means to enable analysis even when malware writers change approach, targeted architectures or programming languages. The paper assed some of the most popular techniques used by both black and white hat entities and exemplified the targeted infrastructures such SCADA systems and x86 architectures.

Acknowledgement

Parts of this paper were presented at The 8th International Conference on Security for Information Technology and Communications (SECITC 2015), Bucharest, Romania, 11-12 June 2015.

References

[1] http://en.wikipedia.org/wiki/Malware#History_of_viruses_and_worms
[2] http://www.cisco.com/web/about/security/intelligence/virus-worm-diffs.html
[3] https://cve.mitre.org/
[4] https://software.imdea.org/~juanca/papers/ppi_usenixsec11.pdf
[5] http://securityintelligence.com/3-ways-steal-corporate-credentials/#.VTU8WfmUd8E
[6] http://www.wordstream.com/black-hat-seo
[7] http://en.wikipedia.org/wiki/Ransomware
[8] http://www.kaspersky.com/about/news/virus/2013/Kaspersky_Lab_Identifies_Operation_Red_October_an_Advanced_Cyber_Espionage_Campaign_Targeting_Diplomatic_and_Government_Institutions_Worldwide
[9] http://www.symantec.com/connect/blogs/regin-top-tier-espionage-tool-enables-stealthy-surveillance
[10] http://en.wikipedia.org/wiki/Regin_(malware)#cite_note-intercept20041124-3
[11] http://www.symantec.com/connect/blogs/regin-top-tier-espionage-tool-enables-stealthy-surveillance
[12] http://en.wikipedia.org/wiki/Flame_%28malware%29
[13] The "Practical Malware Analysis" book by Michael Sikorski and Andrew Honig
[14] http://upx.sourceforge.net/
[15] http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide
[16] https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf
[17] http://www.autosectools.com/process-hollowing.pdf
[18] https://www.blackhat.com/presentations/bh-usa-07/Butler_and_Kendall/Presentation/bh-usa-07-butler_and_kendall.pdf
[19] http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html
[20] Bitdefender "Reverse Engineering Malware" course notes
[21] http://blogs.cisco.com/security/talos/threat-spotlight-dyre
[22] http://malwaremusings.com/2012/10/13/self-modifying-code-changing-memory-protection/
[23] Mandiant "Advanced Malware Analysis" course notes
[24] https://www.hex-rays.com/products/ida/support/download.shtml
[25] http://arstechnica.com/security/2015/05/gpu-based-rootkit-and-keylogger-offer-superior-stealth-and-computing-power/
[26] http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf