School of Information Technology and Communications
Department of Digital Systems

Master of Science

Digital Systems Security

Real World Malware Analysis_

STARTEX Ransomware

Master Thesis in Computer Science

Author: Papadopoulos Polymenis - Fotios MTE1629

Supervisor: Professor Dr. Christoforos Ntantogian Uni.Pi.-D.S.

Piraeus, February 2019

Abstract

The goal of this paper is to analyze a real-world malware, step by step, from an academic perspective. The steps to be followed, are predefined, from Basic Analysis to Advanced Static and Dynamic Analysis. There will be a detailed description of the techniques, the tools and the architecture of the lab environment. Consider that the purpose of this paper is to analyze malware once it has been found and not to reveal the malware. The under-examination malware is a ransomware, found on the Windows operating system, by far the most common operating system in use today. But the techniques and the procedures that will be used to analyze it, could work on any operating system, as long as executables would be mainly examined. Notice that, executables are the most common and the most difficult files that an incident response team will encounter.

Keywords: ransomware, malware analysis

Acknowledgments

It is a great pleasure to acknowledge everyone who helped me writing this thesis successfully. First of all, I would like to thank my advisor Professor Dr. Christoforos Ntantogian. He gave me the opportunity and the flexibility to solely focus on research topics that I was interested in, without any pressure or dictation at all. Besides that, he offered me productive environments at the System Security Laboratory within the Department of Digital Systems of the University of Piraeus.

During my master sessions, I enjoyed working with friendly and talented research staff and enthusiastic graduate students, that share the same outstanding enthusiasm and expertise for security research in Computer Science. I have been enlightened by many valuable discussions with all my Professors; C. Xenakis, C. Lambrinoudakis, S. Katsikas, L. Mitrou and P. Rizomiliotis.

I would like to show my gratitude to colleagues in the CyberCrime Prosecution Division of Hellenic Police, since they make my daily work fun and give me that satisfying feeling of belonging to some great community that is pulling in the same direction.

I owe sincere and earnest appreciation to my section head Alexandros Filippidis, my division head Vasilios Papakostas and my former division head Georgios Papaprodromou, for their trust in me and being helpful facilitating with the attendance days of courses. This gave me the autonomy to follow my aims without any constraints and provided me with the chance to get into touch with so many interesting and helpful people in the academic field.

Finally, I would like to express my gratitude to mother, to my companion and especially dedicate my work to my deceased father and my recently deceased siblings, grandmother, grandfather and aunt. My deepest thankfulness goes to them for their love, understanding, and inspiration. Without their blessings and encouragement, I would not have been able to either start or finish this work.

Contents

1. Introduction

1.1 Definition of Malware

Malware, a shortened form of malicious software, is defined as the software that does something that causes harm to a user, computer, or network. Malwares play a part in most computer intrusion and security incidents. The ultimate goal of gaining control is to disrupt the normal operations of the target, obtain sensitive or secret information, or gain access to private computer networks and system for other purposes. For the end user, malware is just software that is doing nasty things to them or their computers, without them knowledge or permission. Some kind of software that can be considered malwares, are viruses, trojan horses, worms, rootkits, scarewares, adwares, spywares and ransomwares.

1.2 Ransomware, the key threat

1.2.1 IOCTA 2017 [i]

By the end of 2016 we had witnessed the first massive attack originating from such devices, as the Mirai malware[1] transformed around 150.000 routers and CCTV cameras into a DDoS botnet. This botnet was responsible for a number of high-profile attacks, including one severely disrupting internet infrastructure on the west coast of the United States).

Ransomware attacks have eclipsed most other global cybercrime threats, with the first half of 2017 witnessing ransomware attacks on a scale previously unseen following the emergence of

---

[1] Mirai is a malware that turns networked devices running Linux into remotely controlled "bots" that can be used as part of a botnet in large-scale network attacks. It primarily targets online consumer devices such as IP cameras and home routers. The Mirai botnet was first found in August 2016 by MalwareMustDie, a whitehat malware research group, and has been used in some of the largest and most disruptive distributed denial of service (DDoS) attacks. Reference source: https://en.wikipedia.org/wiki/Mirai_(malware)

self-propagating 'ransomworms', as observed in the WannaCry and Petya/NotPetya cases. Moreover, while information-stealing malware such as banking Trojans remain a key threat, they often have a limited target profile. Ransomware has widened the range of potential malware victims, impacting victims indiscriminately across multiple industries in both the private and public sectors, and highlighting how connectivity and poor digital hygiene and security practices can allow such a threat to quickly spread and expand the attack vector.

The primary targets - key threat for the majority of cyber-dependent crimes are vulnerable software products, insecure, internet-connected devices or networks, and the users and data behind them. As such, the development and propagation of malware typically sits at the core of cyber-dependent crime. Malware can be coded or repurposed to perform almost any function; however, the two dominant malware threats encountered by EU law enforcement continue to be ransomware and information stealers.

Comparatively, ransomware is easier to monetise. Beyond the initial infection, all the attacker has to do is collect the ransom payment, and by using pseudonymous currencies such as Bitcoin, the subsequent laundering and monetisation is considerably simpler. Furthermore, the nature of the attack means that ransomware can inherently target a much more diverse range of targets – essentially anyone with data to protect – with little requirement for adaption. Victims are atypical from the usual financial targets, and include entities such as hospitals, law enforcement agencies, and government departments and services. While the public also continues to be targeted, small to medium enterprises, who often lack the resources to fully safeguard their data and networks, are also key targets. The success and the demand for ransomware resulted in an explosion in the number of ransomware families throughout 2016, with some reports highlighting an increase of 750% from 2015[2]. The business model for ransomware has also evolved. Developers of early iterations of ransomware produced it for their own use, but now variants such as Satan[3] or Shark[4] are run as affiliate programs, providing ransomware-as-a-service in exchange for a share of the criminal proceeds. The surge in ransomware is also reflected in this year's reporting, with almost every Member State reporting a growing number of cases. Throughout 2016, the emerging threats highlighted in the previous year's report, Locky[5] and Cerber[6], were the most prominent ransomwares. A number of other ransomwares, including CTB-Locker[7], Cryptowall[8], Crysis[9],

[2] Trend Micro, 2017, TrendLabs 2016 Security Roundup, p4

[3] The name "Satan ransomware" is aptly chosen in this regard. The platform acts as a gateway to hell where new minions can be spawned who must contribute a bounty to the Lord of Hell. The platform is so much bigger than just a new type of ransomware users to deal with, as it can create different types of offspring with relative ease. Anyone making use of this service will be hunted down by law enforcement agents, though, as deliberately distributing malware is illegal in most global jurisdictions. Reference source: https://themerkle.com/bitcoin-ransomware-education-satan/

[4] Symantec, Internet Security Threat Report, 2017, p61

[5] Locky is ransomware malware released in 2016. It is delivered by email (that is allegedly an invoice requiring payment) with an attached Microsoft Word document that contains malicious macros. Filenames are converted to a unique 16 letter and number combination. Initially, only the .locky file extension was used for these encrypted files. Subsequently, other file extensions have been used, including .zepto, .odin, .aesir, .thor, and .zzzzz. After encryption, a message (displayed on the user's desktop) instructs them to download the Tor browser and visit a specific criminal-operated Web site for further information. Since the criminals possess the private key and the remote servers are controlled by them, the victims are motivated to pay to decrypt their files. Reference source: https://en.wikipedia.org/wiki/Locky

[6] Ransom.Cerber is a ransomware application that uses a ransomware-as-a-service (RaaS) model where affiliates purchase and then subsequently spread the malware. Commissions are paid to the developers for the use of the malware. Ransom.Cerber uses strong encryption, and there are currently no free decryptors available. Reference source: https://blog.malwarebytes.com/detections/ransom-cerber/

[7] CTB-Locker emerged in June 2014 and is one of the first ransomware variants to use Tor for its C2 infrastructure. CTB-Locker uses Tor exclusively for its C2 servers and only connects to the C2 after encrypting victims' files. Additionally, unlike other ransomware variants that utilize the Tor network for some communication, the Tor components are embedded in the CTB-Locker malware, making it more efficient and harder to detect. CTB-Locker is spread through drive-by downloads and spam emails. Reference source: http://itlaw.wikia.com/wiki/CTB-Locker

[8] Ransom.Cryptowall is a Trojan horse that encrypts files on the compromised computer. It then asks the user to pay to have the files decrypted. The threat typically arrives on the affected computer through spam emails, exploit kits hosted through malicious ads or compromised sites, or other malware. Reference source: https://www.symantec.com/security-center/writeup/2014-061923-2824-99

[9] CrySiS is a ransomware virus that was spotted back in March 2016 and is still active today. Since its initial release, malware had multiple updates, changing the file extension and the contact email to a different one. Reference source: https://www.2-spyware.com/remove-crysis-ransomware-virus.html

Teslacrypt[10], Torrentlocker[11] and Zepto[12] were also reported, but these appear to be localised to specific countries. On 12 May 2017 however, all other ransomware activity was eclipsed by a global ransomware attack of unprecedented scale. While reports vary, the WannaCry ransomware is believed to have rapidly infected up to 300.000 victims in over 150 countries, including a number of high-profile targets such as the UK's National Health Service, Spanish telecommunication company Telefónica, and logistics company Fed-Ex.

There were several key factors in the success of the WannaCry attack. Firstly, unlike most ransomware, WannaCry used the self-propagating functionality of a worm to spread infections. Secondly, and of greater concern, the worm made use of a Windows SMB (Server Message Block) exploit dubbed 'EternalBlue' to infect machines. EternalBlue is one of the exploits allegedly leaked by the NSA and acquired by the ShadowBrokers group. The ShadowBrokers publicly leaked the code for the exploit in April 2017, one month after Microsoft released a patch for it. One month later the WannaCry attack occurred. While the scope and scale of the WannaCry attack was considerable, and the anxiety generated was socially significant, if WannaCry truly was as an attempt at extortion, it was a negligible financial success, with less than 1 percent of the victims paying the ransom. In the month following the WannaCry outbreak, another global ransomware attack was launched, utilising some of the same exploits used by WannaCry. The updated version

---

[10] TeslaCrypt was a ransomware trojan. It is now defunct, and its master key was released by the developers. In its early forms, TeslaCrypt targeted game-play data for specific computer games. Newer variants of the malware also affect other file types. Reference source: https://en.wikipedia.org/wiki/TeslaCrypt

[11] The TorrentLocker ransomware, which has been in a lull as of late, has recently come back with new variants. These new variants are using a delivery mechanism that uses abused Dropbox accounts. This new type of attack is in line with our 2017 prediction that ransomware would continue to evolve beyond the usual attack vectors. Reference source: http://blog.trendmicro.com/trendlabs-security-intelligence/torrentlocker-changes-attack-method-targets-leading-european-countries

[12] Zepto (a new variant of the Locky ransomware) is a file-encrypting ransomware, which will encrypt the personal documents found on victim's computer using RSA-2048 key (AES CBC 256-bit encryption algorithm), appending the .zepto extension to encrypted files. Reference Source: https://malwaretips.com/blogs/remove-zepto-virus/

of the Petya[13] ransomware, dubbed ExPetr or NotPetya, reportedly hit more than 20.000 victim machines in more than 60 countries. Victims were mainly in Europe, but also in Asia, North and South America and Australia; however, more than 70% of the total infections were in the Ukraine[15]. Moreover, reports indicated that more than 50% of the businesses targeted were industrial companies. Some opinions suggest that the attack was staged to appear as another ransomware attack, but it appears to have been designed as a 'wiper', whose sole purpose is to destroy data.

1.2.2 IOCTA 2018 [ii]

In the year 2018, Ransomware retains its dominance, by remaining the key malware threat in both law enforcement and industry reporting. Even though the growth of ransomware is beginning to slow, ransomware is still overtaking banking Trojans in financially-motivated malware attacks, a trend anticipated to continue over the following years. In addition to attacks by financially motivated criminals, a significant volume of public reporting increasingly attributes global cyber-attacks to the actions of nation states. Mobile malware has not been extensively reported in 2017, but this has been identified as an anticipated future threat for private and public entities alike.

The most commonly reported ransomware families are Cerber, Cryptolocker, Crysis, Curve-Tor-Bitcoin Locker (CTBLocker), Dharma[14] and Locky. With the exception of Dharma, for

[13] Petya is a family of encrypting ransomware that was first discovered in 2016[2]. The malware targets Microsoft Windows-based systems, infecting the master boot record to execute a payload that encrypts a hard drive's file system table and prevents Windows from booting. It subsequently demands that the user make a payment in Bitcoin in order to regain access to the system. Reference source: https://en.wikipedia.org/wiki/Petya_(malware)

[14] The Dharma Ransomware is an encryption ransomware Trojan that is being used to extort computer users. There have been numerous computers around the world that have been infected by the Dharma Ransomware. The Dharma Ransomware seems to target only the directories inside the Users directory on Windows, with encrypted files receiving the suffix [bitcoin143@india.com].dharma added to the end of each file name. Variants of the Dharma Ransomware will sometimes not have a ransom note. The Dharma Ransomware does not stop the affected computer from working

which decryption keys are now available, all of these were reported in previous years. Member states reported a wide range of other ransomware families, but in fewer instances and dispersed across Europe. Overall damages arising from ransomware attacks are difficult to calculate, although some estimates suggest a global loss in excess of USD 5 billion in 2017[15]. In comparison, other reporting suggests that over the past two years, 35 unique ransomware strains have earned cybercriminals USD 25 million, with Locky and its many variants accounting for more than 28%[16]. This highlights the huge disparity between the losses to victims, compared to the actual criminal revenue generated.

Ransomware attacks may move from random to targeted

In some Member States attacks appear to remain largely untargeted, affecting citizens and businesses alike; this is perhaps the result of "scattergun" attacks by those engaging ransomware-as-a-service, or those with affiliate programs, such as Cerber, which allegedly allows its authors to sustain an income of USD 200.000 per month[17]. Some other Member States report that campaigns are customized or tailored to specific companies or individuals, suggesting a more organized or professional attack.

---

properly, but every time a file is added to the targeted directories, it will be encrypted unless the Dharma Ransomware infection is removed.

[15] Morgan, S., Global ransomware damage costs predicted to hit $11.5 billion by 2019, Reference source: https://cybersecurityventures.com/ransomware-damage-report-2017-part-2/, 2017.

[16] Spring, T., Google study quantifies ransomware profits, Reference source: https://threatpost.com/google-study-quantifies-ransomware-revenue/127057/, 2017.

[17] Spring, T., Google study quantifies ransomware profits, Reference source:https://threatpost.com/google-study-quantifies-ransomware-revenue/127057/, 2017.

As we have seen with other cyber-attacks, as criminals become more adept and the tools more sophisticated yet easier to obtain, fewer attacks are directed towards citizens and more towards small businesses and larger targets, where greater potential profits lie.

1.2.3 Previous IOCTA reports

In the 2014[iii] IOCTA report, while over half of EU law enforcement had encountered ransomware, this related on the whole to police ransomware, without encryption. Cryptoware was only just emerging with sporadic cases of Cryptolocker. By 2015[iv] cryptoware had become a top emerging threat for EU law enforcement, although non-encrypting police ransomware still accounted for a significant proportion of ransomware cases. By 2016[v] police ransomware had all but vanished, except for on mobile devices, superseded by a growing variety of cryptoware. By 2017 the number of ransomware families had exploded, their impact significantly overshadowing other malware threats such as banking Trojans. Industry reported that ransomware damages had increased fifteen-fold over the previous two years[18].

1.3 Needance of Malware - Ransomware Analysis

With millions of malicious programs in the wild ecosystem of Informatics, and more encountered every day, malware analysis is critical for anyone who responds to computer security incidents. And, with a shortage of malware analysis professionals, the skilled malware analyst is in serious demand.

---

[18] Morgan, S., Global ransomware damage costs predicted to exceed $5 billion in 2017, https://cybersecurityventures.com/ransomware-damage-report-2017-5-billion/, 2017.

1.3.1 Definition of Malware Analysis

Malware analysis is the procedure of identifying the working mechanism of the malware in order to counter it. While the various malware incarnations do all sorts of different things, there are several techniques and tools for analyzing malware.

In order to do a Malware Analysis, several steps of dissecting the malware are being followed. Following these steps, the analyst is able to understand the malware's scope. Reverse-engineering is not malware analysis, as a large audience believes, but is a part of the analysis. It could be said that, is the last technique an analyst will use to reveal the unanswered details of the malware.

Nevertheless, malware analysis is the critical part of incident response. Without the knowledge of the malware's actions, the security experts are not able to respond to an incident, as a result any technical or organizational measures, will not be effective.

In a simple case where a network intrusion, there are several information which are required to respond. At start, it should be revealed what exactly happened and ensure that all infected machines and files have been located. Also, a measurement of the damage should be calculated. Then, in order to counter the network intrusion, signatures should be generated and entered to the intrusion detection systems.

1.3.2 Background of Malware Analysis

In the old days, analysis had to be done with shell commands, built-in system utilities, and a text editor. Of course, back then, the attack surface was small and malwares could not hide behind the few processes running. As malware really began to hit its stride, virtual machine technology started to gain in popularity among security analysts. Researchers could make a snapshot or backup of a virtual machine and proceed to hack it, infect it, and trash it to their heart's content. In addition, the analyst could restore the good copy in just a few short minutes, with this process could be repeated over and over and streamlined analysis in a big way. However, virtual machine detection appears to be trivial nowadays [19] [20] [21]. Furthermore, some malware authors are well aware and take advantage of [22]. With the knowledge that researchers use virtual environments to analyze their code, some malware authors now instruct their creations not to run, or to run differently within these environments. The goal of malware authors is to make it more difficult for researchers that employ the use of virtualized environments to analyze samples of malware.

1.3.3 Malware Analysis Techniques

Currently, there are five general techniques used in malware analysis: basic static or surface analysis, basic dynamic or behavioral analysis, static code analysis, dynamic code analysis, and volatile memory analysis. [23]

- **Surface analysis** examines the structural properties and file attributes of a malware sample [24] without viewing assembly or machine-level instructions *(Sikorski & Honig,*

---

[19] Rutkowska, J. (2004, November). Red Pill... or how to detect VMM using (almost) one CPU instruction, source url: http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html

[20] Klein, T. (2003). Scoopy Doo - VMware Fingerprint Suite. source url: http://www.trapkit.de/research/vmm/scoopydoo/index.html

[21] Klein, T. (2003). Jerry – A(nother) Vmware Fingerprinter. source url: http://www.trapkit.de/research/vmm/jerry/index.html

[22] Zeltser, L. (2006, November 11). Virtual Machine Detection in Malware via Commercial Tools. Retrieved January 18, 2007, from SANS Internet Storm Center. link: http://isc.sans.org/diary.html?storyid=1871&rss

[23] Case Study: 2012 DC3 Digital Forensic Challenge Basic Malware Analysis Exercise, Author: Kenneth J. Zahn, kenneth.j.zahn@gmail.com Advisor: Rick Wanner, Accepted: August 24, 2013, from SANS

[24] (e.g. true file type (useful if the file extension was changed), size, file hash values, file and section headers, strings, contained objects, packing mechanisms)

2012). Surface analysis can provide information artifacts, such as IP addresses, Internet domain names, and command parameters, that prove useful in subsequent analysis steps.

- **Behavioral analysis** observes the actions taken by a malware sample while it is running. Certain key actions taken by the malware sample, such as adding/modifying/deleting Windows Registry keys, dropping files on the file system, and establishing communications with a command-and-control server, may serve as indicators of compromise (IOC) for the particular sample *(Mandiant, 2011)*. The IOC's observed by the analyst during this phase may then be used to produce signatures for intrusion detection and prevention systems. Because behavioral analysis requires executing the malware on a live machine, it is critical to implement appropriate risk mitigations (e.g. using a stand-alone, virtualized test environment or a sandbox) to avoid infecting production systems *(Sikorski & Honig, 2012).*
- **Static code analysis** examines the malware sample's executable instructions and internal data structures by loading the sample into a disassembler. Barring code that has been packed, encrypted, or otherwise obfuscated, all instructions present in the sample can be viewed. Although a time-consuming technique, static code analysis can give investigators full insight into the capabilities of the sample under examination *(Sikorski & Honig, 2012)*.
- **Dynamic code analysis** allows the analyst to execute a malware sample instruction-by-instruction by loading it into a debugging application. Because malware samples may have obfuscated portions, it is sometimes necessary to execute the malware sample up to the completion of the de-obfuscation routine. Once execution is halted at that point in time, the sample in memory may be examined for de-obfuscated data structures or may be dumped to disk for additional static code analysis *(Sikorski & Honig, 2012)*. Dynamic code analysis

also reveals data values that are assigned at run time and not available at compile time.

- **Volatile Memory Analysis** involves the examination of volatile memory at a single point in time. Such analysis is accomplished first by dumping the volatile memory to a file and then by inspecting the contents offline using a specialized tool such as the Volatility Framework *(Case, 2012)*.

1.3.4 Definitions of Analysis Techniques

Short definition of Static Analysis

Static Analysis, examines malware without running it, using a gamma of tools, like disassemblers. More specifically, the under examination malware, is being analyzed in static state, without loading it in RAM or analyses its behavior and without looking at CPU instructions.

Short definition of Dynamic Analysis

On the other hand, on dynamic analysis the malware is being run and monitor its effect. More specifically, the observation take place on running processes, on Windows registry edits and in low level RAM and CPU analysis.

Short definition of Basic Static Analysis

The Basic static analysis, that can be referred as quick and easy but fails for advanced malware, as it can miss important effects, as the malware is being viewed without looking at instructions.

Short definition of Basic Dynamic Analysis

And the Basic dynamic analysis, that can be referred as easy, but requires a safe test environment, with the risk that this method will not be effective on all malware.

Short definition of Advanced Static Analysis

The Advances Static analysis is a complex procedure that requires understanding of assembly code. The main procedure is the Reverse-engineering with a disassembler, without the actual execution of the binary by the CPU.

Short definition of Advanced Dynamic Analysis

The Advances Dynamic analysis examines internal state of a running malicious executable, that also requires understanding of assembly code combined with the understanding of the running code procedure in a debugger.

2. Malware Analysis Environment

2.1. Virtualization Technologies

Virtualization is an important tool for malware researchers and as such, is a large focus in this paper. The fact that some samples of malware are now refusing to run in researchers' labs is an important issue, and one without a simple solution. The aim of this section is to dissect the problem and clarify the solutions available.

If "The Matrix"[25] analogy is getting old, but it really is a perfect example, and a very effective way to explain the relationships between hosts and guests in the world of VMEs. Most important to VM detection is the difference between different types of VMEs, specifically between native virtualization / paravirtualization and emulation.

It is no secret that the Information Security industry takes advantage of virtualization software in order to research security threats. VMWare, Sandboxie, Hyper-V (Virtual PC[26]), Anubis, CWSandbox, JoeBox, VirtualBox, Parallels, QEMU are just of few of these virtual machines. The cornucopia of virtual environments gives the security professional, the opportunity to observe and analyze malicious software in a convenient and easily reproducible manner.

---

[25] The Matrix is a 1999 science fiction action film. It depicts a dystopian future in which reality as perceived by most humans is actually a simulated reality called "the Matrix". Source: en.wikipedia.org/wiki/The_Matrix

[26] Windows Virtual PC (successor to Microsoft Virtual PC 2007, Microsoft Virtual PC 2004, and Connectix Virtual PC) is a virtualization program for Microsoft Windows. In July 2006 Microsoft released the Windows version as a free product. The newest release, Windows Virtual PC, does not run on versions of Windows earlier than Windows 7, and does not officially support MS-DOS or operating systems earlier than Windows XP Professional SP3 as guests. The older versions, which support a wider range of host and guest operating systems, remain available. Starting with Windows 8, Hyper-V supersedes Windows Virtual PC. On the latest Windows version Windows 10 Virtual PC has been replaced by Hyper-V. Source url: https://en.wikipedia.org/wiki/Windows_Virtual_PC.

2.2. Differences between virtual and real world

A malicious software has several ways to detect the system that is being executed, using the VME Technologies Detection. It could be considered as the base operation of a VME. Malware writers, in order to counter the virtual world, include code in their binaries to make it more difficult for computer security professionals to analyze their executables in those virtual environments. Therefore, the VME technologies should be explained, in order to have the clearest view for each anti-virtualization technique.

2.3. VME Technologies

2.3.1 Native Virtualization

In Native Virtualization, the VMM executes guest code on the underlying hardware. Because the host and guest operating systems are sharing the same hardware, certain resources must be relocated by the VMM to prevent conflicts. One of these resources is the interrupt descriptor table register (IDTR). When this resource is relocated by the VMM, the address of the table changes. Using the SIDT instruction, one could write some simple code that will return the location of this table, and thus show whether code is being executed inside the matrix (inside a VME guest), or in “the world of the real” (within the host OS). The positive and the negative effects of the native virtualization implementation, are being listed:

+ Fast, Easy, Flexible, Convenient
- Easy for malware to detect, VME host software is limited to running on x86 architectures.

2.3.2 Paravirtualization

Paravirtualization is similar to Native Virtualization, except that there is a unique relationship between the host and the guest. The host presents an interface, similar to a software API, to the guest. This interface is called an ABI (Application Binary Interface) and is used by the guest to speak indirectly to the hardware. The positive and the negative effects of the Paravirtualization implementation, are listed below:

+ Is claimed to be potentially even faster[27] than Native Virtualization, due to the unique "shortcut" paravirtualization provides for the guest.

- The guest must be modified to work with the host's specific ABI. This generally means that paravirtualization is an approach that generally would not work with commercial operating systems, such as Microsoft Windows.

2.3.3 Native Virtualization and Paravirtualization Detection Techniques

Tools and code demonstrating VM detection techniques are freely available. Joanna Rutkowska's Red Pill[28] is probably the most well-known of these, though Tobias Klein's Scoopy[29] tool is a bit more informative.

---

[27] Barham, P., Dragovic, B., Fraser K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the Art of Virtualization. Source url: www.cl.cam.ac.uk/netos/papers/2003-xensosp.pdf.

[28] Rutkowska, Joanna (2004, November). Red Pill... or how to detect VMM using (almost) one CPU instruction. Source url: www.invisiblethings.org/papers/redpill.html

[29] Klein, Tobias. (2003). Scooby Doo - VMware Fingerprint Suite. Source url: www.trapkit.de/research/vmm/scoopydoo/index.html

2.3.4 Descriptor Table Registers check

The SIDT Instruction (Store Interrupt Descriptor Table) stores the content of the IDTR (Interrupt Descriptor Table Register) register, which in fact, is a selector that points into the Interrupt Descriptor Table. The instruction SGDT (Store Global Descriptor Table) stores the register value of GDTR, which is a selector that points into the global descriptor table. The SLDT instruction (Store Local Descriptor Table) stores the register value LDTR. This register is a selector that points into the local descriptor table (LDT).

There is only one Interrupt Descriptor Table Register (IDTR), one Global Descriptor Table Register (GDTR) and one Local Descriptor Table Register (LDTR) per processor. Since there are two operating systems running at the same time (the host and the guest), the virtual machine needs to relocate the IDTR, GDTR and LDTR for the guest OS to different locations in order to avoid conflicts. This will cause inconsistencies between the values of these registers in a virtual machine and in the native machine. The instructions SIDT, SGDT and SLDT are assembly instructions that can respectively be used to retrieve the values of IDTR, GDTR and LDTR.

2.3.5 The IDTR Detection Technique

When Red_Pill.exe is executed within an OS running directly on hardware, Red Pill informs us that we are “Not inside the Matrix”. When executed within an OS running in a VME like VMWare, Red Pill informs us that we are, indeed, “Inside the Matrix”. Malware authors have taken advantage of the fact that VM detection can be done with a line, or just a few lines of code. It is increasingly common to find malware that will refuse to run in virtualized environments, as their authors know that VMEs commonly used by malware researchers.

To counter this, it is possible that a VME could fake the results of a query for IDT values, but it is unlikely that commercial vendors would take much interest in making these changes. It is also not clear whether such changes would cause detrimental effects on operating systems running within the modified VME.

2.3.6 Thwart virtual machine detection

Most commercial VMEs create many artifacts that allow for easy VM detection. Because anti-VM techniques typically target VMware in this case, the focus stands on anti-VMware techniques. One such example is Tobias Klein's Doo VBScript, included in the Scooby Doo release. This VBScript simply looks for VME artifacts in the Windows registry. These are extremely easy to find if a VME toolset, such as VMWare Tools, or Parallels Tools have been installed on the Guest OS. For example, VMware provides a set of tools called VMware Tools that enhances the overall user experience with the guest OS. The drawback is that installing VMware Tools in a Windows guest OS will leave many clues easily detectable by a piece malware that is running in a virtual machine.

Even if VME toolsets have not been installed, artifacts can still be found, as Doo shows. Doo specifically looks for the names of hardware components, which usually contain the word "virtual" or the name of the VME vendor. It is simply a check for the presence of virtualized hardware, but as a method is effective all the same. Specifically, Malware can check for the presence of certain OUIs (VMware has more than one Organizationally Unique Identifier or OUI) and choose to behave differently or not to display any malignant behavior whatsoever in a virtual machine. In Windows these OUIs can be easily reveal themselves via Registry. Each virtual machine is associated with specific device drivers, registry values that give away their nature. For instance:

- Hard drive driver (VMware):

*HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\IDE\DiskVMware_Virtual_IDE_Hard_*
Drive 00000001\3030303030303030303030303030303030303130\FriendlyName VMware Virtual IDE Hard Drive

- Video driver (VMware):

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}\0000\DriverDesc VMware SVGA II

- Mouse driver (VMware):

%WINDIR%\system32\drivers\vmmouse.sys

- In addition, virtual environments have virtual network interfaces. Just like any network interface, they are assigned a unique MAC address that usually includes the manufacturer's identification number. For example, a network interface for VMware Workstation will have a MAC address that starts with

00:50:56 or *00:0C:29*

Any of these can be used by a malware writer to detect the presence of a virtual machine. Furthermore, there is a full list of detection techniques, which are more thoroughly explored in paper: “On the Cutting Edge: Thwarting Virtual Machine Detection, a paper by Ed Skoudis and Tom Liston”. This list will be used as a reference during the analysis, when specific detection techniques are being identified.

Emulation

Emulation is a different matter altogether. Computer emulators emulate the underlying hardware using code, rather than by sharing the actual physical hardware. As a result, SIDT/IDTR detection techniques do not work within emulated VMEs. Another advantage of emulation is that the emulated hardware can potentially run on top of any other hardware architecture. For example, Bochs running on MacOS X could run x86 versions of Windows XP. The positive and the negative effects of the Emulation implementation, are being listed:

+ x86 emulators such as QEMU and Bochs can run on any architecture where the code is ported to, so they can evade current detection techniques

- Emulation is generally slower than native virtualization or paravirtualization.

2.4. General Local Virtual Machine Detection

There are several ways to detect a VM. Complementary to the above mentioned, the Local Virtual Machine Detection that covers nearly all of the elements of the virtual machine, is divided to four categories of methods for locally detecting the presence of a virtual machine:

1. Look for VME artifacts in processes, file system, and/or registry
2. Look for VME artifacts in memory
3. Look for VME-specific virtual hardware
4. Look for VME-specific processor instructions and capabilities

2.4.1. Exploring Available VMEs

The following Table describes the Notable Emulators and VMEs [30]

Product	Type	Pros	Cons
VMware Server - Services	Native	Can be remotely controlled and configured. Easy setup and free	Easily to detect by malware
Hyper-V (Virtual PC)	Native	Fast. Easy setup	Commercial, money cost. Easily detect by malware
Parallels	Paravirtualization	Easy to Setup and configure	Commercial, money cost. Easily detect by malware
Bochs	Emulation	Free and Open Source. Can not be easily detect by malware	Operating Systems run much more slowly on emulation. High Specification machine needed
QEMU	Emulation	Free and Open Source. Can not be easily detect by malware. Faster than Bochs	Confusing to configure and run

Table 1: Notable Emulators and VMEs[31]

[30]More complete list on Wikipedia source url: en.wikipedia.org/wiki/Comparison_of_virtual_machines
[31] Malware Analysis: Environment Design and Architecture, SANS Institute, Author: Adrian Sanabria, Adviser: Rick Wanner, January 18th 2007. Source url: https://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841

2.4.2. Environment Design and Architecture

At the software level, tools and methods for detecting and analyzing malware have been documented above. However, the design and architecture of malware analysis environments does not often get publicly discussed. Specifically, commercial antivirus vendors use highly customized and specialized environments to explore the goals and inner workings of malware quickly and efficiently. The regular analysts rarely experiment beyond the use of an isolated virtual machine to quarantine the malicious intent of a virus or trojan.

Lab Design due to Malware Type

There are many different ways to classify malware. Antivirus vendors tend to classify by intent (Trojan, worm, mailer, Ransomware, etc) and several aspects of severity (damage potential, potential of outbreak, and actual outbreak reports). These metrics are usually used to create an overall risk rating. The necessity for a method of identifying and classifying malware according to its detection difficulty, was introduced by Joanna Rutkowska, which she calls *Stealth Malware Taxonomy*[32]. The following categorization is not a recommendation to replace currently used categories, but instead, it is another set of criteria to consider when analyzing malware.

Malware Type	Stealth Characteristics	Analysis Considerations
Type 0	Does not use undocumented methods to hide.	Most standard malware falls under this category. Usage of traditional tools to analyze
Type I	Modifies constant resources to hide itself (by patching executables, modifies code, inserting into BIOS, ect)	Compare hashes of running memory with equivalent values on disk. Digitally sign code.

[32] Rutkowska, J. (2006, November), Stealth Malware Taxonomy. Source url: blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf

Type II	Modifies dynamic resources to hide itself (for example: using sections of data within memory)	Unable to compare hashes of application data, as it is constantly changing.
Type III	Hides itself where the operating system cannot see it at all, like a hypervisor. Full control of the running system and interfere with it.	Being nearly undetectable from within the Operating System, detection, prevention and analysis would have to be done at the hypervisor level or outside of the OS. A way for analysis is to compare the timing of instructions executed before and after type 3 malware is introduced or network activity analysis.

Table 2: Brief Overview of J. Rutkowska's Stealth Malware Taxonomy[33]

The relevance to malware analysis and lab architecture exists on the opportunity to specialize a lab or PC environment for the analysis of a specific type or class of malware. One of the most common recent examples is malware that refuses to run in virtualized environments, while these environments are often equated with malware analysis. On the under analysis PE file, the class of malware must be taken into account. During the dynamic analysis, several anti-vm methods have been detected. Furthermore, some specific network and time behavior exists, which should be considered to make the necessary changes to their lab design. This results in several opportunities to specialize an analysis lab.

Guidelines for Lab architecture

The basic guidelines when designing and implementing a malware analysis environment are:

- Simplicity

  Each added bit of complexity can make it more difficult to maintain.

- Containment

---

[33] Rutkowska, J. (2006, November), Stealth Malware Taxonomy. Source url: blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf

Acts as a paramount when designing an environment that may test the digital equivalents of plagues and super flues. Maintaining control is preferred as well, but cannot be guaranteed when dealing with new malware specimens. Containment is the safety net when control is lost.

- Flexibility

A flexible environment is essential. One that is too fragile, or has too much downtime is of little use to a malware researcher.

Suggested Requirements and setups

Physical and Financial Constraints:

A researcher may need to do analysis on the road, could do all of it in a fully funded data center, or could employ a combination of both. In the current case with a non funded malware analysis for educational purposes, there will be a restriction on a single physical machine.

Scenario single PC Lab

The single PC lab is one of the most commonly used environments and especially for researchers. This deployment will take place in current project, because it can be easily deployed on single workstation and also easily deployed on a laptop. The option of using emulators, such as Bochs or QEMU rather than VMware, would be more difficult to isolate the networks and specifically using the VLAN features of QEMU because of the requirement of host-based firewall in order to filter and block the incoming traffic, exposing the host machine.

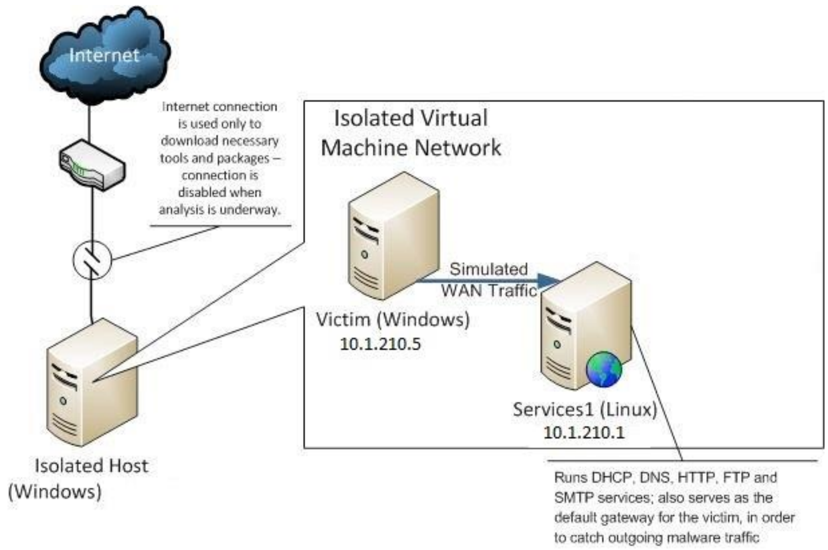


Figure 1: Single PC Lab[34]

Please see the **Appendix A** for the full specifications of Hardware and Software on Host. In addition, Appendix A includes the VM and VME configuration and installation using FlareVM.

Sample files for the analysis

Due to the nature of the ransomware and considering that specific files are being searched in order to encrypt them, we have collected some sample files. The file type of sample files are png, jpg, txt, xls, doc and pdf. They will be placed on the specific directories Desktop, Document, Downloads and on C:/files. Each directory will have a different package of files, with all types included. With the above actions, we are preparing our environment to be helpful and ready for the behavioral analysis. We are expecting the ransomware to encrypt these types of files and we would like to know if the ransomware searches exhaustive or in specific directories. In addition,

[34] Based on figure 8 of Malware Analysis: Environment Design and Architecture, SANS Institute, Author: Adrian Sanabria, Adviser: Rick Wanner, January 18th 2007. Source url: https://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841

we will compare the encrypted file and the original file on hex editor, in order to try to find vulnerability on the entropy. Please find the files on the **Attached zipped files**. Note that all the files are originally publicly posted in the website of www.unipi.gr and its subdomains.

Swift Recovery

Traditionally, recovering a computer system to an earlier state would be a tedious, time intensive operation. In the past five years, however, VMEs have become popular in malware analysis due in part to the ease and speed of recovery possible with these environments. The system will use VMWare virtualization software as an VME in which to run the malicious samples.

A hardware failure is always possible, so the RAID 1 structure of the VMs storage, decreases the probability of both HDDs failure. Keep in mind that, a frequent backup of the VM is being taken, as the last recovery options. The disaster recovery approach is to upload these backups of the VMs in a cloud storage service. Because the University uses the G-suite service with unlimited storage, the disaster recovery backups will be uploaded to Google Drive.

2.4.3. VMware Workstation Setup

Virtual Network Editor

In order to setup the network securely, a custom VLAN should be created. The name of the Network Adapter would be *Malnet10*, acting as custom Host-Only network, which connects the Virtual Machines internally in private network and with no interaction with the host's network. The subnet IP range will be set as *10.1.210.0* and subnet mask as *255.255.255.0*, without local DHCP service activated, so the distribution of IP Addresses to Virtual Machines would be manual settled.

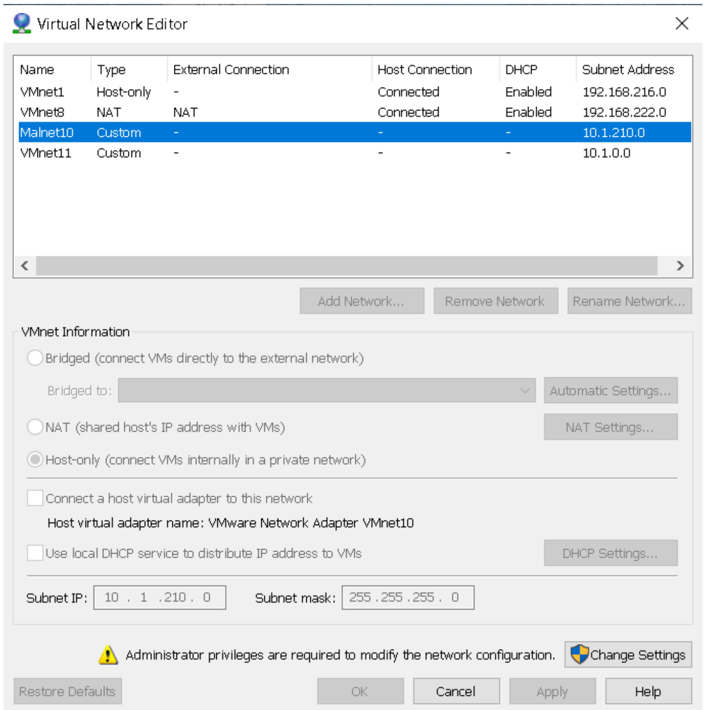


Figure 2: Virtual Network Editor settings

Then we should attach the virtual cable to our VM, by adding or editing a Network Adapted in Hardware/Virtual Machine Settings. Keep in mind that the host's virtual adapter should not be connected.

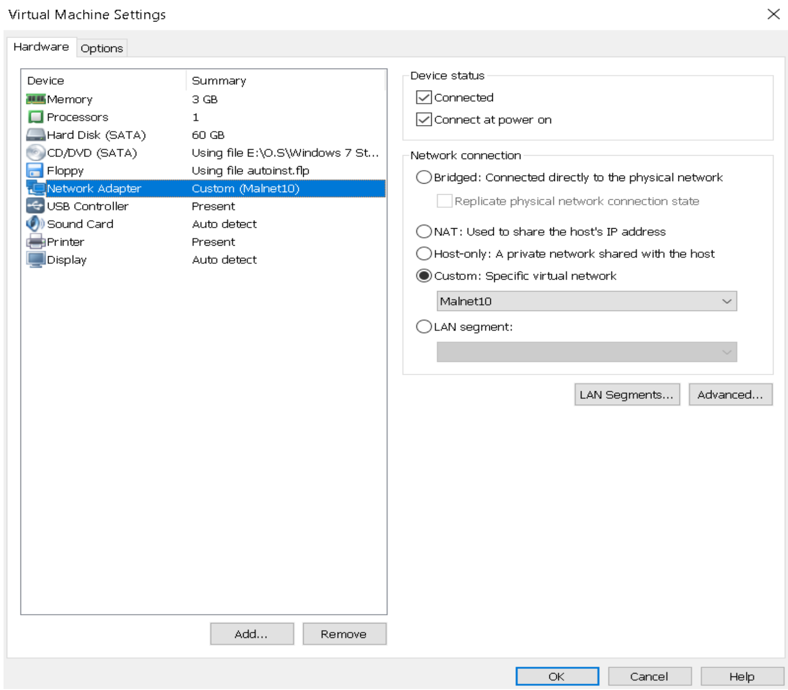


Figure 3: Virtual Machine Settings on network adapter

In addition, each VM should manual adapt an IP manually from the subnet 10.1.210.0.

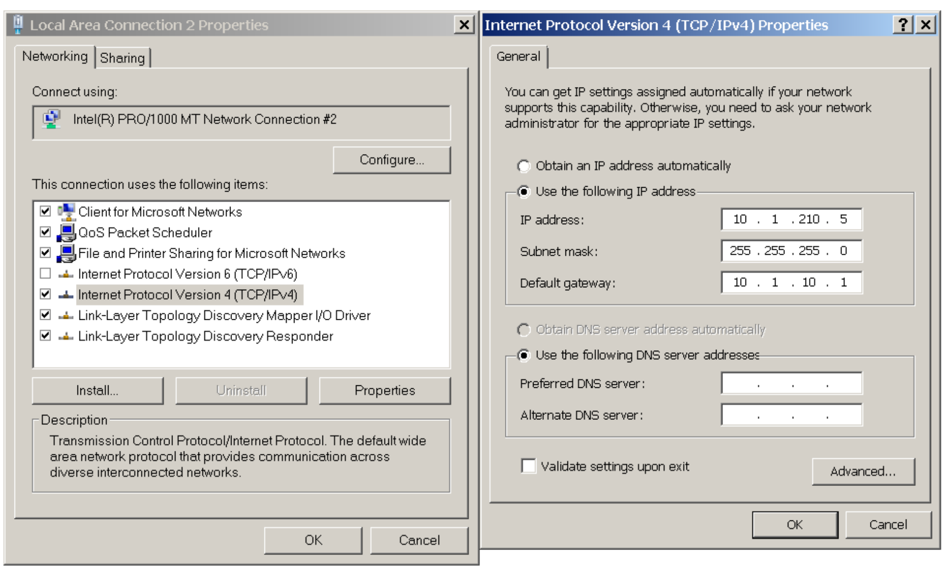


Figure 4: Local Area Connection Properties in VM OS

The selection of the subnet IP range is not random, but it is selected for the under analysis ransomware. More specifically the verification of date and time is being done at binary's location .text:004026CC and it is combined with the verification of the IP is being done by gethostbyname API function call, at binary's location .text:00403FE8. These techniques will be analyzed in further analysis of the subject malware.

Stealthy Tools

Stealthy Tools, that are being included in the Appendix B, are basically a registry file that edit some default registry values. This registry script will make our VME stealthier from VM detection techniques. The default registry values reveal the VME, but after editing them the VME will be spoofed and would not be differed.

Note that, in case of Windows10 VM, go to task manager, click performance tab and click CPU on the left. There is a value 'Virtual Machine: Yes' at right bottom and L1, L2, L3 cache are not being showed. To spoof these finding in VMWare, "Virtualize Intel VT-x/EPT or AMD-V/RVI" in the settings of the VM should be activated, in order to have virtual L1, L2, L3 cache [35].

VMware Tools detection evasion

To hide the VMWare Tools from the list of programs (or any program for that matters), you can just go to:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

Find the program you want to hide in the list. Once you found it, create a DWORD named 'SystemComponent' and set it to 1. In case of non changed state, restart the VME.

[35]CPU cache is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory. A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3).

A helpful PowerShell script to rename same registry values to spoof the VMware Tools, having them fully functional is included in the **Appendix C**.

VMware Tools uninstall

Nowadays the majority of malware authors check, with several ways, if VMware tools if installed. In case that the above VMware tools detection evasion script is not working, the best way to hide it from the control panel is going to the registry editor and going to they following registry value:

```
hkey_local_machine>software>microsoft>windows>currentversion>uninstall
```

Click on every folder there until you find “VMware Tools” in the variable 'displayname' and delete that folder. Restarting Windows after these actions required.

VMX configuration file

The next step is to edit your VMware .vmx file. When you create a new virtual image with VMware, settings about it are stored in a configuration file with the .vmx extension. The file contains information about networking, disk size, devices attached to the virtual machine, etc. The config file is usually located in the directory where you created your virtual image. The recommended VMX setup from SANS paper is the following[36]:

```
isolation.tools.getPtrLocation.disable = "TRUE"
isolation.tools.setPtrLocation.disable = "TRUE"
isolation.tools.setVersion.disable = "TRUE"
isolation.tools.getVersion.disable = "TRUE"
monitor_control.disable_directexec = "TRUE"
monitor_control.disable_chksimd = "TRUE"
```

[36] More vmx file commands can be found at the url: http://sanbarrow.com/vmx/vmx-advanced.html#isolationtools

```
isolation.tools.getPtrLocation.disable = "TRUE"
monitor_control.disable_ntreloc = "TRUE"
monitor_control.disable_selfmod = "TRUE"
monitor_control.disable_reloc = "TRUE"
monitor_control.disable_btinout = "TRUE"
monitor_control.disable_btmemspace = "TRUE"
monitor_control.disable_btpriv = "TRUE"
monitor_control.disable_btseg = "TRUE"
```

Table 3: VMX configuration file recommended by SANS

It should be pointed out that:

```
monitor_control.disable_directexec = "TRUE"
```

will usually thwart descriptor table registers checks. This setting will make VMware interpret each assembly instruction instead of executing them directly on the processor. Therefore, the result of a SIDT instruction will not be an address in the 0xffXXXXXX range as one would get without this setting.

```
isolation.tools.getVersion.disable = "TRUE"
```

Will thwart the backdoor I/O check.

Furthermore, a VMWare virtual machine's SMBIOS data will show VMWare Inc, by default, as the system manufacturer and VMWare Virtual Platform as the system model. While this information is not directly editable in the VM settings, you can however edit the virtual machine's configuration file to instead pass along the SMBIOS System Manufacturer and Model info from the host computer. The config command that should be added to vmx file is:

```
SMBIOS.reflecthost = "TRUE"
```

Please note that, the best and most popular paper for VM Anti Detection is the Thwarting Virtual Machine Detection, that was very helpful on Static and Dynamic Code analysis is: Liston, Tom; Skoudis, Ed;, "On the Cutting Edge:Thwarting Virtual MachineDetection," SANS, 2006 [37].

VMX setup for system time check

The under analysis ransomware has a sophisticated check of system time. More specifically the verification of date and time is being done at binary's location *.text:004026CC.*, where the valid range to execute the ransomware is from the epoch time 1410739200, which is being converted as human readable date to GMT: Monday, September 15, 2014 12:00:00 AM, until the epoch time 1416009600, which is being converted as human readable date to Saturday, November 15, 2014 12:00:00 AM.

The bypass solution of the system time check, without patching the binary, is to set the virtual BIOS real time clock of the virtual system, to the epoch time 1410739300, each time the virtual machine is powered on:

```
rtc.startTime = "1437997063"
tools.syncTime = "FALSE"
time.synchronize.continue ="FALSE"
time.synchronize.restore = "FALSE"
time.synchronize.resume.disk = "FALSE"
time.synchronize.resume.memory = "FALSE"
time.synchronize.shrink = "FALSE"
time.synchronize.tools.startup = "FALSE"
```

Table 4: VMX configuration for the system time check

[37] Source url: https://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

3. Surface Analysis

3.1. Online malware repositories

3.1.1. VirusTotal

The usual first movement of a malware analyst is to upload the suspicious file at an online repository of known malwares and If it is already analyzed, there will be results. The most famous is VirusTotal[38]. Keep in mind that this action may alert the attacker and inform him that you have detected an intrusion. The safest way to check a suspicious file in the VirusTotal database, without having interactions, is to hash the file and search online for its hash value.

The check on VirusTotal was done with the SHA-256 of the suspicious file, which is "6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92". The results were confusing with the 21/67 detection ratio. There are strong suspects that the file is malicious, but no one has done a full analysis yet. The only suspicious indicators are the high entropy .txt section and some mutexes that are being created.

The VirusTotal results are available offline on the **Appendix D** and online on the url: https://www.virustotal.com/#/file/6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92/

[38] VirusTotal was founded in 2004 as a free online service that analyzes files and URLs for viruses, worms, trojans and other kinds of malicious content. VirusTotal inspects items with over 70 antivirus scanners and URL/domain blacklisting services, in addition to a myriad of tools to extract signals from the studied content.

3.1.2. HybridAnalysis

A VirusTotal alternative, HybridAnalysis has richer results and confirms the maliciousness of the file, but it is categorized as Spyware without useful details for its behavior. The addition indicators are the Anti-VM tricks, Anti-Debugging tricks and the TLS[39] callbacks.

Hybrid Analysis is an innovative technology integrated into the flagship product VxStream Sandbox. VxStream Sandbox is a fully automated malware analysis system, as a standalone software package that is automatically deployed within a limited hosted solution that is operated from Hybrid Analysis's servers in Germany.

The feature set of VxStream Sandbox is very extensive with hundreds of generic indicators at its core that have proven to detect unknown threats independent of Anti-Virus signatures. The analysis does not limit only the runtime behavior of the sample, but in the entire process memory, using multiple timed snapshots. This allows extraction of a lot more indicators (Strings/API calls) regardless of execution.

The HybridAnalysis results are available offline on the **Appendix D** and online on the url: www.hybrid-analysis.com/sample/6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92/

3.2. PE headers

The Portable Executable (PE) format is a file format for executables, object code and DLLs. It is used in 32-bit and 64-bit versions of Windows operating systems. The term "portable" refers to format's versatility within numerous environments of operating system software architecture. The PE format is a data structure that encapsulates necessary information so that Windows OS

[39] TLS Callback is Address of Callbacks, functions that are stored on .tls section, that are executed when a process or thread is started or stopped.

loader can manage wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments. PE is a modified version of the Unix COFF file format. PE/COFF is an alternative term in Windows development. General Portable Executable (PE) format file layout can be described with the following graphical representations.

Figure 5: hexcode dump of a PE header

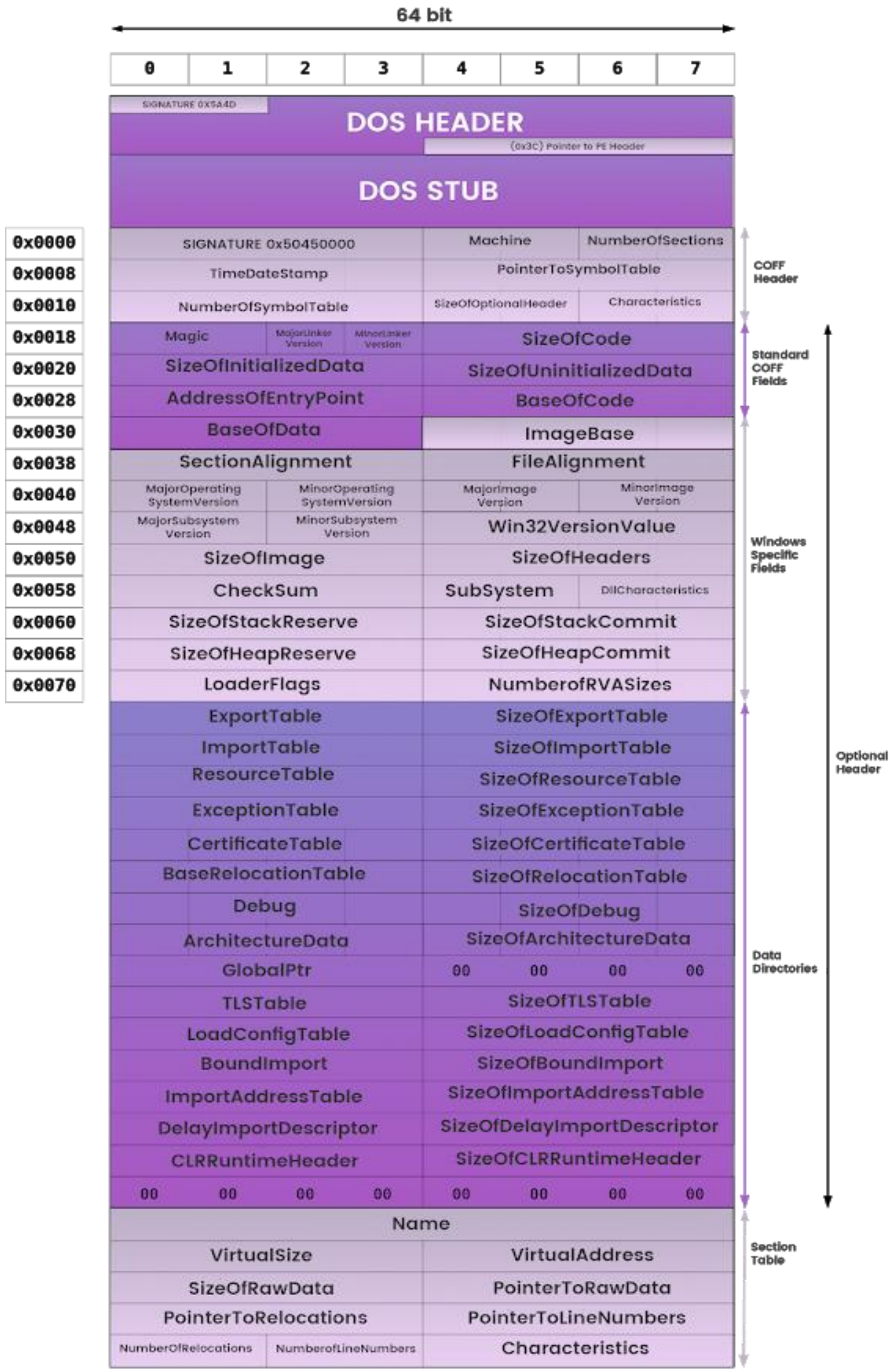


Figure 6: PEheader diagram sections broken up

The specific fields and the structure layout are being detailed described on **Appendix E.**

3.3. Basic Static Analysis with Windows tools

3.3.1. PEView

In order to see what is inside the file, a recommended tool will be used, the Portable Executable viewer Peview [40].

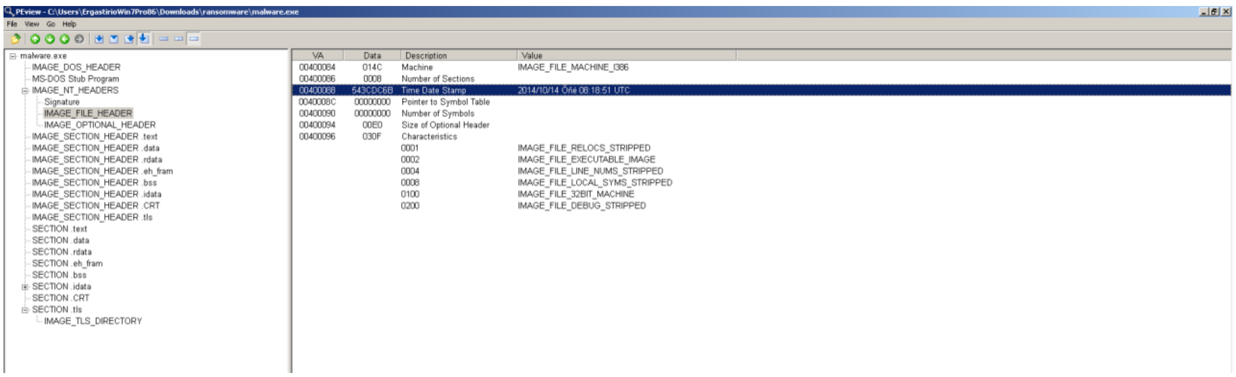


Figure 7: malware.exe/IMAGE_NT_HEADER/IMAGE_FILE_HEADER

The common useful PE section is the IMAGE_NT_HEADERS and its' subsection IMAGE_FILE_HEADERS. The "Time Date Stamp" shows when the files were compiled. This is often used as an indication of the time zone the attackers live in. Also, if the files were both compiled on the same date within a minute of each other, indicating that they are part of the same package. On the current scenario, the timestamp is 14 October 2014 08:18:51 UTC, which indicates that it is crafted. On the following section we will see that the malware has a specific hardcoded lifetime which comes into conflict with the above timestamp. Reasonably the older the sample, the more likely it will be detected by signature-based antivirus if it is malicious.

---

[40] PEview, as the name suggests, is a viewer for PE (Portable Executable) files. It is a program running on Windows OS. More specifically, shows the structure and content of 32-bit Portable Executable (PE) and Component Object File Format (COFF) files. This PE/COFF file viewer displays header, section, directory, import table, export table, and resource information within EXE, DLL, OBJ, LIB, DBG, and other file types.

Figures 8, 9, 10, 11 show the sections from rnsmwr.exe (malware.exe). As you can see, the .text, .data, .rdata and .eh_frame sections, have about the same size on them values on Virtual Size and Size of Raw Data.

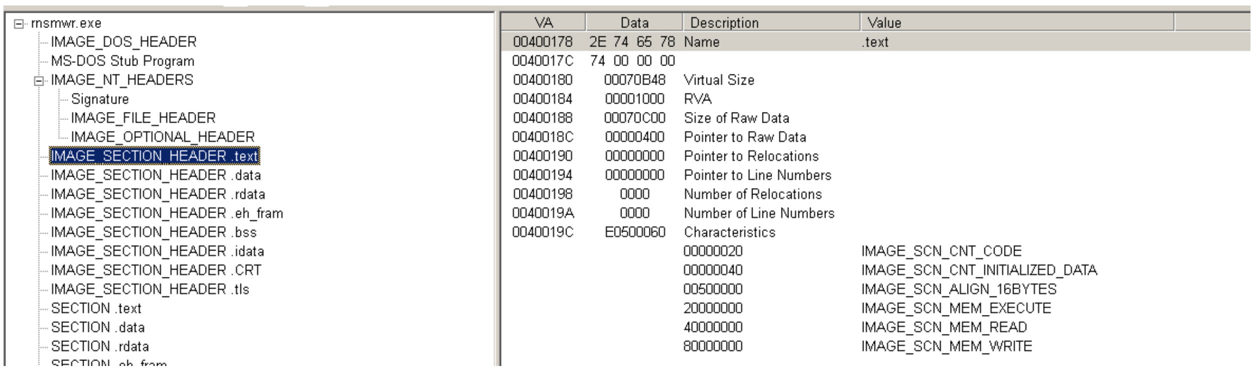


Figure 8: PEview IMAGE_SECTION_HEADER .text

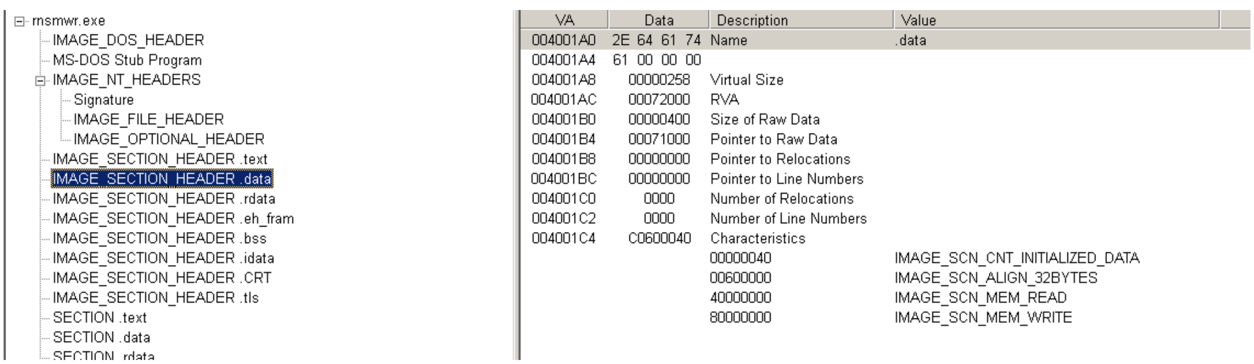


Figure 9: PEview IMAGE_SECTION_HEADER .data

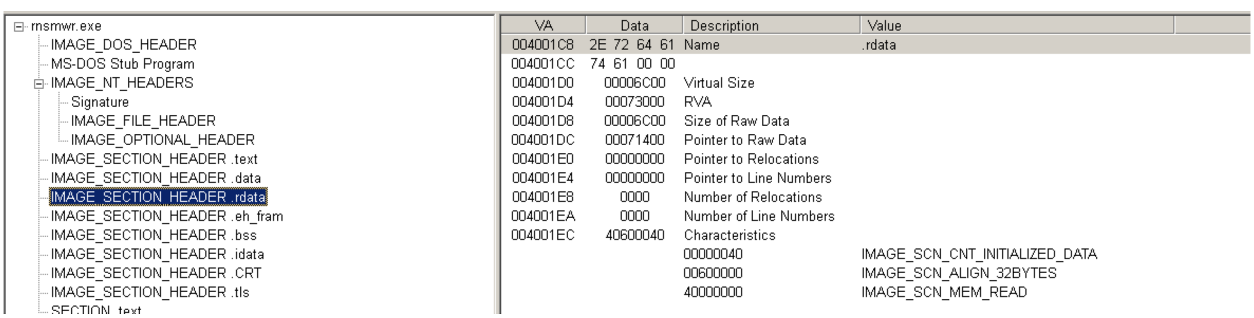


Figure 10: PEview IMAGE_SECTION_HEADER .rdata

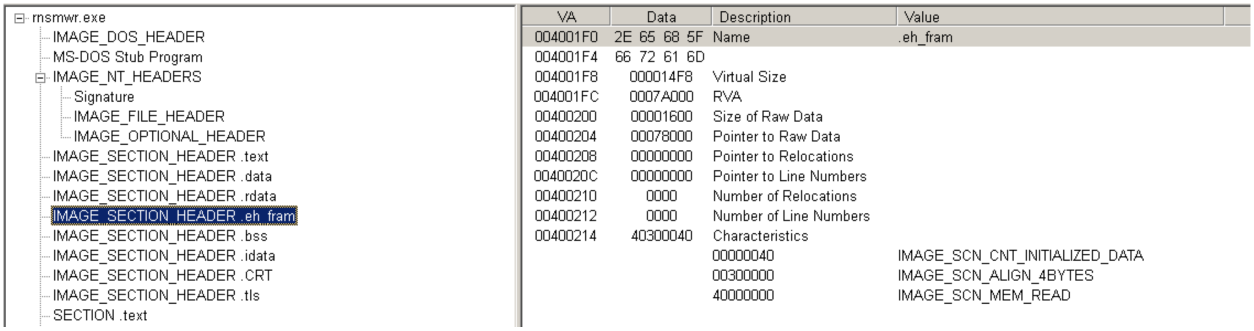


Figure 11: PEview IMAGE_SECTION_HEADER .eh_frame

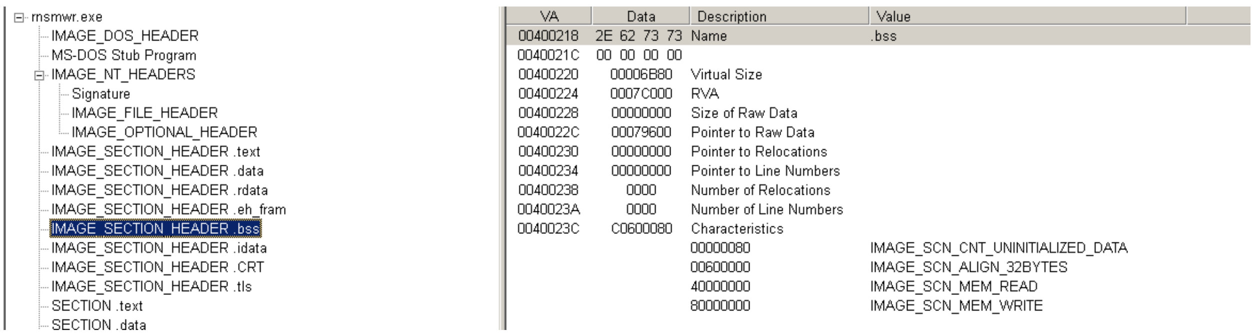


Figure 12: PEview IMAGE_SECTION_HEADER .bss

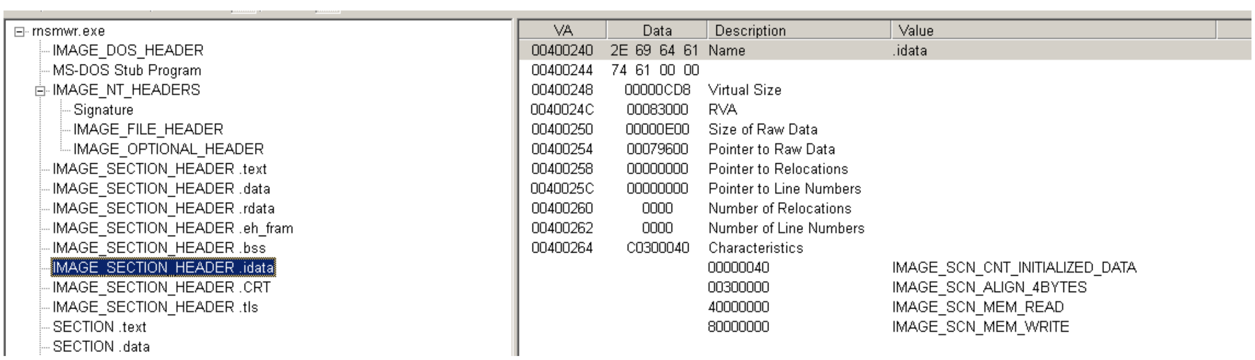


Figure 13: PEview IMAGE_SECTION_HEADER .idata

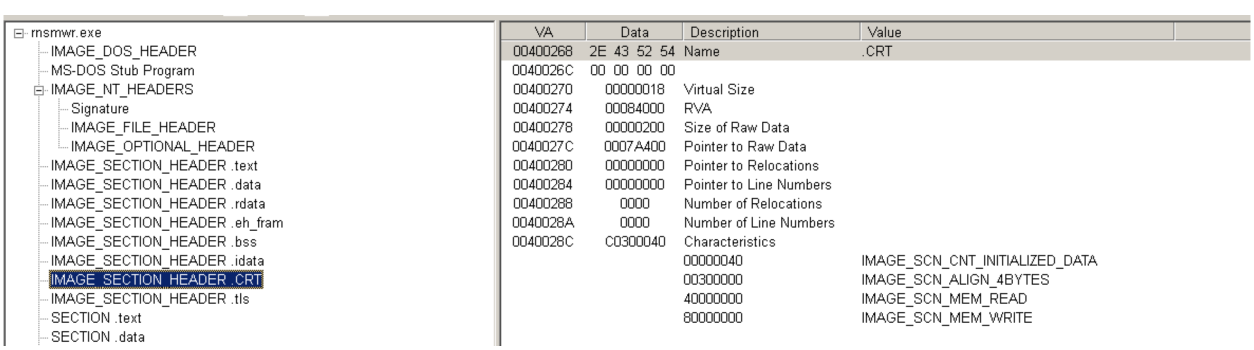


Figure 14: PEview IMAGE_SECTION_HEADER .CRT

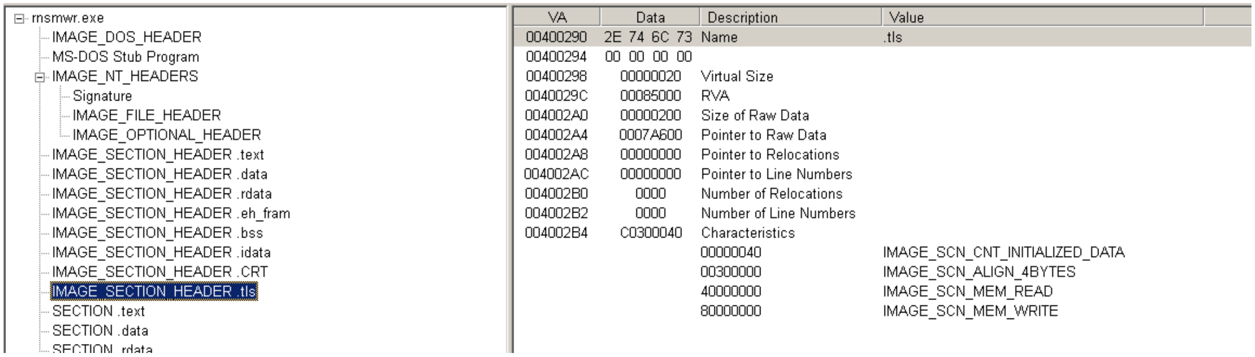


Figure 15: PEview IMAGE_SECTION_HEADER .tls

The .bss section may seem suspicious because it has a much larger virtual size than raw data size, but this is normal for the .data section in Windows programs. But note that this information alone does not tell us that the program is not malicious; it simply shows that it is likely not packed and that the PE file header was generated by a compiler.

Another useful section is the .idata with the IMPORT Address Table, from where we can gather information for the functions from other libraries that are used by the malware.

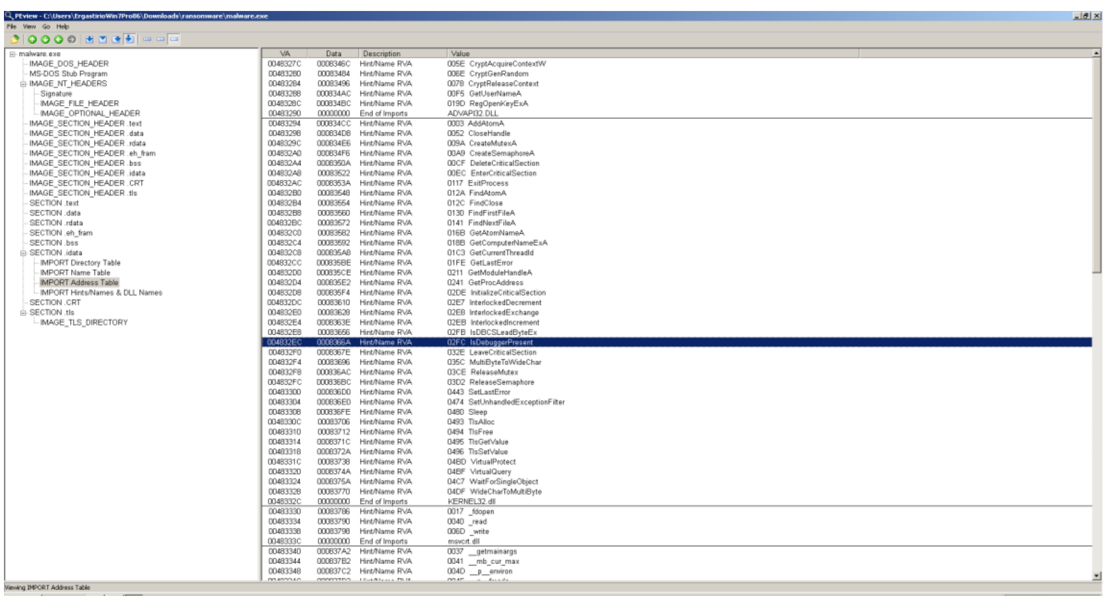


Figure 16: Section .idata/IMPORT Address Table

The full structured list of imports can be found at Appendix F/**List of**

Last but not least, we figure out a not usual section, the .tls section. Malware authors employ numerous and creative techniques to protect their executables from reverse-engineering. The arsenal includes an anti-debugging technique called *TLS callback.* The approach is not new, yet it is not widely understood by malware analysts.

TLS explanation

According to Microsoft, Thread Local Storage (TLS)[41] is a mechanism that allows Microsoft Windows to define data objects that are not automatic (stack) variables, yet are "local to each individual thread that runs the code. Thus, each thread can maintain a different value for a variable declared by using TLS." This information is stored in the PE header. (Windows uses the PE header to store meta information about the executable to load and run the program.)

A programmer can define TLS callback functions, which were designed mainly to initialize and clear TLS data objects. From the malware author's perspective, the beauty of TLS callbacks is that Windows executes these functions before executing code at the traditional start of the program. Since, windows loader first create a thread for the process to run, the code in TLS Callback runs even before the program reach at entry point. Malwares use these functions/Callbacks to store their

[41] All threads of a process share its virtual address space. The local variables of a function are unique to each thread that runs the function. However, the static and global variables are shared by all threads in the process. With thread local storage (TLS), you can provide unique data for each thread that the process can access using a global index. One thread allocates the index, which can be used by the other threads to retrieve the unique data associated with the index. The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems. The maximum number of indexes per process is 1,088. When the threads are created, the system allocates an array of LPVOID values for TLS, which are initialized to NULL. Before an index can be used, it must be allocated by one of the threads. Each thread stores its data for a TLS index in a TLS slot in the array. If the data associated with an index will fit in an LPVOID value, you can store the data directly in the TLS slot. However, if you are using a large number of indexes in this way, it is better to allocate separate storage, consolidate the data, and minimize the number of TLS slots in use. Source url: https://docs.microsoft.com/en-us/windows/desktop/ProcThread/thread-local-storage

malicious code or Anti-Debug methods. It makes malware analyst confused while they are debugging the code since they first break at Entry Point, but the malicious code is already executed.

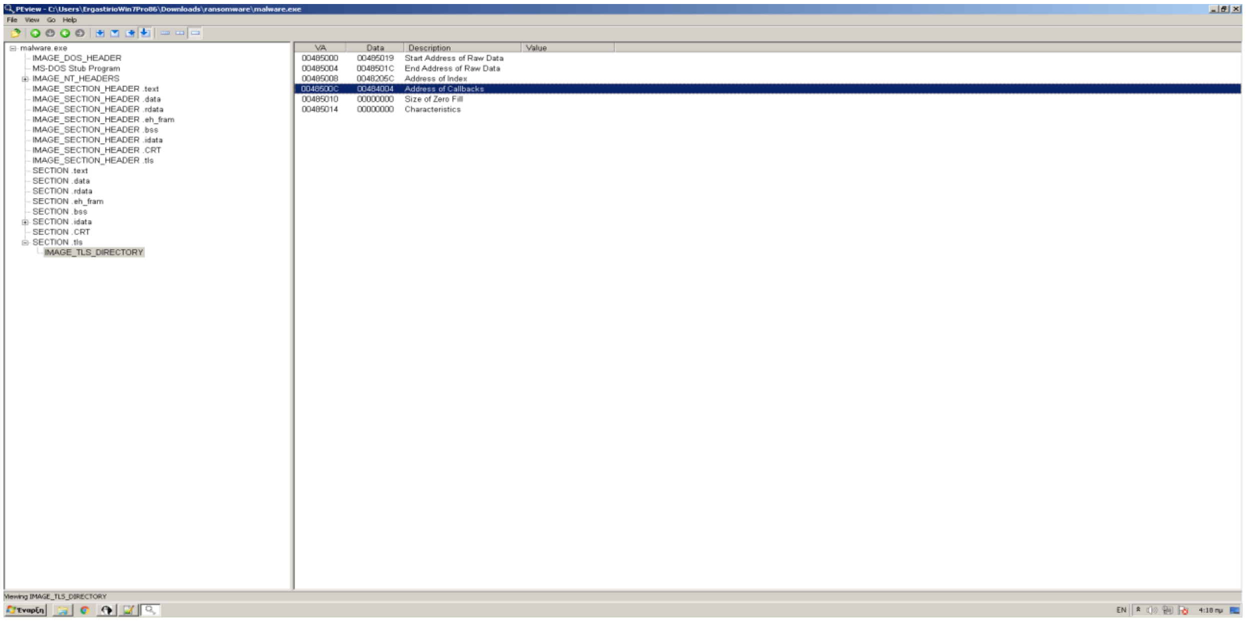


Figure 17: Section .tls/Address of Callbacks

The Memory address 00484000 is written down, and we will be very useful to start correctly the dynamic analysis. More specifically this address will be the entry point of the executable, during the execution and not the start of the program. This is the purpose of TLS anyway, that in this case is being abused from a malicious software.

3.3.2. PEiD

One way to detect packed files is with the PEiD program. PEiD can detect the type of packer or compiler employed to build an application, which makes analyzing the packed file much easier.

Packing and Obfuscation

Malware writers often use packing or obfuscation to make their files more difficult to detect or analyze. Obfuscated programs are ones whose execution the malware author has attempted to hide. Packed programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Both techniques will severely limit your attempts to statically analyze the malware. When the packed program is run, a small wrapper program also runs to decompress the packed file and then run the unpacked file. When a packed program is analyzed statically, only the small wrapper program can be dissected.

In mls.exe case, in order to define if a Portable Executable file is packed or not, the PEiD[42] have been used.

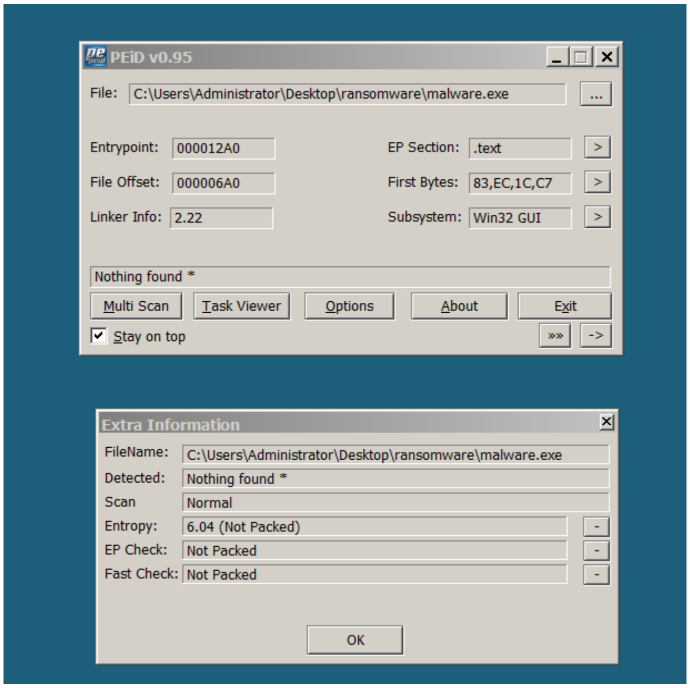


Figure 18:PEiD results

[42] PEiD (PE iDentifier) detects most common packers, crypters and compilers for PE files. It can detect more than 470 different signatures in PE files. There are 3 different and unique scanning modes in PEiD. The *Normal Mode* scans the PE files at their Entry Point for all documented signatures, the *Deep Mode* scans the PE file's Entry Point containing section for all the documented signatures. This ensures detection of around 80% of modified and scrambled files, and the *Hardcore Mode* does a complete scan of the entire PE file for the documented signatures. The hardcore mode should be used as a last option as the small signatures often tend to occur a lot in many files and so erroneous outputs may result.

PEiD shows that the mls.exe is not packed and the programming language that the file was written in cannot be detected. An important information for the PE is the Entropy which is significantly high[43] for an unpacked version. On the submenu of the application, we can also have detailed information for the PE directory.

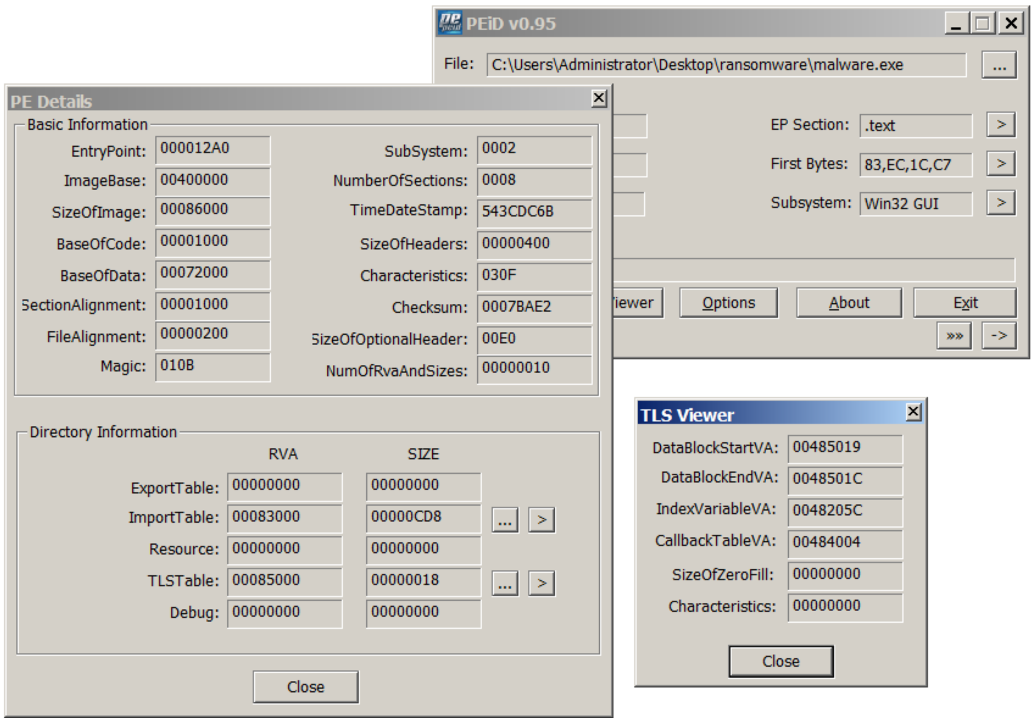


Figure 19: PEiD Details & TLS table view

[43] High refers to a value that is more than 5. Entropy analysis is used for a more generalized insight into the contents of PE files, mostly in regard to packing, compression and cryptography [5, 7] that are common with packers. When analyzing entropy, PE structural information such as sections can be taken into account. The main challenge with this approach is achieving sufficient expressiveness in presenting entropy information, because naive approaches can be fooled by file manipulation such as padding.

Changing the parameters on PEiD and adding some plugins[44], the results were the same. Note that Virtual Address (VA) is the original address in the virtual memory, whereas RVA is the relative address with respect to the ImageBase[45].

3.3.3. Detect It Easy

Due to the results of PEiD we force to dig more on the PE file and its structure. The tool that will give more information for the PE will be the Detect It Easy, or abbreviated "DIE"[46]. Other programs of the kind (PEID, PE tools) allow to use third-party signatures. Unfortunately, those signatures scan only bytes by the pre-set mask, and it is not possible to specify additional parameters. As the result, false triggering often occurs. More complicated algorithms are usually strictly set in the program itself. Hence, to add a new complex detect one needs to recompile the entire project, by the authors themselves. On the other hand, Detect It Easy has totally open architecture of signatures. Third-party algorithms of detects or modify those that already exist, is possible This is achieved by using scripts. The possibilities of open architecture compensate these limitations.

---

[44] Note that many PEiD plugins will run the malware executable without warning, so it is crucial to use this tool under a safe environment. In addition, alike other programs, especially those used for malware analysis, PEiD can be subject to vulnerabilities. In particular, PEiD version 0.92 contained a buffer overflow that allowed an attacker to execute arbitrary code, which would have allowed a clever malware writer to write a program to exploit the malware analyst's machine.

[45] In calculation, RVA = VA - ImageBase. Means for VA = 400100 and ImageBase = 400000, RVA will be 100.

[46] "Detect It Easy" is a cross-platform application, apart from Windows version there are also available versions for Linux and Mac OS. Detect It Easy, or abbreviated "DIE" is a program for determining types of files. First, DIE determines the type of file, and then sequentially loads all the signatures, which lie in the corresponding folder. Currently the program defines the following types: MSDOS, PE, ELF, MACH, Text files and Binary all other files. GitHub link of the tool: https://github.com/horsicq/Detect-It-Easy

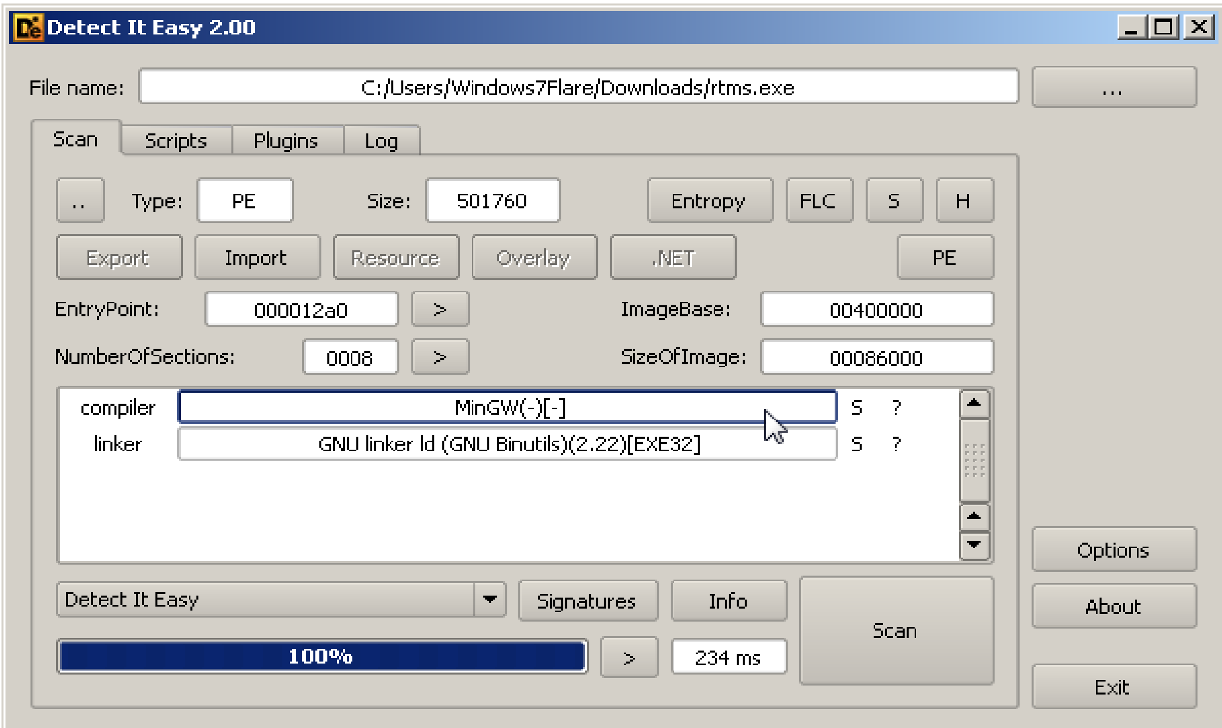


Figure 20:DiE scan results

Some quick information we can get from main GUI panel is that the compiler is MinGW and the linker is the GNU. Also, no packing was detected.

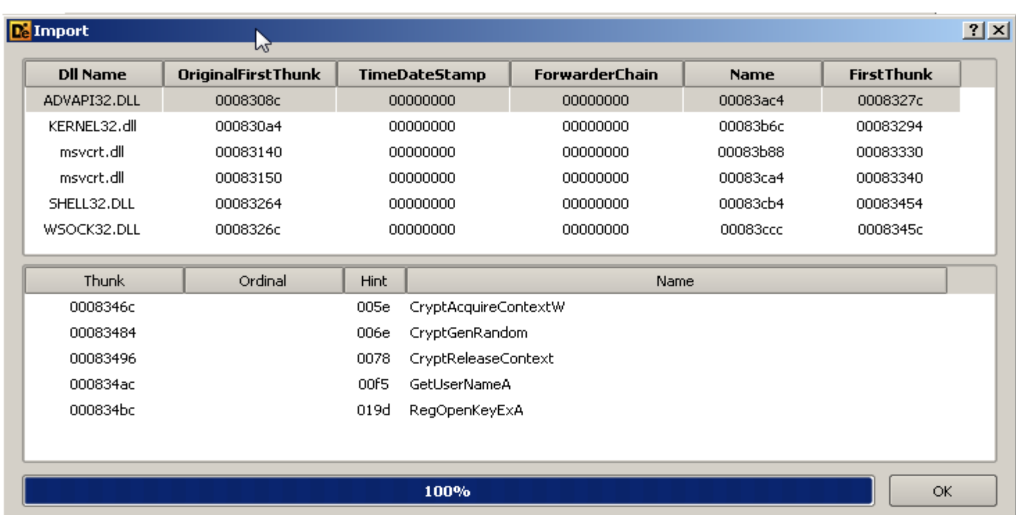


Figure 21: DiE results for imports

The imports of the PE are detailed presented, considering that Crypto Functions and a Registry open, are revealed.

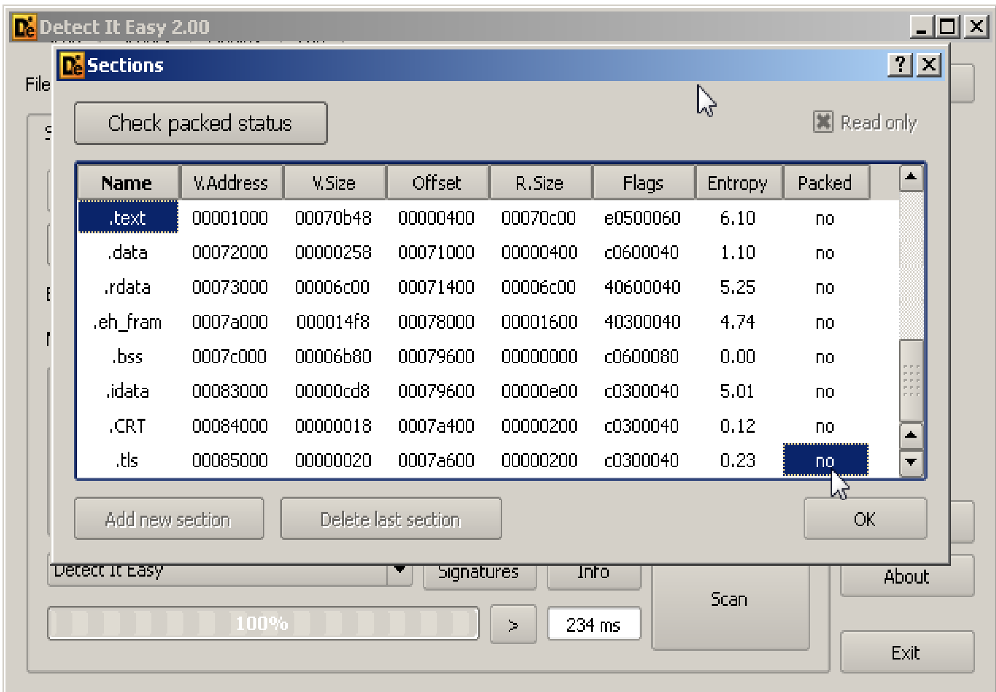


Figure 22: DiE results for several packed sections

In continuous, at the Sections option there are two extra information about possible packaging in each section and entropy measurement for each section also. The sections are (8) eight, as presented at PEview, with the above mentioned .tls section making the difference in this PE file. Nevertheless, the sections *CRT* [47] and *eh_fram* [48] is a confirmation that the PE file is written in C++.

[47] Data added for supporting the C++ runtime (CRT). A good example is the function pointers that are used to call the constructors and destructors of static C++ objects.

[48] When using languages that support exceptions, such as C++, additional information must be provided to the runtime environment that describes the call frames that much be unwound during the processing of an exception. This information is contained in the special sections .eh_frame and .eh_framehdr. Note that, the format of the .eh_frame section is similar in format and purpose to the .debug_frame section.The .eh_frame section shall contain one or more Call Frame Information (CFI) records. The number of records present shall be determined by size of the section as contained in the section header. Each CFI record contains a Common Information Entry (CIE) record followed by 1

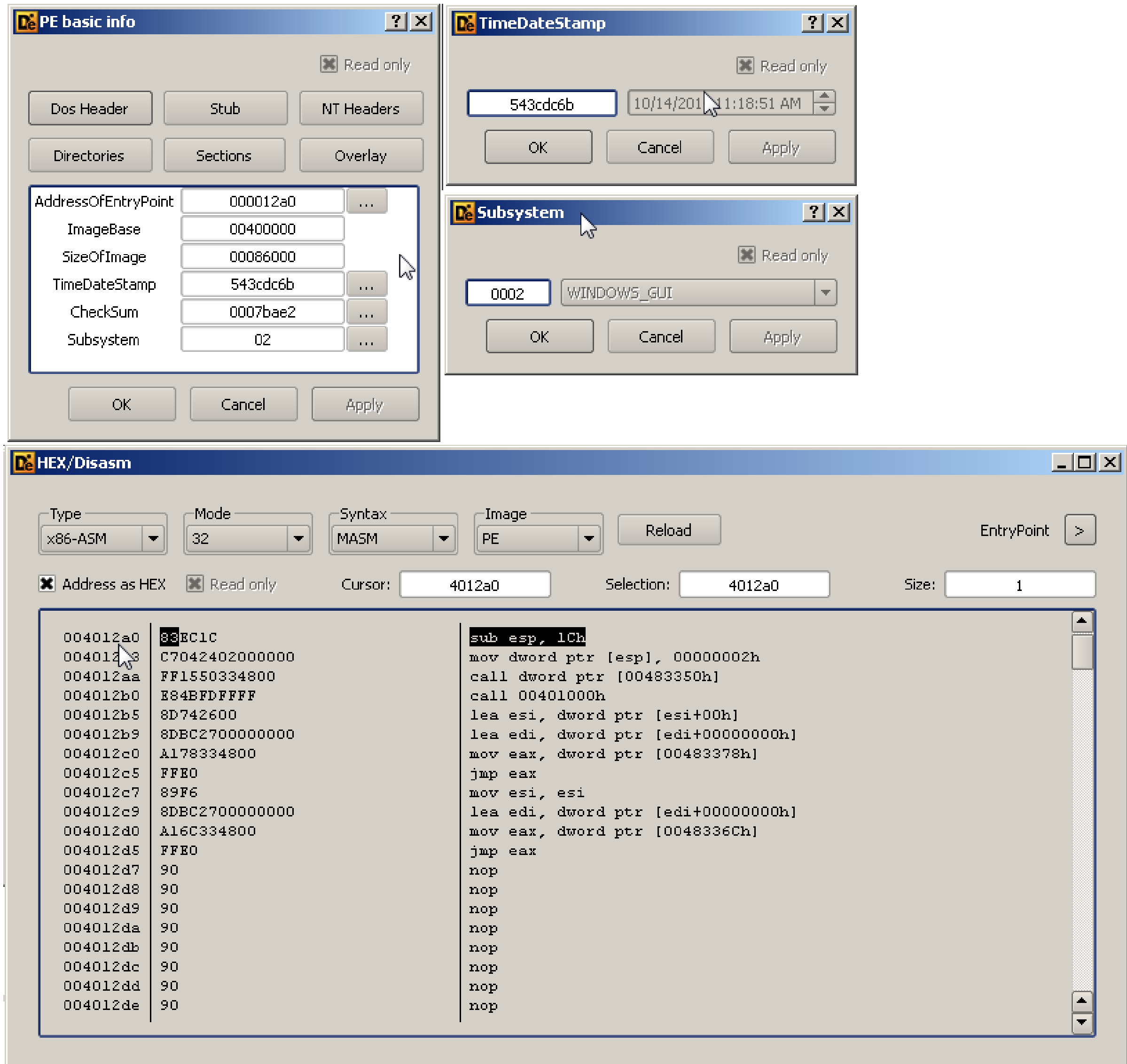


Figure 23:DiE PE basic info on Hex view with disasm

The DIE tool gives the ability to dig in the from a window with PE basic information. The important information for our analysis is that the sample has a GUI, which means that the malware want interaction with the victim or to present something.

---

or more Frame Description Entry (FDE) records. Both CIEs and FDEs shall be aligned to an addressing unit sized boundary.
Source url: http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html

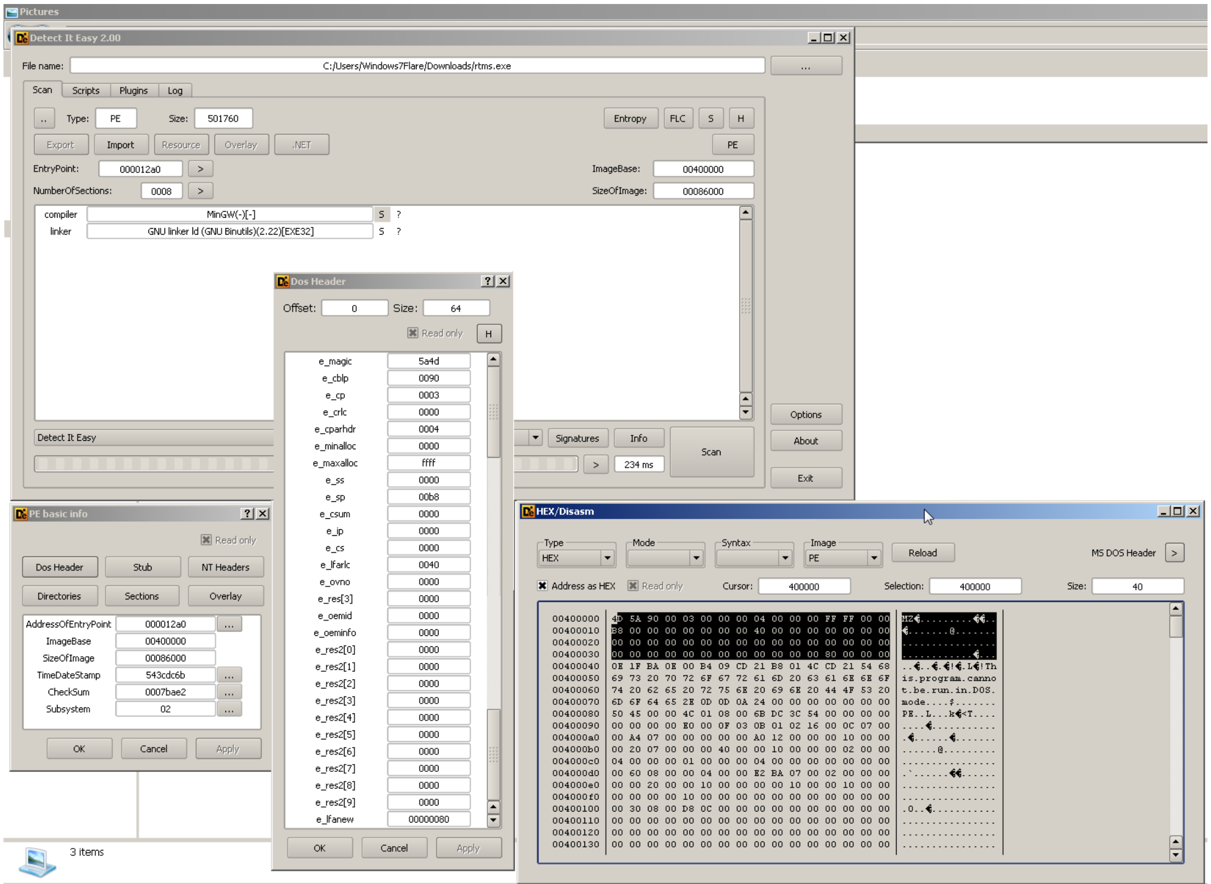


Figure 24: DiE Dos Header detailed preview in Hex disasm

The DOS Header is full of information for the PE file, but for the analysis only the *e_lfanew*[49] attribute is useful. The final field of Dos header, *e_lfanew*, is a 4-byte offset into the file where the PE file header is located. It is necessary to use this offset to locate the PE header in the file. Note that the *e_lfanew* has 80 as value and the size is the Dos Header is 64.

Following the DOS Header is the MS DOS stub. The file under analysis shows the known message, that is not compatible with DOS mode.

---

[49] The *e_lfanew* definition is separated in two parts. The *fanew*, which means: file address of new exe header and the *e*_prefix which helps deal with old K&R compilers that did not yet keep structure members in its own symbol table. The *l* after the prefix, is the system Hungarian for LONG and the "Long" stands because it's from the 16-bit era and the variable size is 32 bits

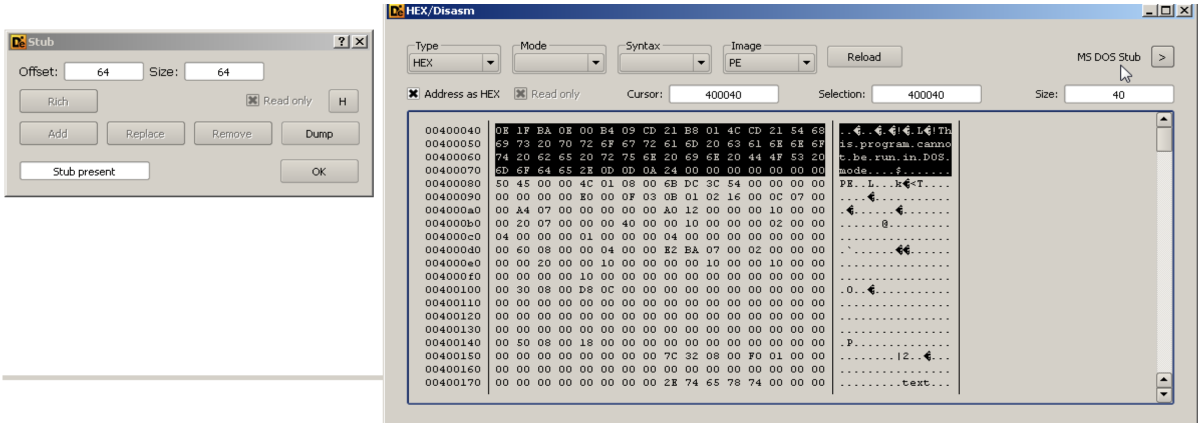


Figure 25:DiE Stub header

The next section of headers is the NT headers section, which is the structure *_IMAGE_FILE_HEADER*. In this section, the TimeDateStamp and the number of the section are taking place, which are already captured and analyzed. The DIE tool explains in depth the Characteristics and the type of machine the executable was built for. Specifically, the PE file under analysis was built for *i386* machine, which means for *Intel x86* architecture. The same information comes from Characteristics, where the *32bit_machine* is checked. The Characteristics field identifies specific attributes about the file and among the others, the *debug_stripped* have our attention in the analysis. The *debug_stripped*[50] indicates that debugging information is removed from the image file.

[50] It is possible to strip debug information from a PE file and store it in a debug file (.DBG) for use by debuggers. To do this, a debugger needs to know whether to find the debug information in a separate file or not and whether the information has been stripped from the file or not. A debugger could find out by drilling down into the executable file looking for debug information. To save the debugger from having to search the file, a file characteristic that indicates that the file has been stripped (IMAGE_FILE_DEBUG_STRIPPED) was invented. Debuggers can look in the PE file header to quickly determine whether the debug information is present in the file or not.

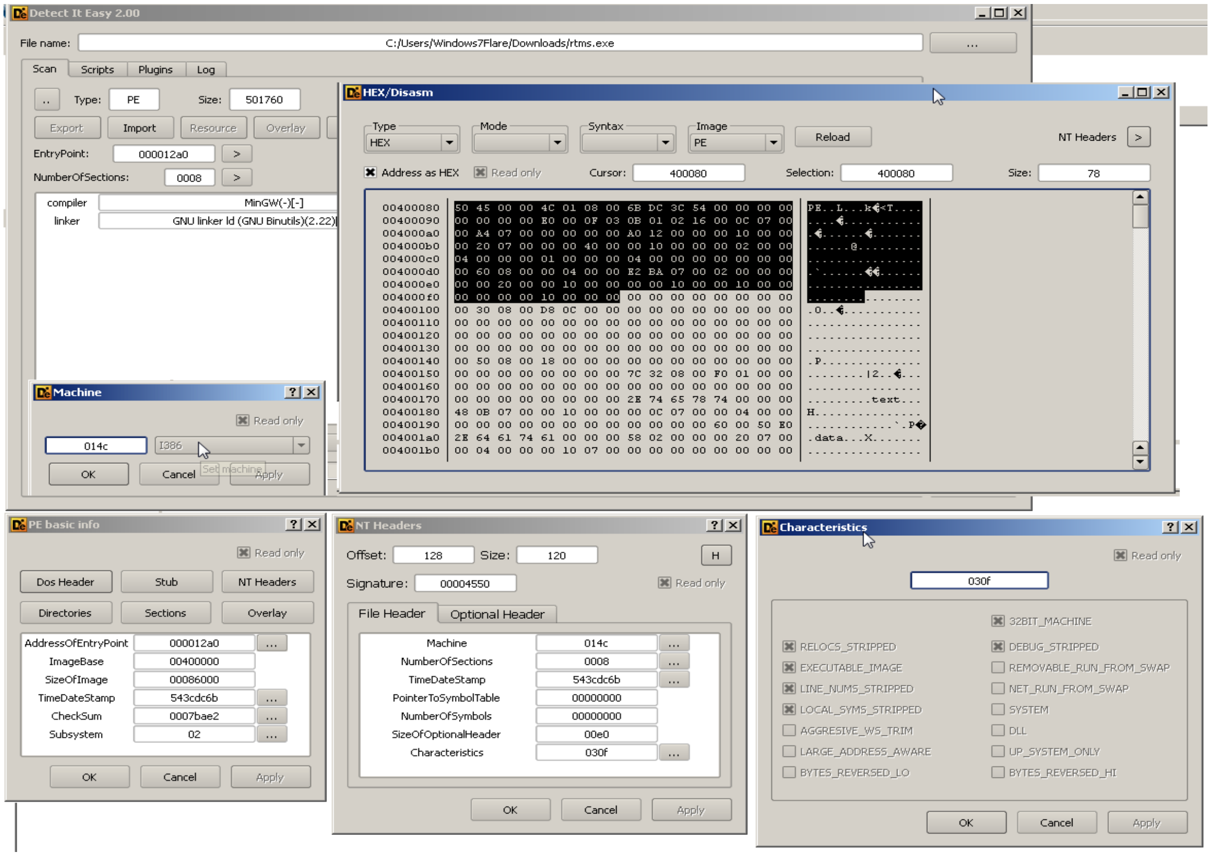


Figure 26: DiE Characteristics on NT Header-File Headers

There are some Optional Header available on File Headers of NT Headers. The AddressOfEntryPoint field has the value *000012a0* and is the most interesting for the PE file format. This field indicates the location of the entry point for the application and, perhaps more importantly to system hackers, the location of the end of the Import Address Table (IAT).

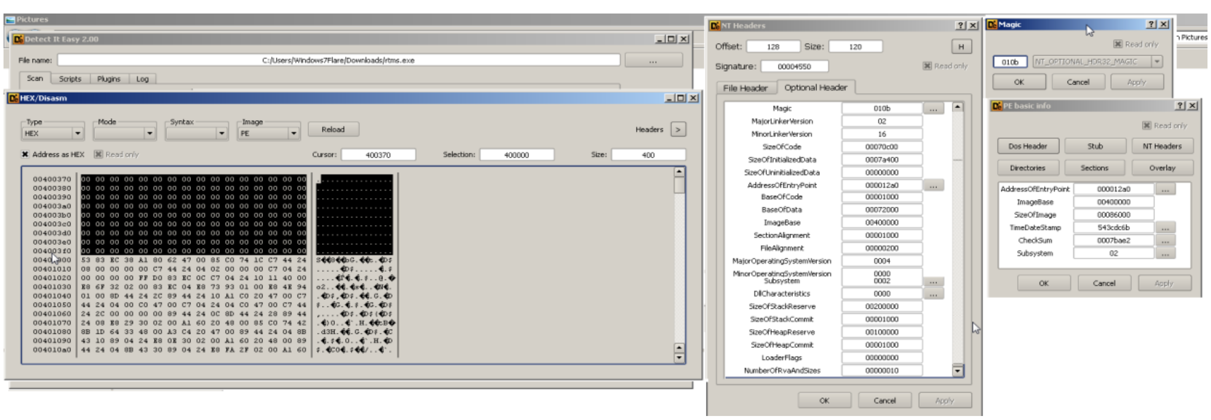


Figure 27: : DiE Characteristics on NT Header-Optional Headers

In continuous, the DIE tool presents the Data Directory[51] as Directories with significant details. The PE file format under analysis, defines 16 possible data directories, 3 of which are now being used. On the following figure, the *IMAGE_DIRECTORY_ENTRY_IMPORT* is being showed in HEX and in an GUI array.

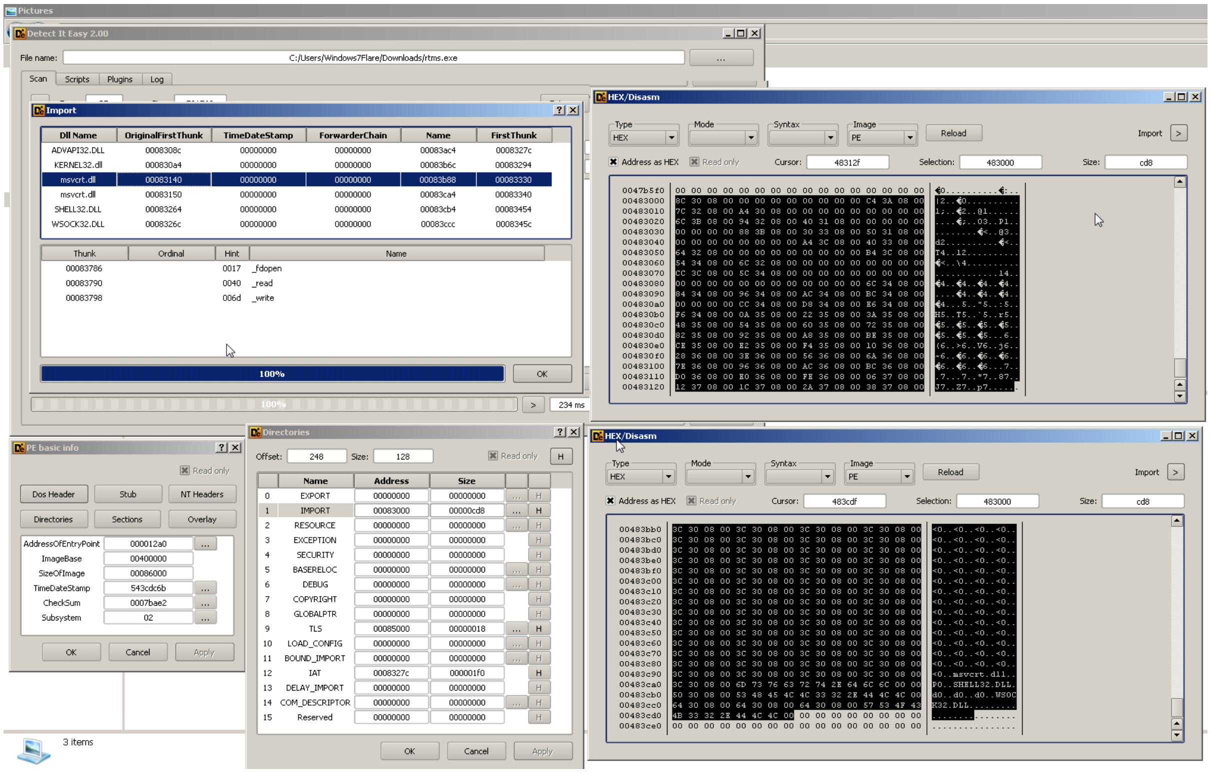


Figure 28: DiE import Directory and its offset

In addition, the DIE tool presents the *IMAGE_DIRECTORY_ENTRY_TLS* as TLS with significant details. The *AddressOfCallBacks* value 00484004 is being noted for our dynamic analysis.

---

[51] DataDirectory. The data directory indicates where to find other important components of executable information in the file. Specifically, is an array of IMAGE_DATA_DIRECTORY structures that are located at the end of the optional header structure.

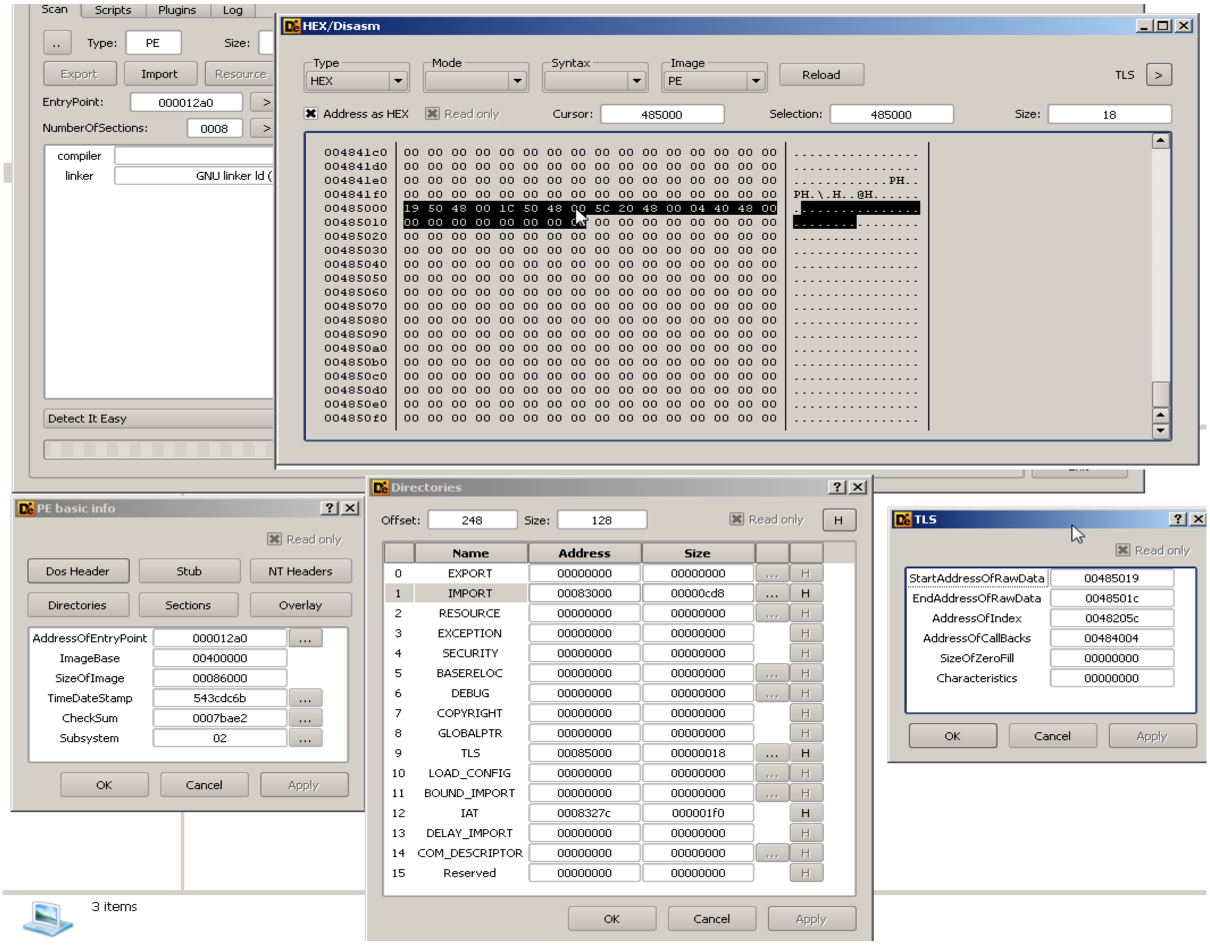


Figure 29: DiE TLS table in detail and in Hex view

At last for Directories, the *ImportAddressTable* (IAT) is located in the .text section immediately before the module entry point[52]. When Windows NT executable images are loaded into a process's address space, the IAT is fixed up with the location of each imported function's physical address. In order to find the IAT in the .text section, the loader simply locates the module entry point and relies on the fact that the IAT occurs immediately before the entry point. And since each entry is the same size, it is easy to walk backward in the table to find its beginning.

[52] The IAT's presence in the .text section makes sense because the table is really a series of jump instructions, for which the specific location to jump to is the fixed-up address.

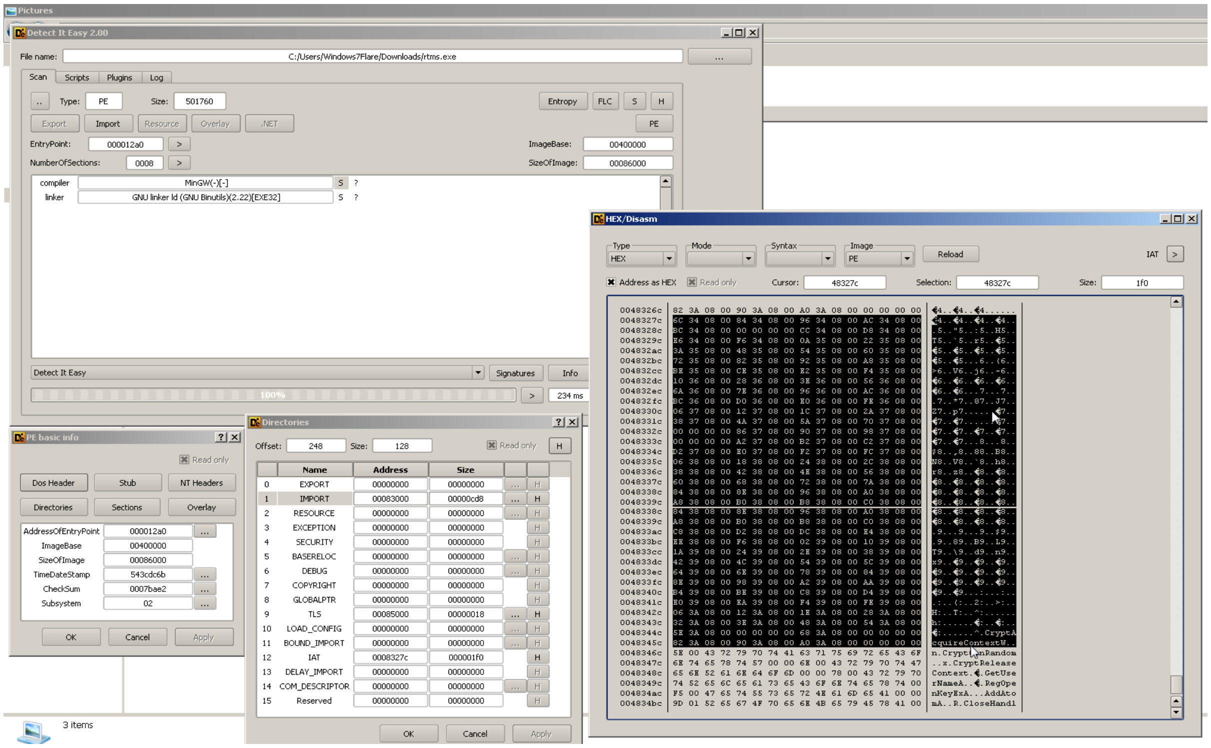


Figure 30: DiE, Directory of the ImportAddressTable (IAT)

At last the DIE tools has some graphical representation of each section, which is very useful as a simple visualization of the PE file content. There are two types of graphs, the Curve graph, which presents on axis X the size of the PE file (bytes), on axis Y the entropy of the hex bytes and the Histogram graph, which presents on axis X each byte of the PE file (decimal), on axis Y the frequency of each byte. Also, there is an array with the content of the Histogram, adding the percentage of the frequency of each byte.

On the following Figure, the .text (section 0) is being selected as the most important section of the PE file and the one with the largest content of bytes.

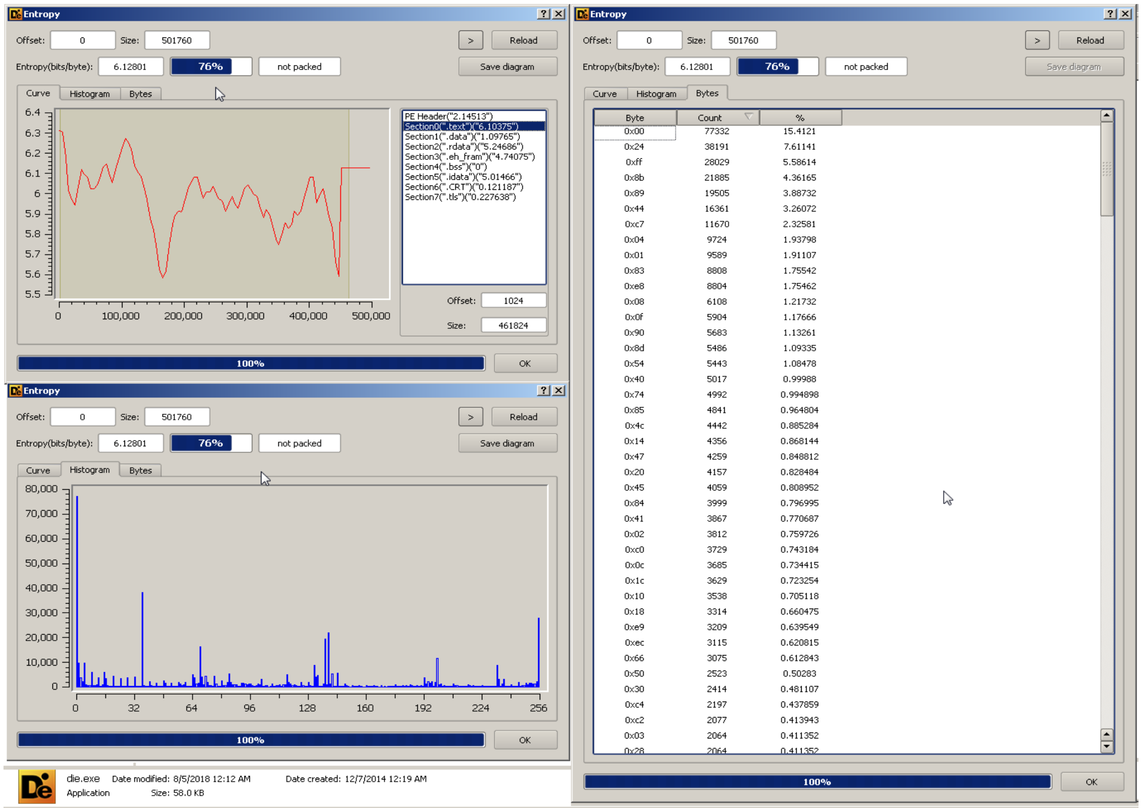


Figure 31: DiE Visualization Entropy of .text section

Keep in mind that the General Entropy of the whole PE file is 6.12801, that differs a bit from PEview’s entropy (6.04).

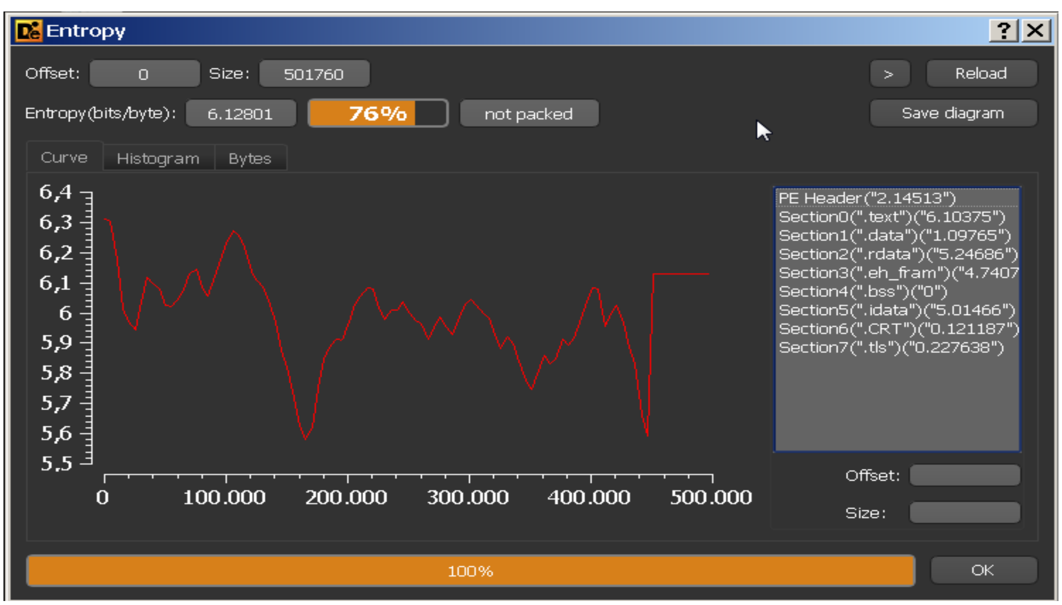


Figure 32: DiE Visualization Entropy of all sections

3.3.4. PortexAnalyzer

Another tool that is great on visualization, is the PortExAnalyzer[53], which generate a graph of colors, to visually detect a packing on a PE file. On the current PE file under analysis, a cross check is being made that no hidden packer is being used.

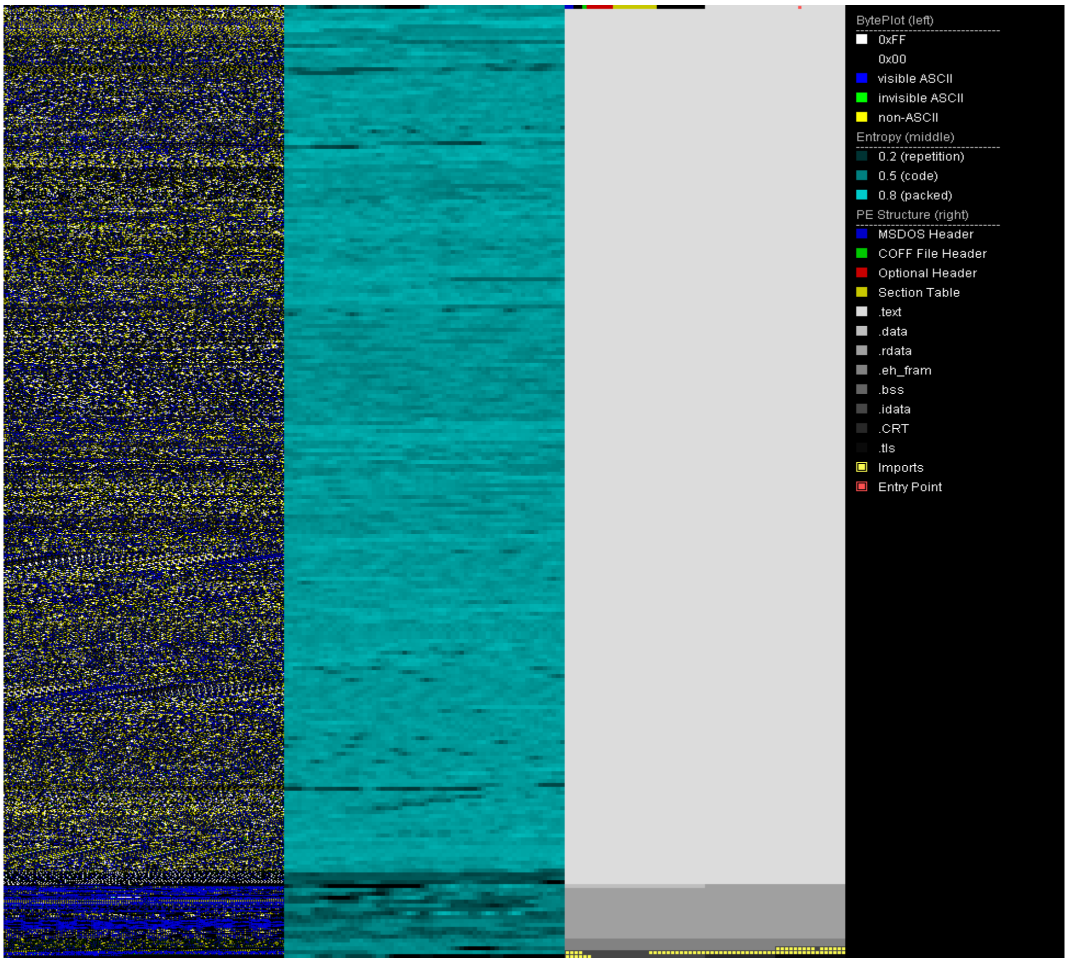


Figure 33: PortexAnalyzer PE structure and Entropy visualization

---

[53] PortExAnalyzer is a command line tool that runs the library PortEx under the hood. PortExAnalyzed is readily compiled command line PE scanner to analyze files with it. Note that, PortEx is a Java library for static malware analysis of Portable Executable files. Its focus is on PE malformation robustness, and anomaly detection. PortEx is written in Java and Scala and targeted at Java applications. GitHub link of the tool: https://github.com/katjahahn/PortEx

On PortExAnalyzer graph, 3 subgraphs are being presented. Each graph present the PE file as it stored in memory (from lower to higher address). More specifically, on the left side a Byte plot is being presented, with a color visualization, focused on possible ASCII characters on the PE file under analysis. On the middle side, the entropy is being colored differently for each memory address - PE file section. And the right side, there are different colors for each PE file section and its subsections. With this type of visualization, the analyst can match and detect visually, the location of possible packing, where possible ASCII characters are being stored and the comparative size of each PE file's section.

In addition, the PortExAnalyzer generates a great summarize report of all the above-mentioned notes, using the command on terminal:

```
java -jar PortexAnalyzer.jar -o report.txt -p graph.png rtms.exe
```

The PortExAnalyzer PE file report, is being attached at **Appendix G.** As a sum up from the PortExAnalyzer report, the malformation[54] characteristics that the PE file are:

- At the COFF Header, the time date stamp is crafted.
- COFF line numbers have been removed, due to deprecation.
- COFF symbol table entries for local symbols have been removed, due to deprecation.
- Section .text has "write" and "execute" characteristics.
- The writeable section .text is also the entry point
- The import VirtualProtect function may set PAGE_EXECUTE flag for memory region, which will lead to typical for code injection.
- Debugging is removed from the image file.

There is a gap between the PE format that the PE/COFF specification describes and the PE files that are allowed to run. The PE/COFF specification uses misleading field names and descriptions, is more restrictive than the loader. Furthermore, the behavior of the loader varies in

[54] Definition: A **PE malformation** is data or layout of a PE le that violates conventions or the PE/COFF specification. File format malformations represent special case conditions that are introduced to the file layout and specific fields in order to achieve undesired behavior by the programs that are parsing it.
Source url: https://media.blackhat.com/bh-us-11/Vuksan/BH_US_11_VuksanPericin_PECOFF_WP.pdf

different Windows versions, with every new version of Windows possibly introduces formerly unknown malformations.

3.3.5. PEstudio

At this point the Malware Initial Assessment has been done in dept, but the most famous and recognized tool for many Computer Emergency Response Teams (CERT) worldwide in order to perform Malware Initial Assessment is the PEstudio[55]. PEstudio shows Indicators as a human-friendly result of the analyzed image. Indicators are grouped into categories according to their severity. Indicators show the potential and the anomalies of the application being analyzed. The classifications are based on XML files provided with PEstudio. Among the indicators, PEstudio shows when an image is compressed using UPX or MPRESS.

On the first view option of the PEstudio, the basic information about the PE file are being previewed. Note that again the entropy of the PE is 6.132 due to PEstudio, that differs from 6.04 of PEview and 6.12801 of DIE. This leads to the indication that, the entropy is being measured differently by each tool, as a result be reliable.

---

[55] **PEstudio** is a utility can be used to Triage malware analysis. Runs on Windows Platform and is fully portable. Malicious software often attempts to hide its intents in order to evade early detection and static analysis. In doing so, it often leaves suspicious patterns, unexpected metadata, anomalies and other valuable indicators. The goal of PEstudio is to spot these artifacts in order to ease and accelerate Malware Initial Assessment. The tool uses a powerful parser and a flexible set of XML configuration files that are used to detect various types of indicators and classify items. Note that, since the file being analyzed is not under execution yet, the inspection of the unknown or malicious executable file can be done without any risk of infection.

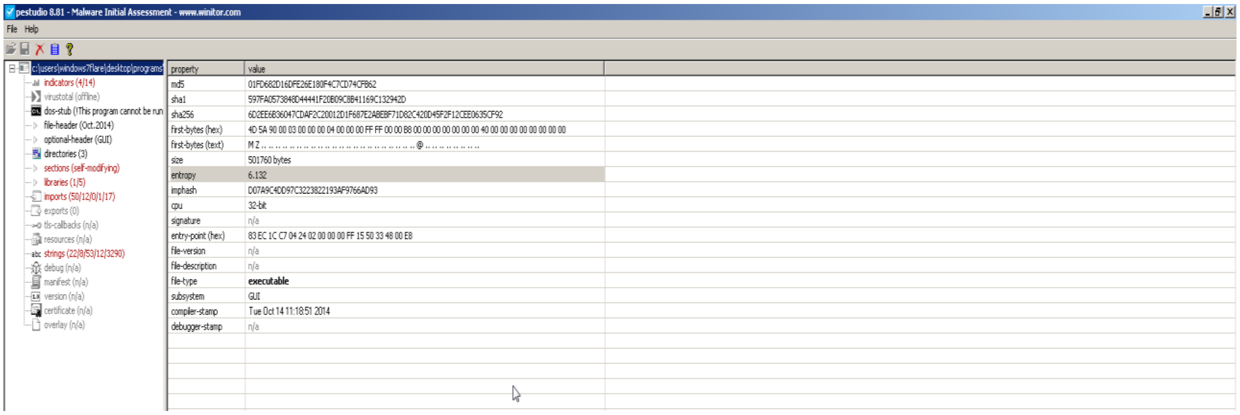


Figure 34: PEstudio general information

The indicator window explains why PEstudio show this file as suspicious, with a severity ranking order.

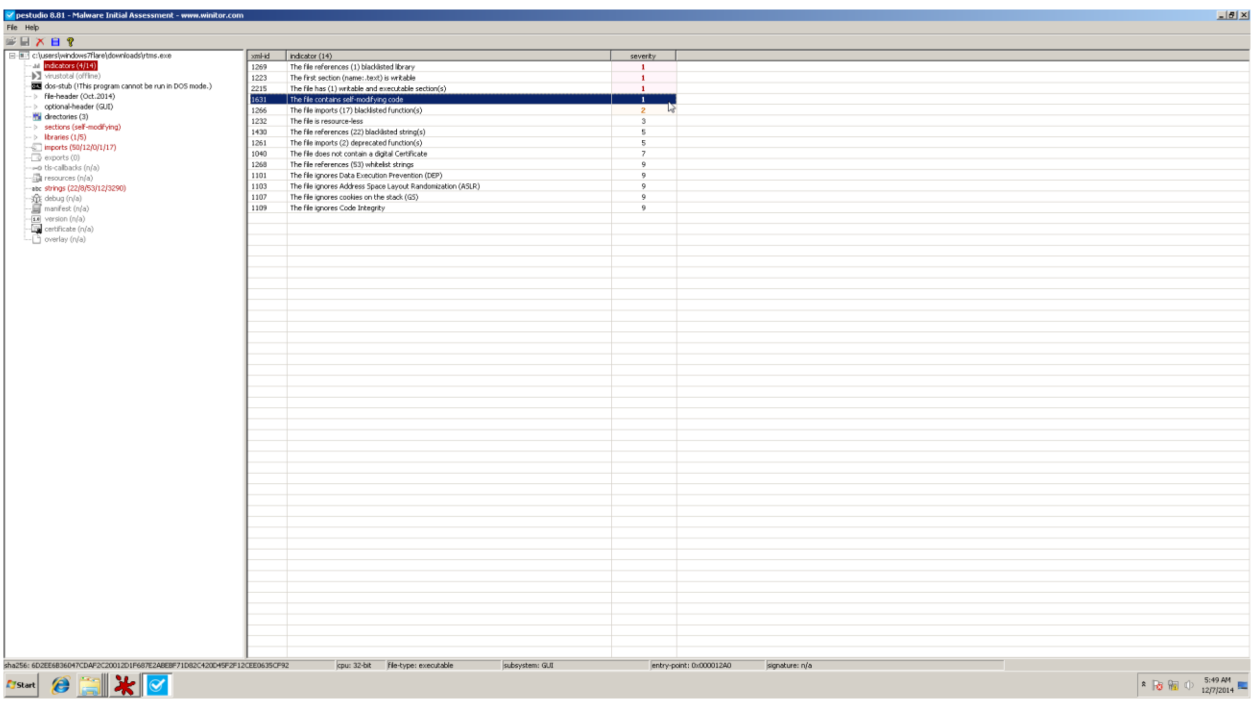


Figure 35: PEstudio Indicators

It summarizes the indicators found further down in the menu tree. The new finding on the file under analysis, is the detection of that the file contains self-modifying code[56].

In addition, the under analysis file ignores Address Space Layout Randomization (ASLR)[57]. It also ignores Data Execution Prevention (DEP) which would allow for code execution from the Data Section in memory.

By default, PEstudio will send a MD5 hash of the file to VirusTotal and it will retrieve the results, but this procedure already have been done manually.

The DOS-stub is next. This window displays information about the DOS application header which comes before the PE header information. It is very rare that an application has much in the dos-stub. In addition, PEstudio displays in DOS-stub the MD5 hash the size, and entropy of the dos-stub.

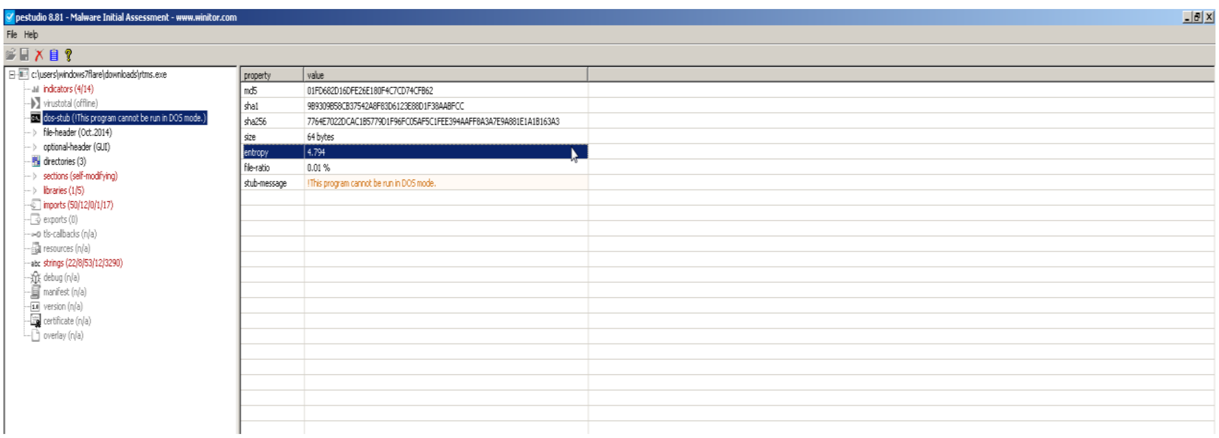


Figure 36: PEstudio dos-stub

---

[56] Self-modifying code is a technique where the actual opcodes of the binary are changed dynamically (at run-time), making it impossible to see what the code does without stepping through it. There are plenty of reasons this technique is used: the function call encrypted in this section will not show up in the intermodular calls, the random data can trick disassemblers into thinking its code, and after the opcodes get decrypted, you must tell the disassembler to re-analyze these bytes as opcodes instead of data.

[57] ASLR is a feature which simply loads an application into memory at a somewhat randomized preventing the ability to successfully perform a buffer overflow attack.

File-header is interesting if simply because it contains some useful information to accurately describe a sample. This window provides information that would be in the PE header if you were analyzing this in another application. In fact, the signature field 0x00004550, converts to ASCII "EP" and reading it flipped (endianness), it states "PE". Note that the debug information stripped, is being also confirmed by PEstudio.

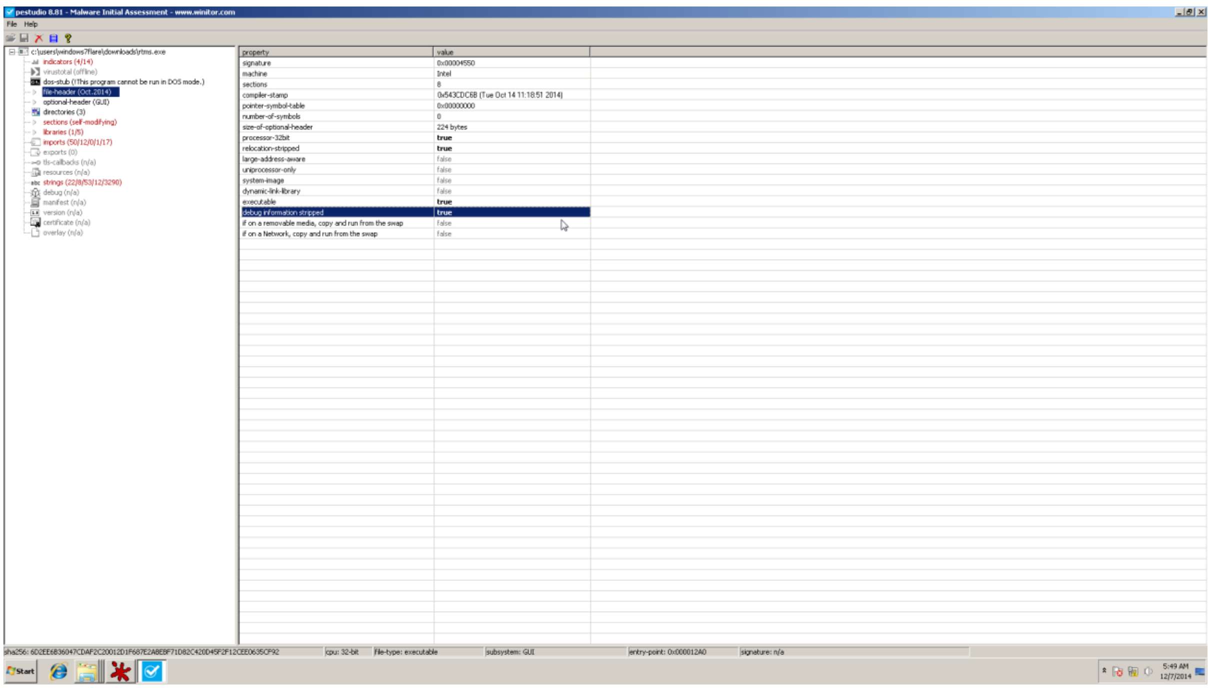


Figure 37: PEstudio file header

The optional header contains information that was at one time completely optional but is not mostly required for an application to execute inside a modern Windows environment. At the bottom of the window though we have information about ASLR, DEP (which the indicators have already show them) and Structured Exception Handling (SEH)[58].

---

[58] SEH is the ability of an application to handle exceptions on its own. The common applications crash is actually an exception. The ability of the developers to define on them applications an execution of another subroutine, if an exception were to occur during runtime, gives the ability to malware authors though SEH code, to use it as a mechanism to obfuscate their malicious code.

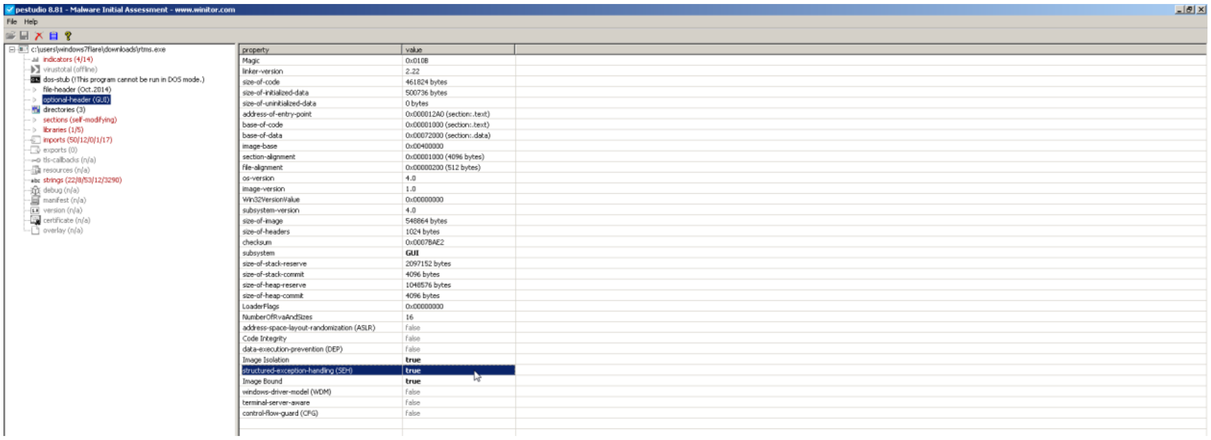


Figure 38: PEstudio optional headers

Sections is a useful piece of information when trying to determine if a file is malicious. Note that, the top indicator was the self-modifying code section. The .text section contains the executable code. Each of the sections has a read, write, and/or execute permission. What permission is applied to the section is denoted by an x in the appropriate field. The normal expectation on the .text section is to have Read and Execute permissions. The .text section should never have written permissions, otherwise this means the application can actively modify itself. Also, in the .text section is the entry-point, where the first line of executable code, when the application is loaded into memory.

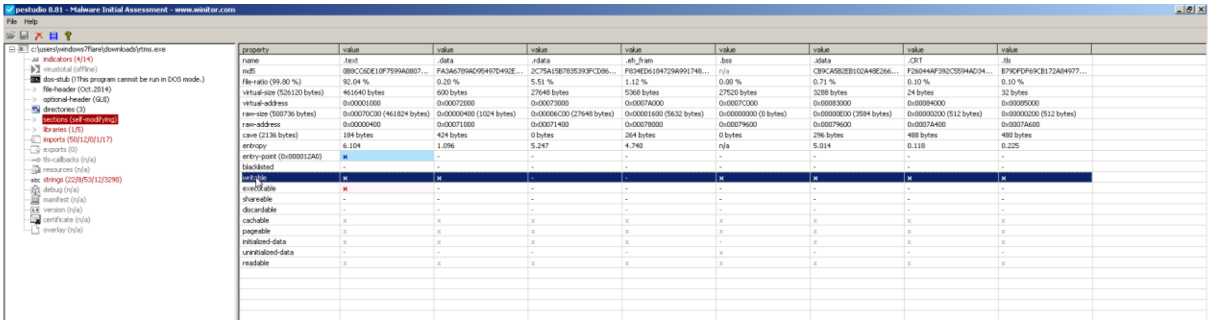


Figure 39: PEstudio sections and them RWE rights

Imports contain the actual imported function names. PEstudio has a list of blacklisted imports, which are all API functions in Windows which are not malicious in their own right but can be used to perform functions which may be considered malicious.

Function imports can be referenced by ordinal number as well. Libraries which contain exports assign a number to each export. The author of the PE can choose to use the number rather than the name of the import, which is often a technique to obfuscate what the application is importing. PEstudio is pretty good at finding the actual name of imports referenced by ordinal.

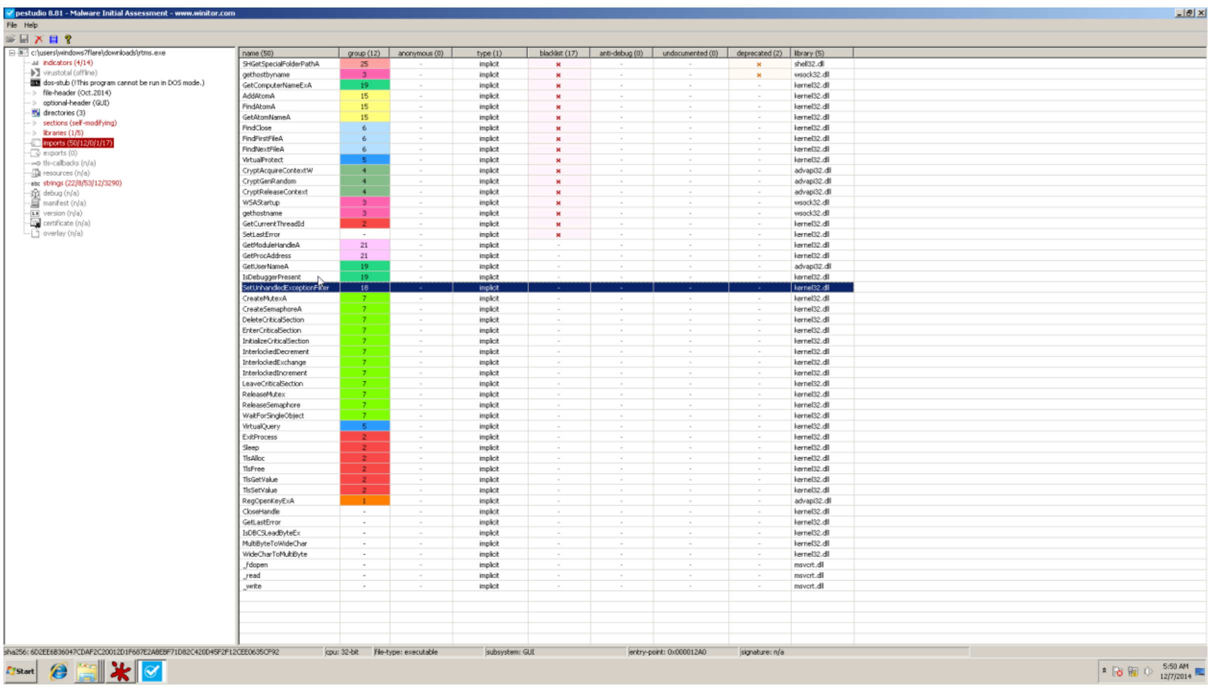


Figure 40: PEstudio imports grouping and ranking

Strings actually is any string from the raw bytes which can be read as ASCII or a UNICODE character, which is parsed and placed in PEstudio's table. Unlike linux/unix strings command[59], PEstudio will mark any suspicious string, that comes with a predefined list of a suspicious strings.

It is concerning that there are very few readable strings. Having a minimal number of readable strings would indicate the application is being obfuscated.

Note that a serial of ASCII character-set has been detected. Such a string indicates that an encoding schema is being used. This ASCII character-set seems to be a Base64 input, but this will be confirmed only on dynamic analysis.

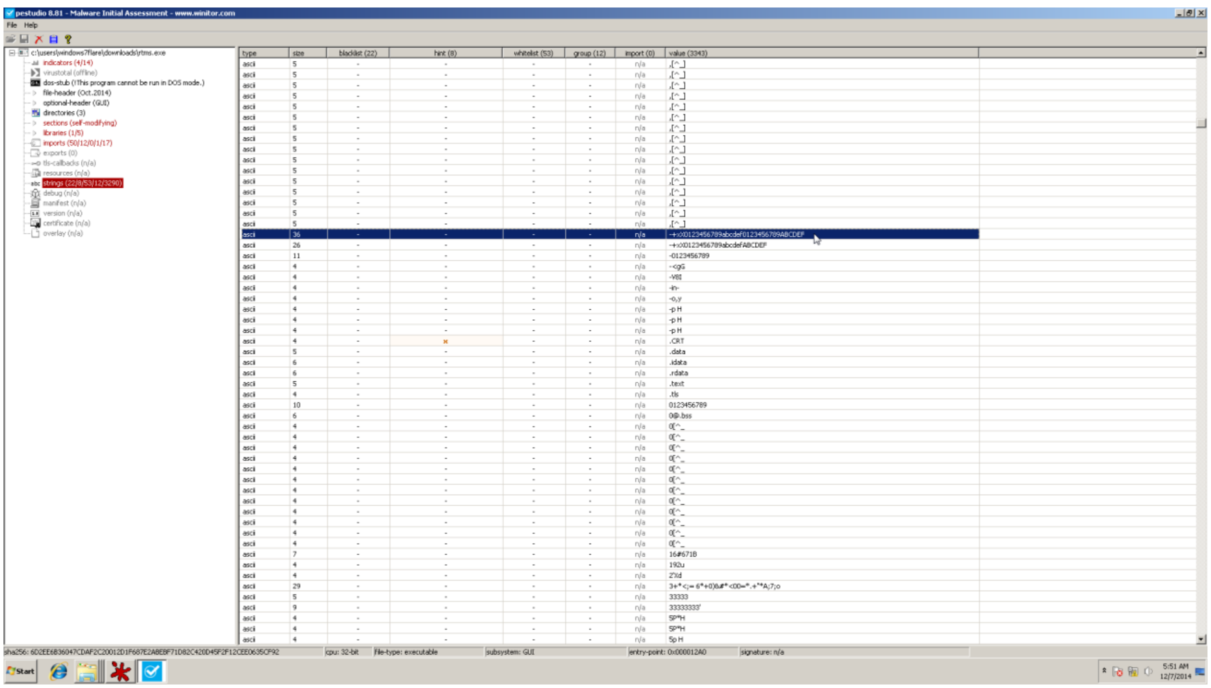


Figure 41: PEstudio strings ranking and evaluation

[59] In computer software, strings is a program in Unix-like operating systems that finds and prints text strings embedded in binary files such as executables. It can be used on object files and core dumps. Strings are recognized by looking for sequences of at least 4 (by default) printable characters terminating in a NUL character (that is, null-terminated strings). Some implementations provide options for determining what is recognized as a printable character, which is useful for finding non-ASCII and wide character text. Source: https://en.wikipedia.org/wiki/Strings_(Unix)

3.3.6. BinText

Another tool digesting string theory of a PE file, we can use several tools. An application for Windows OS is the BinText[60].

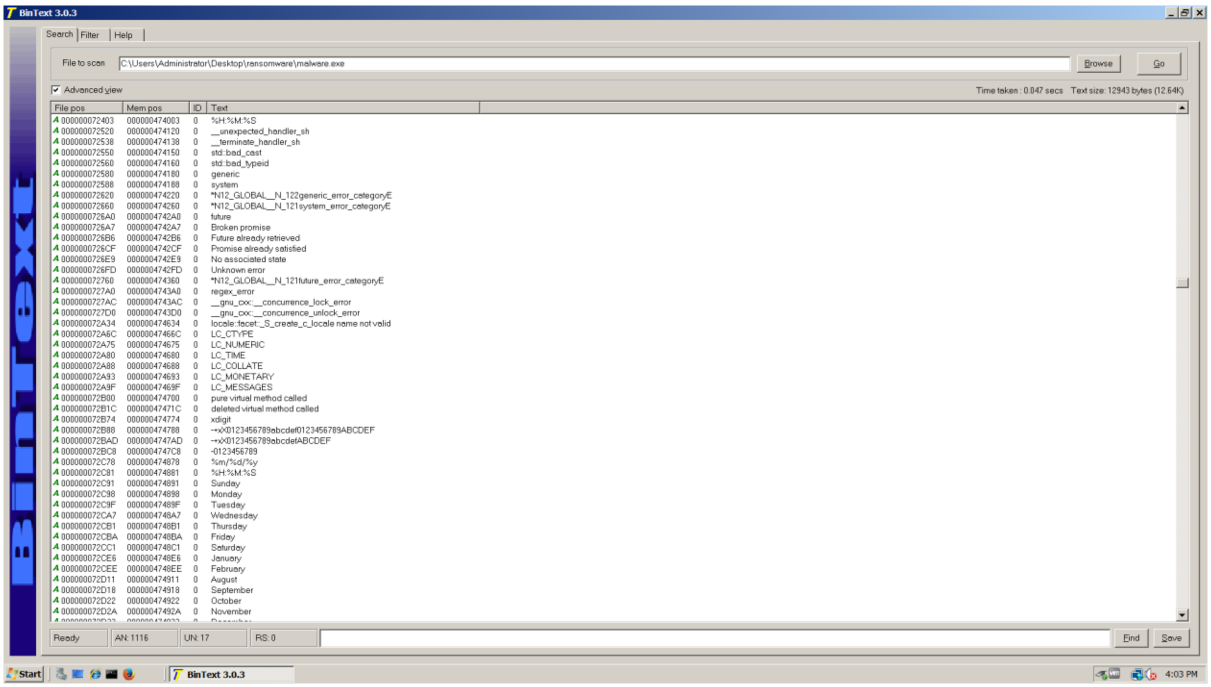


Figure 42: BinText string search and filtering

The strings of the file under analysis are reasonably the same with all the above tools. Considering that most of the strings are non-human-readable ASCII characters, we assume that an obfuscation is taken place. On Appendix H/BinText, the results of BinText's are being extracted.

The main advantage of BinText and the purpose of using this tool, are BinText's filters. More specifically, as it is being shown on the following Print screen, BinText has GUI to exclude or include any character in the definition of a string, giving the ability to specify some unique

---

[60] BinText is a file text scanner / extractor that helps find character strings buried in binary files. The program can extract text from any kind of file and display plain ASCII text, Unicode (double byte ANSI) text, as well as Resource strings. Additional useful information for each item is included in the "Advanced" mode. Uniquely, the program will show both the file offset and the memory offset of each string found.

strings with special characters. Unfortunately, in the PE file under analysis, the addition of more filters, prints more non-human-readable strings and with a specific selection of filters, some strings continue to be non-human-readable.

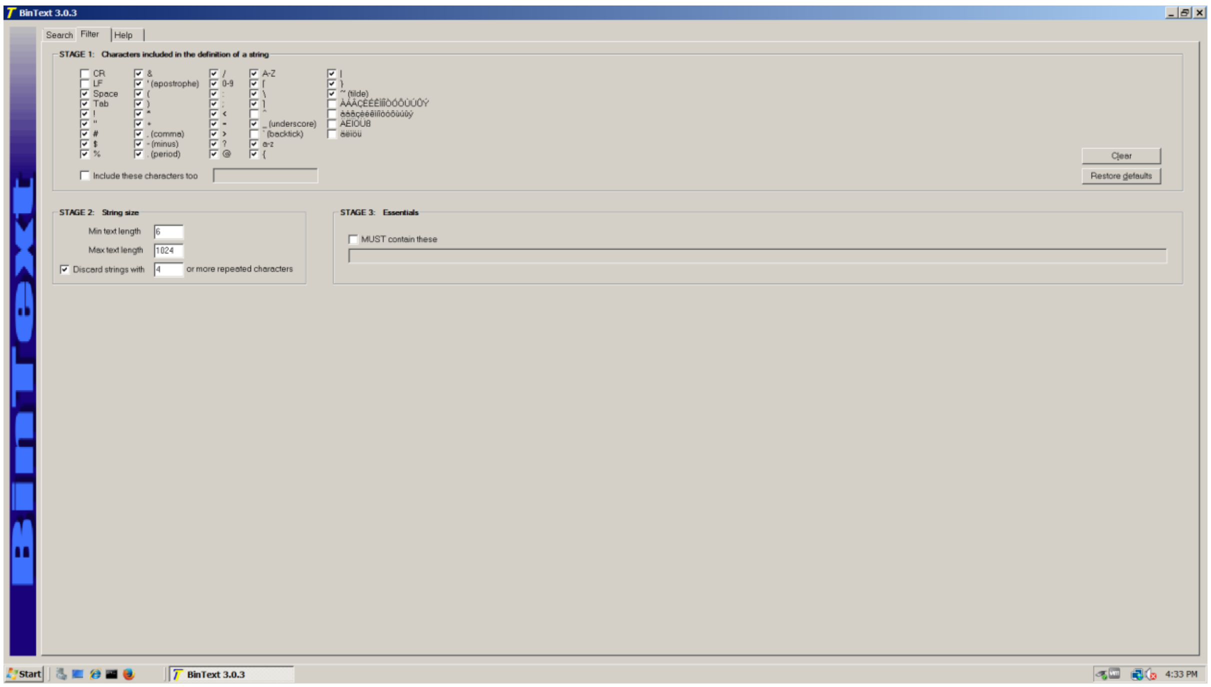


Figure 43: BinText filtering settings and strings length

4. Behavioral Analysis

This section describes the basic dynamic analysis techniques. Dynamic analysis is any examination performed after executing malware. Dynamic analysis techniques are the second step in the malware analysis process. Dynamic analysis is typically performed after basic static analysis has reached a dead end, whether due to obfuscation, packing, or the analyst having exhausted the available static analysis techniques. It can involve monitoring malware as it runs or examining the system after the malware has executed. Unlike static analysis, dynamic analysis lets you observe the malware's true functionality, as the existence of an action string in a binary does not mean the action will actually execute.

Although dynamic analysis techniques are extremely powerful, they should be performed only after basic static analysis has been completed, because dynamic analysis can put your network and system at risk. There are limitations in Dynamic techniques also, because not all code paths may execute when a piece of malware is run.

4.1.Basic Dynamic Analysis with free Sandboxes

Why invent a new wheel when you can walk to the store and buy one? Why invent a wheel when you can invent the engine?[61]

Our first step on Surface Analysis, was to upload the file under analysis in an online service and check the past work from other analysts. These online tools have been expanded, not only to characterize a file as a malicious, via its hash value, but analyze them header and some of them, does on step further, a Basic Dynamic Analysis report.

[61] An idiom common amongst engineers and developers.

The HybridAnalysis results are available offline on the Appendix D/HybridAnalysis results and online on the source url:

www.hybrid-analysis.com/sample/6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92/

4.1.1. Results explanation

Malicious Indicators

Environment Awareness: The input sample contains a known anti-VM trick

This indicator, anti-virtual machine (anti-VM) is a set of techniques to thwart attempts at analysis. With these techniques, the malware attempts to detect whether it is being run inside a virtual machine. If a virtual machine is detected, it can act differently or simply not run.

Suspicious Indicators

Environment Awareness: Contains ability to measure performance - Anti-Debugging

The most common timing check[62] method uses the **RDTSC** instruction (opcode 0x0F31), which returns the count of the number of ticks since the last system reboot as a 64-bit value placed into EDX:EAX. Malware will simply execute this instruction twice and compare the difference between the two readings. The malware checks the difference between the two calls to RDTSC.

[62] Timing checks are one of the most popular ways for malware to detect debuggers because processes run more slowly when being debugged. For example, single-stepping through a program substantially slows execution speed. There are a couple of ways to use timing checks to detect a debugger: a) Record a timestamp, perform a couple of operations, take another timestamp, and then compare the two timestamps. If there is a lag, you can assume the presence of a debugger. b) Take a timestamp before and after raising an exception. If a process is not being debugged, the exception will be handled really quickly; a debugger will handle the exception much more slowly. By default, most debuggers require human intervention in order to handle exceptions, which causes enormous delay. While many debuggers allow you to ignore exceptions and pass them to the program, there will still be a sizable delay in such cases.

Anti-Debugging : Contains ability to query CPU information

This indicator, **CPUID**, is an anti-Debugging technique. The virtual machine monitor program monitors the virtual machine's execution. It runs on the host operating system to present the guest operating system with a virtual platform. It also has a couple of security weaknesses that can allow malware to detect virtualization[63].

Some instructions access hardware-based information without generating interrupts. Among others, these are SIDT, SGDT, SLDT and **CPUID**. In order to virtualize these instructions properly, VMware would need to perform binary translation on every instruction (not just kernel-mode instructions), resulting in a huge performance hit. To avoid huge performance hits from doing full-instruction emulation, VMware allows certain instructions to execute without being properly virtualized. Ultimately, this means that certain instruction sequences will return different results when running under VMware than they will on native hardware.

Malware exploit the usage of these instructions in order to perform VMware detection. Keep in mind, that these instructions are not useful if executed in user mode, so if you see them, they're likely part of anti-VMware code.

Remote Access Related

This indicator, the registry input, remote access related action, which reads terminal service related keys. The registry key: "HKLM\SYSTEM\CONTROLSET001\CONTROL\TERMINAL

[63] In kernel mode, VMware uses binary translation for emulation. Certain privileged instructions in kernel mode are interpreted and emulated, so they don't run on the physical processor. Conversely, in user mode, the code runs directly on the processor, and nearly every instruction that interacts with hardware is either privileged or generates a kernel trap or interrupt. VMware catches all the interrupts and processes them, so that the virtual machine still thinks it is a regular machine.

SERVER"[64]; Key: "**TSUSERENABLED**"[65], seems that a backdoor is being established. Backdoors are the most commonly found type of malware, and they come in all shapes and sizes with a wide variety of capabilities. Backdoor code often implements a set of capabilities, so when using a backdoor attack would not need to download additional malware or code.

Unusual mutants

The creation of these mutants[66] have been used as al technique in the context of other malwares. The malware under analysis seems to create the mutex to ensure that only one version of the malware is running at a time. Mutexes can provide an excellent fingerprint for malware if they are unique enough. The creation of mutexes are the followings:

```
"gcc-shmem-tdm2-use_fc_key"
"gcc-shmem-tdm2-fc_key"
"gcc-shmem-tdm2-sjlj_once"
```

Anti-debugging TLS callbacks Related

This indicator, the TLS Callback, has been already described in detail from basic static analysis, when the .tls section was found on the PE. More specifically from the advanced static

---

[64] The HKLM\SYSTEM\ControlSet001HKLM\SYSTEM\ControlSet001\Control\Terminal Server hive allows you to configure general settings, just as you can under Terminal Services configuration or Group Policies.

[65] The **TSUserEnabled** value, indicates whether users can log on to the terminal server.

[66] Mutants or Mutexes are global objects that coordinate multiple processes and threads. Mutexes are mainly used to control access to shared resources and are often used by malware. For example, if two threads must access a memory structure, but only one can safely access it at a time, a mutex can be used to control access. Only one thread can own a mutex at a time. Mutexes are important to malware analysis because they often use hard-coded names, which make good host-based indicators. Hard-coded names are common because a mutex's name must be consistent if it's used by two processes that aren't communicating in any other way. The thread gains access to the mutex with a call to **WaitForSingleObject**, and any subsequent threads attempting to gain access to it must wait. When a thread is finished using a mutex, it uses the **ReleaseMutex** function. A mutex can be created with the **CreateMutex** function. One process can get a handle to another process's mutex by using the **OpenMutex** call. Malware will commonly create a mutex and attempt to open an existing mutex with the same name to ensure that only one version of the malware is running at a time.

analysis view, a malware can use thread local storage (TLS) callbacks as a technique to interfere with normal debugger operation, trying to disrupt the program’s execution only if it is under the control of a debugger. Note that, Thread Local Storage (TLS) callback injection also involves manipulating pointers inside a portable executable (PE) to redirect a process to malicious code before reaching the code's legitimate entry point.

Although that on PEview and other basic static analysis tools the entrypoint address is being defined as the address: 0x484000, Hybrid analysis mentions as entrypoint 1 the address: 0x41a310 and as entrypoint 2 the address: 0x41a2c0.

Imports suspicious APIs

This suspicious APIs indicator contains a set of techniques that are mainly Anti-Debugging oriented. The following APIs functions [67] are being characterised as suspicious from HybridAnalysis:

GetUserNameA
RegOpenKeyExA
IsDebuggerPresent
VirtualProtect
GetProcAddress
GetComputerNameExA
GetModuleHandleA
FindFirstFileA
FindNextFileA
Sleep
WSAStartup

[67] **Function naming conventions**
When evaluating unfamiliar Windows functions, a few naming conventions are worth noting because they come up often and might confuse you if you don’t recognize them. For example, you will often encounter function names with an **Ex** suffix. When Microsoft updates a function and the new function is incompatible with the old one, Microsoft continues to support the old function. The new function is given the same name as the old function, with an added **Ex** suffix. Functions that have been significantly updated twice have **two Ex suffixes** in their names. Many functions that take strings as parameters include an **A** or a **W** at the end of their names. This letter does not appear in the documentation for the function; it simply indicates that the function accepts a string parameter and that there are two different versions of the function: one for **ASCII** strings and one for **Wide** character strings.

More specifically for each one:

- GetUserNameA

The GetUserNameA function retrieves the name of the user associated with the current thread. If the function succeeds, the return value is a nonzero value, and the variable pointed to by lpnSize contains the number of TCHARs copied to the buffer specified by lpBuffer, including the terminating null character.

- RegOpenKeyExA

Opens a handle to a registry key for querying-reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system and application setting information.

- IsDebuggerPresent

Determines whether the calling process is being debugged by a user-mode debugger. If the current process is running in the context of a debugger, the return value is nonzero. The simplest API function for detecting a debugger is IsDebuggerPresent. This function searches the Process Environment Block (PEB) structure for the field IsDebugged, which will return zero if you are not running in the context of a debugger or a nonzero value if a debugger is attached. We'll discuss the PEB structure in more detail in the next section.

- VirtualProtect

Changes the protection on a region of committed pages in the virtual address space of the calling process. By changing the memory protection to execute, read, and write access, the malware can modify the instructions. Then with another call to VirtualProtect at the end of the function restores the original memory-protection settings.

- GetProcAddress

Retrieves the address of a function in a DLL loaded into memory. Used to import functions from other DLLs in addition to the functions imported in the PE file header. Note that packed and obfuscated code will often include the function GetProcAddress, which could be used to load and gain access to additional functions.

- GetComputerNameExA

Retrieves a NetBIOS or DNS name associated with the local computer. The names are established at system startup, when the system reads them from the registry. If the function succeeds, the return value is a nonzero value.

- GetModuleHandleA

Used to obtain a handle to an already loaded module. Malware may use GetModuleHandle to locate and modify code in a loaded module or to search for a good location to inject code.

- FindFirstFileA and FindNextFileA

These functions are being used to search through a directory and enumerate the filesystem. Them combination also show that the program searches the filesystem for files and it can open and modify files. At the moment it is unsure what the program is searching for.

- Sleep

The Sleep function suspends the execution of the current thread until the time-out interval elapses and does not return a value. Sleep function takes a single parameter containing the number of milliseconds to sleep. It pushes 0xEA60 on the stack, which corresponds to sleeping for one minute (60,000 milliseconds).

- WSAStartup

The WSAStartup function initiates use of the Winsock DLL by a process. If successful, the WSAStartup function returns zero, otherwise, it returns one of some listed error codes.

PE file contains unusual section name

As we mentioned in Static Analysis, the unusual sections named ".eh_fram" and ".CRT" demonstrates that the PE file is written in C++.

Informative indicators

Anti-Reverse Engineering

This indicator, **SetUnhandledExceptionFilter** function, is often used by malwares as an Anti-Reverse Engineering technique, that contains ability to register a top-level exception handler.

- SetUnhandledExceptionFilter@KERNEL32.DLL at address 0x401030

```
@401000: push ebx
@401001: sub esp, 38h
@401004: mov eax, dword ptr [00476280h]
@401009: test eax, eax
@40100b: je 00401029h
@40100d: mov dword ptr [esp+08h], 00000000h
@401015: mov dword ptr [esp+04h], 00000002h
@40101d: mov dword ptr [esp], 00000000h
@401024: call eax
@401026: sub esp, 0Ch
@401029: mov dword ptr [esp], 00401110h
@401030: call 004242A4h ;SetUnhandledExceptionFilter@KERNEL32.DLL
@401035: sub esp, 04h
@401038: call 0041A3B0h
@40103d: call 0041A490h
@401042: lea eax, dword ptr [esp+2Ch]
@401046: mov dword ptr [esp+10h], eax
@40104a: mov eax, dword ptr [004720C0h]
@40104f: mov dword ptr [esp+04h], 0047C000h
@401057: mov dword ptr [esp], 0047C004h
@40105e: mov dword ptr [esp+2Ch], 00000000h
@401066: mov dword ptr [esp+0Ch], eax
@40106a: lea eax, dword ptr [esp+28h]
@40106e: mov dword ptr [esp+08h], eax
@401072: call 004240A0h ;__getmainargs@MSVCRT.DLL
@401077: mov eax, dword ptr [00482060h]
@40107c: test eax, eax
@40107e: je 004010C2h
@401080: mov ebx, dword ptr [00483364h]
```

Figure 44: Assembly: SetUnhandledExceptionFilter function call

- SetUnhandledExceptionFilter@KERNEL32.DLL at address 0x4014FB

```
@4014ee: push ebp
@4014ef: mov ebp, esp
@4014f1: sub esp, 18h
@4014f4: mov dword ptr [esp], 004014ACh
@4014fb: call 004242A4h ;SetUnhandledExceptionFilter@KERNEL32.DLL
@401500: sub esp, 04h
@401503: leave
@401504: ret
```

Figure 45: Assembly: SetUnhandledExceptionFilter function call 2

Another informative indicator for Anti-Reverse Engineering is that the PE file contains **zero-size section**. Specifically, the raw size of *.bss* [68] is zero. The section .bss is a data segment there global and static uninitialized variables are being stored.

Network Related

The HybridAnalysis has found that a potential URL in binary exists. Specifically, using heuristic match on the string: "tL<EtH<.tD", several known and analyzed malwares, are using this string also. At the moment, we can not resolve this string, but it is for sure encoded.

We could analyze more the imports of the under analysis executable, but this is a static procedure for the other section.

4.2. Running Malware

Basic dynamic analysis techniques demand to run the malware. Although it is usually simple enough to run executable malware by double-clicking the executable or running the file

[68] BSS (from Block Started by Symbol): The uninitialized data are rarely found in executables created with recent linkers. Instead, the VirtualSize of the executable's .data section is expanded to make enough room for uninitialized data. In C, statically-allocated objects without an explicit initializer are initialized to zero (for arithmetic types) or a null pointer (for pointer types). Implementations of C typically represent zero values and null pointer values using a bit pattern consisting solely of zero-valued bits (though this is not required by the C standard). Hence, the BSS segment typically includes all uninitialized objects (both variables and constants) declared at file scope (i.e., outside any function) source: https://en.wikipedia.org/wiki/.bss#BSS_in_C

from the command line, it has been proven that is trickier to run and activate a malware. Note that all execution of the malware will be done with administrator privileges in order to avoid any privilege conflict.

4.2.1. Hands on Basic Dynamic - Behavioral Analysis Tools

The tools for basic dynamic analysis should be used in concert to maximize the amount of information gleaned. The toolset includes the followings:

1. Setting up your virtual network as the VMware Setup Appendix describes.
2. Examine with Process Explorer and its open source alternative Process Hacker.
3. Running Process Monitor and setting a filter on the malware executable PID and clearing out all events just before running.
4. Gathering a first snapshot of the registry using Regshot.
5. Setting up network traffic logging using Wireshark.

Again, it should be warned that testing malware dynamically should be done ensuring the host computer and networks, as discussed in the previous chapter.

Process Explorer

The Process Explorer, free from Microsoft, is an extremely powerful task manager that should be running when you are performing dynamic analysis. It can provide valuable insight into the processes currently running on a system, to list active processes, DLLs loaded by a process, various process properties, overall system information, to kill a process, log out users, and launch and validate processes.

Process Explorer monitors the processes running on a system and shows them in a tree structure that displays child and parent relationships. The user can view five columns: Process (the process name), PID (the process identifier), CPU (CPU usage), Description, and Company Name, with services being highlighted in pink, processes in blue, new processes in green, and terminated processes in red. Green and red highlights are temporary, and are removed after the process has started or terminated.

They key point with Process Explorer is when analyzing malware to look for changes or new processes, in order to investigate them thoroughly.

On the following screenshot the malware is running, due to the continuously high CPU usage.

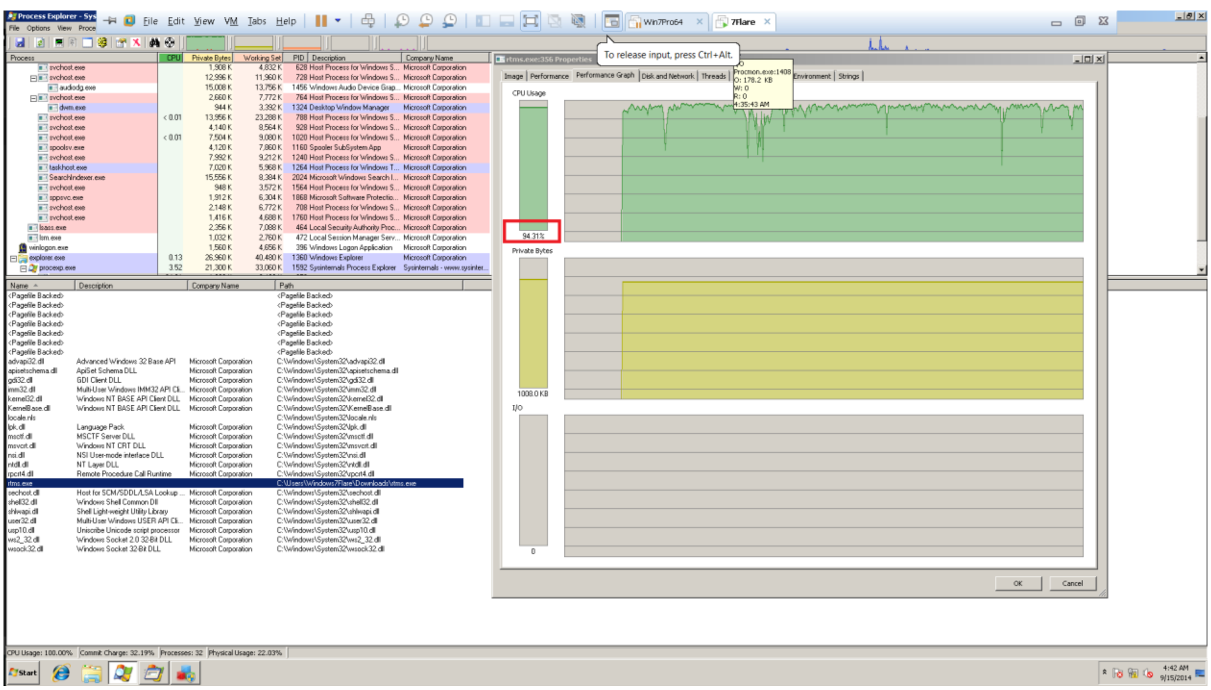


Figure 46: Process Explorer: malware's Properties - Performance Graph - CPU usage

The following screenshot shows that the malware has debugging privilege is enables, as long as we run the malware as Administrators. SeImpersonatePrivilage is also enabled by default.[69]

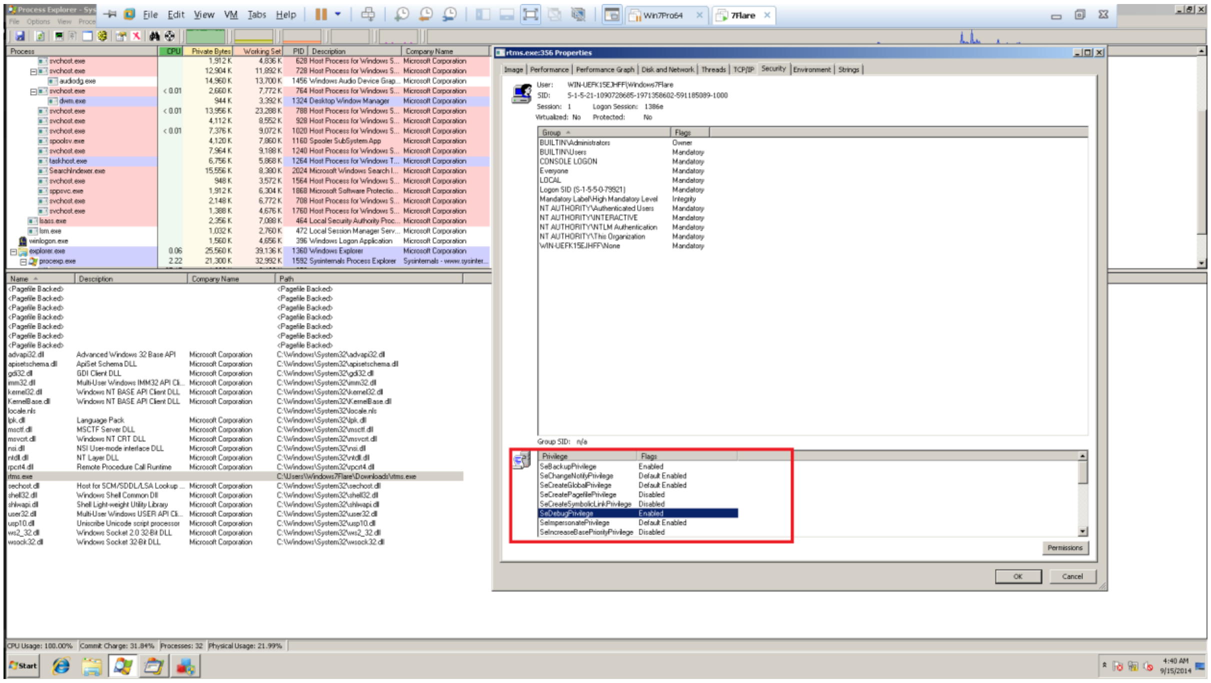


Figure 47: Process Explorer: malware's Properties - Security - Permissions

The following screenshot shows that the malware has a specific stack already built. Specifically, on the thread were the malware is being executed, the stack of this thread contains the malicious code which is being built during the execution. The last function that has been added to the thread's stack is RtlInitializeExceptionChain[70] from the known ntdll.dll. This assumes that

---

[69] When you assign the "Impersonate a client after authentication" user right to a user, you permit programs that run on behalf of that user to impersonate a client. This security setting helps to prevent unauthorized servers from impersonating clients that connect to it through methods such as remote procedure calls (RPC) or named pipes. Source: https://support.microsoft.com/en-us/help/821546/overview-of-the-impersonate-a-client-after-authentication-and-the-crea

[70] RtlInitializeExceptionChain is an internal function in the Run-Time Library, a collection of kernel-mode support functions used by kernel-mode drivers and the OS itself. It's kind of the kernel-mode version of the C run-time library. If your application is 32-bit and you're profiling it on a 64-bit machine, profiling it on a 32-bit machine or building a 64-bit version will probably move RtlInitializeExceptionChain out of the top 10 list since it's always used in thunking.

an Exception error is taken place and the malware does not actually executes the whole of its execution procedure.

Moreover, the 4th place of the thread's stack the KeUpdateSystemTime[71] function exist. This function does the time-check and we have successfully pass it as long as the procedure does not stop there.

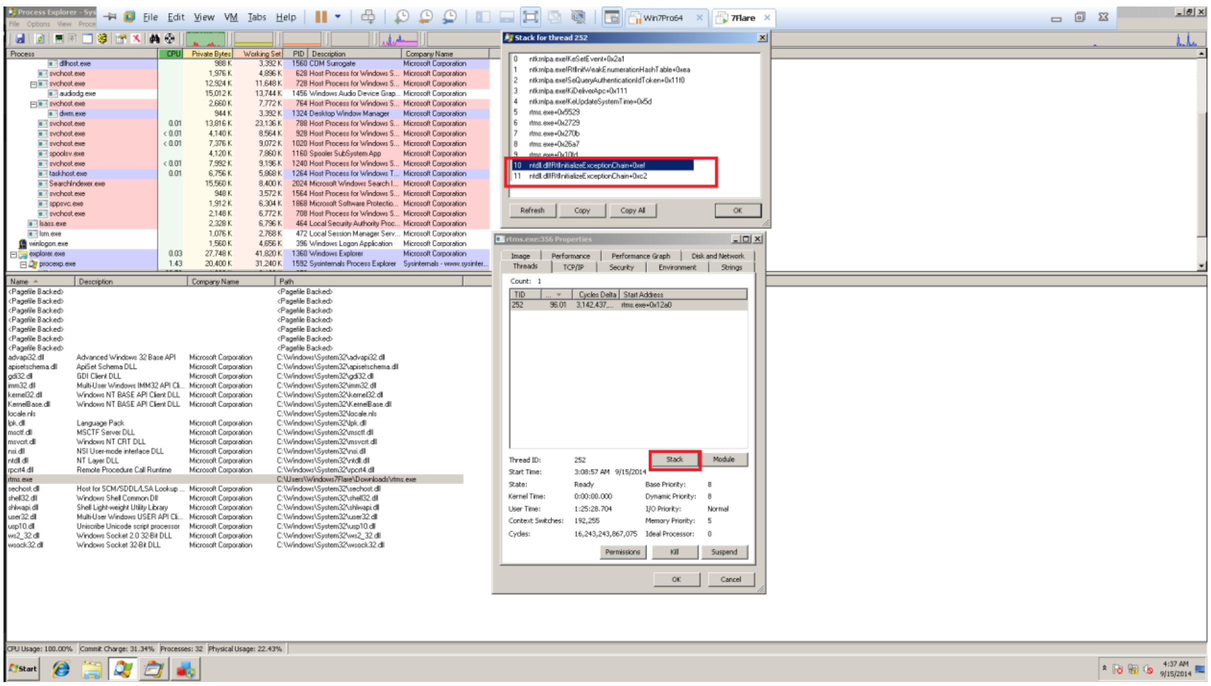


Figure 48: Process Explorer: malware's Properties - Threads - Stack - information on current stack

The following screenshot shows that the malware has 1 hour and 27 minutes runtime in User-land. This is a lot of time using the maximum of CPU usage, as it is being previews above, without any actual behavior from the malware or its infection. It should be assumed that the malware does not execute its main procedure but is idling on purpose.

[71] KeUpdateSystemTime routine is executed on a single processor in the processor complex. Its function is to update the system time and to check to determine if a timer has expired.

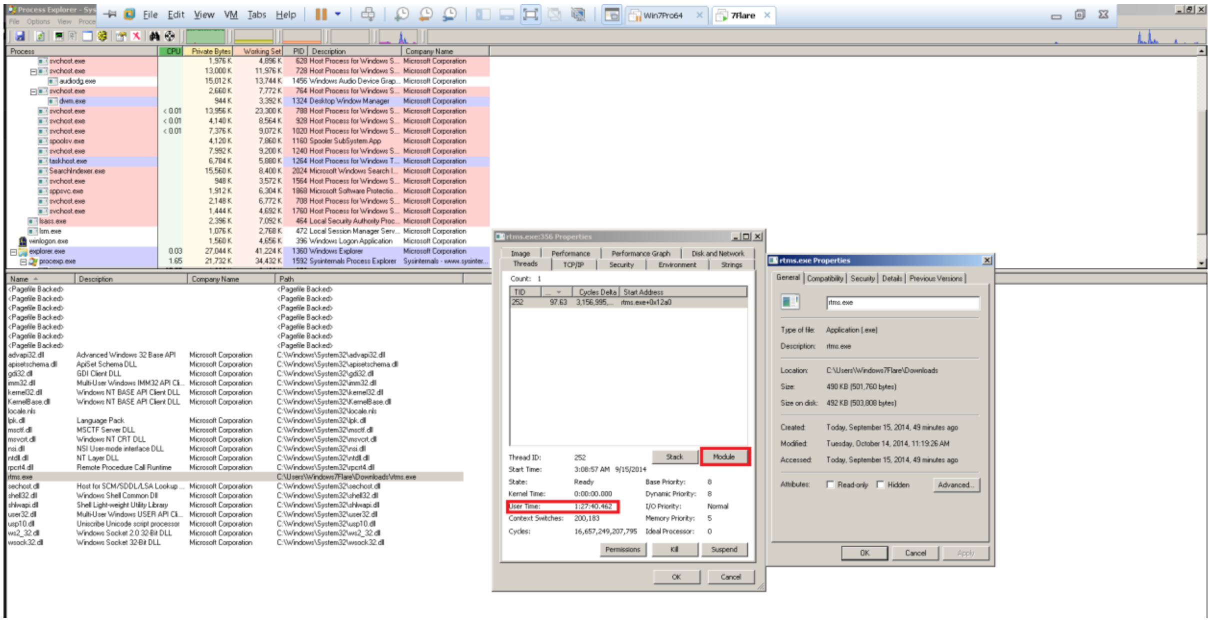


Figure 49: Process Explorer: malware's Properties - Threads - Module - General details for the malicious file

The following screenshot shows that the malware's executable file does not have any metadata, which does not help analyzing it.

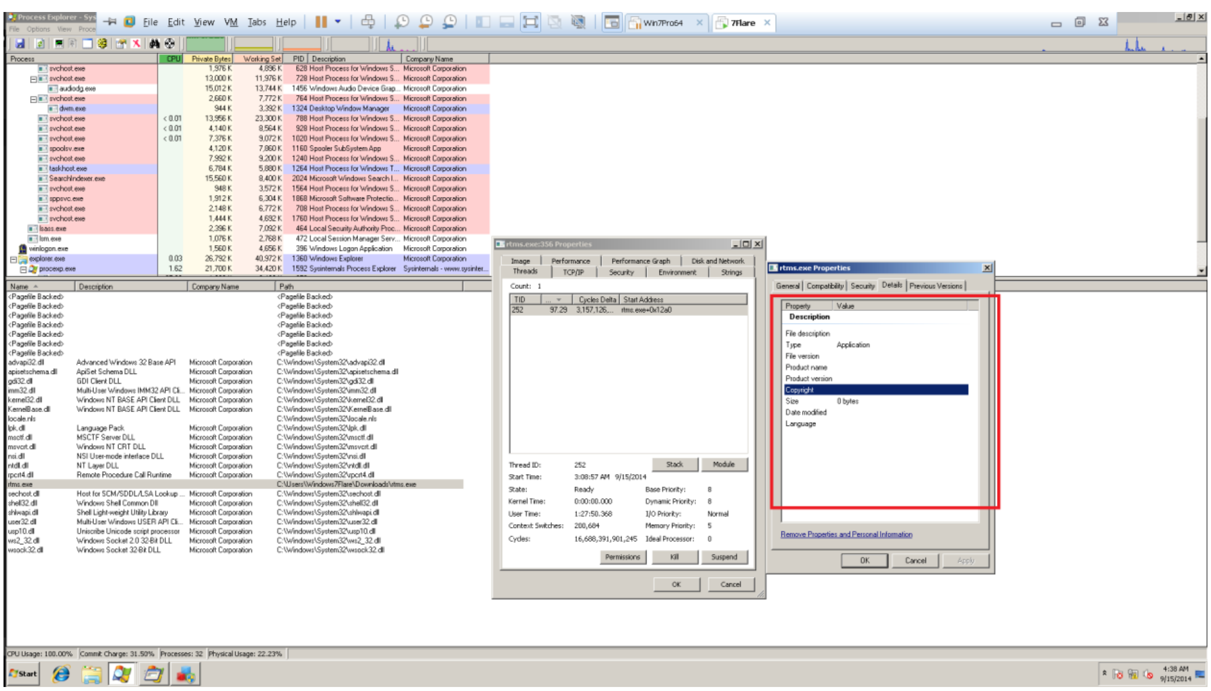


Figure 50: Process Explorer: malware's Properties - Threads - Module - no metadata for the malicious file

The following screenshot shows that the malware has not any network activity.

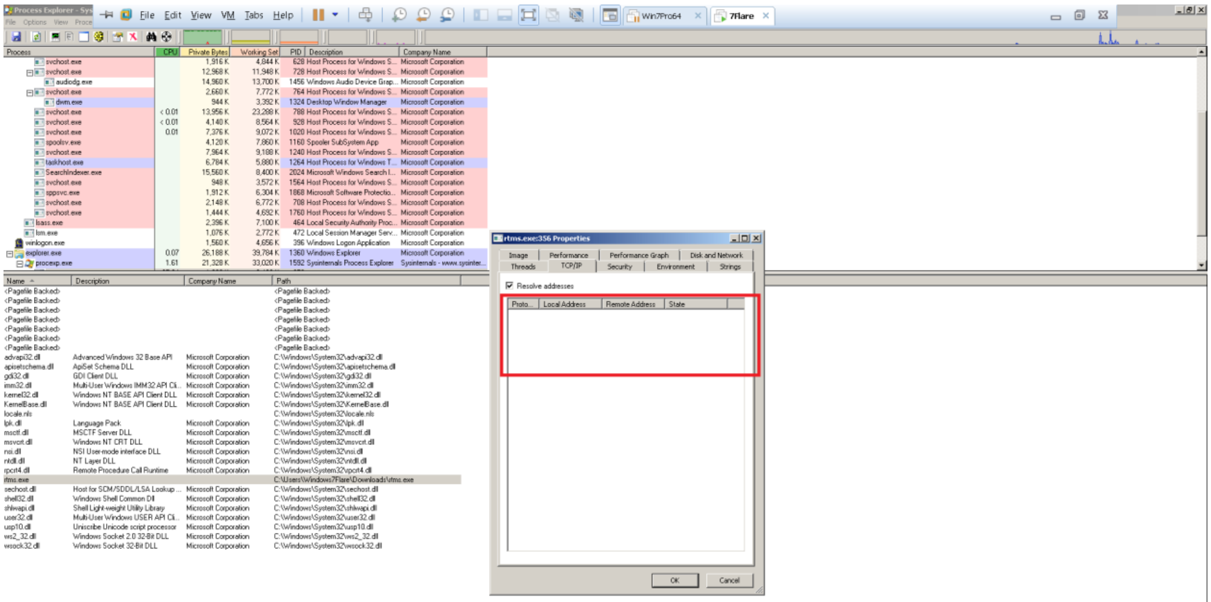


Figure 51: Process Explorer: malware's Properties - TCP/IP - no network activity

The following screenshot shows the environment where the malware is being executed.

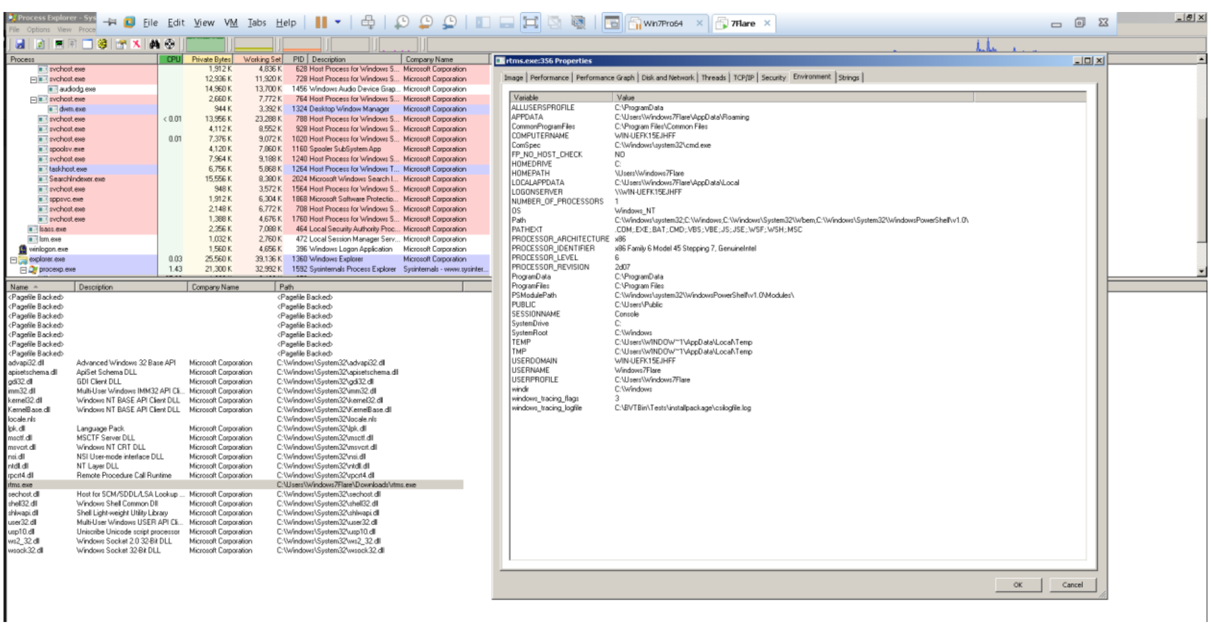


Figure 52: Process Explorer: malware's Properties - Environment

As a result, information gathering from Process Explorer was successful, but they key point to reveal a new, related to the malware, process did not happen.

4.2.2. Comparing the image and memory Strings

One way to recognize process replacement is to use the Strings tab in the Process Properties window to compare the strings contained in the disk executable (image) against the strings in memory for that same executable running in memory. Both options exported in text file and being compares with the WinMerge[72] Windows Tool.

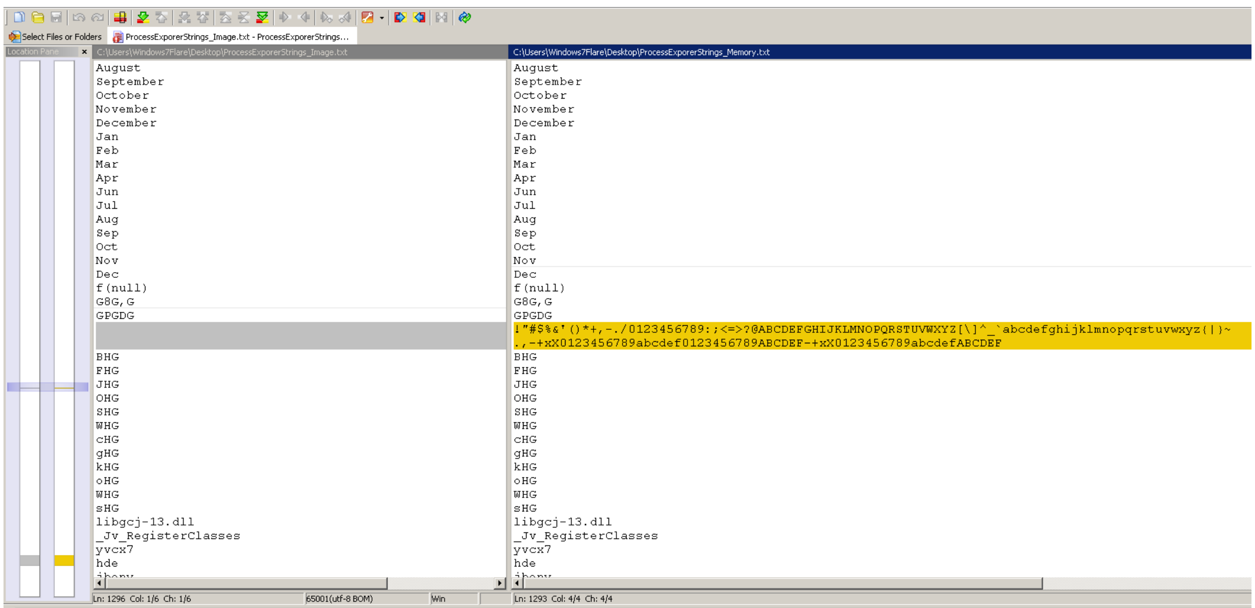


Figure 53: WinMerge Strings txt files comparison from Binary's Image and Memory's executable

[72] WinMerge is a Windows tool for visual difference display and merging, for both files and directories. It is highly useful for determining what has changed between file versions, and then merging those changes. Side-by-side line difference and highlights differences inside lines. A file map shows the overall file differences in a location pane. The user interface is translated into several languages.

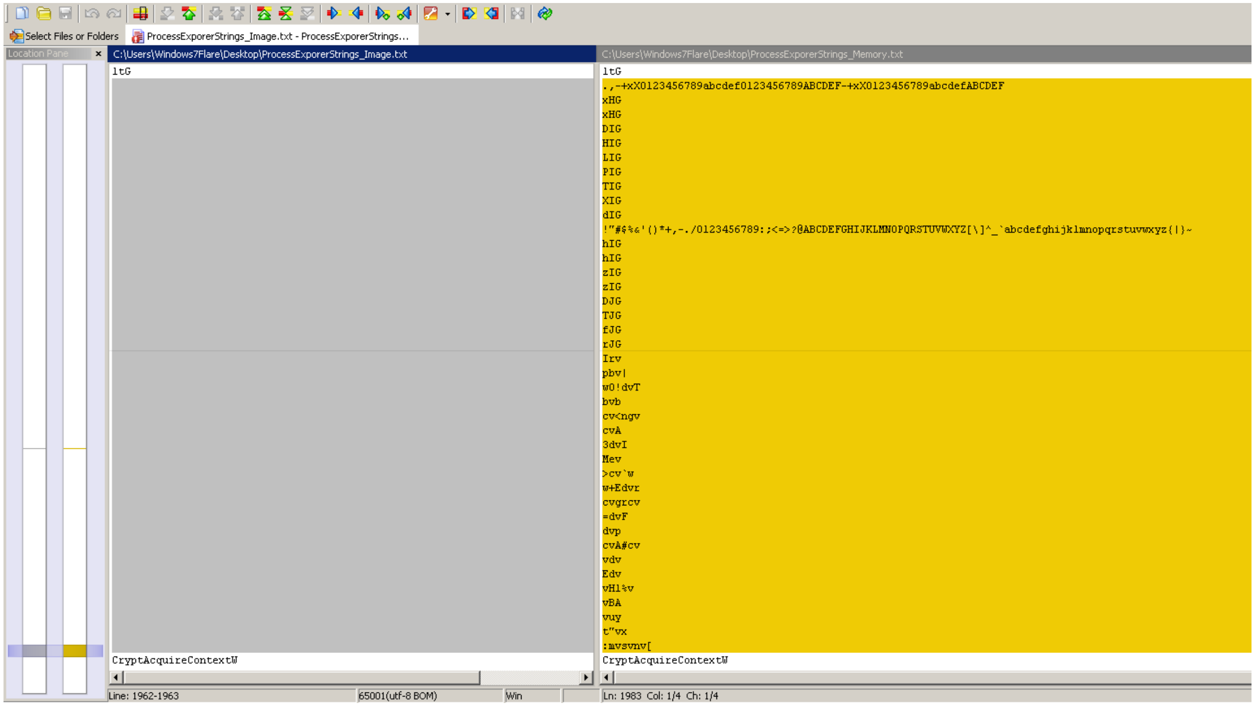


Figure 54: WinMerge Strings txt files comparison from Binary's Image and Memory's executable-2

The comparison of the two string listings did not drastically different, so it is sure that process replacement did not occurred. On the other hand, belong the highlighted strings three Base64 alphabets are being revealed. The encoding scheme and the further analysis of Base64 alphabet will be presented on the Static code Analysis section.

4.2.3. Examine with Process Hacker

An open source alternative of Process Explorer is Process Hacker that includes detailed network activity, but in this case, no additional information was usable.

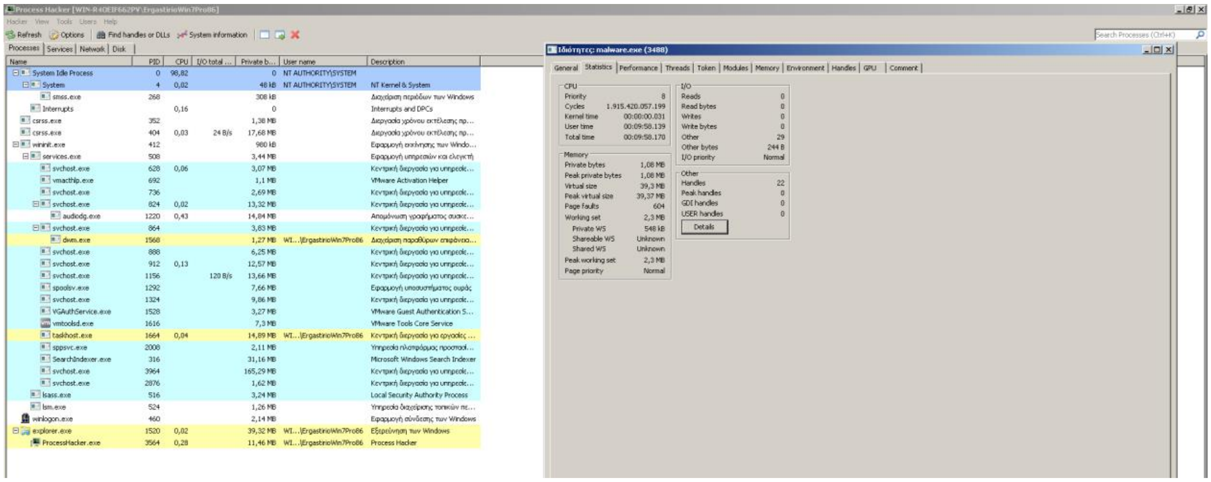


Figure 55: Process Hacker: malware.exe's Statistics on Properties

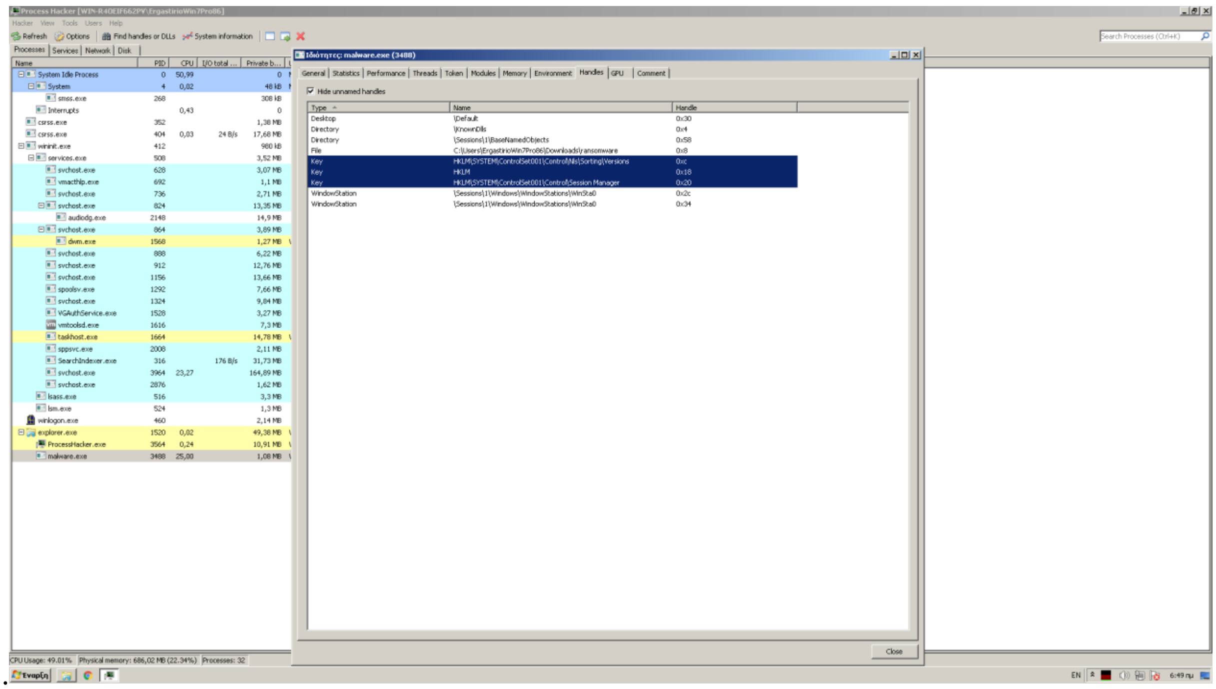


Figure 56: Process Hacker: malware.exe's Handles on Properties.

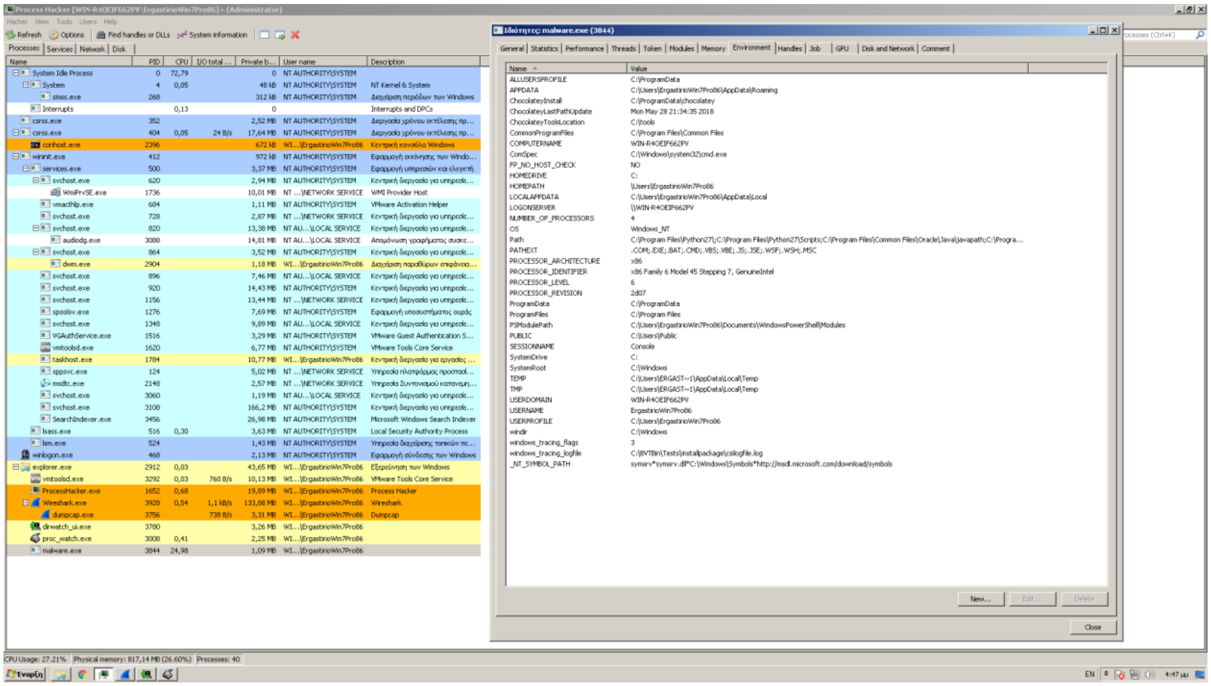


Figure 57: Process Hacker: malware.exe's Environment on Properties.

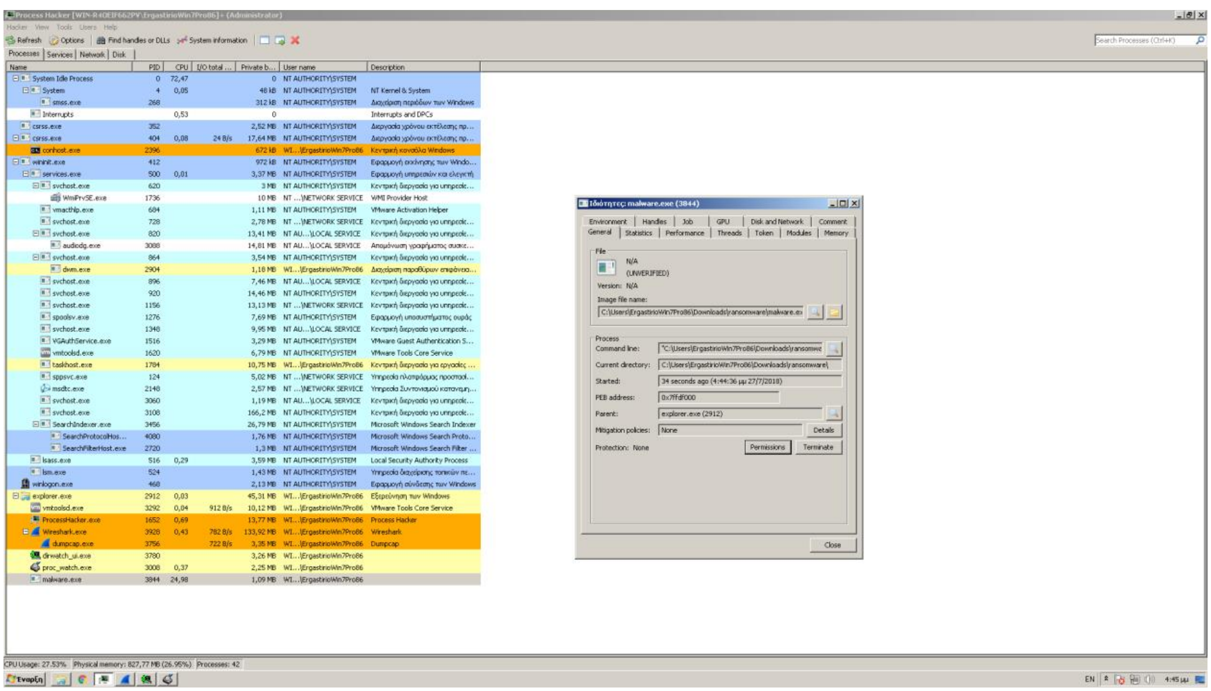


Figure 58: Process Hacker: malware.exe's General Properties. (PEB address 0x7ffdf000)

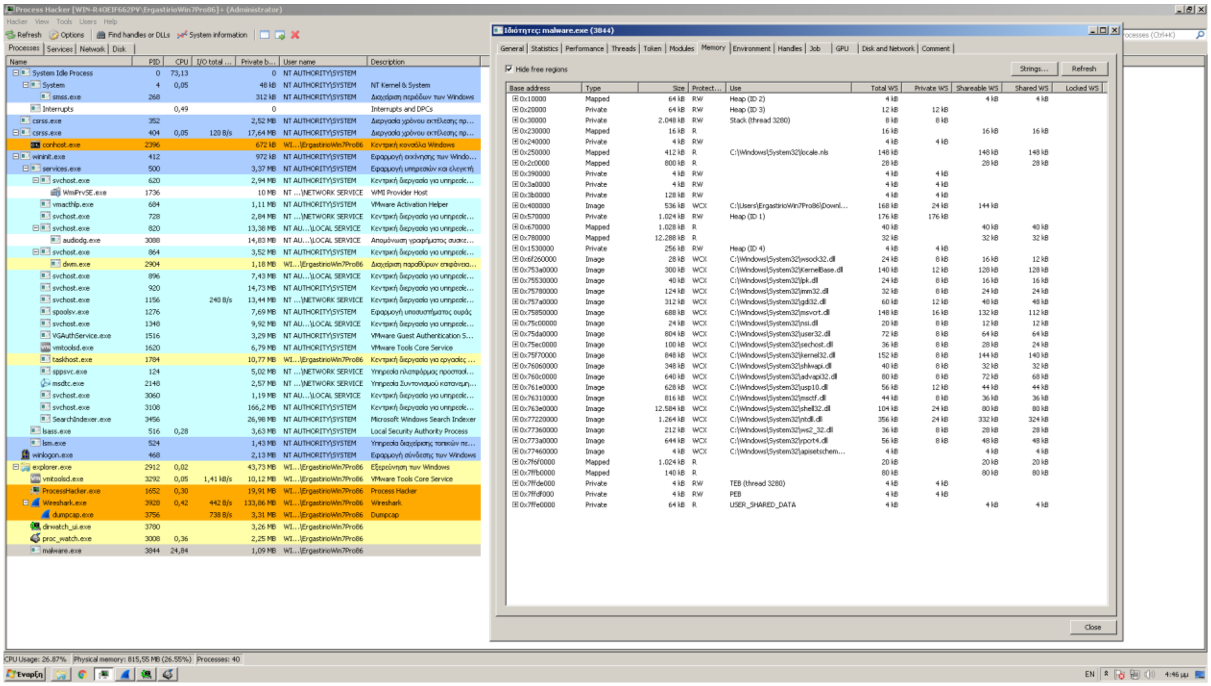


Figure 59: Process Hacker: malware.exe's Memory on Properties.

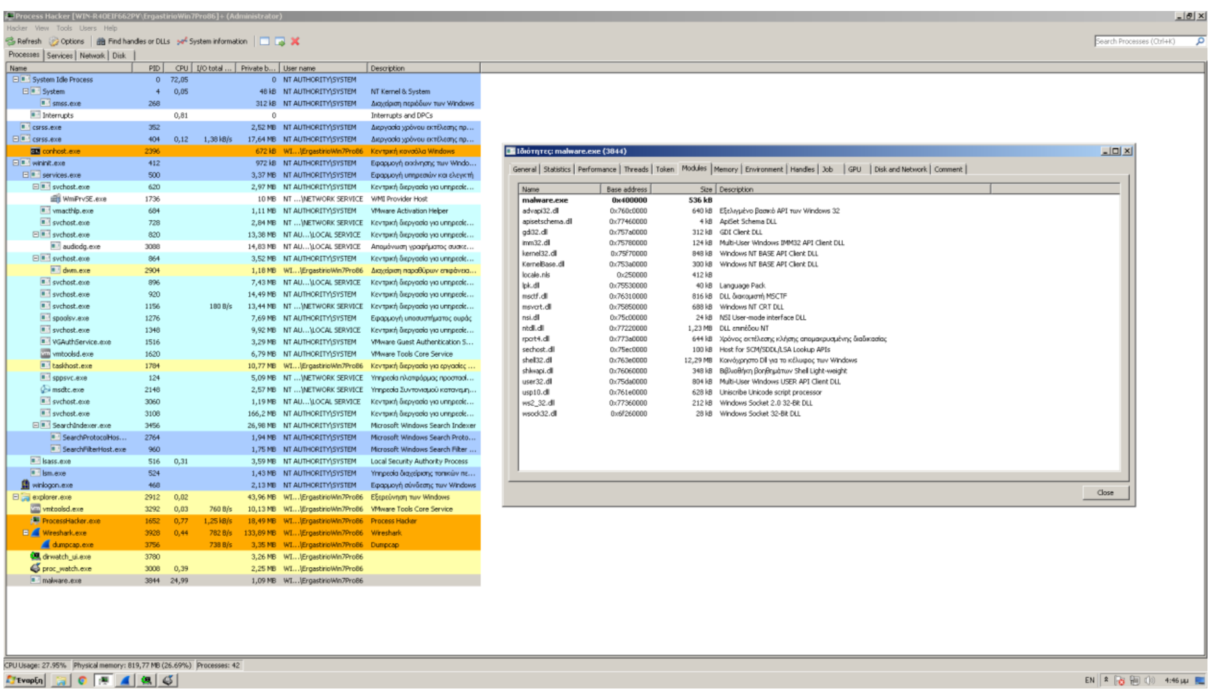


Figure 60: Process Hacker: malware.exe's Modules on Properties.

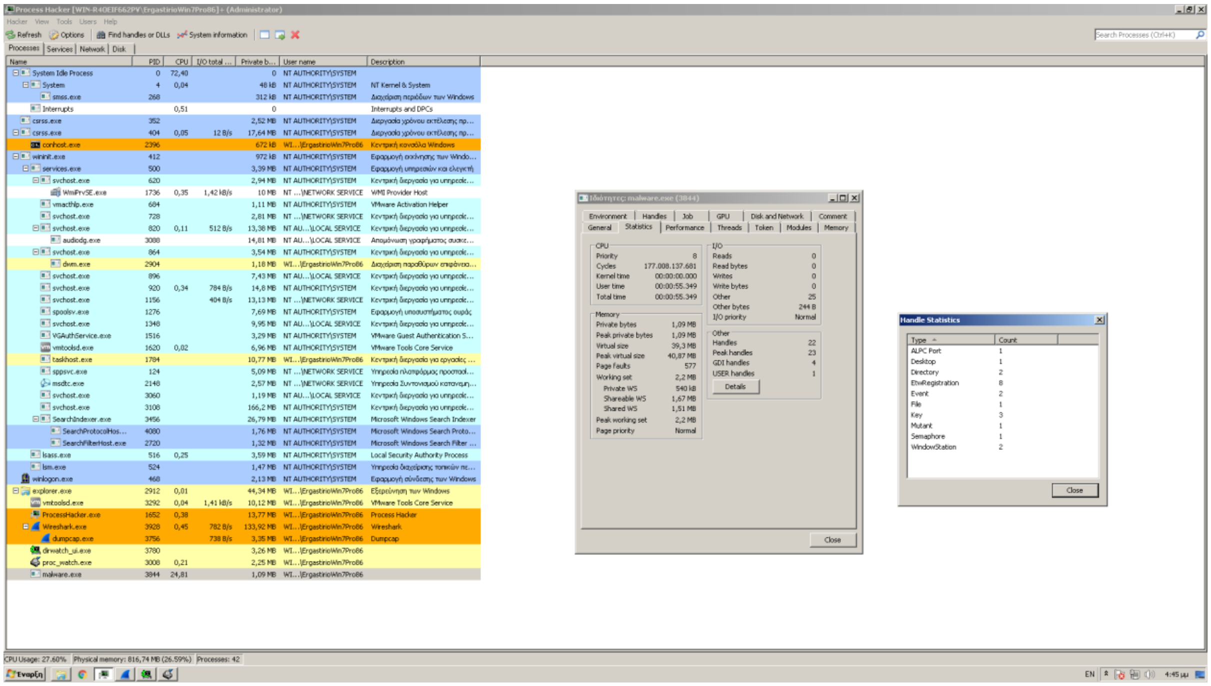


Figure 61: Process Hacker: malware.exe's Handle's Statistics on Properties.

As a result, some information was gathered from Process Hacker in compare with Process Explorer. On the other hand, no new process appeared but they key point to reveal a new, related to the malware, process did not happen.

4.2.4. Monitoring with Process Monitor

Process Monitor, or procmon, is an advanced monitoring tool for Windows that provides a way to monitor certain registry, file system, network, process, and thread activity[73]. It combines and enhances the functionality of two legacy tools: FileMon and RegMon.

[73] Although procmon captures a lot of data, it doesn't capture everything. For example, it can miss the device driver activity of a user-mode component talking to a rootkit via device I/O controls, as well as certain GUI calls, such as SetWindowsHookEx. In addition, it should not be used for logging network activity, because it does not work consistently across Microsoft Windows versions.

Procmon monitors all system calls and because many system calls exist on a Windows machine (more than 50,000 events a minute), procmon uses RAM to log events. Keep in mind that Procmon can crash a virtual machine using all available memory.

Before using procmon for analysis, first clear all currently captured events to remove irrelevant data by choosing Edit/Clear Display. Next, run the rtms.exe (malware) as Administrator, with capture turned on. Then filter the results showing only the PID of rtms.exe, in screenshot's case the PID is 3000.

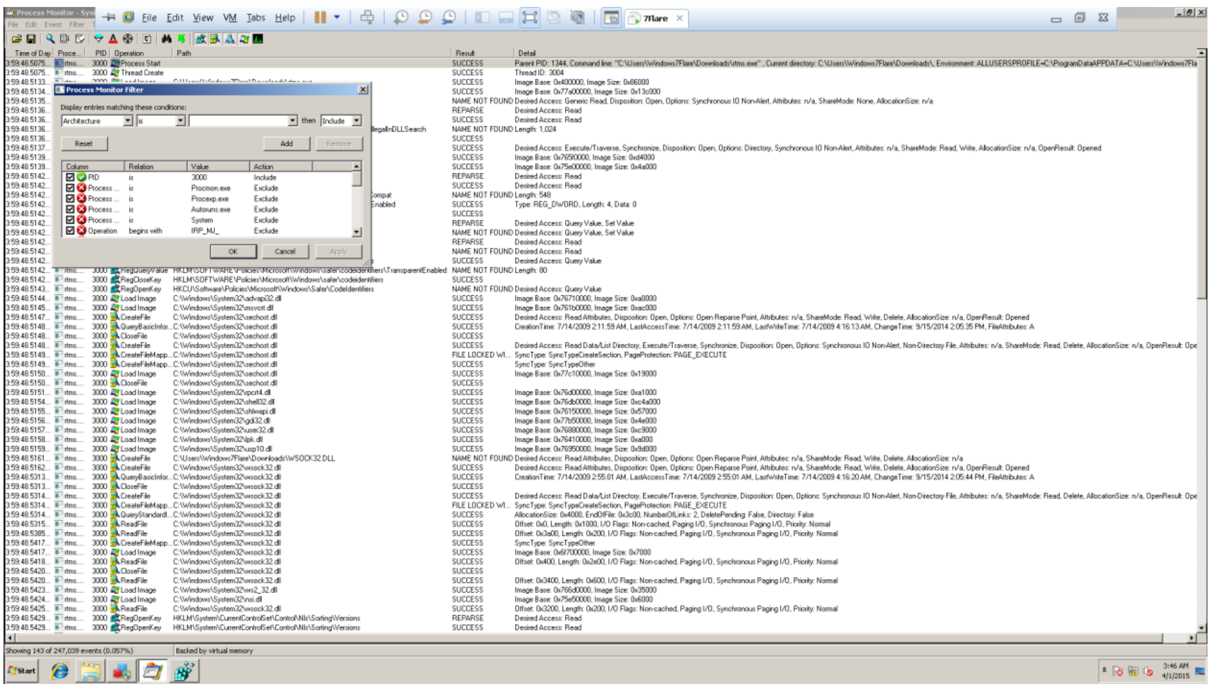


Figure 62: Process Monitor: Filter apply in PID of the under analysis malware (rtms.exe)

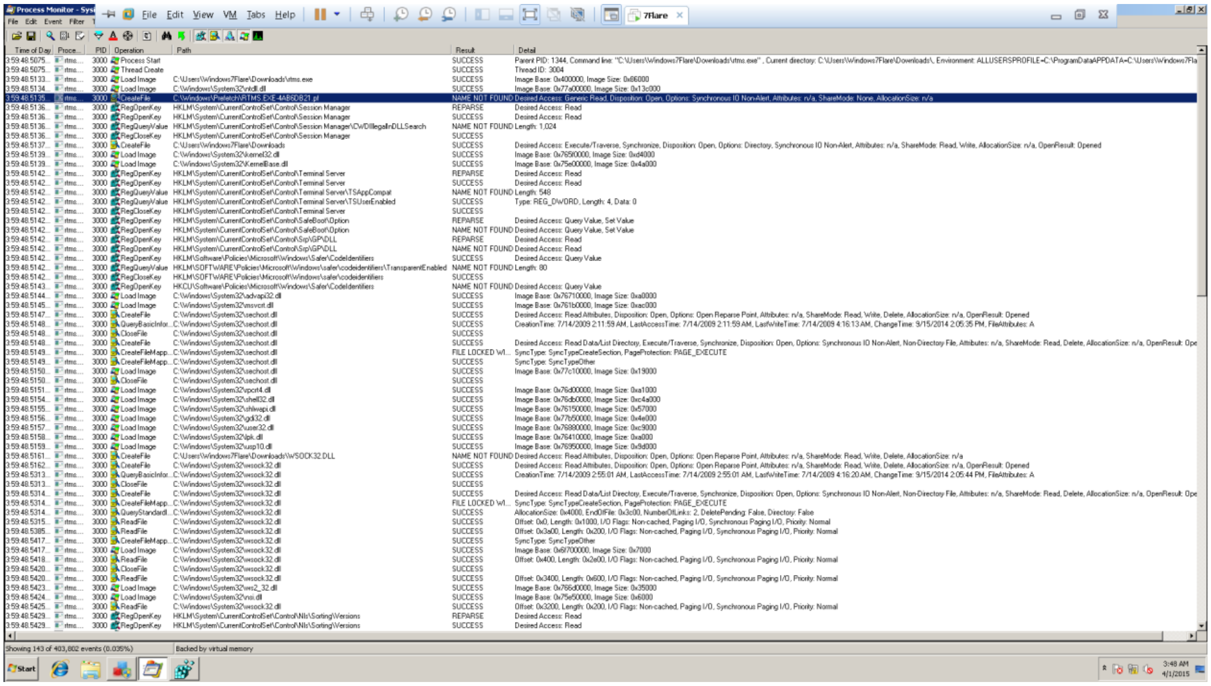


Figure 63: Process Monitor: List1 of all events

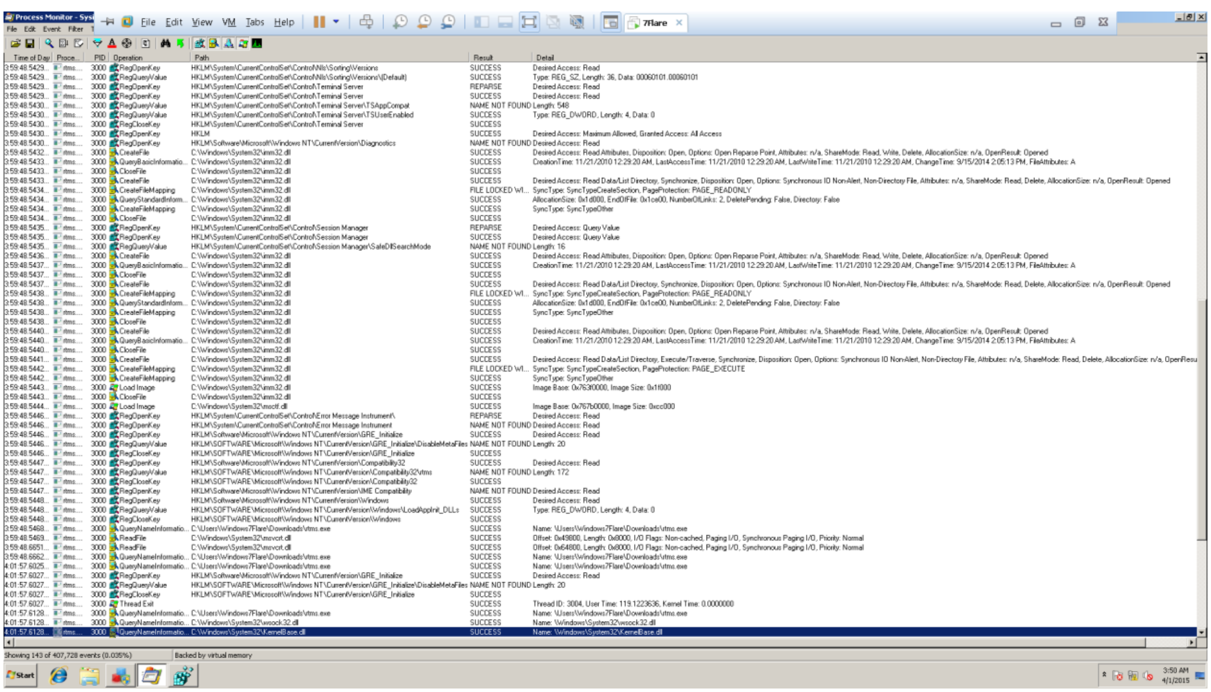


Figure 64: Process Monitor: List2 of all events

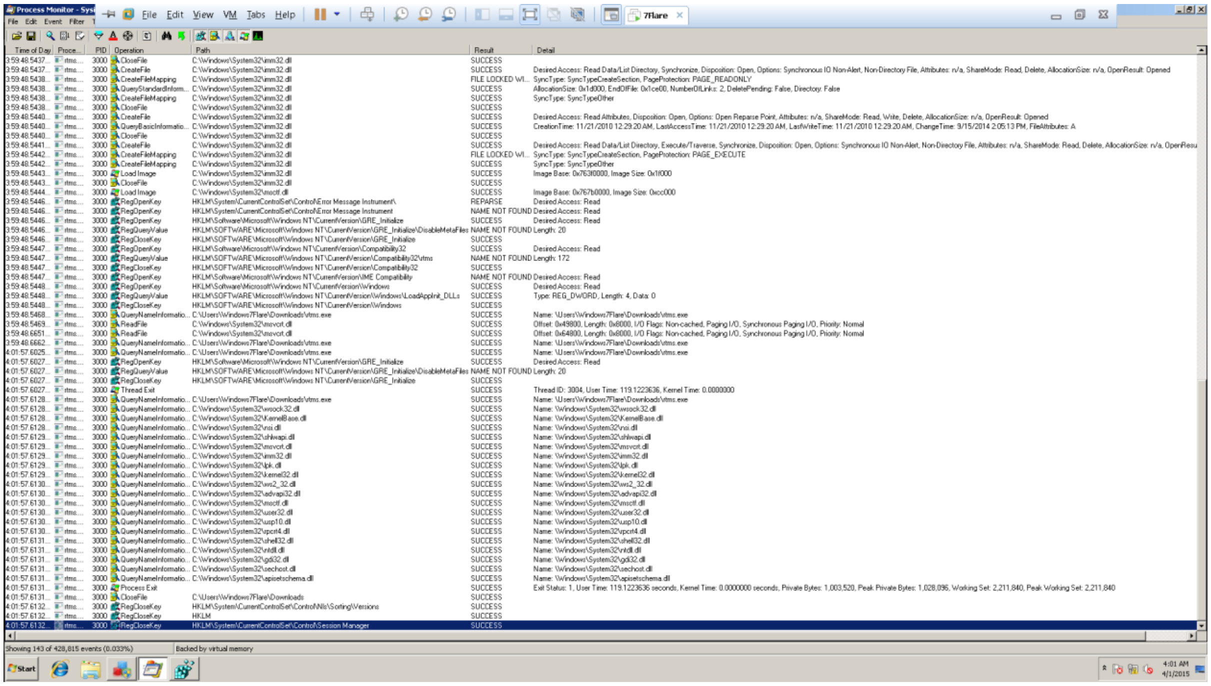


Figure 65: Process Monitor: List3 of all events

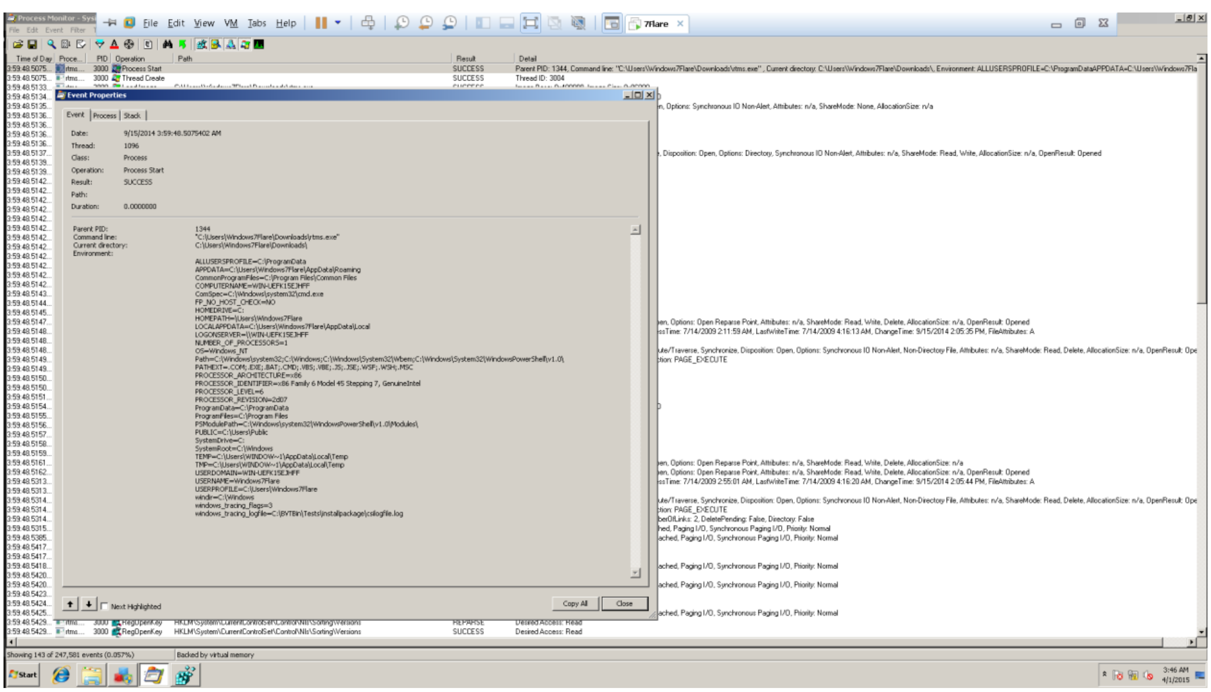


Figure 66: Process Monitor: Process start event - Event Properties - General

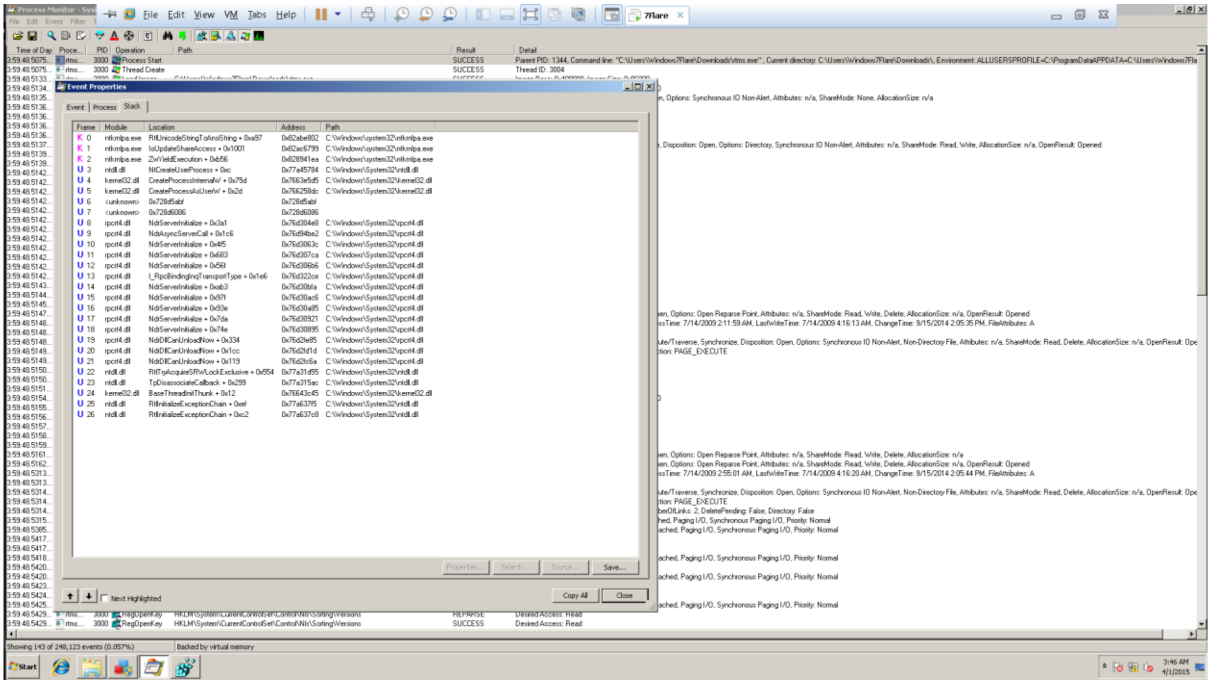


Figure 67: Process Monitor: Process start event - Event Properties - Stack

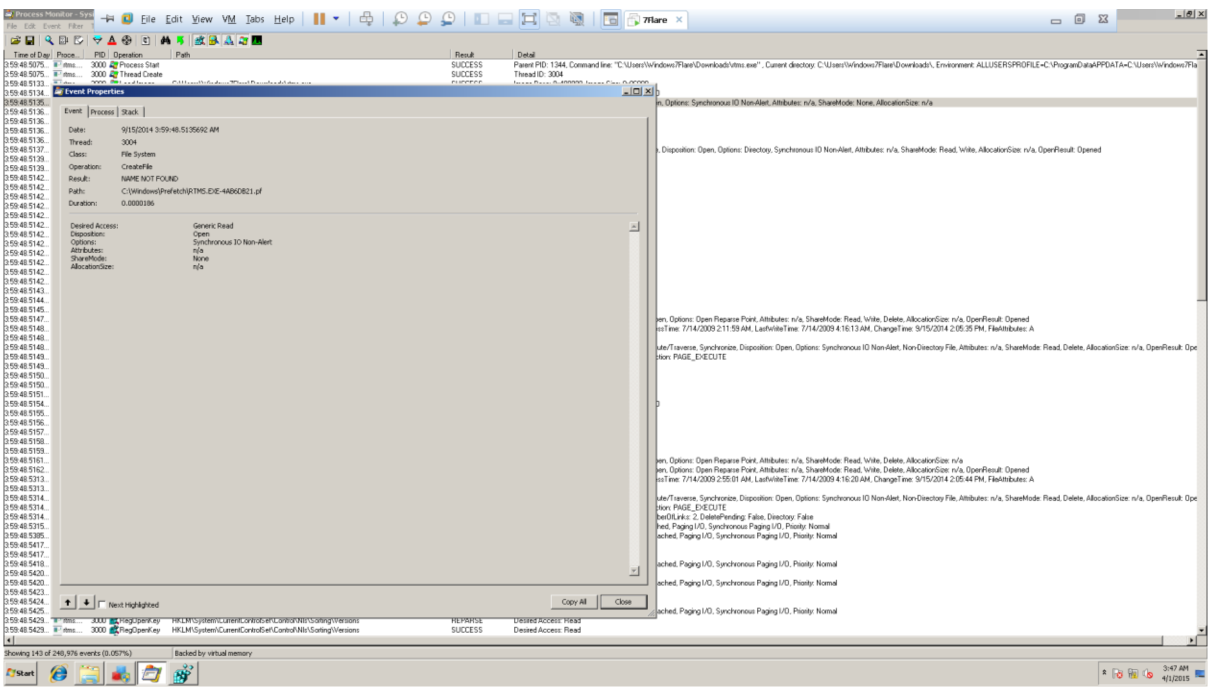


Figure 68: Process Monitor: Create File event - Event Properties

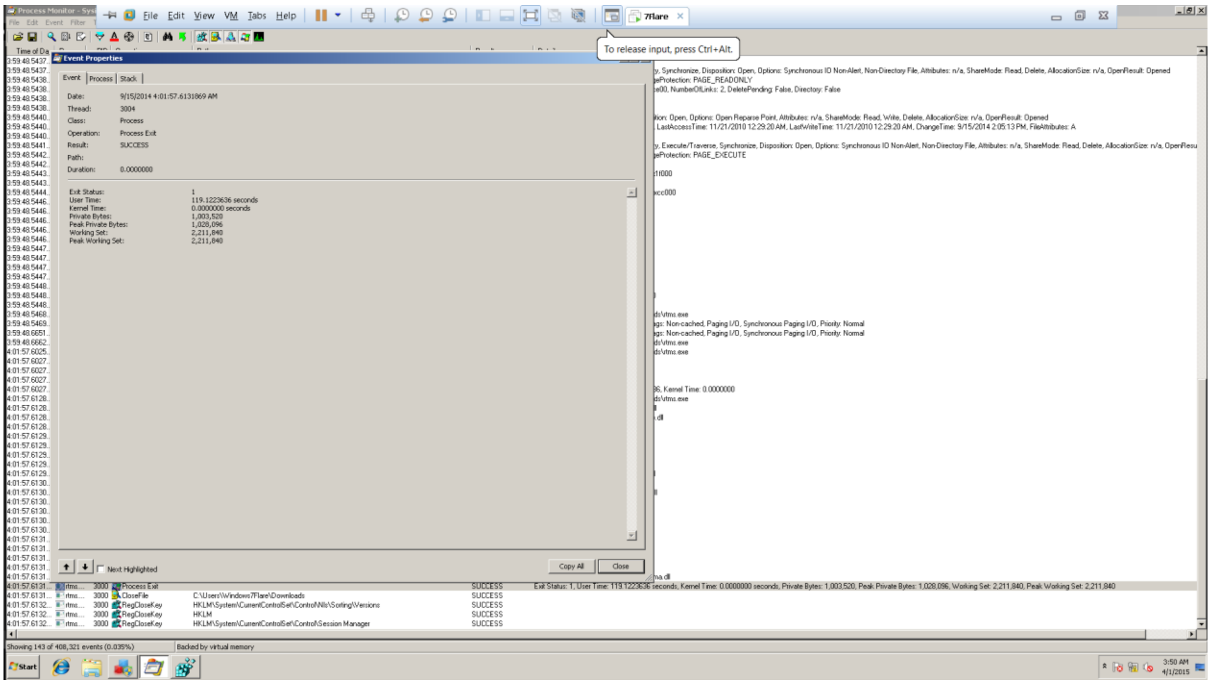


Figure 69: Process Monitor: Process Exit

- Registry: By examining registry operations, it is unsure how malware installs itself in the system.
- File system: Exploring file system interaction shows all files that the malware creates or configuration files it uses. There are files created that were not useful at this point of analysis.
- Process activity: Investigating process activity, the malware did not spawn any additional processes.
- Network: Identifying network connection, which is in an isolated subnet, did not show any communication in ports on which malwares usually listening.

4.2.5. Regshot

Regshot is an open source registry comparison tool that allows you to take and compare two registry snapshots. To use Regshot for malware analysis, simply take the first shot by clicking the 1st Shot button, and then run the malware and wait for it to finish making any system changes. Next, take the second shot by clicking the 2nd Shot button. Then, click the Compare button to compare the two snapshots displays a subset of the results generated by Regshot during malware analysis.

Registry snapshots were taken before and after running the malware rtms.exe. As you can see 1875 changes occurred in registry. The amount of noise is huge in these results.

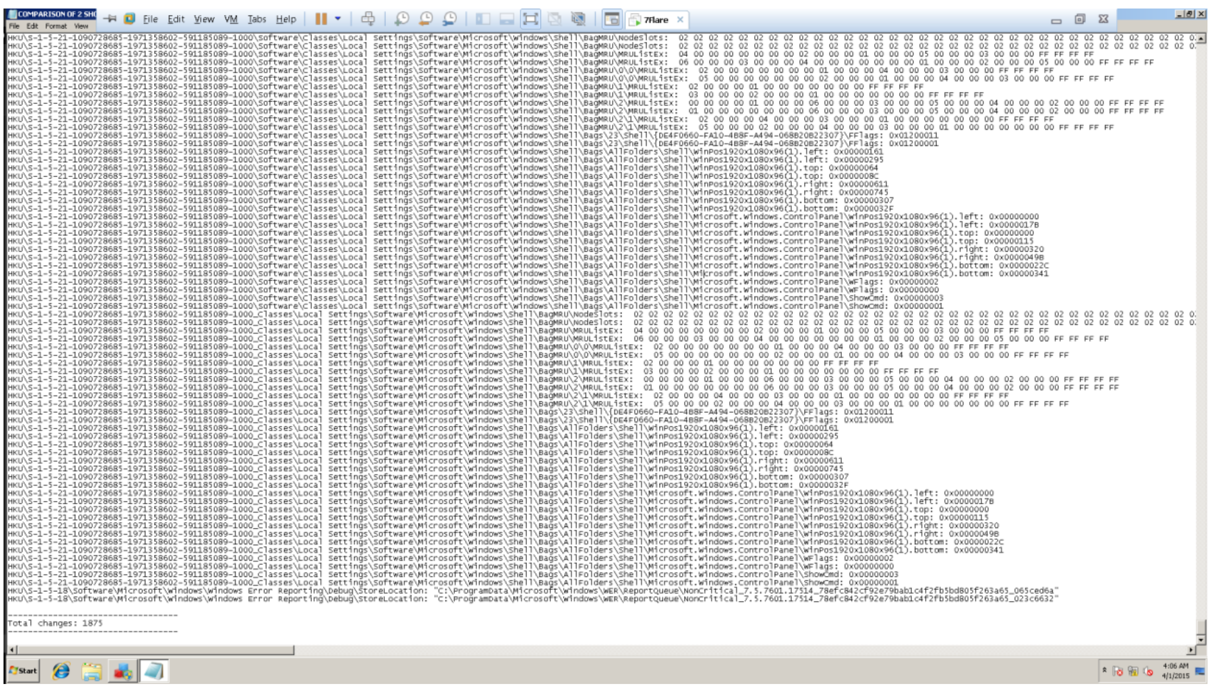


Figure 70:Regshot: comparison results of registry snapshots before and after rtms.exe run

4.2.6. Basic Dynamic Analysis is not enough

As a conclusion in basic dynamic analysis, it should be noticed that many tries and steps back had been done. The malware was renamed to random names in case there was a naming detection technique. In addition, many changes had been done on the VMware's configuration file - VMX and inside VME OS settings, in order to deflect any tools detection.

As a last try, the ransomware was installed in a Windows 7 SP1 x86 in bare metal machine without Virtualization Technologies and Debuggers - Disassemblers installed, in order to prevent any detection and even then, the malware did not execute all its procedures. After a long research on the faulty side of the malware, the problem was detected in the IDT instruction behavior of Intel i3 processor.

A spoil from the advanced malware analysis is being done at this point, but it should be clarified why basic analysis did not and would not work in this case.

Joanna Rutkowska came across this strange behavior of SIDT instruction a few years ago on her RedPill paper, when Joanna Rutkowska was testing “Suckit” rootkit on VMWare. Joanna Rutkowska noticed that it failed to load on VMWare whereas it seemed to work fine on the same distribution ran outside VM. After spending many hours Joanna Rutkowska figured out that the problematic instruction was actually SIDT, which was used by “Suckit” to get the address of the IDT table, and to hook its 0x80 entry through /dev/kmem device.

However, Joanna Rutkowska was not the first one who discovered this trick. Shortly after her adventure with “Suckit” Joanna Rutkowska found a very good USENIX paper about problems when implementing Virtual Machines on Intel processors, discussing of course SIDT problem, as well as many others.

So now, here is the simple code, written in C, which should compile on any all Intel based OS. Just in case you don't have the C compiler for Windows, there is also a binary version attached.[74]

[74] Paragraph’s source URL: https://securiteam.com/securityreviews/6Z00H20BQS/

On the other hand, Oliver Schneider's paper conclusion (for the conclusions drawn from observation of RedPill results being wrong)[75], says that among the others, RedPill Technique does not take into account multiprocessor machines. As a result, the under analysis malware detects all the multiprocessor machines, the bare metal ones, as Virtual Environments!

[75] RedPill getting colorless?, Oliver Schneider, published 01/04/2007, source url: https://blog.assarbad.net/wp-content/uploads/2007/04/redpill_getting_colorless.pdf

5. Static code Analysis

As discussed in introduction chapter, basic static and dynamic malware analysis methods are good for initial triage, but they do not provide enough information to analyze malware completely and there is where disassembly comes in. Assembly is the highest-level language that can be reliably and consistently recovered from machine code when high-level language source code is not available.

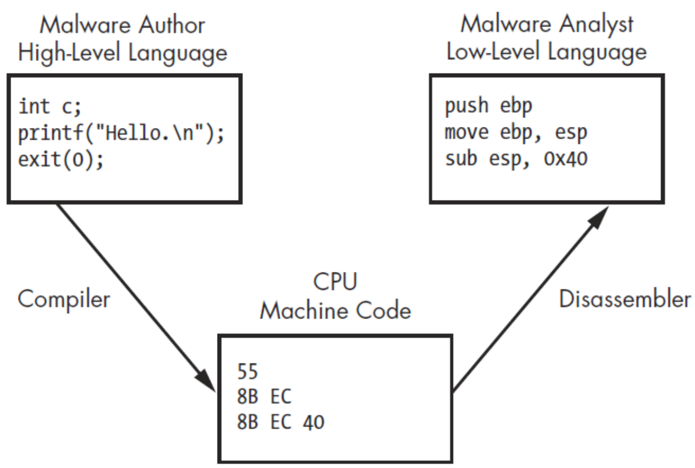


Figure 71: Three coding levels example[76]

The above Figure shows the three coding levels involved in reverse-engineering on malware analysis. Malware authors create programs at the high-level language level and use a compiler to generate machine code to be run by the CPU. Conversely, malware analysts and reverse engineers operate at the low-level language level. Using disassembler, assembly code is being generated in order to figure out how a program operates.

In under analysis case, the malware targets Windows platforms and interacts closely with the OS. The understanding of basic Windows coding concepts is principal to allow the identification host-

[76] Sikorski, Michael; Honig, Andrew; Lawler, Stephen, Practical Malware Analysis, San Francisco, CA: No Starch Press, 2012, pp. 66.

based indicators of malware, follow malware as it uses the OS to execute code without a jump or call instruction, and determine the malware's purpose. Windows uses two processor privilege levels: kernel mode and user mode. Nearly all code runs in user mode, except OS and hardware drivers, which run in kernel mode. In user mode, each process has its own memory, security permissions, and resources. If a user-mode program executes an invalid instruction and crashes, Windows can reclaim all the resources and terminate the program. Normally, user mode cannot access hardware directly, and it is restricted to only a subset of all the registers and instructions available on the CPU. In order to manipulate hardware or change the state in the kernel while in user mode, you must rely on the Windows API. When you call a Windows API function that manipulates kernel structures, it will make a call into the kernel. Kernel code is very important to malware writers because more can be done from kernel mode than from user mode.

The following figure illustrates a schematic overview of the involved parts.

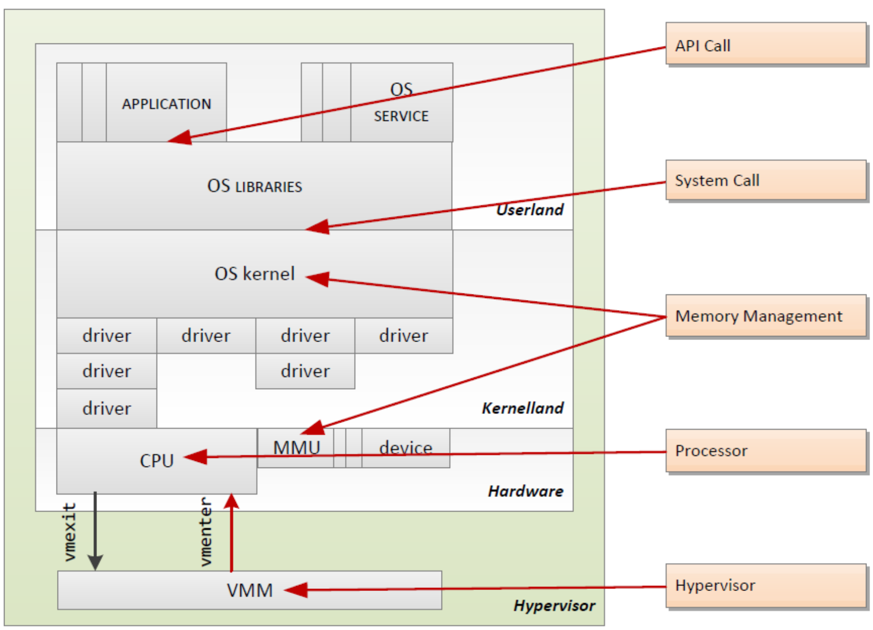


Figure 72: Schematic overview of Userland, Kernelland and Hardware, under a VM Hypervisor

5.1. IDA Pro

The Interactive Disassembler Professional (IDA Pro) is an extremely powerful disassembler distributed by Hex-Rays. Although IDA Pro is not the only disassembler, it is the disassembler of choice for many malware analysts, reverse engineers, and vulnerability analysts.

5.1.1 Loading the executable

When loading a PE file into IDA Pro, the program maps the file into memory as if it had been loaded by the operating system loader. The following figures presents our loading procedure and the relevant options of kernel and processor.

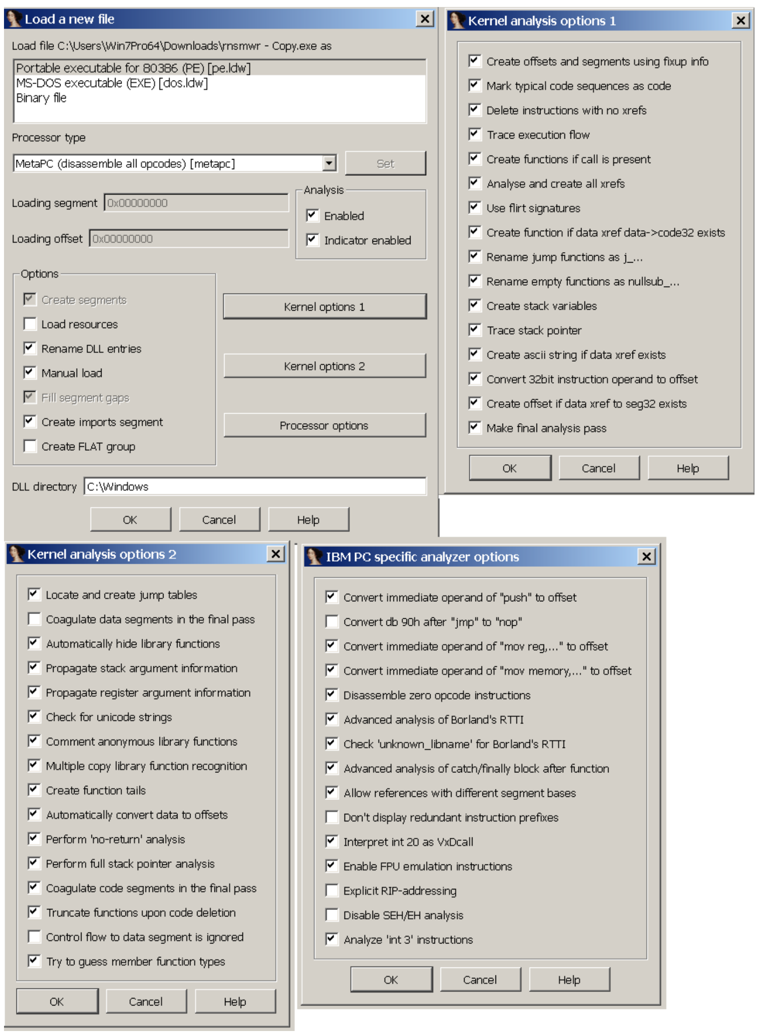


Figure 73: IDA Pro: Load PE file with analysis options

By default, IDA Pro does not include the PE header or the resource sections in its disassembly. Because malware often hides malicious code in such places, the manual load option, will load each section, one by one, including the PE file header, so that these sections would not escape IDA's analysis.

5.1.2 IDA's First glance

At first glance, the executable's entry point is at 401000 address. There are different views of IDA Pro that can be used to analyze the PE, the schematic view with diagrams and the text view where the analyzed, by IDA Pro, assembly is being previewed.

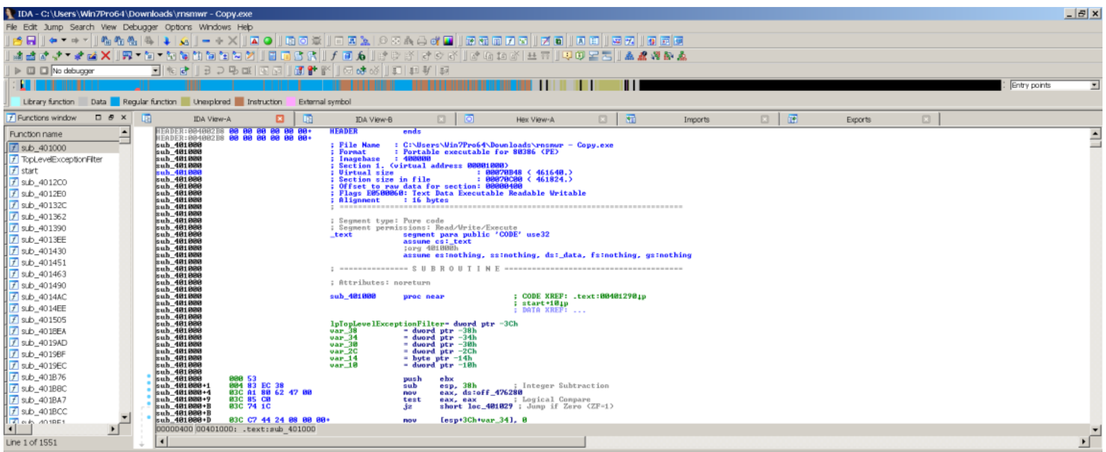


Figure 74: IDA View - text mode, PE entrance

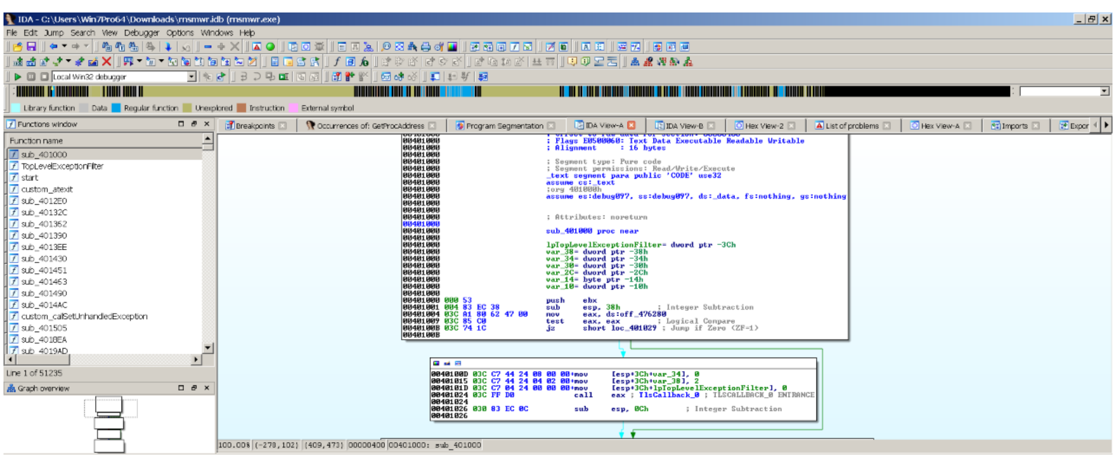


Figure 75: IDA View - graph mode, PE entrance

Please note that, because of the manual load of the PE file, the PE header is also loaded. The assembly code of PE Header is places before the entrance point from 400000 address until 401000.

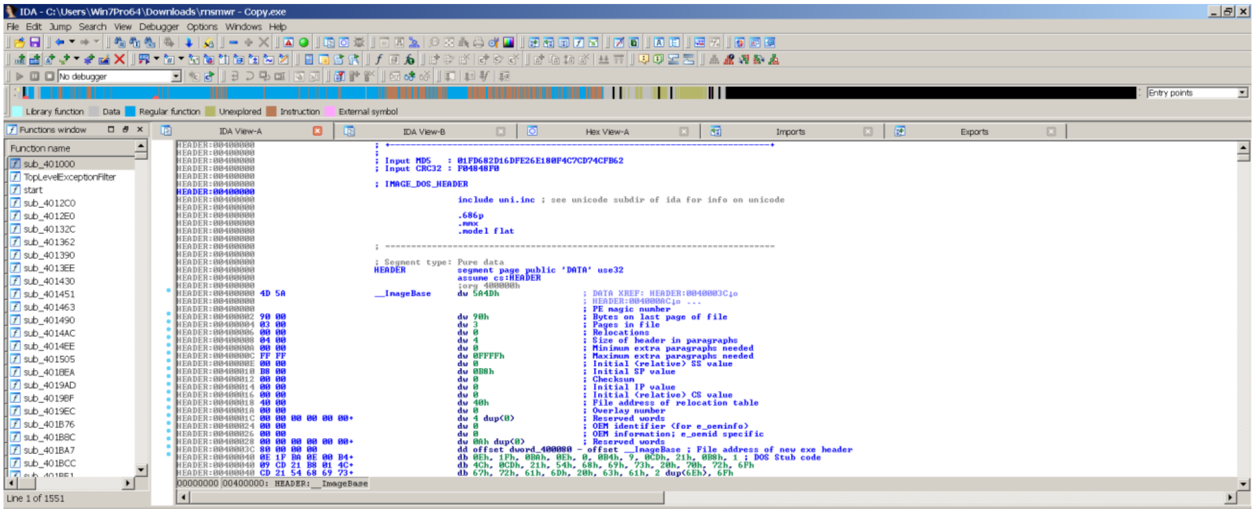


Figure 76: IDA View - PE headers on assembly

Double checking the results of Surface analysis, is helpful to focus on specific points on this stage. The size of the PE file and the number of its function is extremely high. The 1551 functions, that the IDA Pro reveal with its analysis stage, show that the malware author spent a lot of time writing the under analysis executable and from the malware analyst perspective a lot of work should be done.

Although the entrance point is on 401000 address, from the PE exports it is known that there are two TLS Call back functions that will be executed before that.

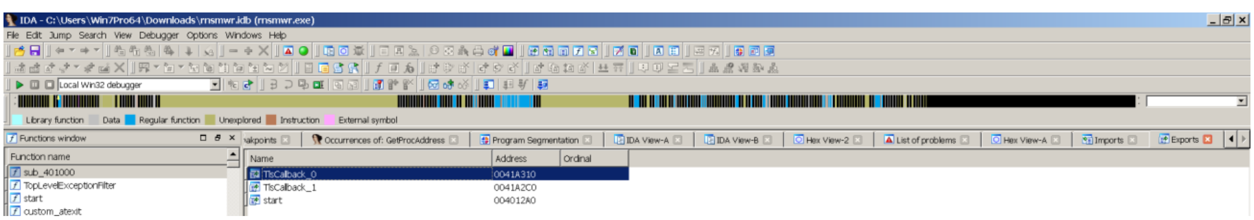


Figure 77: IDA View - list of Exports

Using IDA Pro, a crosscheck should be done on the suspicious functions. The full structured list of the 118 imports can be found at Appendix F/**List of .**

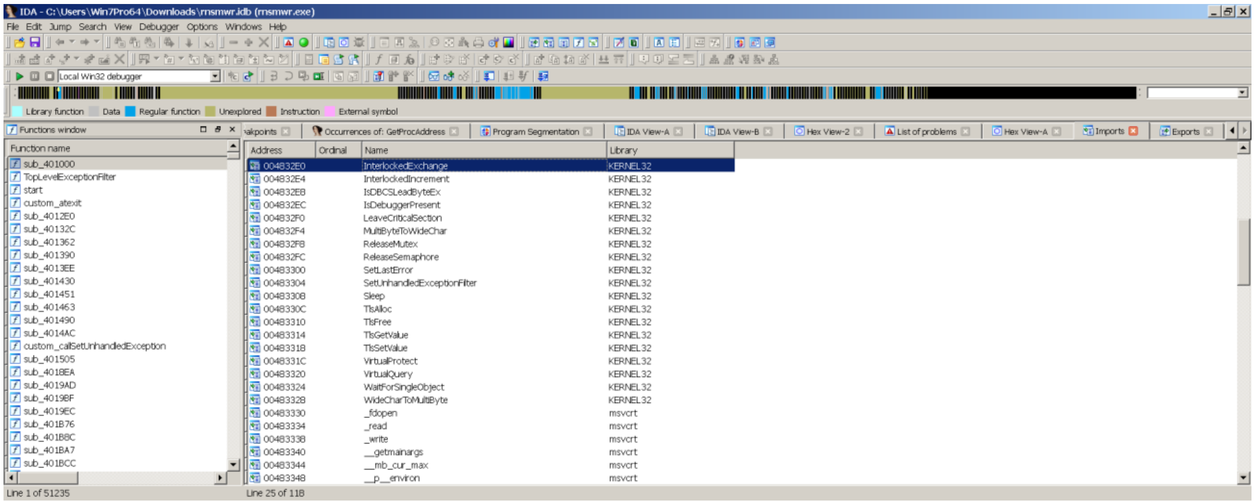


Figure 78: IDA Pro - imports

5.1.3 Custom Date Validation Check

The subject malware has an advanced anti-analysis feature. The malware author seems to have specific intentions, because the malware was programmed to be executed only in specific time range. As it is already mentioned in section **2.4.3. VMware Workstation Setup,** the under analysis ransomware has a sophisticated check of system time. More specifically the verification of date and time is being done at binary's location *.text:004026CC*, where the valid range to execute the ransomware is from the epoch time 1410739200, which is being converted as human readable date to GMT: Monday, September 15, 2014 12:00:00 AM, until the epoch time 1416009600, which is being converted as human readable date to Saturday, November 15, 2014 12:00:00 AM. The bypass solution of the system time check, without patching the binary, is already provision from the BIOS clock. Otherwise the binary should be patched with different time ranges.

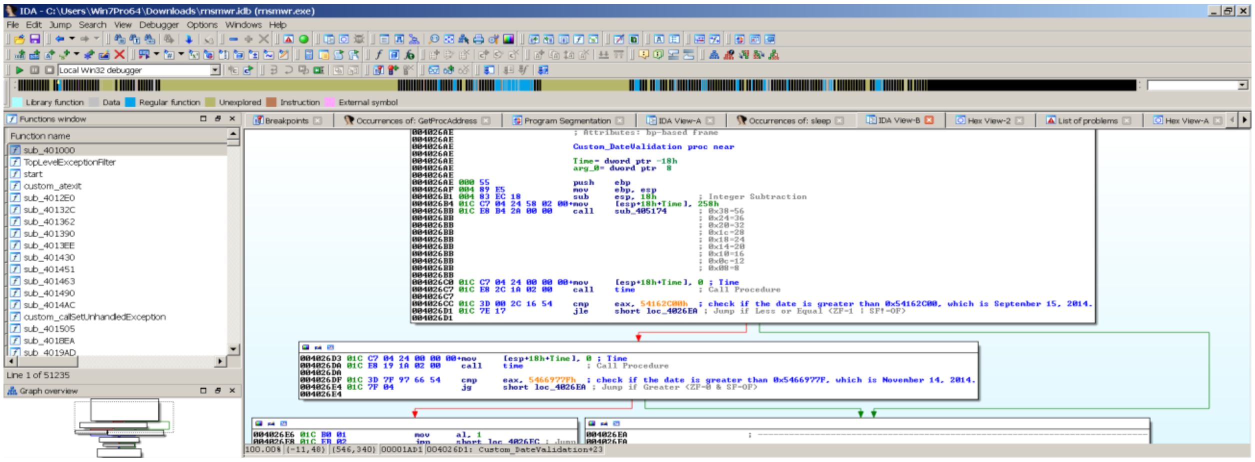


Figure 79: IDA - graph mode, Custom Date Validation Function

5.1.4 TLS Callback Functions

Malware authors employ numerous and creative techniques to protect their executables from reverse-engineering. The anti-debugging technique called *TLS callback* and has been explained on section **3.3.1./TLS explanation**. TLS callback functions are actually executed before executing code at the traditional Original Entry Point (OEP). To find the TLS callback in IDA Pro and press Ctrl+E.

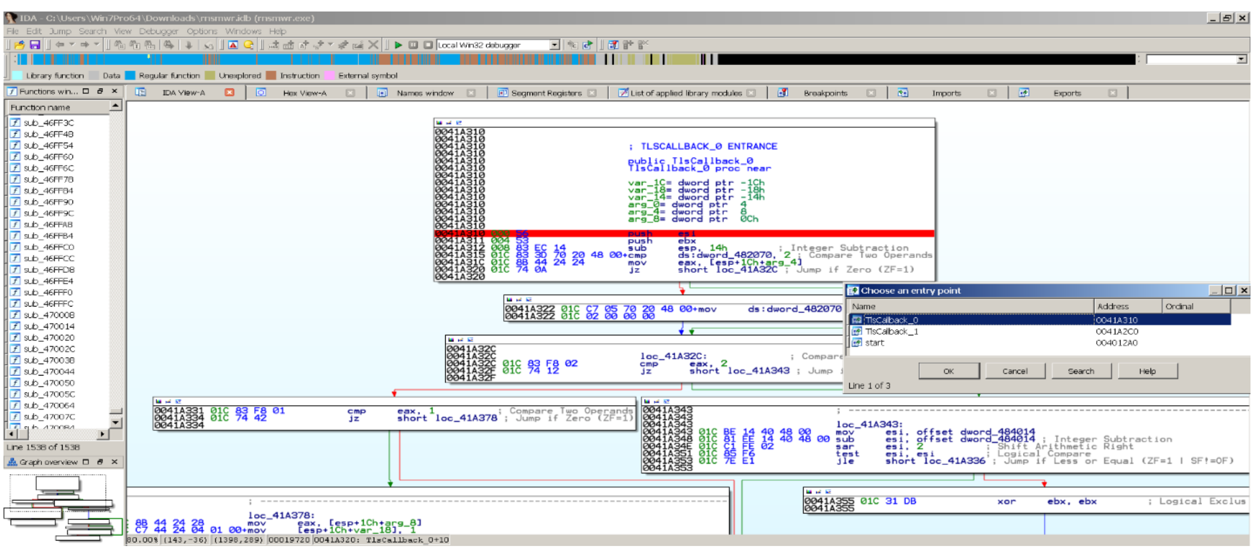


Figure 80: IDA Pro - Entry point choice

We can clearly see the structure of the execution. This program will execute three functions in a specific order, first the TlsCallback_0, then the TlsCallback_1 and at last the start – main program. Despite it is the first and only complete program called after the entry point, the start with will be executed last. The explanation of this chain of prosecution sourcing from the 'AddressOfCallBacks' value 00484004. The address is on .crt section and points to the TlsCallback_0. By default, most debuggers break at the entry point and consequently the TLS callbacks function are executed, but this will be discussed on the next section. On this case, the TLS_Callbacks are not only executed before the main - start function, but they are dynamically called, via call eax command. The indirect call procedure, is and would be a frequent technique, from the malware author.

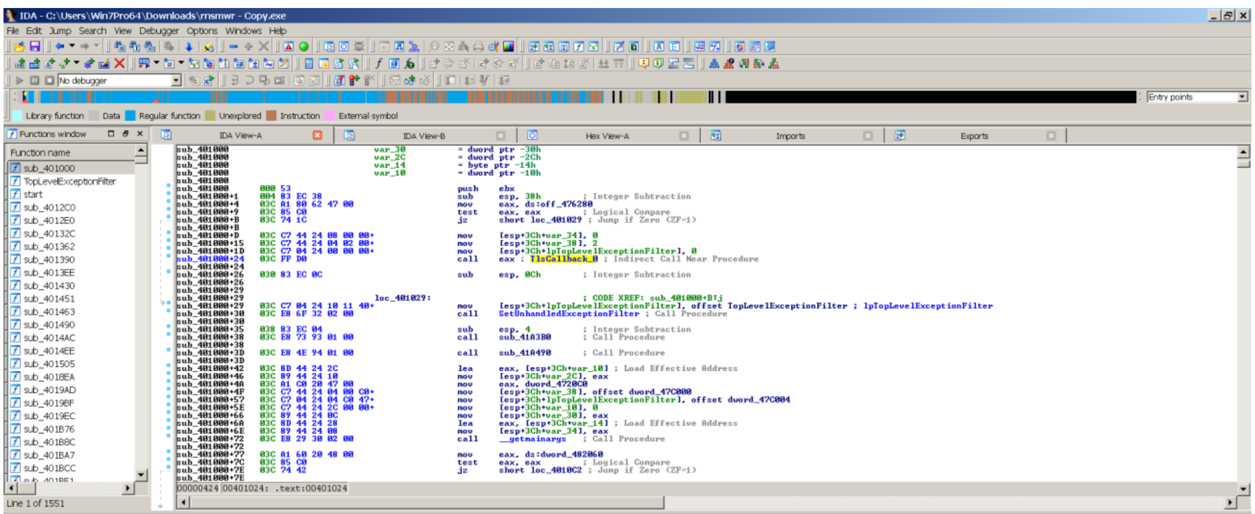


Figure 81: IDA Pro - TLScallback dynamic call

Nevertheless, the attacker had inserted anti-debugging routines inside the TLS callback functions to mislead the malware analyst.

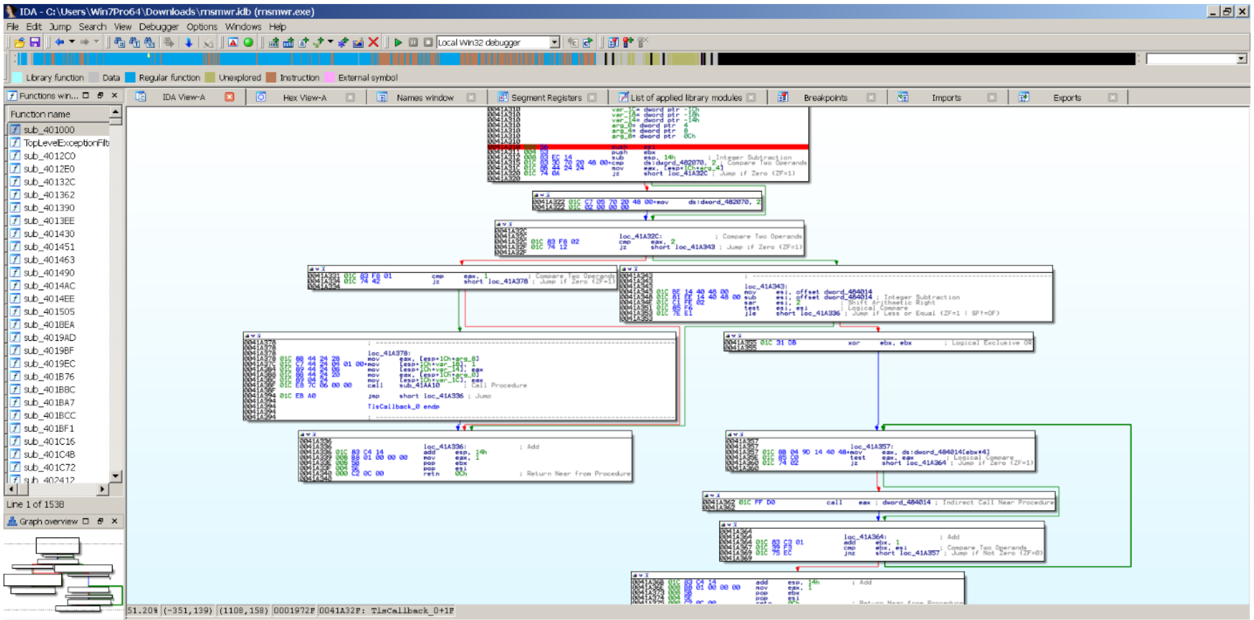


Figure 82: IDA Pro - TLScallback_0

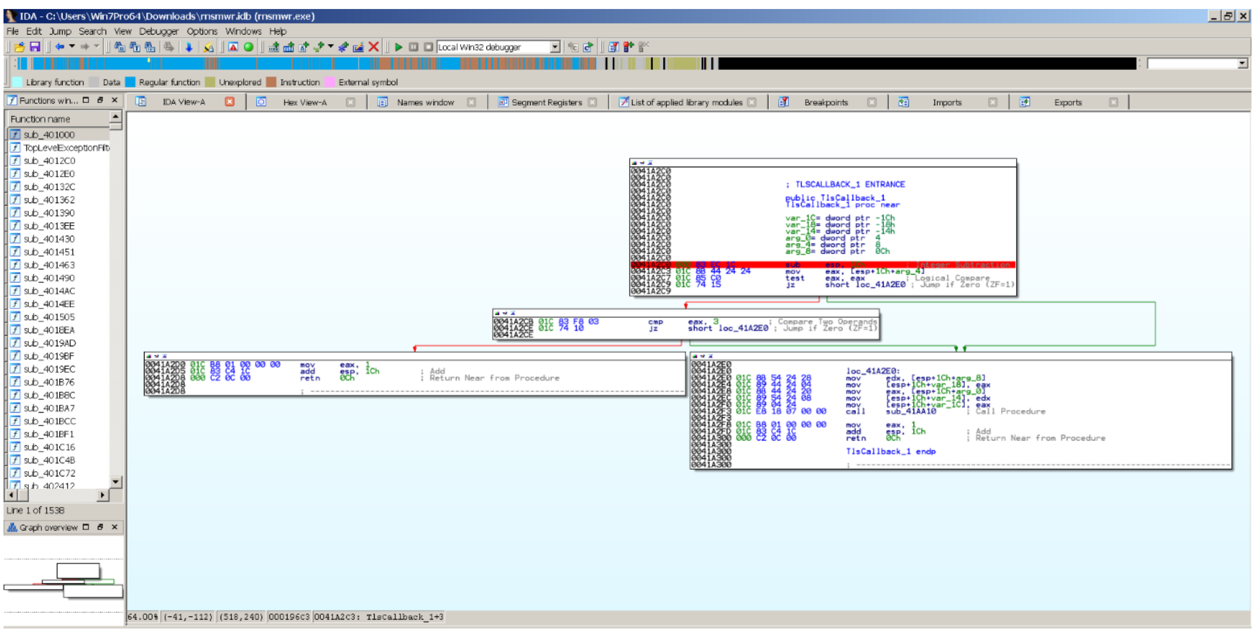


Figure 83: IDA Pro - TLScallback_1

Both of the TLS callback functions are leading to 0041AA10 function call that is related to EnterCriticalSection, LeaveCriticalSection, InitializeCriticalSection or DeleteCriticalSection. Note that for the calling the thread EnterCriticalSection twice, will lead to stuck an eternity loop. Specifically, with the thread call EnterCriticalSection getting stuck forever at the call. In addition ,a critical section object cannot be moved or copied. The process must also not modify the object, but must treat it as logically opaque. The usage of critical section functions is to manage critical section objects.

5.1.5 Debugger Presence

IsDebuggerPresent API

The most distinct point in the list of import functions is the IsDebufferPresent function. The explanation of IsDebuggerPresent function can be found in 4.1.1. section. Searching for all the occurrences for the IsDebuggerPresent function, the function is being called in at 00402736 address.

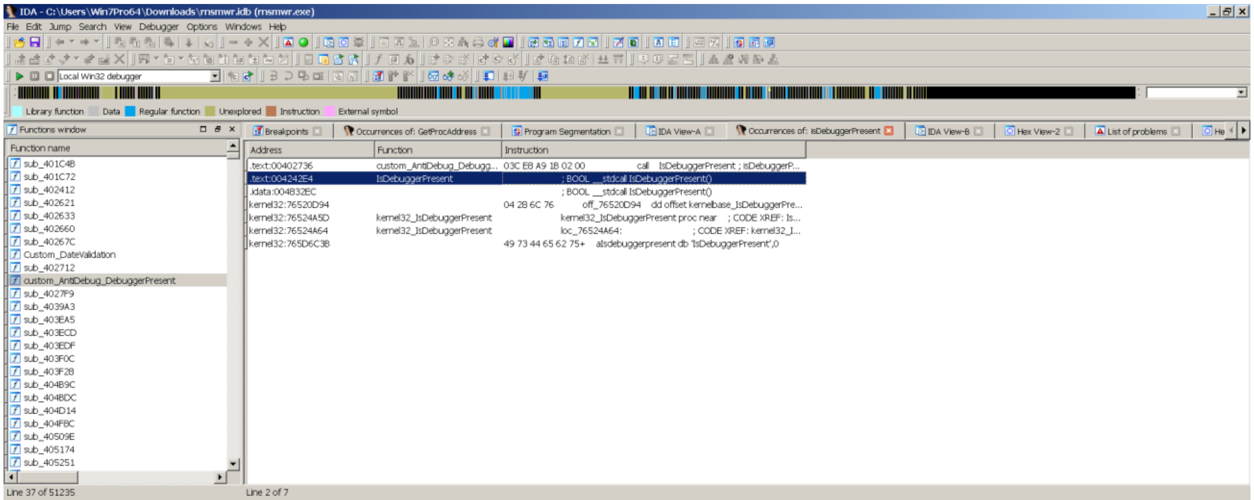


Figure 84: IDA View, IsDebuggerPresent all occurrences

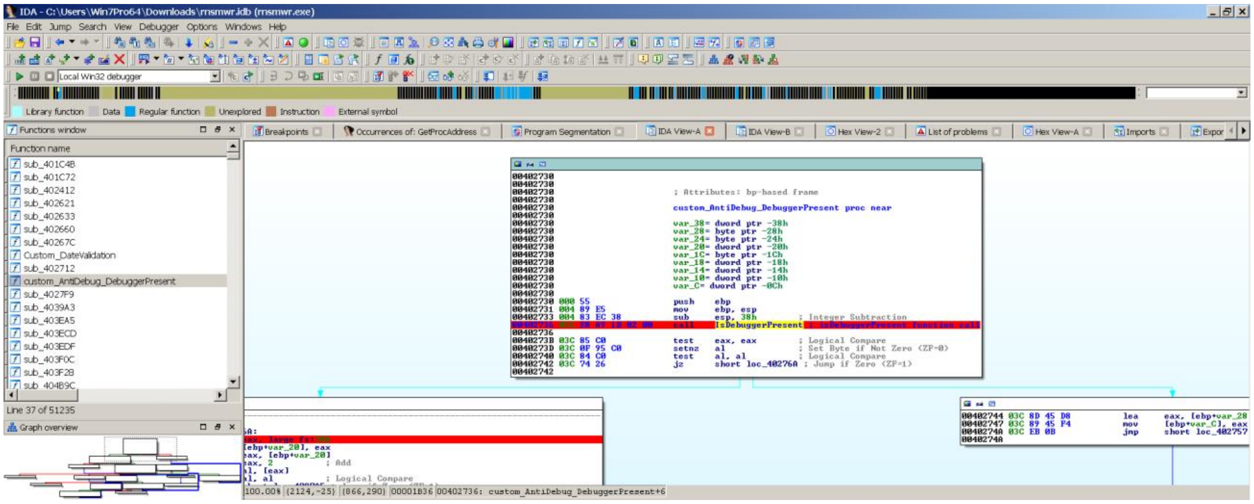


Figure 85: IDA View - graph mode, IsDebuggerPresent at 00402730

The figure 84 depicts a custom process that determines whether the calling process is being debugged (by a user-mode debugger). If the current process is running in the context of a debugger, the return value is nonzero. The simplest API function for detecting a debugger is IsDebuggerPresent. This function searches the Process Environment Block (PEB) structure for the field IsDebugged, which will return zero if you are not running in the context of a debugger or a nonzero value if a debugger is attached.

The Process Environment Block (PEB) is a user-mode data structure that can be used by applications (and by extend by malware) to get information such as the list of loaded modules, process startup arguments, heap address, check whether program is being debugged or even find image base address of imported DLLs.

IsDebugged PEB Flag

If we examine the API in a debugger we can see that it uses FS[30] segment register which is the linear address of Process Environment Block (PEB) and then reach the offset 0x002 which is the BeingDebugged. So instead of calling IsDebuggerPresent(), the malware manually check the PEB (Process Environment Block) for the BeingDebugged flag.

In the under analysis case, the malware author created a custom procedure of checking the existence of a Debugger. The check does not stop on the Windows API return value, but continues with custom checks of PEB. The Process Environment Block (PEB) structure for the field IsDebugged, which will return zero if you are not running in the context of a debugger or a nonzero value if a debugger is attached.

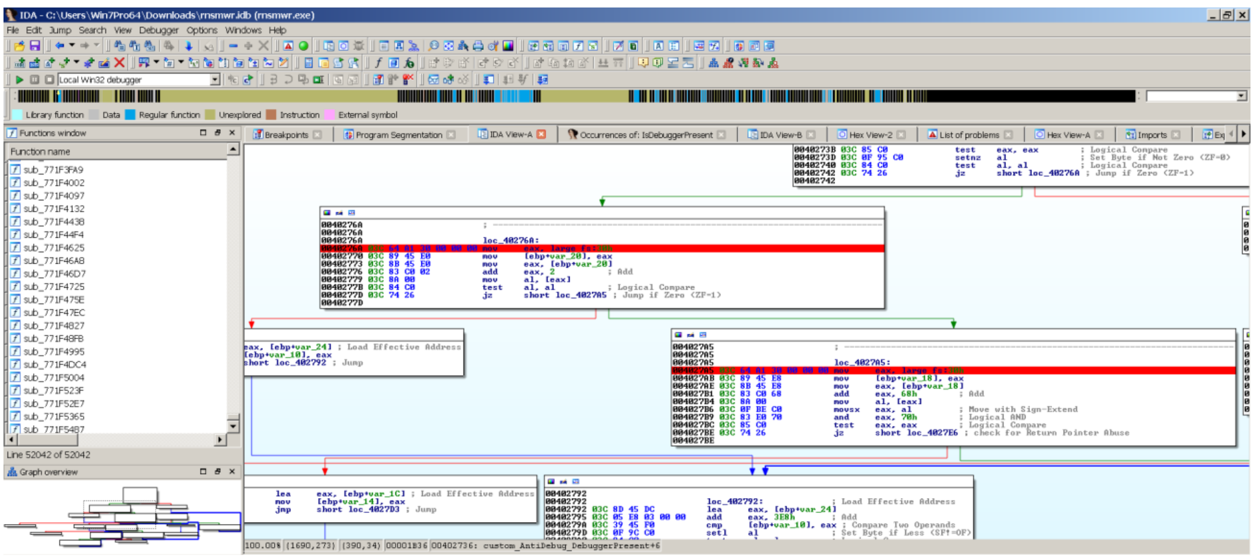


Figure 86: IDA View - graph mode, custom PEB check IsDebugged

More specifically, at address 0040276A the large fs:30 segment register leads to the address of PEB and then the offset 0x02 is added and checked, which is the BeingDebugged flag.

NtGlobalFlag Flag

Moreover, at the address 004027A5, large FS[30] segment register leads also to the address of PEB and then the offset 0x68 is added and checked, which is the NtGlobalFlag flag. This is another simple anti-reversing trick used to detect a debugger. At the TEB structure and the PEB structure, NtGlobalFlag is located in the PEB Structure at offset PEB+104.

So this flag can also challenge identification of whether the process is being debugged. Normally, when a process is not being debugged, the NtGlobalFlag field contains the value 0x0. When the process is being debugged, the field will usually contain the value 0x70. The 0x70 value is a total of checks, which indicates that the following flags are set:

- *FLG_HEAP_ENABLE_TAIL_CHECK   0x10*
- *FLG_HEAP_ENABLE_FREE_CHECK   0x20*
- *FLG_HEAP_VALIDATE_PARAMETERS   0x40*
- *Total   0x70*

That is the reason the malware author makes a comparison at 004027B9 address. If we examine the API in a debugger we can see that it uses FS[30] segment register which is the linear address of Process Environment Block (PEB) and then reach the offset 0x68 which is the NtGlobalFlag.

Nevertheless, searching for all the occurrences for large FS[30], it can be figured that the malware author has implemented the anti-debugging PEB checks in several places, with several ways.

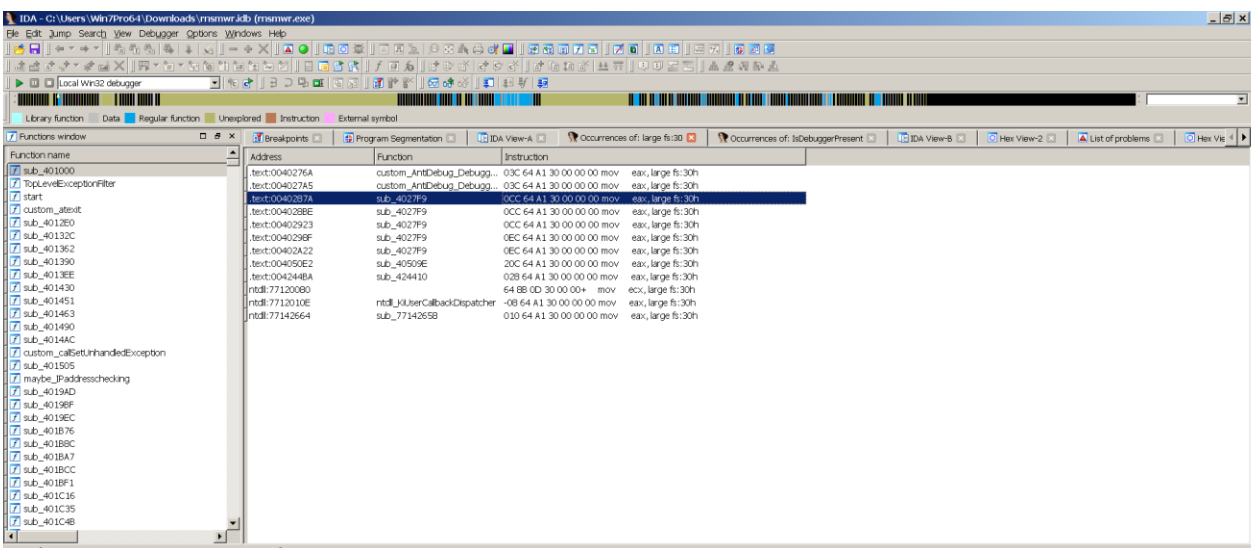


Figure 87: IDA View, large fs:30 - all occurrences

Specifically, at the addresses 0040287A, 00402923, 0040298F and 004050E2 the BeingDebugged flag is being checked. Also, at the addresses 004028BE, 00402A22 and 004244BA the NtGlobalFlag flag is being checked. Special attention is needed to the technique the malware author is using, the eax register is not doing the comparison immediately, but each check preceded with a no operation trick, by using the EBP plus to a non-stable variable.

5.1.6 Anti-VMware

The most popular anti-VMware techniques are being used, in order to slow down analysis, so it was important to recognize them at early points, as it has been done in basic surface and behavioral analysis.

As it is already mentioned, when performing basic dynamic analysis, a virtual machine should be used. However, if your subject malware does not seem to run, a different virtual environment (like VirtualBox or Parallels) or even a physical machine, should be tried. As with anti-debugging techniques, anti-VM techniques can be spotted using common sense while slowly debugging a process. For example, code terminating prematurely at a conditional jump, it may be doing so as a result of an anti-VM technique. As always, be aware of these types of issues and look ahead in the code to determine what action to take.

The Red Pill Anti-VM Technique

Red Pill is an anti-VM technique that executes the SIDT instruction to grab the value of the IDTR register. The virtual machine monitor must relocate the guest's IDTR to avoid conflict with the host's IDTR. Since the virtual machine monitor is not notified when the virtual machine runs the SIDT instruction, the IDTR for the virtual machine is returned. For more detailed explanation of the Descriptor Table Registers and them detection technique, please check at the section **2.3. VME Technologies**.

The Red Pill tests for this discrepancy to detect the usage of VMware. The malware issues the SIDT instruction at, which stores the contents of IDTR into the memory location pointed to by EAX. The IDTR is 6 bytes, and the fifth byte offset contains the start of the base memory address. That fifth byte is compared to 0xFF, the VMware signature.

The attached short exploit code can be used to detect whether the code is executed under a VME or under a real environment. [77]

```
int swallow_redpill ()
{
  unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
  *((unsigned*)&rpill[3]) = (unsigned)m;
  ((void(*)())&rpill)();
  return (m[5]>0xd0) ? 1 : 0;
}
```

Table 5: "Swallowing" the Red Pill has been published as this four line code, generating almost a single CPU instruction and that returns nonzero when in "Matrix".

The heart of this code is actually the SIDT instruction (encoded as 0F010D[addr]), which stores the contents of the interrupt descriptor table register (IDTR) in the destination operand, which is actually a memory location. What is special and interesting about SIDT instruction is that, it can be executed in non-privileged mode (ring3) but it returns the contents of the sensitive register, used internally by operating system.

Because there is only one IDTR register, but there are at least two OS running concurrently (i.e. the host and the guest OS), VME needs to relocate the guest's IDTR in a safe place, so that it

[77] Red Pill... or how to detect VMM using (almost) one CPU instruction, Joanna Rutkowska, originally published at URL: http://invisiblethings.org/, on November 2004, current access URL: http://web.archive.org/web/20110726182809/http://invisiblethings.org/pa

will not conflict with a host's one. Unfortunately, VME cannot know if (and when) the process running in guest OS executes SIDT instruction, since it is not privileged (and it doesn't generate exception). Thus, the process gets the relocated address of IDT table. It was observed that on VMWare, the relocated address of IDT is at address 0xffXXXXXX, whereas on Hyper-V (Virtual PC) it is 0xe8XXXXXX.

Joanna Rutkowska came across this strange behavior of SIDT instruction a few years ago, when Joanna Rutkowska was testing Suckit rootkit on VMWare. Joanna Rutkowska noticed that it failed to load on VMWare whereas it seemed to work fine on the same distribution ran outside VM. After spending many hours Joanna Rutkowska figured out that the problematic instruction was actually SIDT, which was used by Socket to get the address of the IDT table, and to hook its 0x80 entry through /dev/kmem device.

Please note that Red Pill succeeds only on a single-processor machine, because it would not work consistently against multicore processors, as long as each processor (guest or host) has an IDT assigned to it. Therefore, the result of the SIDT instruction can vary, and the signature used by Red Pill can be unreliable. To thwart this technique, run on a multicore processor machine or simply NOP-out the SIDT instruction.

The No Pill Technique

The SGDT and SLDT instruction technique for VMware detection is commonly known as No Pill. Unlike Red Pill, No Pill relies on the fact that the LDT structure is assigned to a processor, not an operating system. And because Windows does not normally use the LDT structure, but VMware provides virtual support for it, the table will differ predictably.

Specifically, the LDT location on the host machine will be zero, and on the virtual machine, it will be nonzero. A simple check for zero against the result of the SLDT instruction does the trick.

The SLDT method can be subverted in VMware by disabling acceleration. To do this, select VMware Settings > *Settings*, on the Analysis VM > at *Processors* option tab and check *the Disable Acceleration* box. No Pill solves this acceleration issue by using the SMSW instruction if the SLDT method fails. This method involves inspecting the undocumented high-order bits returned by the SMSW instruction.

The I/O Communication Port

The most common anti-VMware technique currently in use is that of querying the I/O communication port. This technique was discovered by Ken Kato[78]. VMware uses virtual I/O ports for communication between the virtual machine and the host operating system to support functionality like copy and paste between the two systems. The port can be queried and compared with a magic number to identify the use of VMware. The success of this technique depends on the x86 in instruction, which copies data from the I/O port specified by the source operand to a memory location specified by the destination operand.

VMware monitors the use of the in instruction and captures the I/O destined for the communication channel port 0x5658 (VX). Therefore, the second operand needs to be loaded with VX in order to check for VMware, which happens only when the EAX register is loaded with the magic number 0x564D5868 (VMXh)[79]. ECX must be loaded with a value corresponding to the

---

[78] Ken Kato, VMware Backdoor I/O Port, source URL: chitchat.at.infoseek.co.jp/vmware/backdoor.html
[79] Methods for Virtual Machine Detection, Alfredo Andr´es Omella, Grupo S21sec Gesti´on S.A., 20th June 2006

action you wish to perform on the port. The value 0xA means "get VMware version type," and 0x14 means "get the memory size." Both can be used to detect VMware, but 0xA is more popular because it may determine the VMware version.

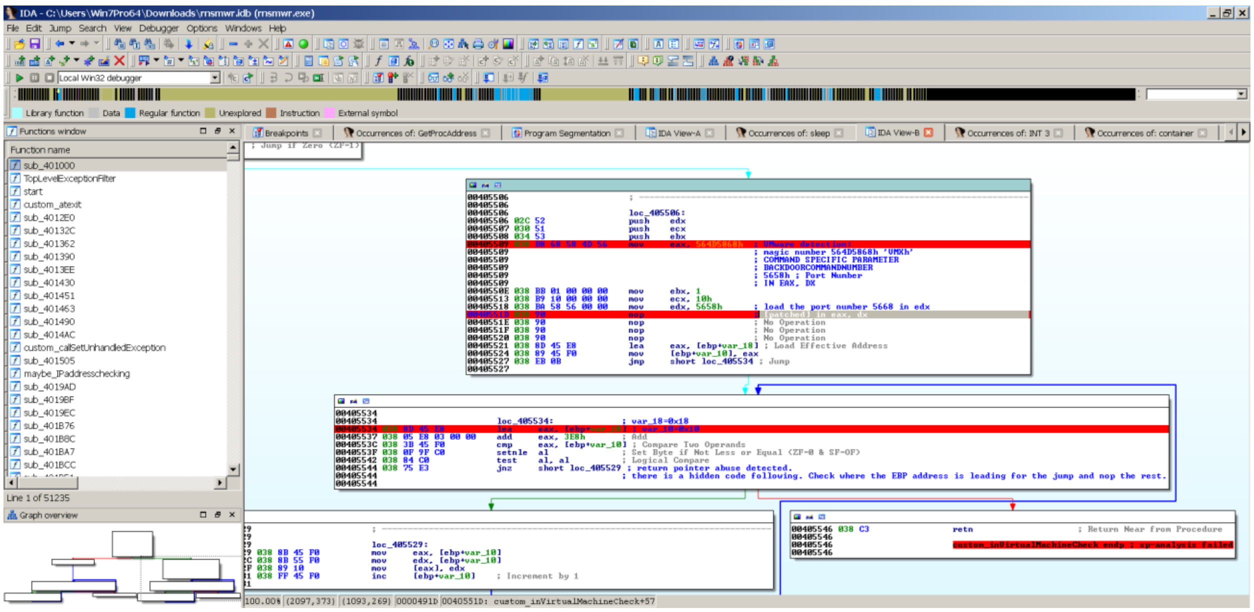


Figure 88: Red Pill VMware detection with Backdoor Command Number - patched

On the above Figure, at the address 00405509 the command MOV EAX, 564D5868h has been detected, which is the famous VMware Magic Number (VMXh). The malware first loads the magic number *0x564D5868* (VMXh) into the EAX. Next, it loads the value *1* into EBX, a memory address that will return any reply from VMware. ECX is loaded with the value 0x10 to get the VMware version type. Next, the **0x5658** (VX) is loaded into EDX, to be used in the following in instruction to specify the VMware I/O communication port. Upon execution, the in instruction is trapped by the virtual machine and emulated to execute it. The in instruction uses parameters of EAX (magic value), ECX (operation), and EBX (return information). If the magic value matches VMXh and the code is running in a virtual machine, the virtual machine monitor will echo that back in the memory location specified by the EBX register. The next immediate check determines whether the code is being run in a virtual machine. Since the get version type option is selected, the ECX register will contain the type of VMware (**1=Express**, 2=ESX, 3=GSX, and 4=Workstation).

The easiest way to overcome this technique is to **NOP**-out the in instruction **IN EAX, DX** or to patch the conditional jump to allow it regardless of the outcome of the comparison. At the figure 89, the NOP-out technique has been chosen.

6. Dynamic code Analysis

The dynamic code analysis is the hard part of debugging a software. The tool to make a dynamic analysis is the debugger. A debugger is a piece of software, in this case, used to test or examine the execution of the subject malware. Debuggers help in the process of developing software, since programs usually have errors in them when they are first written. Debuggers gives the insight into what a program is doing while it is executing. Specifically, debuggers are designed to allow developers to measure and control the internal state and execution of a program. Because theory of debuggers and instructions using them are not part of this thesis and the document is already long enough, in continuous only the vital parts of code are being presented during the debugging.

6.1 Structured Exception Handlers

Generally, the exceptions allow a program to handle events outside the flow of normal execution. The Structured Exception Handling (SEH) mechanism provides a method of flow control that is unable to be followed by disassemblers and will fool debuggers. SEH is a feature of the x86 architecture and is intended to provide a way for the program to handle error conditions intelligently.

The common exceptions are caused by errors and when an exception occurs, execution transfers to a special routine that resolves the exception. Some exceptions, such as division by zero, are raised by hardware. Some others, such as an invalid memory access, are raised by the OS. Specifically, the Structured Exception Handling (SEH) is the Windows mechanism for handling exceptions, where SEH information are stored on the stack.

At the beginning of each function, an exception-handling frame is put onto the stack, with the special location *fs:0* points to an address on the stack, that stores the exception information. When an exception occurs, Windows looks in fs:0 for the stack location that stores the exception information, and then the exception handler is called. After the exception is handled, execution returns to the main thread. So exception handlers are nested, and not all handlers respond to all exceptions. The SEH chain is a list of functions designed to handle exceptions within the thread. If the exception handler for the current frame does not handle an exception, it will be passed to the exception handler for the caller's frame. Eventually, if none of the exception handlers responds to an exception, the top-level exception handler crashes the application.

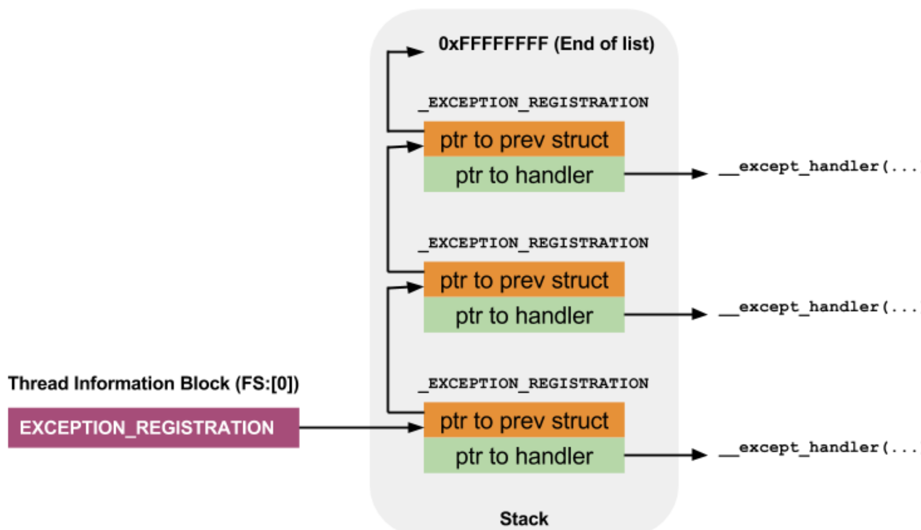


Figure 89: SEH Chain [80]

To find the SEH chain, the OS examines the FS segment register. This register contains a segment selector that is used to gain access to the Thread Environment Block (TEB). The first structure within the TEB is the Thread Information Block (TIB). The first element of the TIB (and

[80] The source URL of the image: www.aldeid.com/wiki/Category:Architecture/Windows/SEH-Structured-Exception-Handling

consequently the first bytes of the TEB) is a pointer to the SEH chain. The SEH chain is a simple linked list of 8-byte data structures called EXCEPTION_REGISTRATION records.

The first element in the EXCEPTION_REGISTRATION record points to the previous record. The second field is a pointer to the handler function. This linked list operates conceptually as a stack. The first record to be called is the last record to be added to the list. The SEH chain grows and shrinks as layers of exception handlers in a program change due to subroutine calls and nested exception handler blocks. For this reason, SEH records are always built on the stack.

Misusing Structured Exception Handlers

In the subject malware, the exception handlers are being used in exploit code to gain execution. A pointer to exception-handling information is stored on the stack, and during a stack overflow, an attacker can overwrite the pointer. By specifying a new exception handler, the attacker gains execution when an exception occurs.

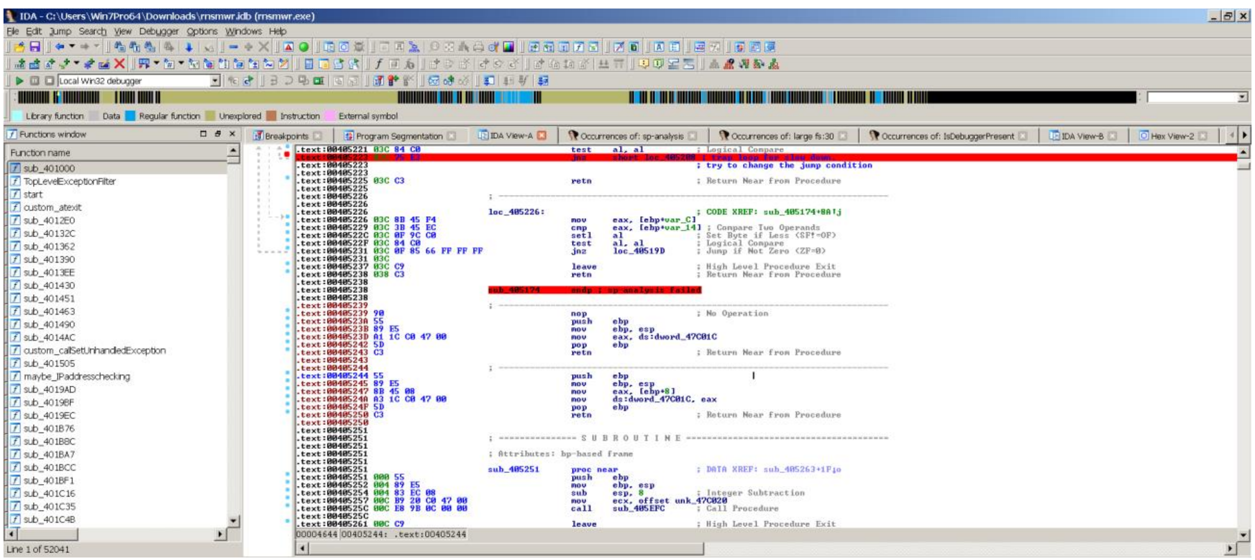


Figure 90: IDA Pro, text view, sp-analysis failed

In this figure, IDA Pro has not only missed the fact that the subroutine at location 405239 was not called, but it also failed to even disassemble this function (sp-analysis failed). Stack-frame anti-analysis techniques depend heavily on the compiler used. Of course, if the malware is entirely written in assembly, then the author is free to use more unorthodox techniques. However, if the malware is crafted with a higher-level language such as C or C++, special care must be taken to output code that can be manipulated.

Anti-disassembly is not confined to the studied techniques. It is a class of techniques that takes advantage of the inherent difficulties in analysis. Anti-disassembly is more difficult with a flow-oriented disassembler but still quite possible, once you understand that the disassembler is making certain assumptions about where the code will execute. Obscuring flow control is a way that malware can cause the malware analyst to overlook portions of code or hide a function's purpose by obscuring its relation to other functions and system calls.

Please keep in mind that in **Behavioral Analysis section, 4.1.1.** at **Informative Indicators** the anti-reverse engineering technique of *SetUnhandledExceptionFilter* has been already detected from an online automotive analysis tool. Specifically, at the addresses 00401030 and 004014FB the call of function *SetUnhandledExceptionFilter* has been done and at address 004242A4, another indirect near jump is taken place. Furthermore, the call of function *ltTopLevelExceptionFilter* at the addresses 00401026 and 004014F1, in addition with the indirect near jump at address 004242A4.

Function lpTopLevelExceptionFilter is a pointer to top-level exception filter function that will be called whenever the UnhandledExceptionFilter function gets control, and the process is not being debugged. A value of null for this parameter specifies default handling within UnhandledExceptionFilter. Usually, in absence of an UnhandledExceptionFilter the topmost handler called when an unhandled exception occurs, is the default one provided by Windows Itself, the classical MessageBox that advices the user that an Unhandled Exception has occurred.

Debugging detection using Unhandled Exceptions

On the other hand, Windows allow programmers to use custom Handlers for UnhandledException. The core of the trick is here, if the application is not debugged, the application is able to call the Custom Handler, but if the application is debugged the Custom Handler will be never called.

Please note that inside UnhandledExceptionFilter function, the function NtQueryInformationProcess is called that has as first parameter the subject process and next DebugPort, this is done to know if the process is debugged.

This anti-debugging and also anti-reversing technique was caught being called in several parts of assembly code, in the subject malware. As long as these are custom handlers, the counter technique should be manual.

- At First a search for "All intermodular calls" should be done and due to the results breakpoints at the call of GetProcAddress function should be added and then resolve the imports of the pack file.
- The next move is to run the binary of the subject malware and when it breaks, the stack should be checked for the function SetUnhandledExceptionFilter that is being loaded. The SetUnhandledExceptionFilter handles the exceptions that are not being hardcoded with some exception function. At this point the function lpTopLevelExceptionFilter will be executed only if the binary is not being debugged.
- Because the subject malware is obviously running under a debugger, the return value of GetCurrentProcess function should be search. Firstly, "UnhandledExceptionFilter" should be searched (CTRL+G) as an expression.

- Then breakpoint at the call of "kernel32.GetCurrentProcess" function should be added.
- By executing the binary, the return value at EAX register should manually changed from -1 to 0

Keep in mind that there are some dynamic calls of GetCurrentProcess functions, via other functions as a parameter. These functions are "RtlEncodePointer" and "En/DecodePointer".

GetProcAddress(LoadLibraryA(kernel32.dll),	EncodePointer);
GetProcAddress(LoadLibraryA(kernel32.dll),	DecodePointer);

- The next function call will be the "ZwQueryInformationProcess,", that will check the value of EAX register. Keep in mind that the new version of ZwQueryInformationProcess is NtQueryInformationProcess, both mentioned in ntdll. In case that the value will be -1, this will lead to a stop function, because the debugged process is revealed.

It should be noted that, a generic measurement to counter this technique, is by editing the return value of GetCurrentProcess function from 0xFFFFFFFF to 0x00000000. In other words, an apparently undebugged process should be obtained in order to modify the first parameter (last pushed at debugging time).

Timing Checks

Single-stepping through a program substantially slows execution speed. There are a couple of ways to use timing checks to detect a debugger, record a timestamp, perform a couple of operations, take another timestamp, and then compare the two timestamps. If there is a lag, you can assume the presence of a debugger. Also, take a timestamp before and after raising an exception.

If a process is not being debugged, the exception will be handled quickly; a debugger will handle the exception much more slowly. By default, most debuggers require human intervention in order to handle exceptions, which causes enormous delay. While many debuggers allow you to ignore exceptions and pass them to the program, there will still be a sizable delay in such cases.

Nevertheless, on the subject malware, another anti-debugging SEH technique due to the dynamic code analysis revealed. The anti-debugging timing checks are successful because the malware causes and catches an exception that it handles by manipulating the Structured Exception Handling (SEH) mechanism to include its own exception handler in between two calls to the timing checking functions. Exceptions are handled much more slowly in a debugger than outside a debugger. On the following screenshot a Custom top level exception handler is installed, at the address .text:004014FB.

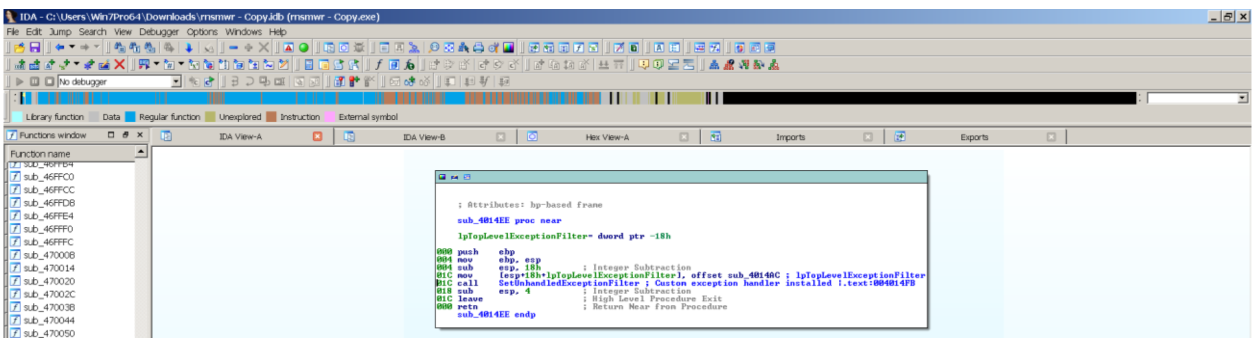


Figure 91: IDA Pro, graph view, Top level Exception Custom Handler

This exception will lead to about 10 minute sleep at the beginning and somewhere else dynamically called. On the following figure, at the address .text:00405174, some implemented with time function calls are being presented, combined with the above mentioned techniques, are adding some additional protections against fast forwarding time.

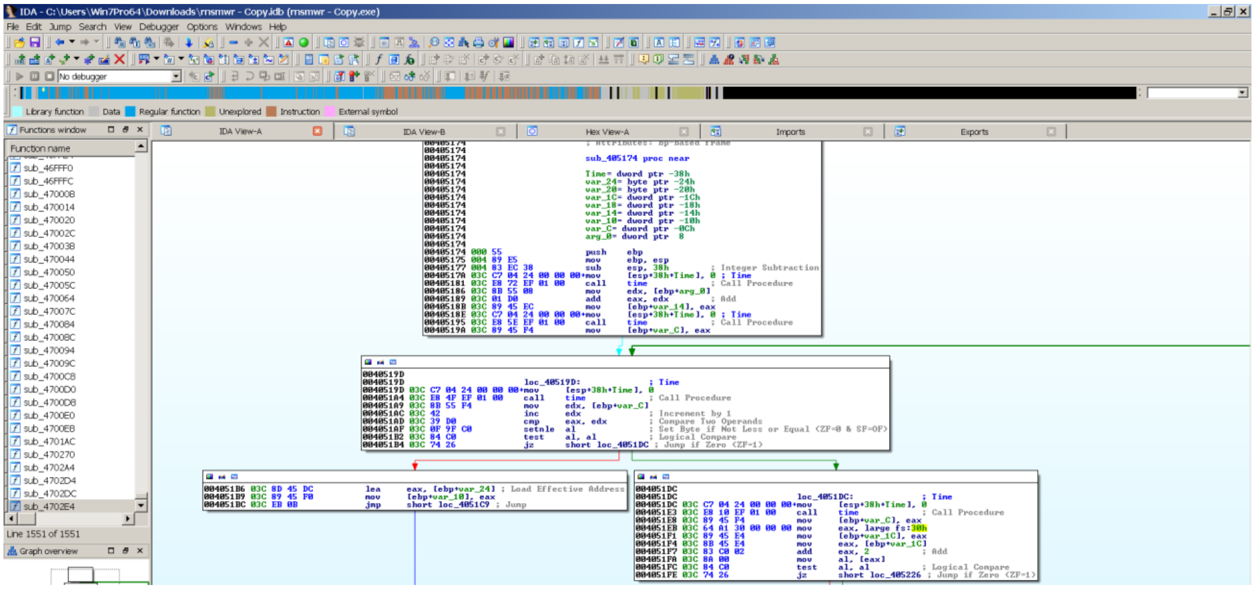


Figure 92: IDA Pro, graph view, time function calls

As a result, a nonstop loop is being detected. The cause was from the REPNE SCASB instruction. The usage of REPNE SCASB is to scan bytes of a string until the trailing null character is found. A common use of the REPNE SCASB instruction, in the subject malware, is to determine the length of a string. Below is a code that checks whether the string passed to the function is 4 characters long.

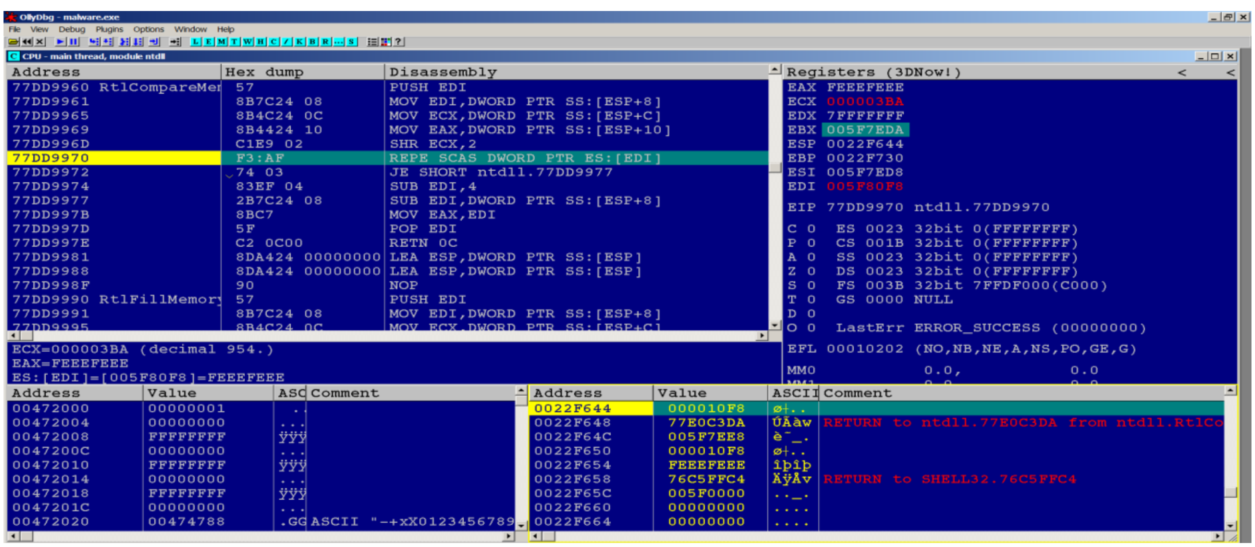


Figure 93:OllyDbg REPE SCAS instruction

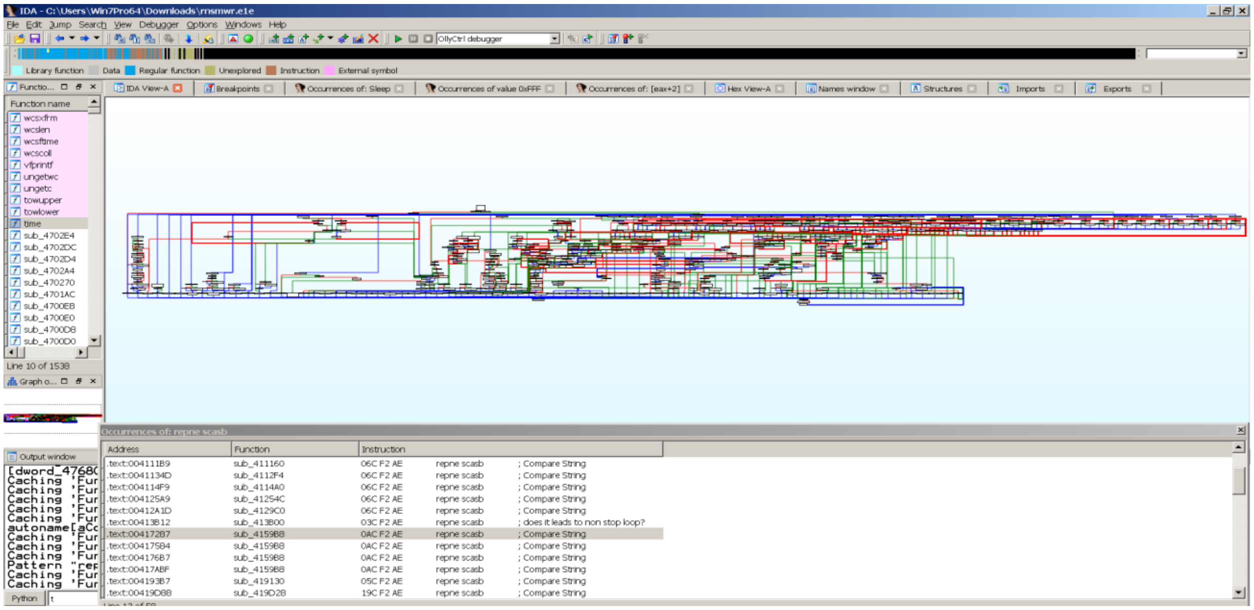


Figure 94: IDA View, REPNE SCASB instruction all occurrences

On the previous screenshot, there are tons of these instruction been detected. So, all occurrences search will not help. In continuous, the endless loop is being detected on the subfunction text.405208. During execution debugging, the stack was filled endlessly with ASCII characters, without finding on a fly solution by patching the binary.

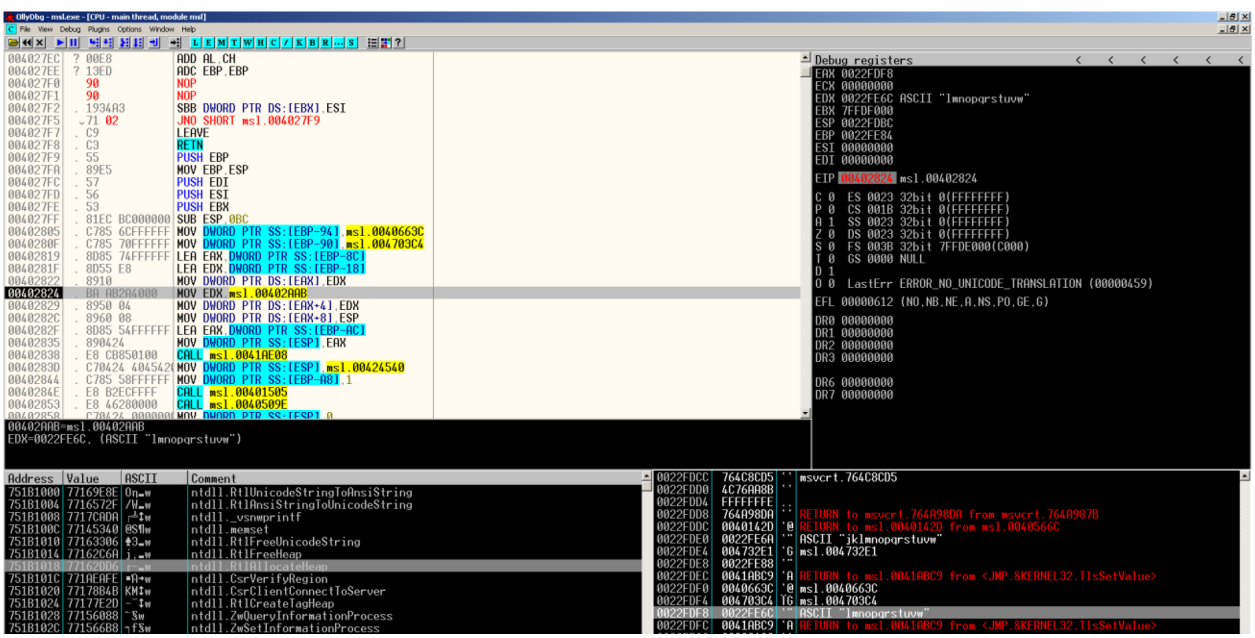


Figure 95: OllyDbg series of ASCII characters loaded in memory endlessly

So, the current solution is not to take the specific jump.

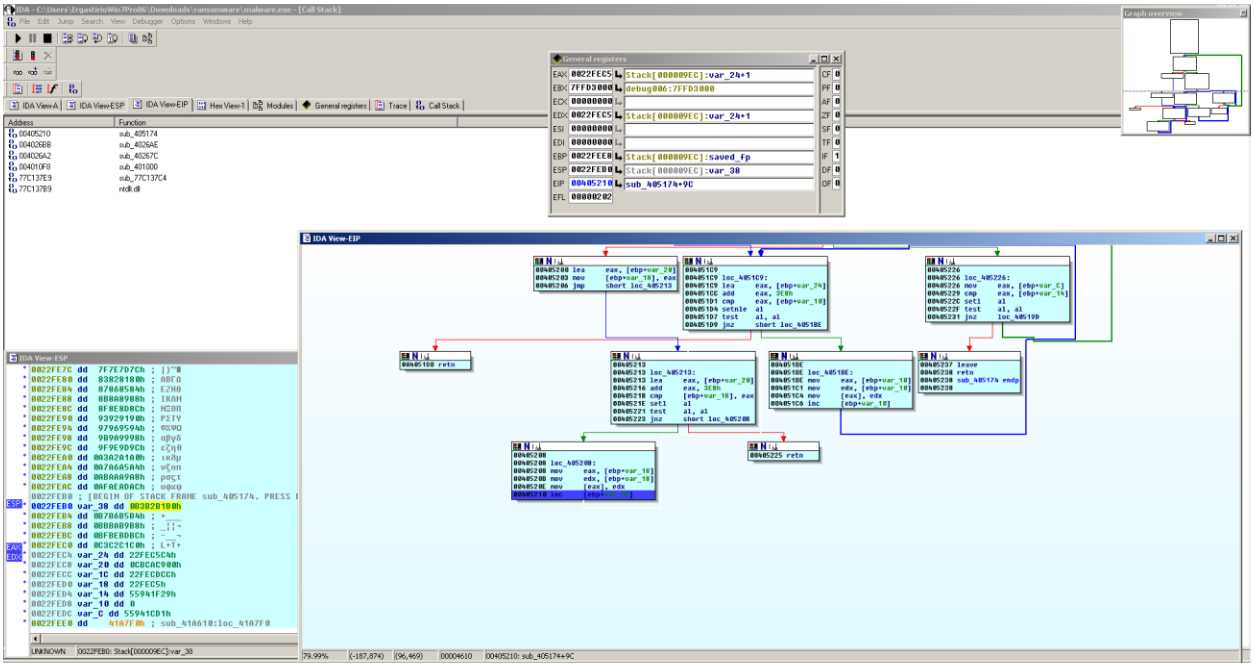


Figure 96: IDA graph view, nonstop loop subfunction text.405208

6.2 Manipulation of CPUID instructions

CPUID is an instruction-level detection method and these kinds of methods are really hard to detect, as long as in order to trap on every execution of CPUID, instructions should be executed step by step (which is really slow and almost impossible) or instrument the target program. Using instrumentation, then anti-instrument techniques might also defeat.

On the subject malware, searching for CPUID occurrences reveals that they are being called four times in the .text section. Exploring them, reveals that the malware author is using difference appliances and techniques with them and reuse them by calling the mother functions several times on his checks.

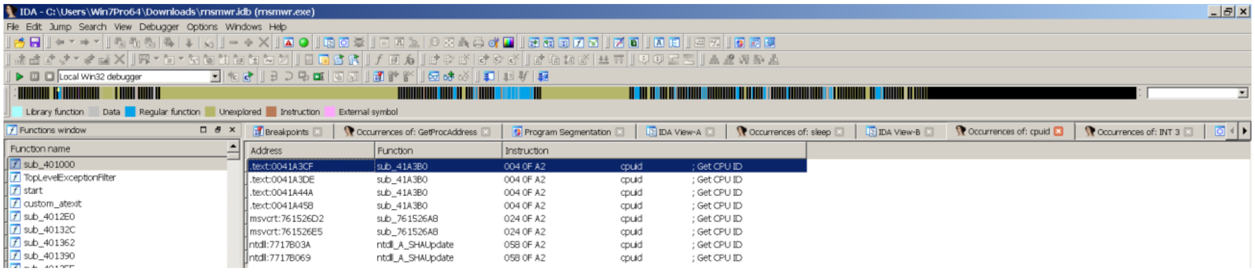


Figure 97: IDA View, CPUID instructions all occurrences

When CPUID instruction is executed with EAX=0 as input, *xor eax, eax* brings the same result, the return value will increase EAX by 1. On the figure 102 the first check CPUID check is doing this check.

In addition, when CPUID instruction is executed with EAX=1 as input, the return value describes the processors features. The 31st bit of ECX or EDX on a physical machine will be equal to 0, but on a guest VM it will equal to 1. On the figure 102 the second check CPUID check is doing this check.

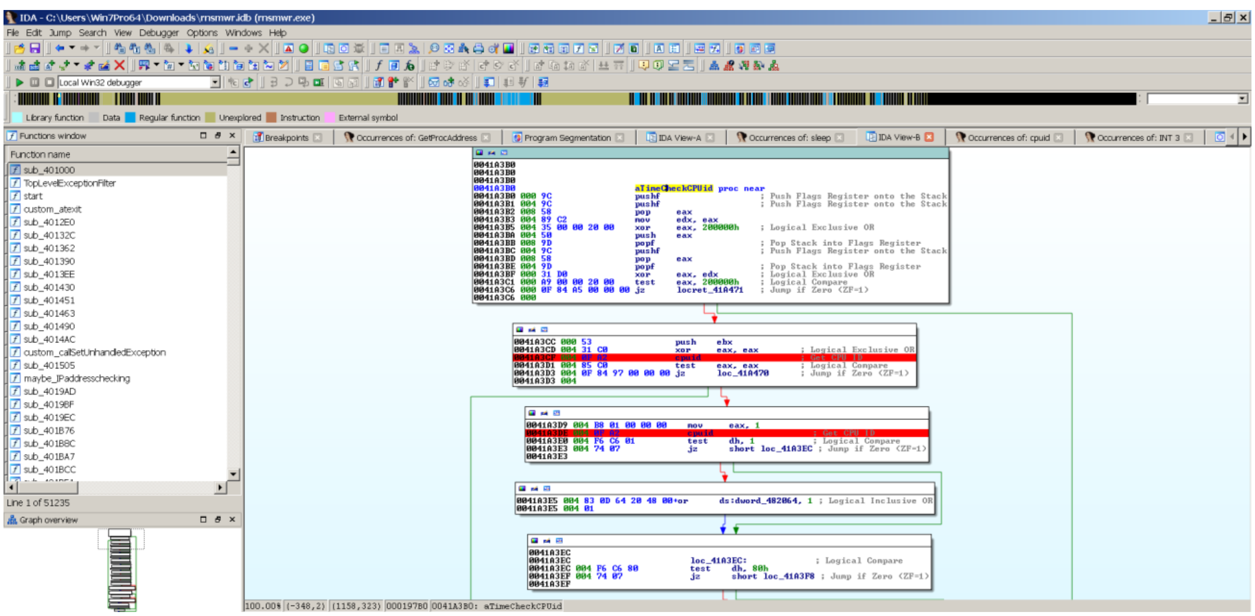


Figure 98: IDA View, CPUID instructions, using eax = 0 and eax = 1 as parameter

Furthermore, more methods are being used with CPUID instruction. When CPUID executes with EAX set to 80000000, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

When CPUID instruction with EAX=0x80000001 as input, requests to Extended Processor Info and Feature Bits. This returns extended feature flags in EDX and ECX. The EDX's Bit 4 is a timestamp counter and Bit 2 is debugging extensions. In the subject malware case, the counter will be measuring the time in case of breakpoint of debugging is active.

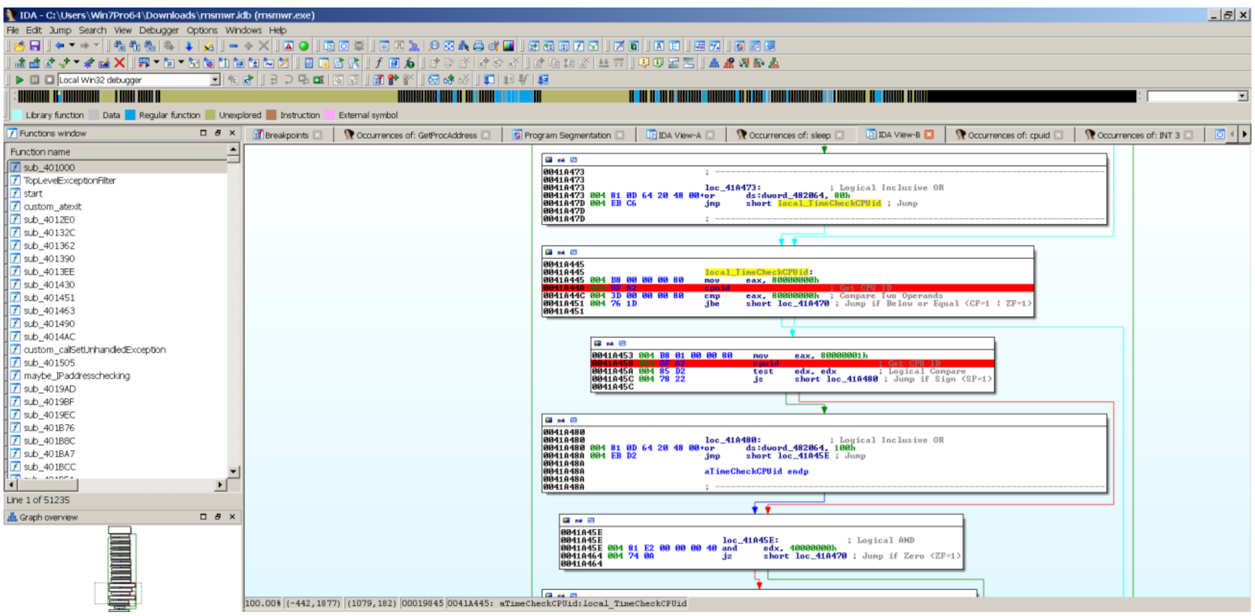


Figure 99: IDA View, CPUID instructions, using eax = 0x80000000 and eax = 0x80000001 as parameter

NOP-ing the CPUID instructions is the again the answer for most of the cases. Defeating results that come from asm instruction level, seems to be impossible but there is always a solution. To change the CPUID results of the target virtual machine from host perspective, is possible via the VMware's configuration file .vmx, that gives the host machine the opportunity to modify CPUID and CPU features. This is because every time your virtual machine fetches a CPUID instruction and wants to execute it, a VM-Exit happens and now hypervisor passes the execution to VMM.

At the .vmx configuration file, the following line should be added, to counter the figure's 106 technique. Keep in mind to put the line at the end of the file when the VM is not running.

cpuid.1.eax="0---:----:----:----:----:----:----:----"

Table 6: VMX configuration file line addition CPUID and EAX manipulation

Also at the .vmx configuration file, the following line should be added, to counter the figure's 107 technique. Keep in mind to put the line at the end of the file when the VM is not running.

cpuid.80000001.edx="0000:0000:0000:0000:0000:0000:0000:0000"

Table 7: VMX configuration file line addition CPUID and EDX manipulation

Anti-VM detection with python in IDA Pro

The python script that it is attached on **Appendix I** will scan the assembly code in IDA-Pro and highlight with green color the instructions corresponding to Anti-VM techniques. All the techniques have been already mentioned in the previous section of **Static code Analysis/Anti-VMware.** By using the script, there are several instructions that are being searched in the binary, such as SGDT, SLDT, SMSW, STR, IN and CPUID.

On the following two figures, the CPUID instruction that we have already analyzed, it is highlighted with green color.

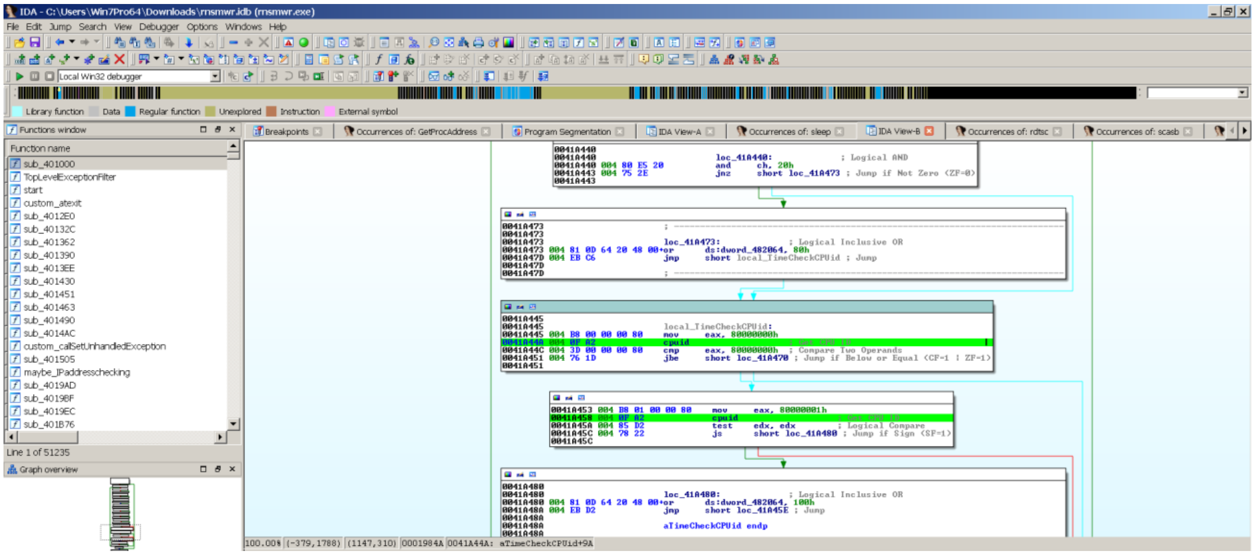


Figure 100: IDA Pro, graph view, CPUID highlighted green

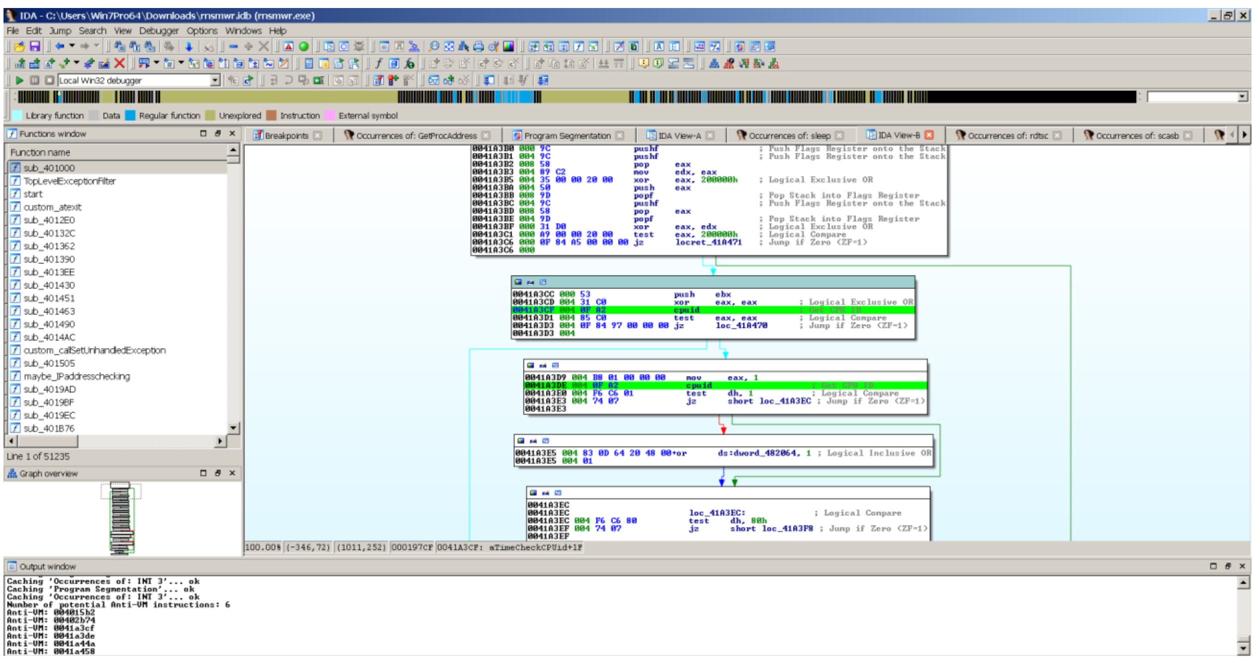


Figure 101: IDA Pro, graph view, CPUID highlighted green2

Except the four CPUID instructions, two more IN instruction have been characterized as potentially Anti-VM technique and been highlighted as red. On the following two figures, the command IN EAX is the suspicious one but unfortunately there are a lot of bad disassembly code as prefix. As a result, the functionality of the showed assembly cannot be clarified.

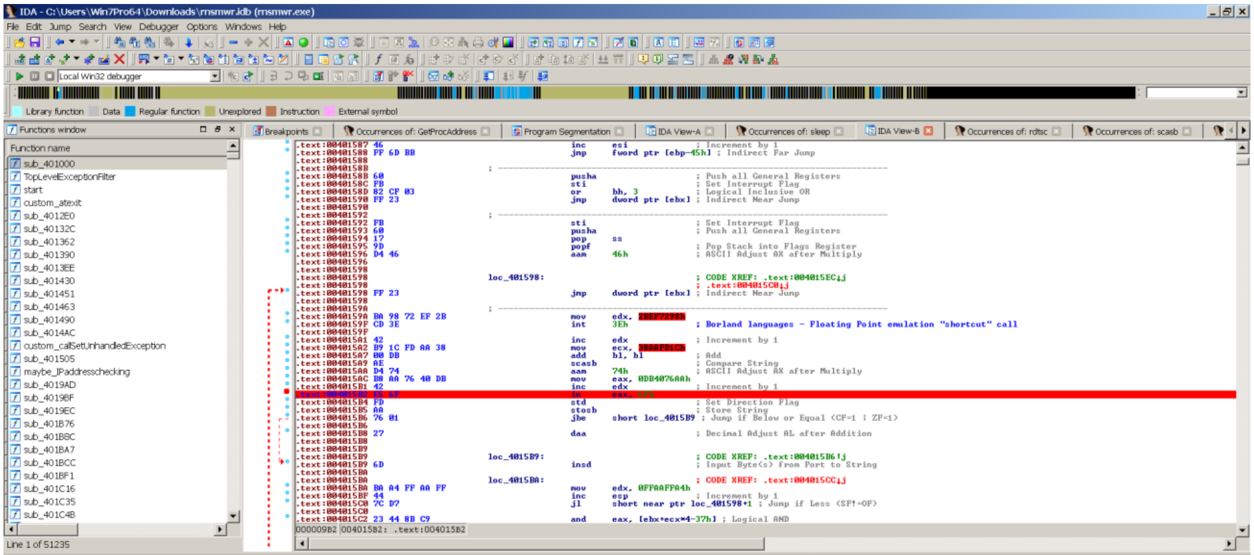


Figure 102: IDA Pro, text view, IN highlighted red

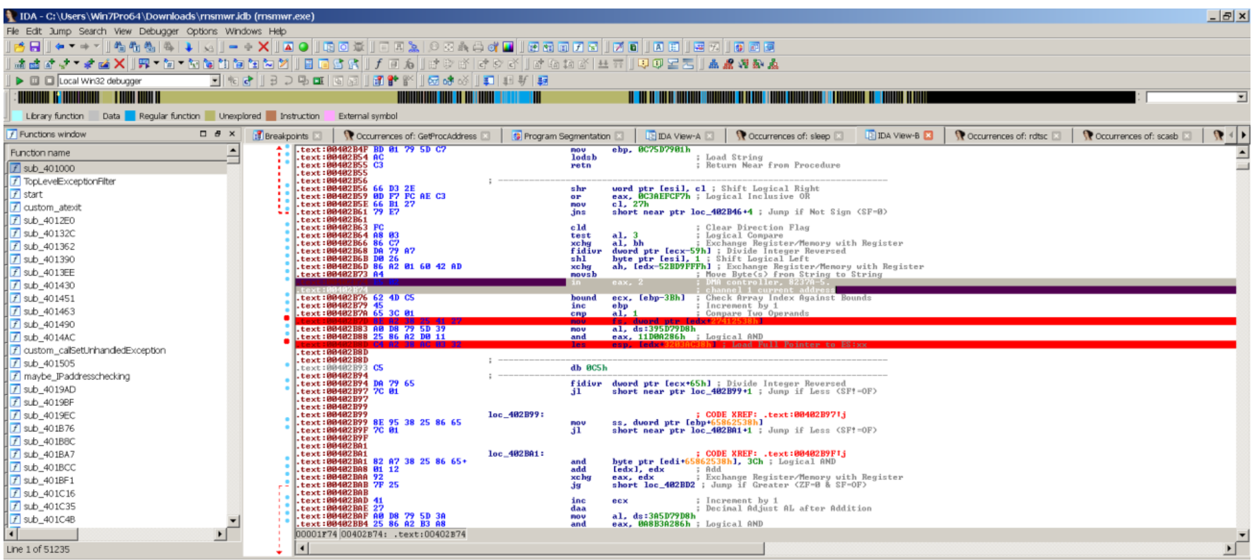


Figure 103: IDA Pro, text view, IN highlighted red2

6.3 Interrupts on Debugging

During the dynamic analysis of the code, some interrupts have been revealed, that were not added breakpoints. So, an INT 3 technique was detected. INT 3 is the software interrupt used by debuggers to temporarily replace an instruction in a running program and to call the debug exception handler. On other words it is a basic mechanism to set a breakpoint. The opcode for INT

3 is 0xCC. Whenever you use a debugger to set a breakpoint, it modifies the code by inserting a 0xCC. In addition to the specific INT 3 instruction, an INT immediate can set any interrupt, including 3 (immediate can be a register, such as EAX). The INT immediate instruction uses two opcodes: 0xCD value.

On the subject malware, four occurrences were found with the 0xCC opcode. On the following figure the traps of the debugger are being presented.

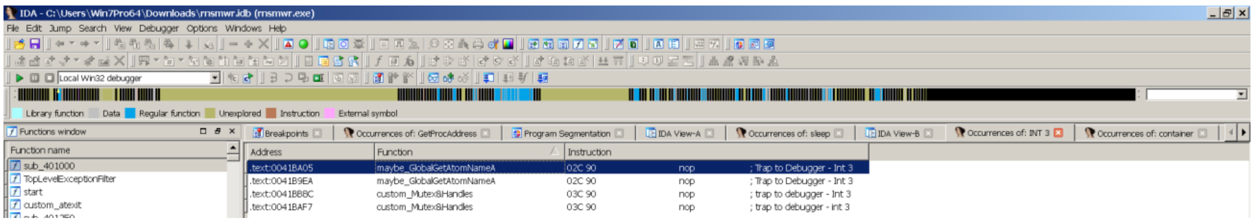


Figure 104: IDA Pro view, INT 3 occurrences

If a 0xCC byte is found, it knows that a debugger is present. This technique can be overcome by using hardware breakpoints instead of software breakpoints or manually by modifying the execution path with the debugger at runtime. On the following screenshot, we manually NOP-ed out the INT 3 fake instruction.

Figure 105: IDA Pro graph view, INT 3 trap to debugger NOP-ed

6.4 Thwarting Stack-Frame Analysis

Advanced disassemblers can analyze the instructions in a function to deduce the construction of its stack frame, which allows them to display the local variables and parameters relevant to the function. This information is extremely valuable to a malware analyst, as it allows for the analysis of a single function at one time, and enables the analyst to better understand its inputs, outputs, and construction.

However, analyzing a function to determine the construction of its stack frame is not an exact science. As with many other facets of disassembly, the algorithms used to determine the construction of the stack frame must make certain assumptions and guesses that are reasonable but can usually be exploited by a knowledgeable malware author.

The call and jmp instructions are not the only instructions to transfer control within a program. The counterpart to the call instruction is retn. The call instruction acts just like the jmp instruction, except it pushes a return pointer on the stack. The return point will be the memory address immediately following the end of the call instruction itself.

As call is a combination of jmp and push, retn is a combination of pop and jmp. The retn instruction pops the value from the top of the stack and jumps to it. It is typically used to return from a function call, but there is no architectural reason that it can't be used for general flow control.

When the retn instruction is used in ways other than to return from a function call, the most disassemblers are left in the dark. The most obvious result of this technique is that the disassembler does not show any code cross-reference to the target being jumped to. Another key benefit of this technique is that the disassembler will prematurely terminate the function.

On the following figure a short jump is taken place, with the return pointer being abusive. Specifically, there is a hidden code following if we switch to text mode in IDA.

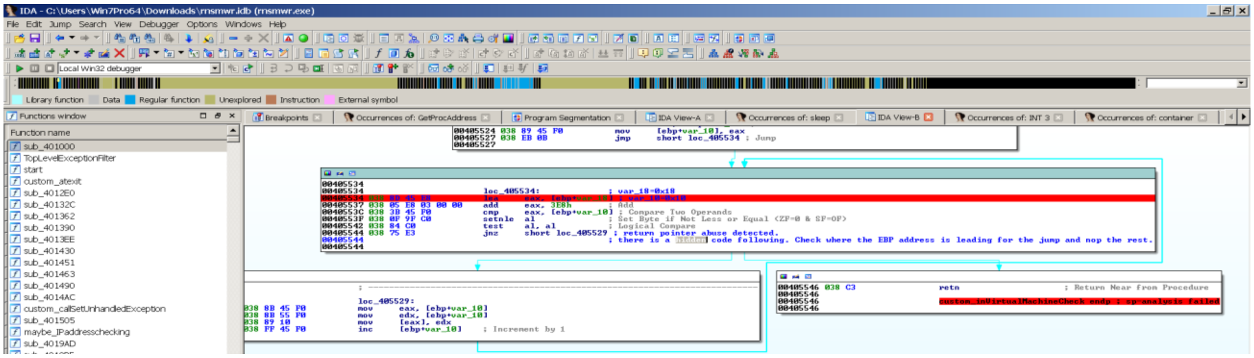


Figure 106: IDA Pro, graph view, sp-analysis fail return pointer abuse

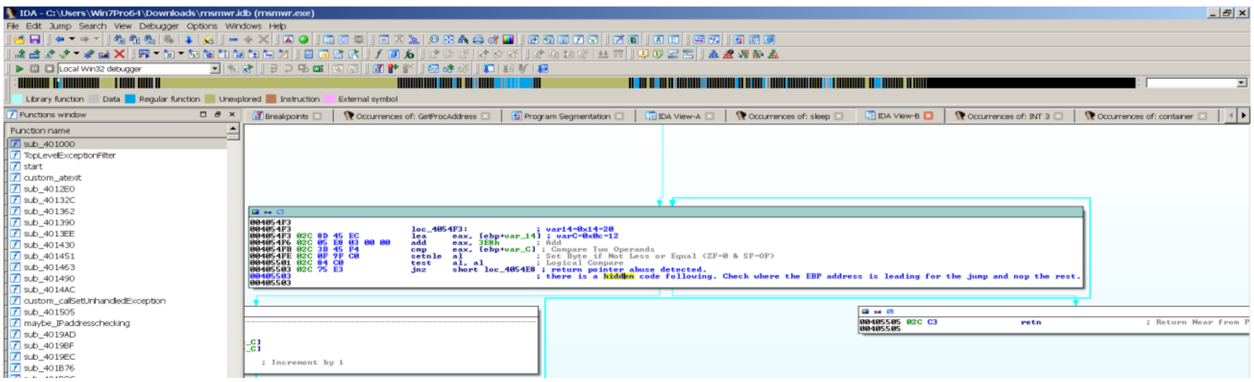


Figure 107: IDA Pro, graph view, sp-analysis fail return pointer abuse2

In order to resolve this sp-analysis fail error and disassemble the assembly correctly the EBP address should be followed from the jump and then the rest of the code should be NOP-ed.

6.5 Escaping the control of debuggers by Sleeping

One of the simplest ways to escape from the control of a debugger is for a process to execute another copy of itself. Typically, the process will use a synchronization object, such as a mutex, to prevent being repeated infinitely. The first process will create the mutex, and then execute the copy of the process. The second process will not be under the debugger's control, even if the first process was. The second process will also know that it is the copy since the mutex will exist.

On the following figure, there are several occurrences where the sleep function is messing, but actually the call of the function is being made at the addresses 0042222E, 0041AD0B and 0041B6A7.

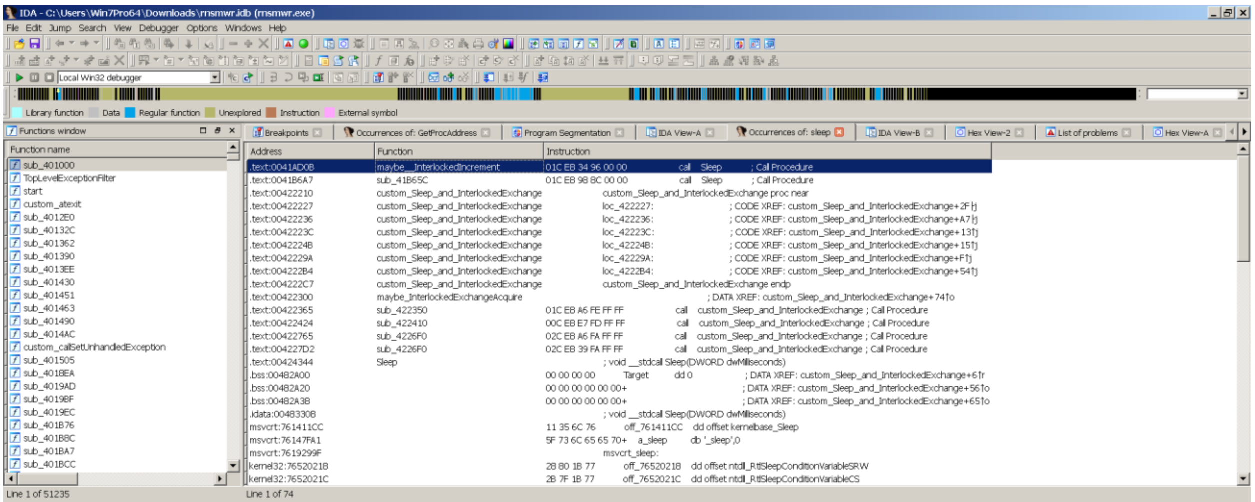


Figure 108: IDA View, sleep function all occurrences

It is quite common to see the use of the kernel32.Sleep() function, instead of the kernel32.WaitForSingleObject() function, but this introduces a race condition. The problem occurs when there is CPU-intensive activity at the time of execution. This could be because of intentional delays in the second process.

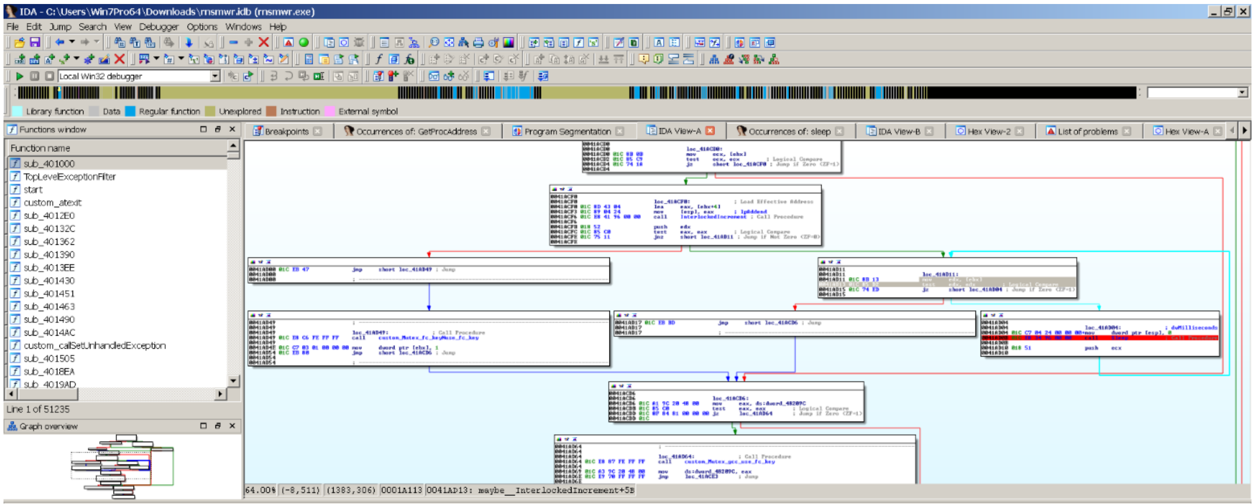


Figure 109: IDA View, sleep function in InterlockedIncrement thread mutex

On the following figure, the parameter of the function is a double word integer that gives the input of time sleep in milliseconds.

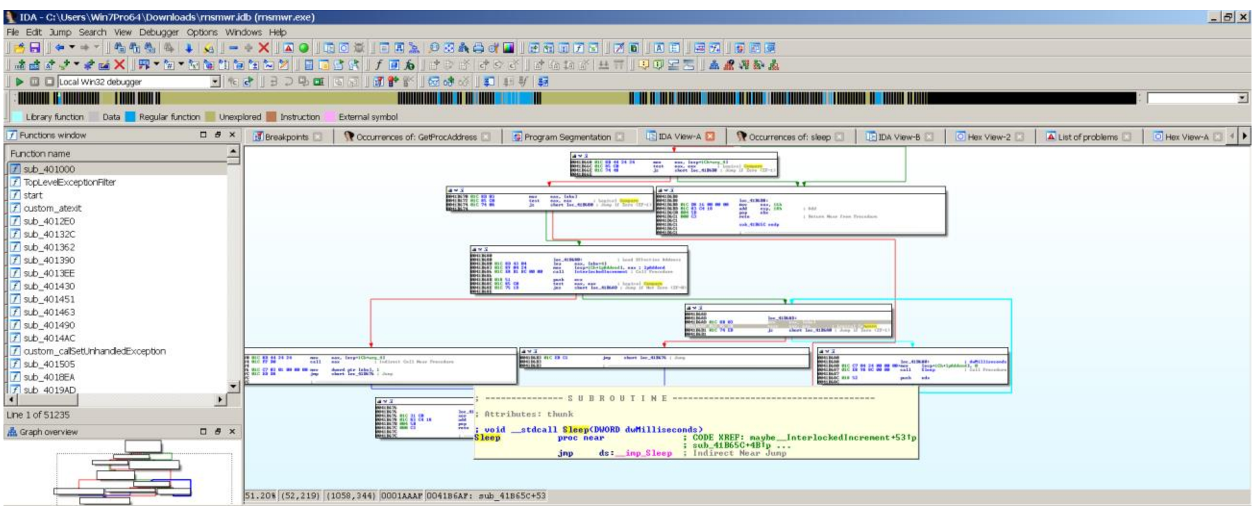


Figure 110: IDA View, sleep function millisecond parameter

6.6 Anti-analysis technique terminating the process

exit Function

In result of the above mentioned techniques, the malware author terminates the process of the malware, in case of detection of VME, debugging presence, execution manipulation, any false validation of the time and the IP address of a specific subnet.

On the following figure, the list of exit function occurrences is being presented.

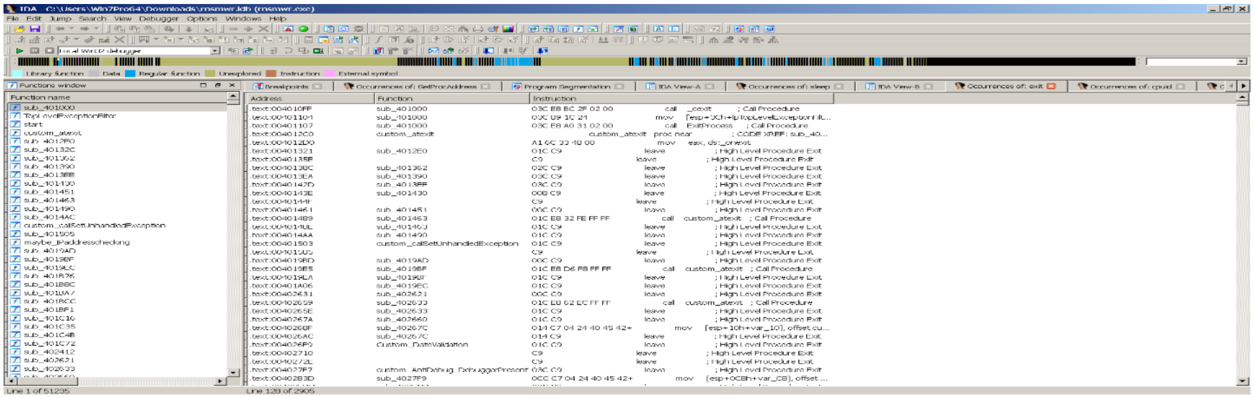


Figure 111: IDA View, exit function all occurrences

In addition, a custom function seems to be written by the malware author, that also terminated the execution of the binary. In the following figure, the address *.text:0042454D* is completely unlinked and without references. It is assumed that this function is also dynamically being called during the execution of the malware, so it should be an exit after a sophisticated check.

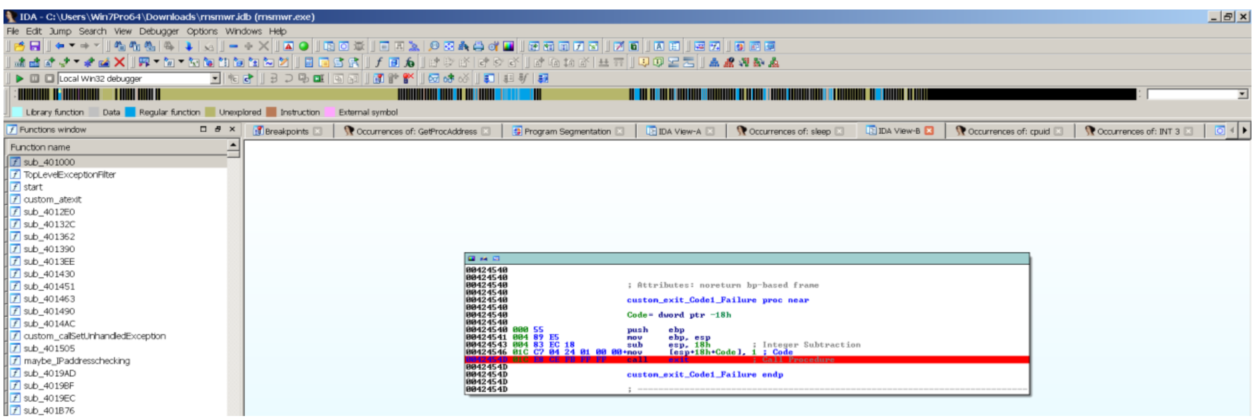


Figure 112: IDA graph mode, custom exit function

abort Function

Nevertheless, except the common exit function, the malware author is using abort function in order to crash the execution flow. More specifically, the abort does not return control to the calling process. By default, it checks for an abort signal handler and raises SIGABRT if one is set. Then abort terminates the current process and return an exit code to the parent process.

On the following figures, the abort function is being presented, after conditional jumps, custom switch cases and indirect call procedures.

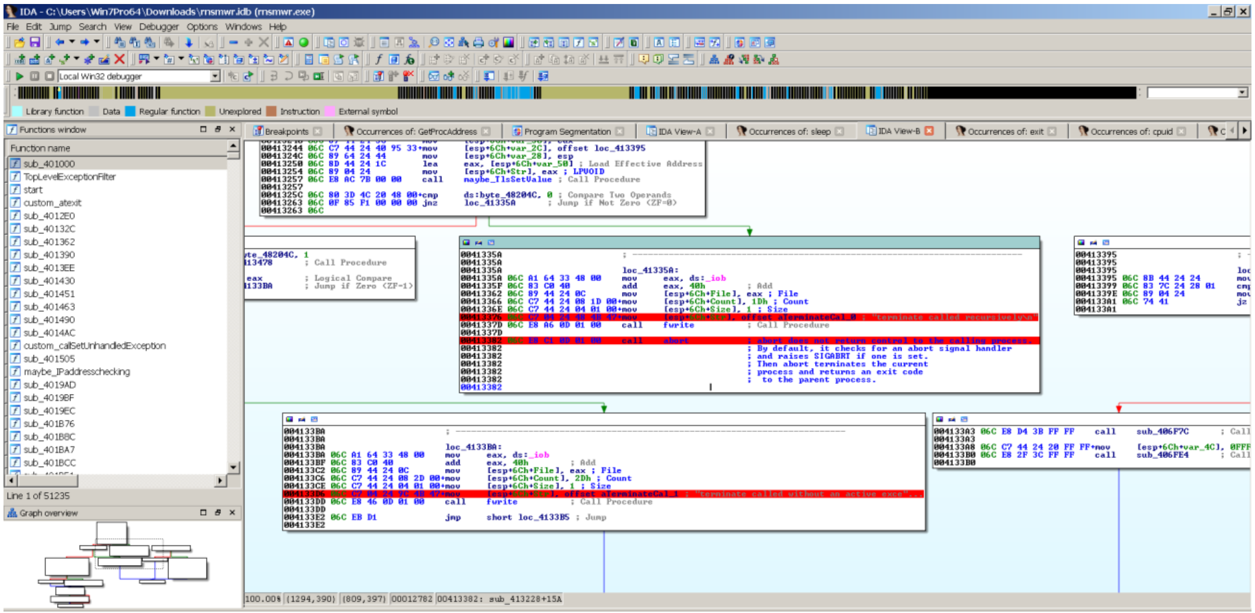


Figure 113: IDA graph mode, custom abort function

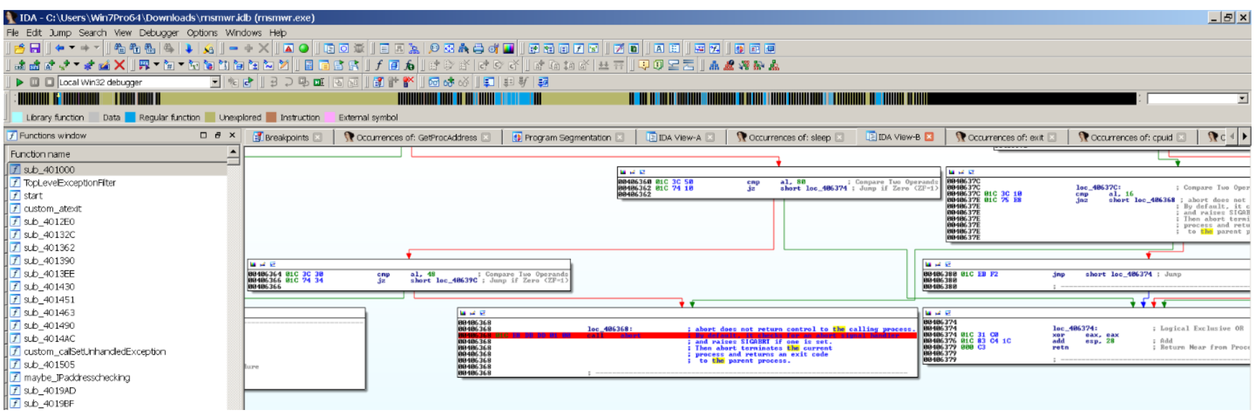


Figure 114: IDA graph mode, conditional jump abort function

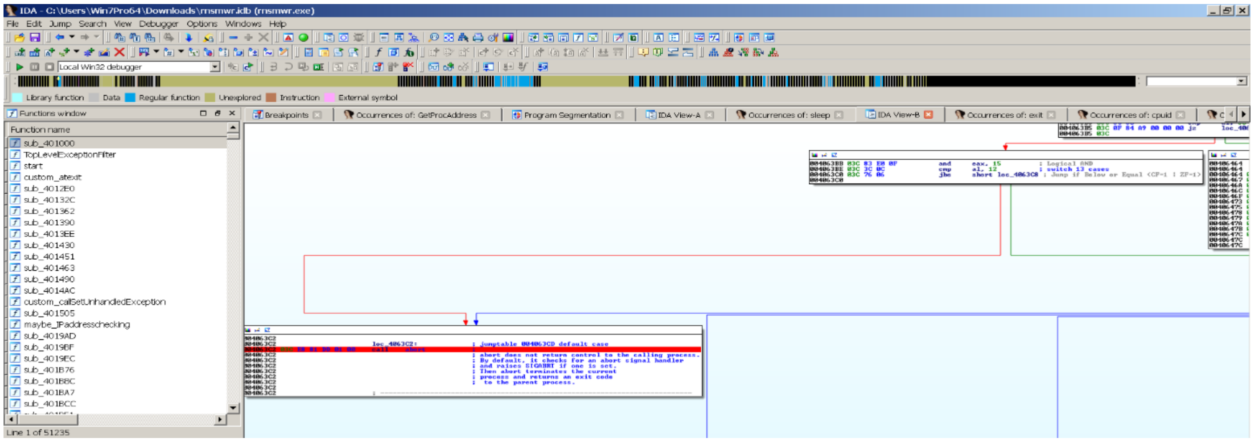


Figure 115: IDA graph mode, switch case abort function

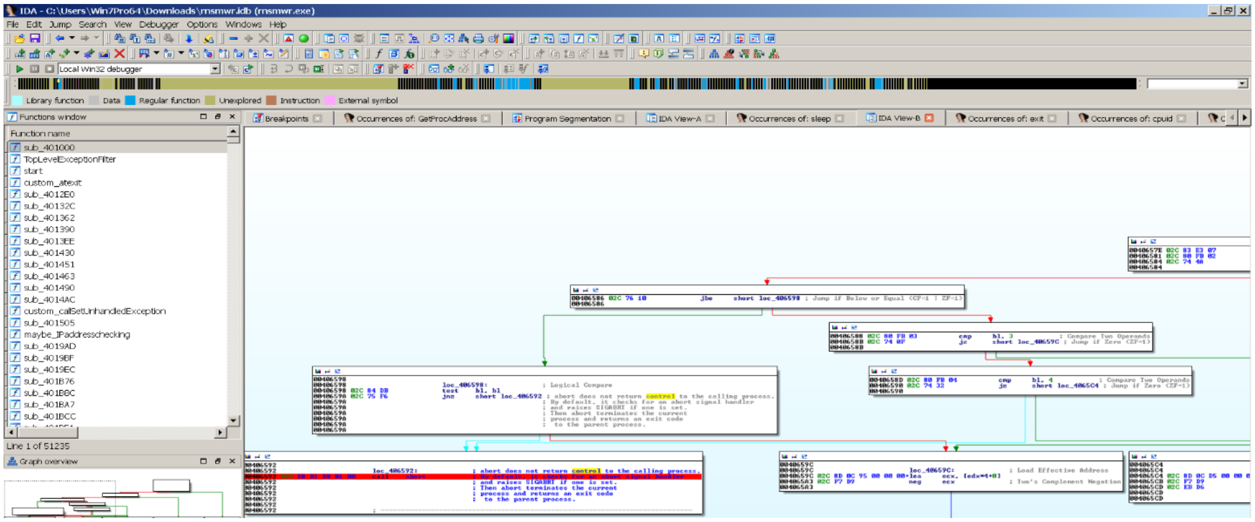


Figure 116: IDA graph mode, logical comparison abort function

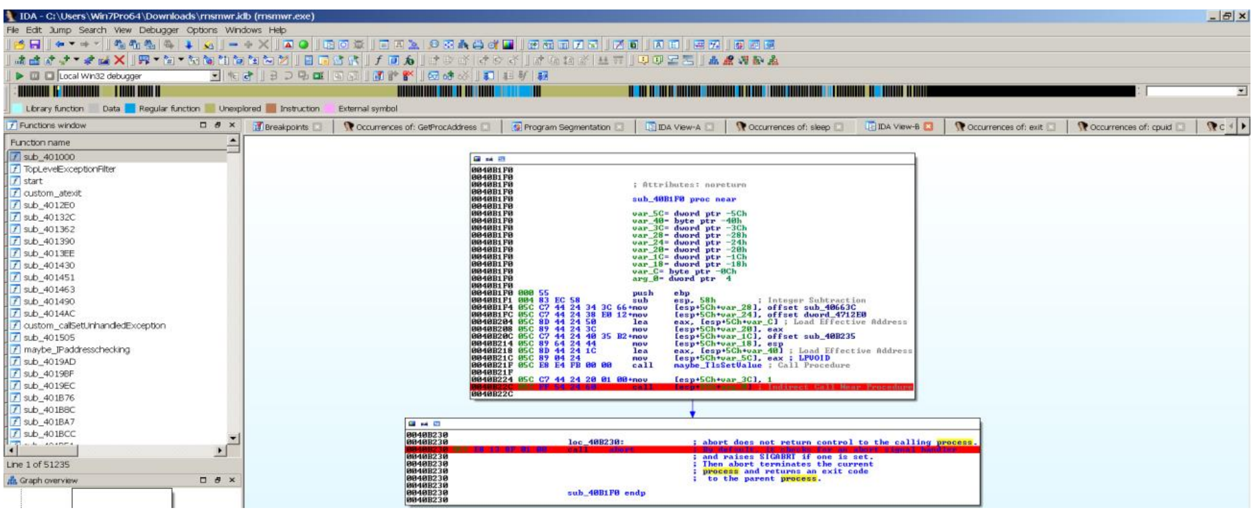


Figure 117: IDA graph mode, TLS check abort function

6.7 Antivirus Evasion

In order to achieve evasion from some antivirus software using this methodology, the following steps need to be implemented:

1. Allocate a location to place the TLS Directory structure defined as *_IMAGE_TLS_DIRECTORY32.*
2. Fill in the addresses for callback functions (our supposed constructors).
3. Allocate a location to place the code for the TLS callback functions.
4. Write code that uninstalls the initial hooks from the EP or *ZwTestAlert.*
5. Modify the PE Header's DataDirectory to use the newly created TLS Directory.

ZwTestAlert

The above ZwTestAlert function tests whether the current thread has been alerted (and clears the alerted flag). It also enables the delivery of queued user APCs. NextDisableThreadLibraryCalls disables the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications for the DLL. By disabling the notifications, the DLL initialization code is not paged in because a thread is created or deleted, thus reducing the size of the application's working code set. This use of DisableThreadLibraryCalls increases invisibility for the injected DLL.

6.8 Anti-Dump Trick "Header Erase"

The Anti-Dump trick is erasing the header of the process running, so the dumping techniques will fail, as long as, no header to identify exists, used as anti-reversing trick.

More specifically, we start calling the function "GetModuleHandleA", using the parameter 0, in order to handle the same process. After that, using the function "VirtualProtect" we can make the header of a file writable. Keep in mind that headers of files are usually read-only, because the header exists on the memory region. In continuous, with XORing the registers, the memory is being filled with zero bytes.

7

7. Conclusion

In conclusion the procedure of the code decryption during runtime will be presented, with some specific binary's addresses, where these actions are taken place. From Surface Analysis and the examination of the binary's strings, they are for sure encrypted and obfuscated. During the runtime, the used ones are dynamically being decrypted.

7.1 Encryption and Decryption procedure

Large parts of the subject malware binary's code are encrypted. It is already presented that the disassembler either fails to disassembly due to anti-disassembly techniques but also due to encrypted parts of code inside the binary. Some of them, they are dynamically being loaded, because as if figured out on the surface analysis, the .text section is writable – not read only. The first part of binary that is being detected as encrypted is on the .text:402B25 address, where it starts with the value 0x1111111111111111.

At start the infected machine should meet some circumstances. Except the specific time range of execution, a normal machine must be assigned in a specific subnet with a specific IP address. On the following screen at the binary's address .text:00405011, the call of API function gethostbyname is detected.

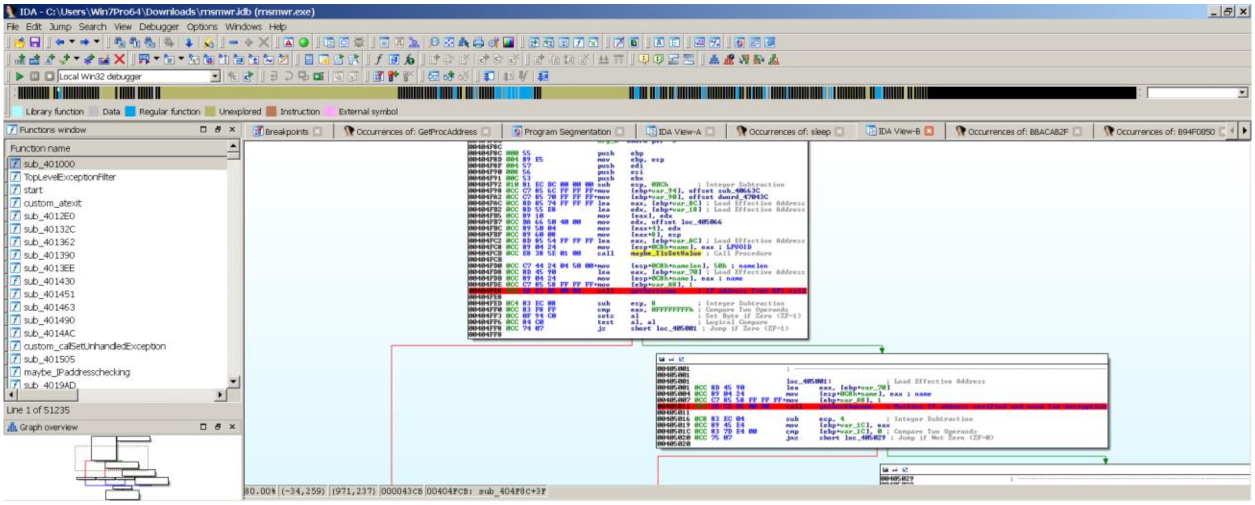


Figure 118: IDA graph view, gethostbyname function API call

In continuous, on the following figure the part of the code that gets the IP address of the current machine is being detected and hashing it. The function starts at binary's address .text:004018EA.

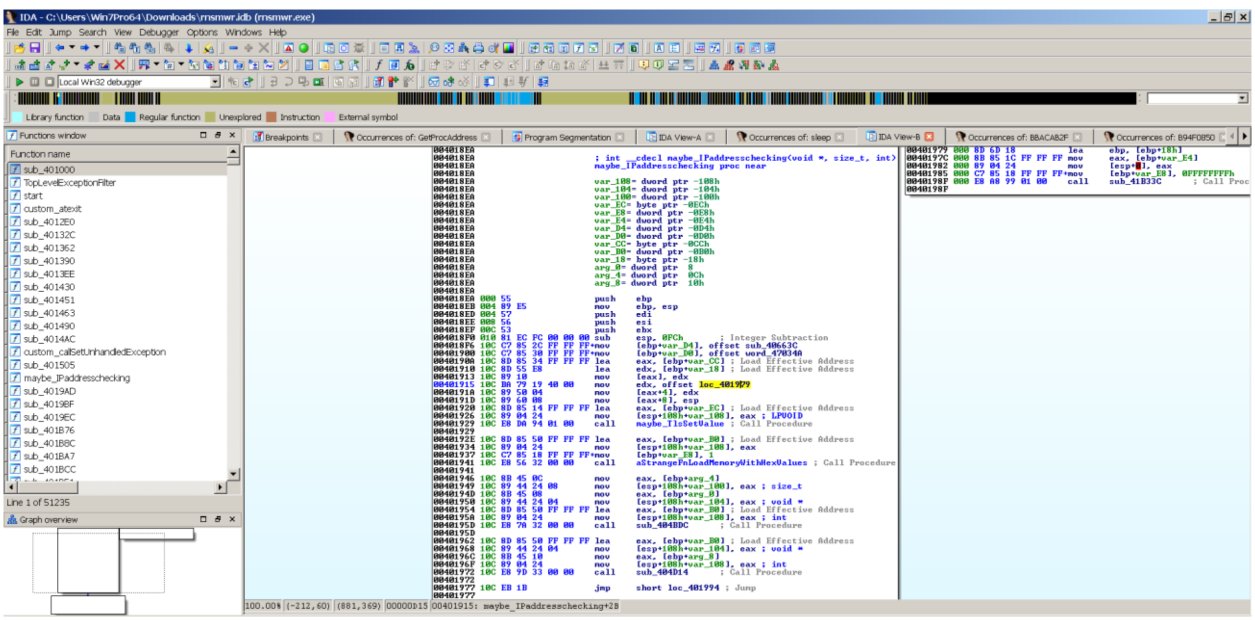


Figure 119: IDA graph view, custom function for hashing the IP address

Parts of this result is used to verify the IP. More specifically, it is compared in two pieces. On the following figure we detect at binary's address .text:0040297E a comparison with the value 0xB94F0850 and at binary's address .text:00402988 a comparison with the value 0xBBACAB2F.

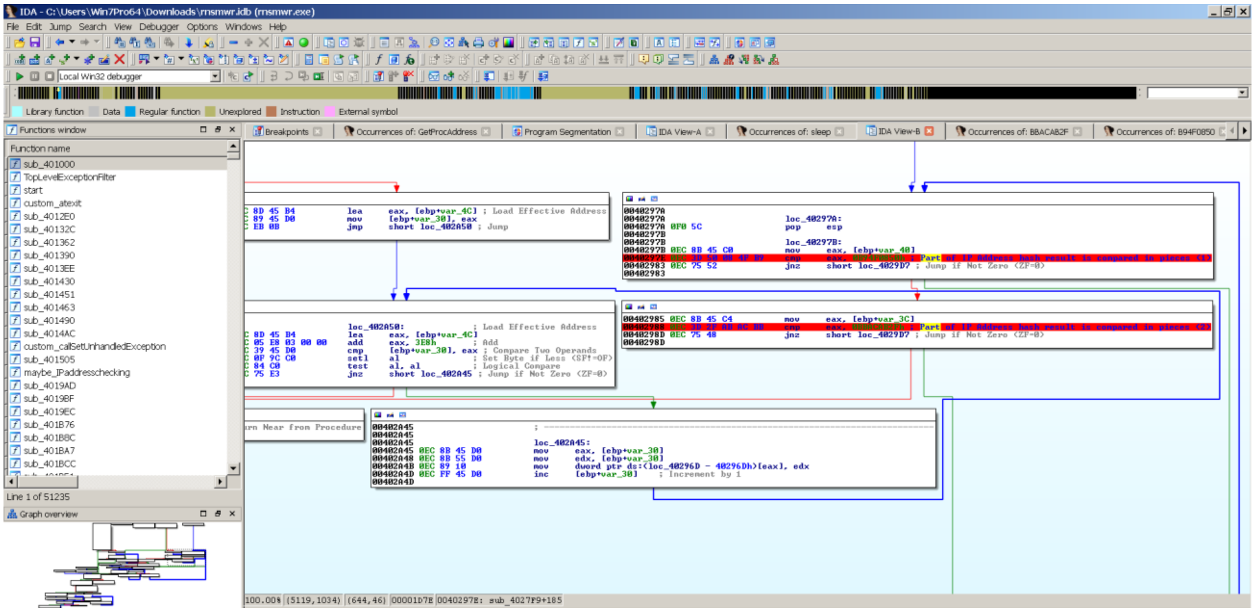


Figure 120: IDA graph view, custom function for IP validation in two pieces

Keep in mind that in section **2.4.3 VMware Workstation Setup/Virtual Network editor**, a provision is being made, so the IP is correctly configured in the right subnet. It would be hard to patch the return bytes of this function during execution each time, so we bypass this check by configuring correctly the virtual network.

The malware author used a custom sophisticated technique, where some part of the result is used to decrypt part of the next code. The IP Address that gives resulting hash is 10.1.210.*. The star symbol stands for all possible values, because only the first 3 bytes are being used. The result of the IP address hashing is the hex value 49C60C2B94F0850BBACAB2F2538A286. This value must be delaminated in four parts of 4 bytes, like the following structure: 49C60C2 B94F0850 BBACAB2F 2538A286.The first part, last 4 bytes in endian, the 0x2538A286 hex value is used to decrypt the first part of the encrypted code in the binary.

The encryption and decryption have been done with the XOR procedure using a 4 byte key. As it mentioned before, at binary's address .text:402B25 where the hex value 0x1111111111111111 exists, the XOR is done using the key 0x2538A286.

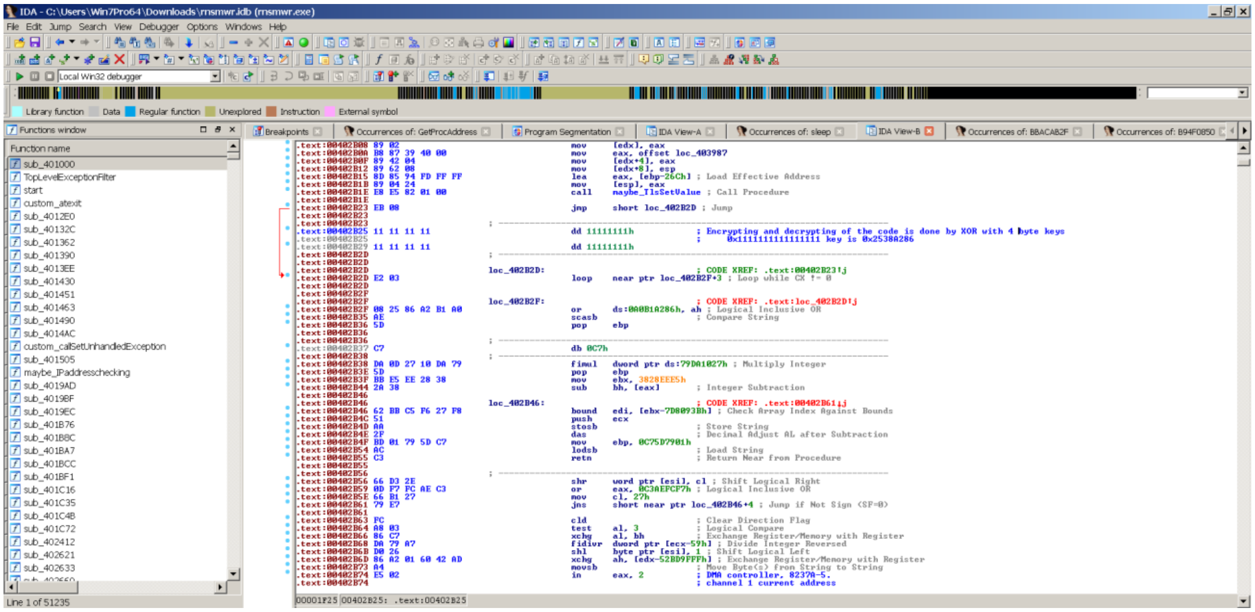


Figure 121: IDA graph view, custom en/decryption XOR function with 4 byte key (1)

Although on the following encrypted parts of the code, another key is being used. Specifically, at binary's address .text:403A11where the hex value 0x2222222222222222 exists, the XOR is done using the key 0x2387645A.

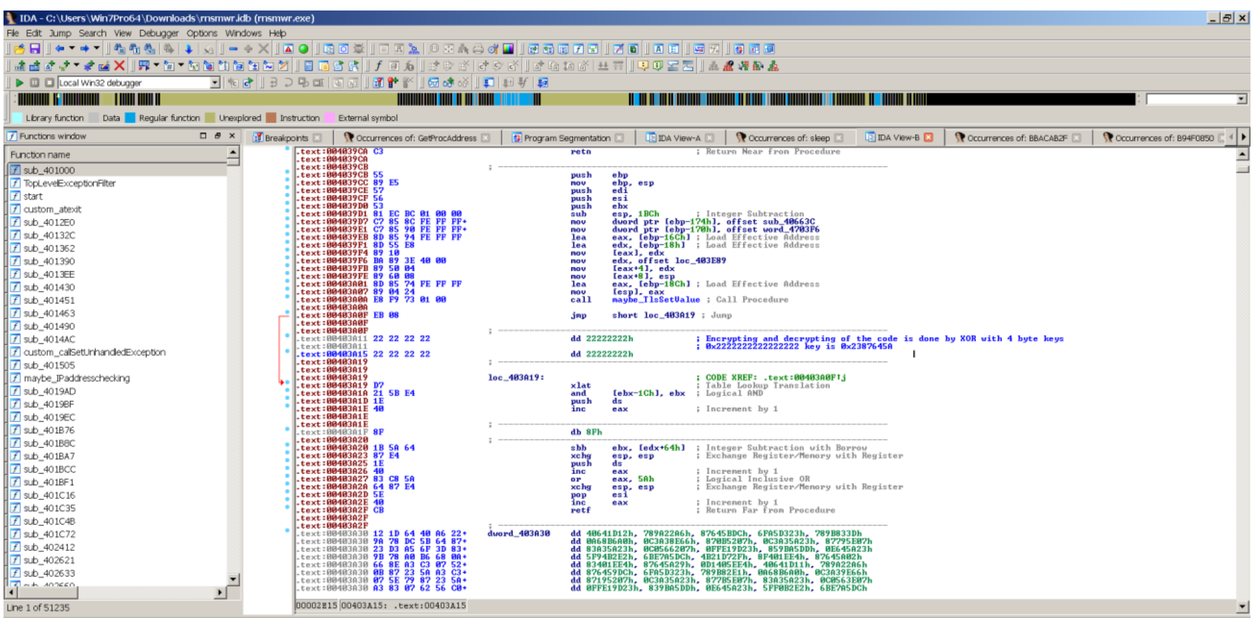


Figure 122: IDA graph view, custom en/decryption XOR function with 4 byte key (2)

Also, at binary's address .text:401721 where the hex value 0x3333333333333333 exists, the XOR is done using the key 0xA345FFE0.

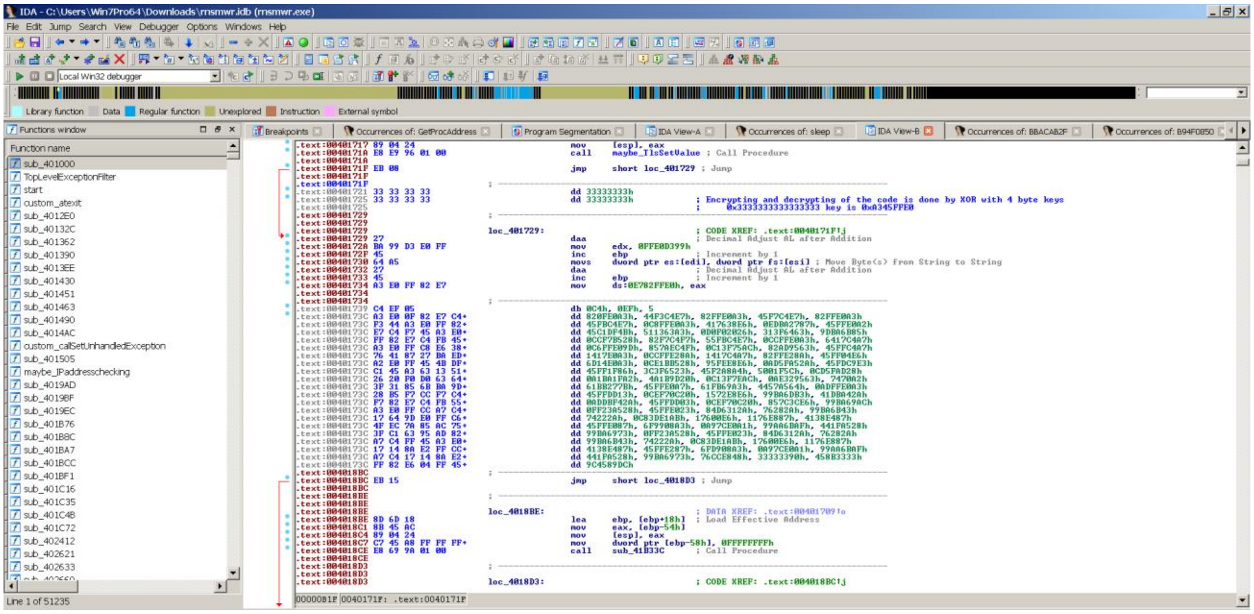


Figure 123: IDA graph view, custom en/decryption XOR function with 4 byte key (3)

At last, at binary's address .text:401550 where the hex value 0x4444444444444444 exists, the XOR is done using the key 0xFF44FFAA.

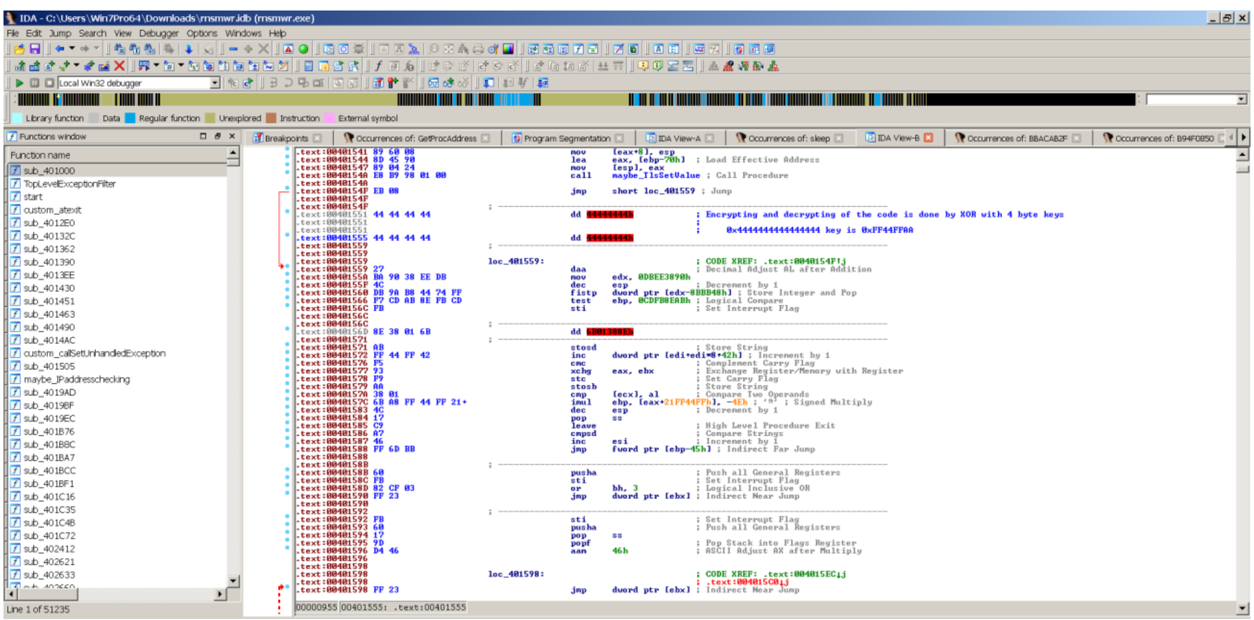


Figure 124: IDA graph view, custom en/decryption XOR function with 4 byte key (4)

Using OllyDbg, the CryptGenRandom API call has been detected and analyzed. This function leads to the above mention results. Note that, CryptGenRandom is a cryptographically secure pseudorandom number generator function that is included in Microsoft CryptoAPI.

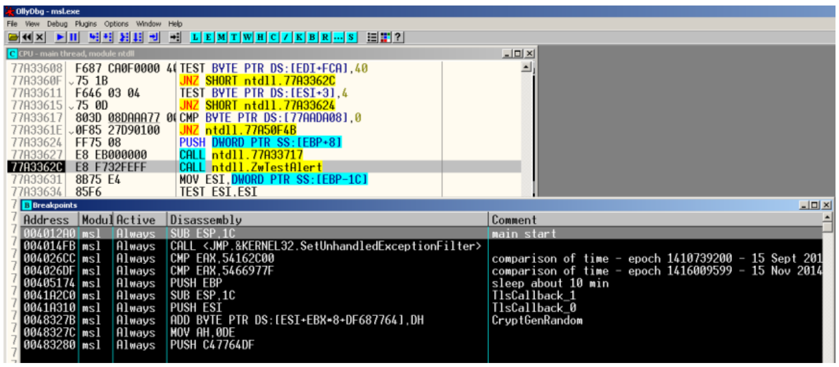


Figure 125: OllyDbg Breakpoints on CryptGenRandom API call

7.2 Malware deflection

Malware authors and specially ransomware authors, are creating mutexes[81] in order to check if a machine is already infected. If the analyst locates the hard-coded mutex name, can emulate it and fool the ransomware that the machine is already infected.

7.3 The smart-dumb alternative way to deflect the Ransomware

Base64 encoding code and strings

Base64 is an encoding scheme originally designed to allow binary data to be represented as ASCII text. Widespread in its use, Base64 seems to provide a level of security by making sensitive information difficult to decipher. In reality, the use of Base64 provides a significant

[81] Mutexes are global objects that coordinate multiple processes and threads. In the kernel they are called mutants. Keep in mind that mutexes are usually hard-coded names.

advantage to attackers while providing minimal benefit to defenders. The use of Base64 can result in the disclosure of passwords, bypass of data leakage protection systems and can even be used to create a one click, obfuscated and self-contained cross site scripting attacks. [82]

In malware analysis, is another well-known encoding technique utilized by malware authors. Keep in mind that Base64 is from the MIME standard, which recognized the need for converting binary to text for email attachments. Base64 has a set of only 64 characters (as the name describes), and a standard for translating data within this limited set.

The MIME Base64 “alphabet” looks like this:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=

Note that due to Base64 being a smaller set of characters, encoded data is often “longer” than encoded data. Typically, we should expect an increase of about 33% (or 4 Base64 encoded characters for every three decoded characters), give or take. Furthermore, attackers can also define their own Base64 alphabets, which make standard conversion techniques useless.[83]

Identification and Decoding Base64

The characteristics that make up a Base64 encoded string are fairly simple; it will typically contain letters (A-Z and a-z), numbers (0-9) and the characters “/”, “+” and “=” where the equal sign, if found, will always be found at the end of the string. Base64 strings usually contain a multiple of 4 characters (e.g. 4, 8, 12, 16, etc.). In such cases, the minimum size for a Base64-

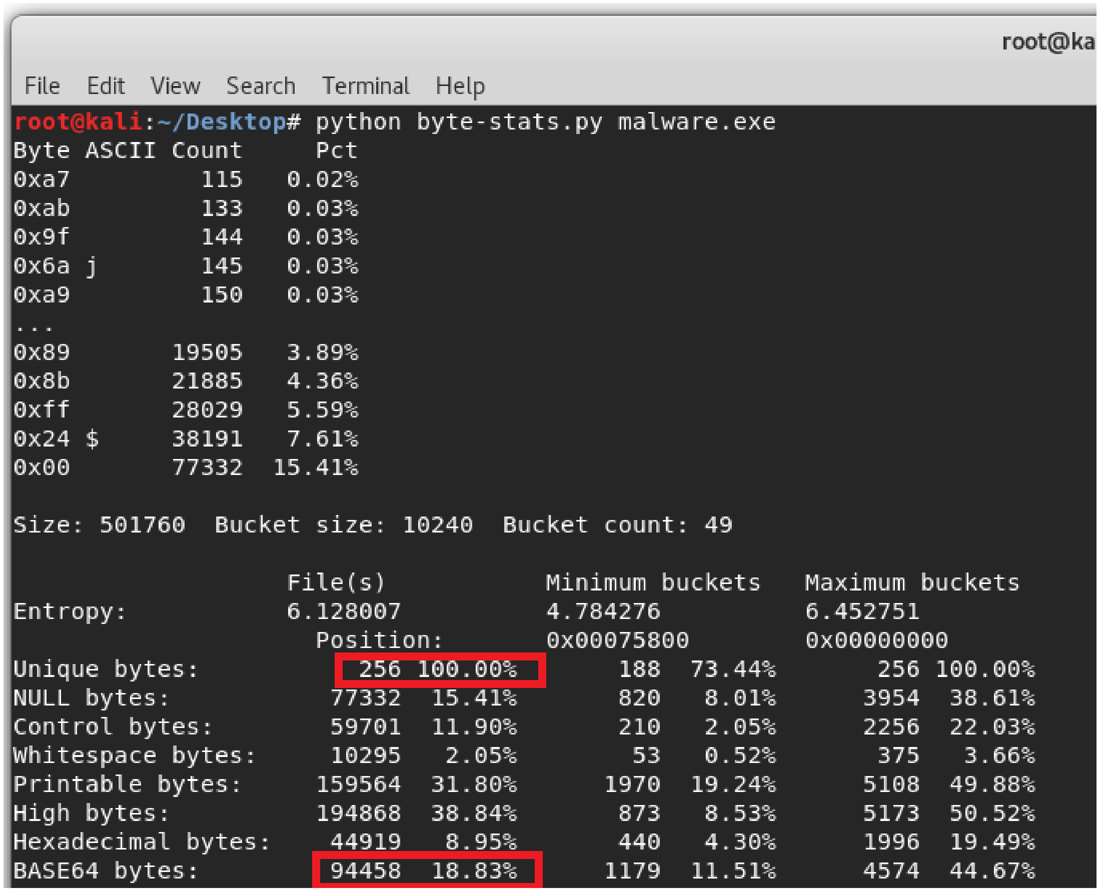
---

[82] Fiscus Kevin, SANS Institute (2011, April), Base64 Can Get You Pwned. Source url: https://www.sans.org/reading-room/whitepapers/auditing/base64-pwned-33759. [Accessed 24 02 2019].
[83] M. B, "Malware Monday: Obfuscation," 19 12 2016. Source url: https://medium.com/@bromiley/malware-monday-obfuscation-f65239146db0. [Accessed 24 02 2019].

encoded string is 4 characters. If the source string is not long enough to generate an output of 4 characters, one or two equal signs will be added for padding. This padding is found in most Base64 encoded strings where the encoding does not generate a number of characters that is divisible by 4, thus you often see either one or two equal signs at the end of Base64 encoded data. Based on this definition however, the words “data”, “Data” and “Database” are all potentially valid Base64 (although they decode to random binary data) making positive validation of Base64 data difficult. Making things worse, Base64 does not always use the special characters / and +. In some implementations of Base64 a number of other special characters are used including the dash (-), the underscore (_), the period (.), the colon (:), and the exclamation point (!). In addition, some implementations of Base64 don’t use padding. As a result, Base64 can contain any combination of letters (upper and lower case), numbers and various special characters (/+-_:!) that may or may not have one or two equal signs at the end.

With byte-stats.py[84], statistics are being generated for the different byte values found in the under analysis PE. When we use this to analyze our Base64 encoded executable, we the following output:

[84] D. Stevens, "Decoding malware via simple statistical analysis," Didier Stevens Labs, 30 08 2017. Source url: https://blog.nviso.be/2017/08/30/decoding-malware-via-simple-statistical-analysis/. [Accessed 24 02 2019].



```
root@kali:~/Desktop# python byte-stats.py malware.exe
Byte ASCII Count     Pct
0xa7         115   0.02%
0xab         133   0.03%
0x9f         144   0.03%
0x6a j       145   0.03%
0xa9         150   0.03%
...
0x89       19505   3.89%
0x8b       21885   4.36%
0xff       28029   5.59%
0x24 $     38191   7.61%
0x00       77332  15.41%

Size: 501760  Bucket size: 10240  Bucket count: 49

                   File(s)            Minimum buckets   Maximum buckets
Entropy:           6.128007           4.784276          6.452751
                     Position:        0x00075800        0x00000000
Unique bytes:         256 100.00%       188  73.44%       256 100.00%
NULL bytes:         77332  15.41%       820   8.01%      3954  38.61%
Control bytes:      59701  11.90%       210   2.05%      2256  22.03%
Whitespace bytes:   10295   2.05%        53   0.52%       375   3.66%
Printable bytes:   159564  31.80%      1970  19.24%      5108  49.88%
High bytes:        194868  38.84%       873   8.53%      5173  50.52%
Hexadecimal bytes:  44919   8.95%       440   4.30%      1996  19.49%
BASE64 bytes:       94458  18.83%      1179  11.51%      4574  44.67%
```

Figure 126: Base64 bytes check with byte-stats.py

In the screenshot above see that we have 256 different byte values, and that 19% of the byte values are Base64 characters. This is not a strong indication that the data in the under analysis PE are Base64 encoded.

Using the option -r of byte-stats.py, an overview of the ranges of byte values is being presented:

```
root@kali:~/Desktop# python byte-stats.py -r malware.exe
Byte ASCII Count     Pct
0xa7         115   0.02%
0xab         133   0.03%
0x9f         144   0.03%
0x6a j       145   0.03%
0xa9         150   0.03%
...
0x89       19505   3.89%
0x8b       21885   4.36%
0xff       28029   5.59%
0x24 $     38191   7.61%
0x00       77332  15.41%

Size: 501760  Bucket size: 10240  Bucket count: 49

                   File(s)          Minimum buckets   Maximum buckets
Entropy:           6.128007         4.784276          6.452751
                     Position:      0x00075800        0x00000000
Unique bytes:          256 100.00%       188  73.44%       256 100.00%
NULL bytes:          77332  15.41%       820   8.01%      3954  38.61%
Control bytes:       59701  11.90%       210   2.05%      2256  22.03%
Whitespace bytes:    10295   2.05%        53   0.52%       375   3.66%
Printable bytes:    159564  31.80%      1970  19.24%      5108  49.88%
High bytes:         194868  38.84%       873   8.53%      5173  50.52%
Hexadecimal bytes:   44919   8.95%       440   4.30%      1996  19.49%
BASE64 bytes:        94458  18.83%      1179  11.51%      4574  44.67%

Number of ranges: 1
Fir. Last Len. Range
0x00 0xff 256: ................................ !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
tuvwxyz{|}~........................................................................................................................
.........
root@kali:~/Desktop#
```

Figure 127: Base64 bytes check with specific range

Usually the range check of byte-stats.py would reveal the pattern of Base64 alphabet, but with the 256 length we assume that these characters constitute an alphabet of another encoding scheme.

XORSearch

Having no clue of the encoding scheme on our PE file, XOR operation could reveal additional information. It is perspective of reverse engineering the static information that a PE file offers. As an alternative of brute forcing any known encoding scheme on the under analysis PE file, XORing definitely would be time effective. A tool is needed to try all possible combinations, for every total of bytes that compose a string.

XORSearch[85] is a program to search for a given string in an XOR, ROL, ROT or SHIFT encoded binary file [86]. XORSearch will try all XOR keys (0 to 255), ROL keys (1 to 7), ROT keys (1 to 25) and SHIFT keys (1 to 7) when searching. XORSearch also includes key 0, because this allows to search in an unencoded binary file (X XOR 0 equals X). XORSearch does a bruteforce attack with 8-bit keys and smaller [87].

At this point of our analysis we need given strings that are certainly contained as strings in the PE file. On the sections 2.3. «VME Technologies» and 2.4. «General Local Virtual Machine Detection», anti-virtualization techniques have been detected, so is a good start to search for them as strings in the PE file:

- VBOX
- VMware

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 19 malware.exe VBOX
Found XOR 4C position 716F4(-19): HARDWARE\ACPI\DSDT\VBOX__.Lf..L...L*..L...
```

Figure 128: XORSearch: VBOX string found XORing with 4C

On the figure 73, the results of XORSearch is being presented, for the string "VBOX", which is found on the position 716F4 in the PE file. Furthermore, with parameter "-n 19", 19 neighbor characters are being also printed. The registry path "HARDWARE\ACPI\DSDT\VBOX__" is

---

[85] D. Stevens, "XORSearch & XORStrings," Didier Stevens Labs, 30 01 2007. Source url: https://blog.didierstevens.com/programs/xorsearch/. [Accessed 24 02 2019].

[86] An XOR encoded binary file is a file where some (or all) bytes have been XORed with a constant value (the key). A ROL (or ROR) encoded file has its bytes rotated by a certain number of bits (the key). A ROT encoded file has its alphabetic characters (A-Z and a-z) rotated by a certain number of positions. A SHIFT encoded file has its bytes shifted left by a certain number of bits (the key): all bits of the first byte shift left, the MSB of the second byte becomes the LSB of the first byte, all bits of the second byte shift left, … XOR and ROL/ROR encoding is used by malware programmers to obfuscate strings like URLs.

[87] If the search string is found, XORSearch will print it until the 0 (byte zero) is encountered or until 50 characters have been printed, whichever comes first. Unprintable characters are replaced by a dot.

revealed. The malware searched on registry for this specific value, so it can detect the Virtual Box existence.

The idea of searching for known strings in the PE file that might be encoded, XORed in our case, was accurate. On the next steps a list has been created with all the strings that could be contained in the PE file.

Some functions that are already been detected as anti-debugging techniques would also help to reveal the XOR pattern.

- debug
- time
- sleep

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 19 malware.exe debug
Found XOR 00 position 79C6E(-19): BCSLeadByteEx....IsDebuggerPresent...LeaveC
Found XOR 20 position 79C6E(-19): bcslEADbYTEeX  ."iSdEBUGGERpRESENT .#lEAVEc
```

Figure 129: XORSearch: Debug string found XORing

The “debug” string was searched without case sensitivity, using the “-i” parameter and found as a string in position 79C6E XORing with 00. This means that the actual input string was found without XORing. The second result, XORing with 20, is being printed because of the case sensitive parameter, which converts the capital to lower case and the opposite. As a result, no hidden “debug” string was found in the PE file.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 19 malware.exe time
Found XOR 00 position 7468D(-19): ........A.Mingw runtime failure:..  Virtua
Found XOR 00 position 75D0E(-19): nctIwLb1EE...St11__timepunctIcE..St11__tim
Found XOR 00 position 75D22(-19): imepunctIcE..St11__timepunctIwE..St11logic
Found XOR 00 position 75FBB(-19): ages_base...St13runtime_error.............
Found XOR 00 position 762A4(-19): nct_bynameIwE..St15time_get_bynameIcSt19is
Found XOR 00 position 76304(-19): ...............St15time_get_bynameIwSt19is
Found XOR 00 position 76364(-19): ...............St15time_put_bynameIcSt19os
Found XOR 00 position 763C4(-19): ...............St15time_put_bynameIwSt19os
Found XOR 00 position 76482(-19): _argument....St17__timepunct_cacheIcE....S
Found XOR 00 position 7649E(-19): _cacheIcE....St17__timepunct_cacheIwE....S
Found XOR 00 position 76863(-19): nctIwE..........St8time_getIcSt19istreambu
Found XOR 00 position 768A3(-19): raitsIcEEE......St8time_getIwSt19istreambu
Found XOR 00 position 768E3(-19): raitsIwEEE......St8time_putIcSt19ostreambu
Found XOR 00 position 76923(-19): raitsIcEEE......St8time_putIwSt19ostreambu
Found XOR 00 position 76B23(-19): traitsIwEEE.....St9time_base....St9type_in
Found XOR 00 position 79FDA(-19): ...strerror....strftime....strlen....strto
Found XOR 00 position 7A000(-19): trtod....strxfrm...time....towlower....tow
Found XOR 00 position 7A04E(-19): f....wcscoll...wcsftime....wcslen....wcsxf
Found XOR 20 position 72A83(-19): type lc.numeric lc.time lc.collate lc.mone
Found ADD 27 position 72A83(-19): type lc.numeric lc.time lc.collate lc.mone
```

Figure 130: XORSearch: time string found XORing

The “time” string was found as a string in various positions, XORing with 00. This means that the actual input string was found without XORing. As a result, no hidden “time” string was found in the PE file.

Figure 131: XORSearch: sleep string found XORing

The “sleep” string was searched without case sensitivity, using the “-i” parameter and found as a string in position 79D00 XORing with 00. This means that the actual input string was found without XORing. The second result, XORing with 20, is being printed because of the case sensitive parameter, which converts the capital to lower case and the opposite. As a result, no hidden “sleep” string was found in the PE file.

As long as a ransomware is being analyzed, some certain type of files is interested in the attackers. Searching on a huge list of file types extensions, the following file type extensions has been detected:

txt	doc	docx	xls	xlsx

Table 8: selected file extentions for XORsearch

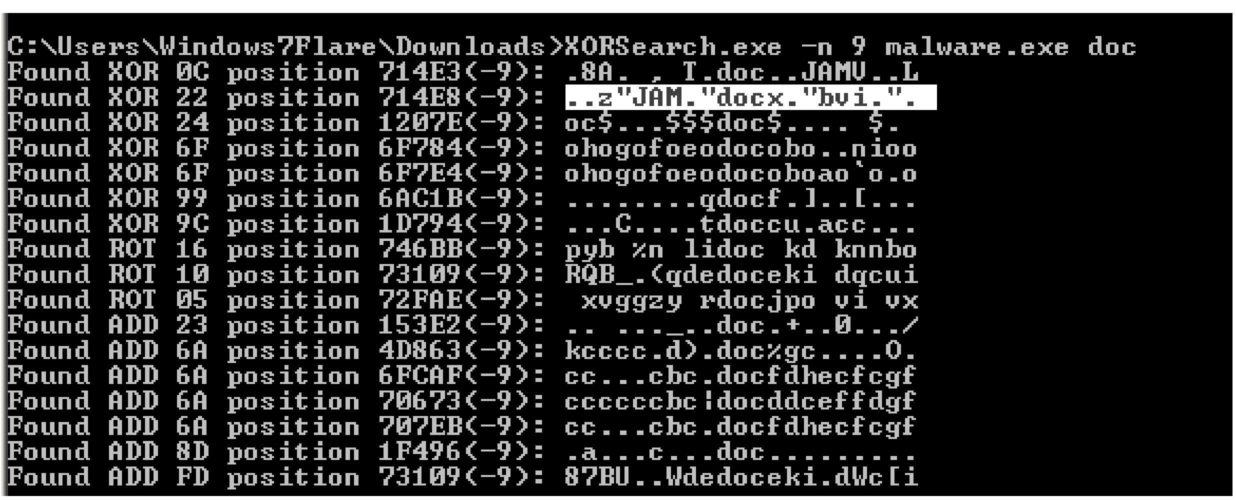


Figure 132: XORSearch: doc and docx string found XORing

On the figure 77, the results for the string “doc”, was found on the position 714E3 XORing with 0C, in the PE file. The string “docx” was also found on the same position (714E8 is next to 714E3), but it is XORed with 22.

The malware author seems to have a sophisticated pattern using XOR with different keys for each malware’s operation. Further analysis is needed so on the Figure 78, the string “xls” was searched, which also contains the “xlsx” like the doc one.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe xls
Found XOR 1E position 79FFA(-9): z....mjlfxls...jws<..
Found XOR 1E position 7A064(-9): p....i>mfxls.t.MVY<jM
Found XOR 38 position 714EE(-9): W48~uyb.8xls.8....v88
Found XOR 4E position 714F3(-9): ..lN...vNxlsx.NNNN.w.
Found XOR 91 position 17976(-9): ..W......xlsnn.n.....
Found ROT 22 position 72B0F(-9): mvxyep qixlsh geppih.
Found ROT 22 position 72B2E(-9): mvxyep qixlsh geppih.
Found ROT 22 position 72FAF(-9): geppih amxlsyx er egx
Found ROT 22 position 7A094(-9): xyt..).kixlswxfcreqi.
Found ROT 22 position 7A0A4(-9): reqi.*.kixlswxreqi...
Found ADD 0B position 72B0F(-9): mvxyep$qixlsh$geppih.
Found ADD 0B position 72B2E(-9): mvxyep$qixlsh$geppih.
Found ADD 0B position 72FAF(-9): geppih${mxlsyx$er$egx
Found ADD 0B position 7A094(-9): xyt..-.kixlswxf}reqi.
Found ADD 0B position 7A0A4(-9): reqi...kixlswxreqi...
Found ADD 5B position 39841(-9): .xpVTTT..xlsTTT..xhUT
Found ADD 5B position 39F8D(-9): .xpVTTT..xlsTTT..xh^T
Found ADD 5B position 3BF85(-9): .xpVTTT..xlsTTT..xhUT
Found ADD 5B position 3C219(-9): .xpVTTT..xlsTTT..xh^T

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe xlsx
Found XOR 4E position 714F3(-9): ..lN...vNxlsx.NNNN.w.N
```

Figure 133: XORSearch: xls and xlsx string found XORing

On the figure 77, the results for the string “doc”, was found on the position 714E3 XORing with 0C, in the PE file. The string “docx” was also found on the same position (714E8 is next to 714E3), but it is XORed with 22. On the figure 78, the results of “xls”, was found on the positing 714EE XORing with 38 and the result of “xlsx” was found on 714F3 XORing with 4E.

All these strings indicate that the attacker is searching for the extensions of certain type of files. This is a strong clue that his malware is a ransomware.

But we have not searched for “txt” yet. On the following figure, the txt with case sensitivity, returns a lot of junk results and them position is not near the above-mentioned type of files extensions. The interesting results here are on the position 7167F, where the string is being XORed with 5D and on the position 71651 where is being XORed with 4F. A file “readme.txt” appeared.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe txt
Found XOR 1B position 746F0(-9): ortu;kitotxtw;m~ihrtu
Found XOR 2B position 71798(-9): tX_YBEL..txtHYNJ_N+IJ
Found XOR 2B position 7184E(-9): tX_YBEL..txtHDEX_Y^H_
Found XOR 2B position 71C94(-9): +GDHJGN..txtEDYFJGBQN
Found XOR 2B position 71E88(-9): tX_YBEL..txtHYNJ_N+IJ
Found XOR 2B position 71F3E(-9): tX_YBEL..txtHDEX_Y^H_
Found XOR 2B position 72A43(-9): ..MJHN_..txtHYNJ_NtHt
Found XOR 4F position 71651(-9): ..readme.txt O2....@.
Found XOR 58 position 714DE(-9): Xqpeyzl.Xtxt.X0;7TX..
Found XOR 5D position 7167F(-9): ..readme.txt ]S*Ltqdq
Found XOR 74 position 6F785(-9): st!t}t~t.txtyt..urttu
Found XOR 74 position 6F7E5(-9): st!t}t~t.txtytzt{tdte
Found XOR 74 position 73D57(-9): ttuttt.#3txttt.#3t!tt
Found XOR 74 position 7410F(-9): 3tettt+/3txttt.ttt./3
Found XOR 74 position 7412B(-9): 3t2ttt+/3txttt.ttt./3
Found XOR 74 position 74137(-9): tt.ttt./3txttt./3tEtt
Found XOR 74 position 74153(-9): tt.tttx(3txttth(3tEtt
Found XOR 74 position 74AF3(-9): ttsttt~tttxtttztttett
Found XOR 74 position 74EFF(-9): tttttttttxttttttt4.3
Found XOR 74 position 7517F(-9): tttttttttxttttttt..3
Found XOR 74 position 76EFF(-9): tttttttttxttttttttT.3
Found XOR 74 position 7749F(-9): tttttttttxttttttt..3
Found XOR 75 position 70A5E(-9): }t!t.t~tytxt{tzttuu.u
Found XOR 75 position 70A8E(-9): }t!t.t~tytxt{tzttuu.u
Found XOR 75 position 70D2A(-9): }t!t.t~tytxt{tzttuu.u
Found XOR 75 position 70D5A(-9): }t!t.t~tytxt{tzttuu.u
Found XOR 8F position 4C5A5(-9): .........gtxtp......H.
Found ADD 1C position 72A4C(-9): thtx.zv.ztxt..xv.z5.v
```

Figure 134: XORSearch: txt string found XORing

Searching for this specific “readme.txt” string and its neighbors, a filename that reveals a malicous action is being returned. Specifically, on the figure 80, the path “C:\DESTROYED_FILES__REAME.TXT” is being revealed. Another strong clue of ransomware which destroys the files after encryption.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 9 malware.exe readme.txt
Found XOR 6F position 7164A(-9): D_FILES__README.TXT.o.*/:/`,
Found XOR 7D position 71678(-9): D_FILES__README.TXT.}s.1TQDQ

C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 19 malware.exe readme.txt
Found XOR 6F position 7164A(-19): o\DESTROYED_FILES__README.TXT.o.*/:/`,' NooooQ<N
Found XOR 7D position 71678(-19): :\DESTROYED_FILES__README.TXT.}s.1TQDQ.RY^0}.}..

C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 29 malware.exe readme.txt
Found XOR 6F position 7164A(-29): .563y0".$Wo\DESTROYED_FILES__README.TXT.o.*/:/`,' NooooQ<NVWAF@]KWVM
Found XOR 7D position 71678(-29): r>52\}}}}C:\DESTROYED_FILES__README.TXT.}s.1TQDQ.RY^0}.}..}}}.C=}.C=
```

Figure 135: XORSearch: readme.txt string found XORing

The final position, that will be written down, in this case is 71678, where the string is being XORed with 7D.

The digging starts, searching for known strings in the PE file that might be XORed, but this time on targeted names of strings, related to ransomware. At first, the strings “NATO”, “container”, “training”, “delivery”, “location”, “status” and “deploy” searched:

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe NATO
Found XOR 37 position 71494(-9): 7..w7m.w7NATO.7_SRHJUR

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe container
Found XOR 0B position 7149A(-9): (K.r>hs<.container..tsyt~¦i

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe indicator
Found XOR 16 position 714A5(-9): si¦tsxo..indicator..CEV^Y^Y

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe training
Found XOR 21 position 714B0(-9): S^TVCXE7!training.!ihad{h.

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe delivery
Found XOR 2C position 714BA(-9): .ldcdcj.,delivery.,wtxzort

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe location
Found XOR 00 position 71A62(-9): _words allocation failed..
Found XOR 00 position 746E3(-9): pseudo relocation protocol
Found XOR 00 position 74717(-9): pseudo relocation bit size
Found XOR 37 position 714C4(-9): ~wrm~ib.7location.7......u

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe status
Found XOR 42 position 714CE(-9): .......uBstatus.Bkj.c`v.

C:\Users\Windows7Flare\Downloads>XORSearch.exe -n 9 malware.exe deploy
Found XOR 4D position 714D6(-9): M¦{n{z¦.Mdeploy.Mama.M%.
```

Figure 136: XORSearch: targeted string names found XORing

On the figure 81, the results for the string “NATO” was found on the position 71494 XORing with 37, the results for the string “container” was found on the position 7149A XORing with 0B, the results for the string “indicator” was found on the position 714A5 XORing with 16, the results for the string “training” was found on the position 714B0 XORing with 21, the results for the string “delivery” was found on the position 714BA XORing with 2C, the results for the string “location” was found on the position 714C4 XORing with 37, the results for the string “status” was found on the position 714CE XORing with 42 and the results for the string “deploy” was found on the position 714D6 XORing with 4D.

As long as the string search is focused on ransomware, encryption will take place and then the unknown perpetrators will ask for ransom.

So, searching for string “crypt” and its neighbors, a whole paragraph is revealed from the ransom message. More Specifically, on the figure 82 the phrase “*We have encrypted lot of your files. If you want to get their real content back, then send us 1000 euros and the data.bin file from this directory. We then send you program that decrypts the encrypted files. Our email is aBit@bad.guys*” is revealed.

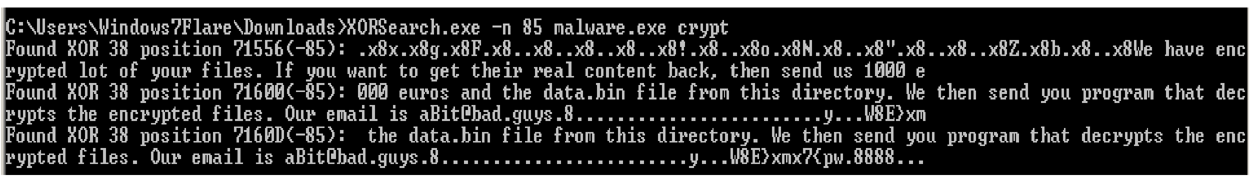


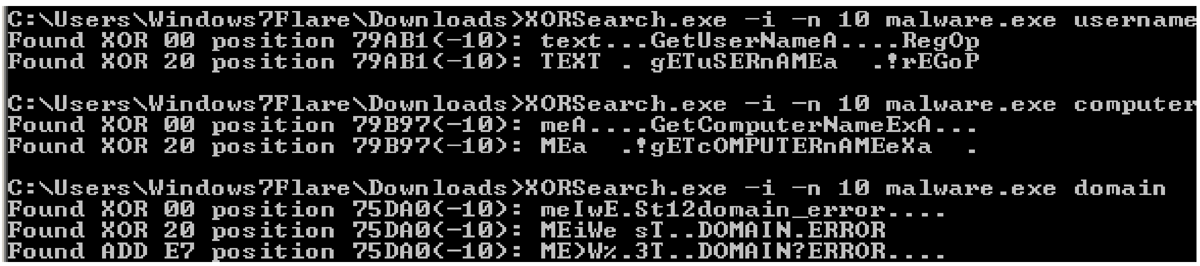
Figure 137: XORSearch: crypt string found XORing

At this point, a lot of information should be analyzed. The ransom cost is 1000 euros. The unknown perpetrators request the ransom and the data.bin file in order to sent back an applocation that decrypts the file. So the data.bin file should contain information for the encryption, its procedure or even the key itself! At last, the email of unknown perpetrators is “*aBit@bad.guys*”.

The digging continues, searching for more strings in the PE, related to ransomware. Considering that the data.bin file could give feedback to the unknown perpetrators for the victim’s PC, the strings “username”, “computer”, “domain” and “money” are searched. The searched results are being screenshotted on the figure 83 as follows:

- the string “username” was found on the position 79AB1 XORing with 00. This result is the function GetUserNameA, that has been already found and analyzed on the 4.1.1. section.
- the string “computer” was found on the position 79B97 XORing with 00. This result is the function GetComputernameExA, that has been already found and analyzed on the 4.1.1. section.

- the string “domain” was found on the position 75DA0 XORing with 20. This result is XORed with 20 because the words are stored in capital (DOMAIN ERROR) and it is a system error.
- the string “money” was found in several positions 75DA0 XORing with 20. This result is XORed with 20 because the words are stored in capital (DOMAIN ERROR) and it is a system error.

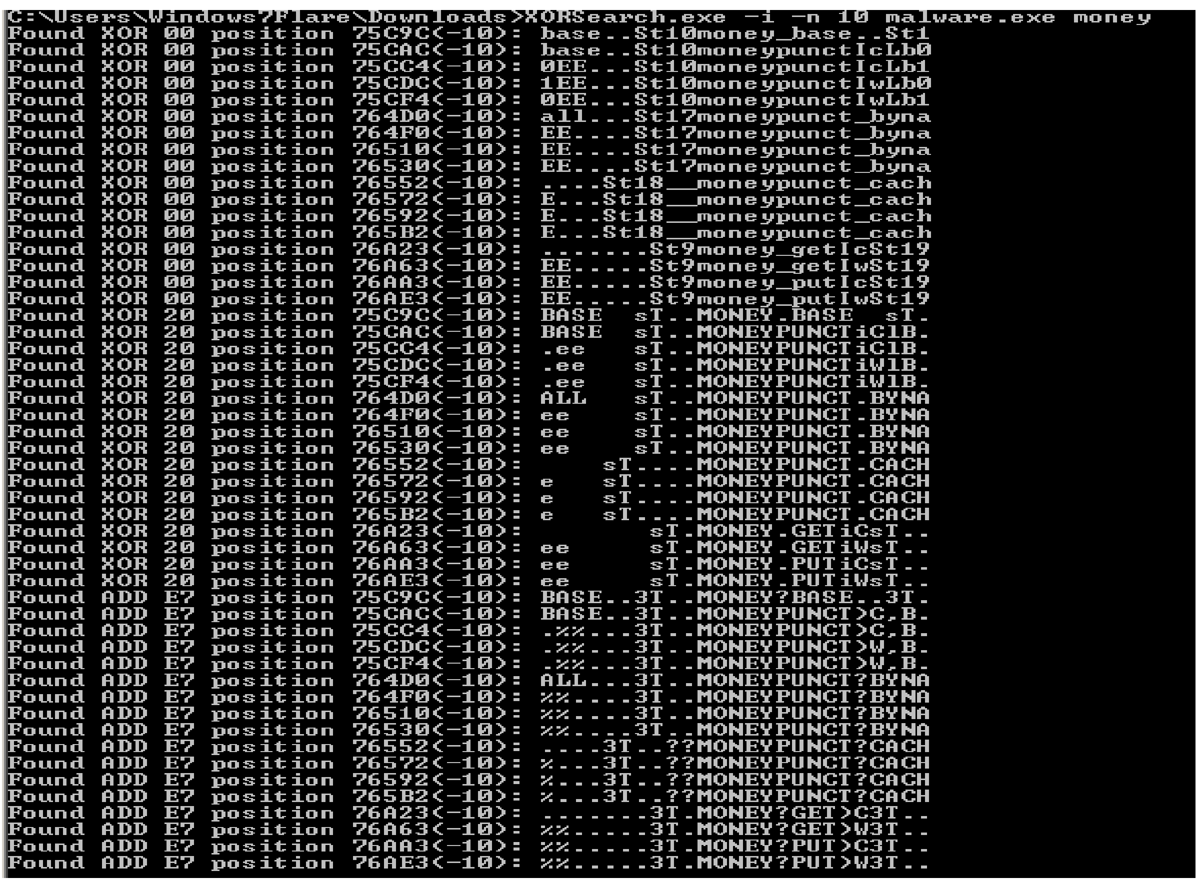


```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe username
Found XOR 00 position 79AB1(-10): text....GetUserNameA....RegOp
Found XOR 20 position 79AB1(-10): TEXT . gETuSERnAMEa  .!rEGoP

C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe computer
Found XOR 00 position 79B97(-10): meA....GetComputerNameExA...
Found XOR 20 position 79B97(-10): MEa  .!gETcOMPUTERnAMEeXa  .

C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe domain
Found XOR 00 position 75DA0(-10): meIwE.St12domain_error....
Found XOR 20 position 75DA0(-10): MEiWe sT..DOMAIN.ERROR
Found ADD E7 position 75DA0(-10): ME>W%.3T..DOMAIN?ERROR....
```

Figure 138: XORSearch: username, computer, domain found



```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe money
Found XOR 00 position 75C9C(-10): base..St10money_base..St1
Found XOR 00 position 75CAC(-10): base..St10moneypunctIcLb0
Found XOR 00 position 75CC4(-10): 0EE...St10moneypunctIcLb1
Found XOR 00 position 75CDC(-10): 1EE...St10moneypunctIwLb0
Found XOR 00 position 75CF4(-10): 0EE...St10moneypunctIwLb1
Found XOR 00 position 764D0(-10): all...St17moneypunct_byna
Found XOR 00 position 764F0(-10): EE....St17moneypunct_byna
Found XOR 00 position 76510(-10): EE....St17moneypunct_byna
Found XOR 00 position 76530(-10): EE....St17moneypunct_byna
Found XOR 00 position 76552(-10): ....St18__moneypunct_cach
Found XOR 00 position 76572(-10): E...St18__moneypunct_cach
Found XOR 00 position 76592(-10): E...St18__moneypunct_cach
Found XOR 00 position 765B2(-10): E...St18__moneypunct_cach
Found XOR 00 position 76A23(-10): .......St9money_getIcSt19
Found XOR 00 position 76A63(-10): EE.....St9money_getIwSt19
Found XOR 00 position 76AA3(-10): EE.....St9money_putIcSt19
Found XOR 00 position 76AE3(-10): EE.....St9money_putIwSt19
Found XOR 20 position 75C9C(-10): BASE  sT..MONEY.BASE  sT.
Found XOR 20 position 75CAC(-10): BASE  sT..MONEYPUNCTiClB.
Found XOR 20 position 75CC4(-10): .ee   sT..MONEYPUNCTiClB.
Found XOR 20 position 75CDC(-10): .ee   sT..MONEYPUNCTiWlB.
Found XOR 20 position 75CF4(-10): .ee   sT..MONEYPUNCTiWlB.
Found XOR 20 position 764D0(-10): ALL   sT..MONEYPUNCT.BYNA
Found XOR 20 position 764F0(-10): ee    sT..MONEYPUNCT.BYNA
Found XOR 20 position 76510(-10): ee    sT..MONEYPUNCT.BYNA
Found XOR 20 position 76530(-10): ee    sT..MONEYPUNCT.BYNA
Found XOR 20 position 76552(-10):     sT....MONEYPUNCT.CACH
Found XOR 20 position 76572(-10): e   sT....MONEYPUNCT.CACH
Found XOR 20 position 76592(-10): e   sT....MONEYPUNCT.CACH
Found XOR 20 position 765B2(-10): e   sT....MONEYPUNCT.CACH
Found XOR 20 position 76A23(-10):        sT.MONEY.GETiCsT..
Found XOR 20 position 76A63(-10): ee     sT.MONEY.GETiWsT..
Found XOR 20 position 76AA3(-10): ee     sT.MONEY.PUTiCsT..
Found XOR 20 position 76AE3(-10): ee     sT.MONEY.PUTiWsT..
Found ADD E7 position 75C9C(-10): BASE..3T..MONEY?BASE..3T.
Found ADD E7 position 75CAC(-10): BASE..3T..MONEYPUNCT>C,B.
Found ADD E7 position 75CC4(-10): .%%...3T..MONEYPUNCT>C,B.
Found ADD E7 position 75CDC(-10): .%%...3T..MONEYPUNCT>W,B.
Found ADD E7 position 75CF4(-10): .%%...3T..MONEYPUNCT>W,B.
Found ADD E7 position 764D0(-10): ALL...3T..MONEYPUNCT?BYNA
Found ADD E7 position 764F0(-10): %%....3T..MONEYPUNCT?BYNA
Found ADD E7 position 76510(-10): %%....3T..MONEYPUNCT?BYNA
Found ADD E7 position 76530(-10): %%....3T..MONEYPUNCT?BYNA
Found ADD E7 position 76552(-10): ....3T..??MONEYPUNCT?CACH
Found ADD E7 position 76572(-10): %...3T..??MONEYPUNCT?CACH
Found ADD E7 position 76592(-10): %...3T..??MONEYPUNCT?CACH
Found ADD E7 position 765B2(-10): %...3T..??MONEYPUNCT?CACH
Found ADD E7 position 76A23(-10): .......3T.MONEY?GET>C3T..
Found ADD E7 position 76A63(-10): %%.....3T.MONEY?GET>W3T..
Found ADD E7 position 76AA3(-10): %%.....3T.MONEY?PUT>C3T..
Found ADD E7 position 76AE3(-10): %%.....3T.MONEY?PUT>W3T..
```

Figure 139: XORSearch: username, computer, domain, money strings found

All the strings that contains the keywork “money” are already revealed in .rdata on Appendix H.

In addition to the focused string search and having the knowledge that the malware checks the IP Address of the infected machine, the string “10.1.210” was found in positions 7048C and 707D0, not XORing but ADDing with 35. Keep in mind that the given information that the machine should be a subnet with range on IP Addresses 10.1.0.0-255 (/24) was incorrect.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe 10.1.0.

C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe 10.1.
Found ADD 35 position 7048C(-10): u.-.K/8.3/10.1.210./.....
Found ADD 35 position 707D0(-10): u.-.K/8.3/10.1.210./.....
```

Figure 140: XORSearch: specific IP Address found

On the following figure 85, some last targeted searched had been done, that reveals that the hash “49C60C2B94F0850BBACAB2F2538A286” (hashed result of IP Address 10.1.0.*) is not contained in the PE file. So it is assumed that another incorrect information was provided.

```
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe copyright
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe -f testXor.txt
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe -f testXor.txt
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe 49C60C2B94F0850BBACAB2F2538A286
C:\Users\Windows7Flare\Downloads>XORSearch.exe -i -n 10 malware.exe Believe
```

Figure 141: XORSearch: fail to find some clues that was provided from external information

Several more searches could be done with XORSearch, using as input the strings that was found in basic static analysis. But the on malware analysis the analyst should focus on keypoints and that is the reason that these searhes are enough, with a lot of information being revealed.

The most useful on the subject malware is the generated key that is encrypted and written to the data.bin file. By brute forcing the data.bin file the key can by revealed, which is the string “Believe you can and you're halfway there”. The bruteforce is applicable as long as only letters and symbols are being contained. All files that are found, are encrypted with XOR operation, using the generated key. At the end of the procedure the old version of the files is deleted.

7.4 Future work

Shellcode authors must employ techniques to work around inherent limitations of the odd runtime environment in which shellcode executes. This includes identifying where in memory the shellcode is executing and manually resolving all of the shellcode's external dependencies so that it can interact with the system. To save on space, these dependencies are usually obfuscated by using hash values instead of ASCII function names. It is not so common for nearly the entire shellcode to be encoded so that it bypasses any data filtering by the targeted process. All of these techniques can easily frustrate beginning analysts, but the provided material should help the reader to recognize these activities, so you can instead focus on understanding the main functionality of the shellcode.

List of tables

List of Figures

.

Bibliography

[1] F. M. Last Name, "Article Title," *Journal Title,* pp. Pages From - To, Year.

[2] F. M. Last Name, Book Title, City Name: Publisher Name, Year.

[3] O. . Martinu and G. . McEwen, "Crime in the age of technology," , 2018. [Online]. Available: https://bulletin.cepol.europa.eu/index.php/bulletin/article/download/337/286. [Accessed 26 3 2019].

[4] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction," Invisible Things Lab, 01 November 2004. [Online]. Available: http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html. [Accessed 01 02 2019].

[5] T. Klein, "Scooby Doo - VMware Fingerprint Suite," 2003. [Online]. Available: http://web.archive.org/web/20061215022409/http://www.trapkit.de/research/vmm/scoopydoo/index.html. [Accessed 01 02 2019].

[6] T. Klein, "jerry - A(nother) VMware Fingerprinter," 2003. [Online]. Available: http://web.archive.org/web/20061215022453/http://www.trapkit.de/research/vmm/jerry/index.html. [Accessed 01 02 2019].

[7] T. Klein, "VMware fingerprint codes," 2003. [Online]. Available: http://web.archive.org/web/20061215022430/http://www.trapkit.de/research/vmm/index.html. [Accessed 01 02 2019].

[8] Quist, Danny; Smith, Val;, "Detecting the Presence of Virtual Machines Using the Local Data Table," Offensive Computing, 25 04 2006. [Online]. Available: http://web.archive.org/web/20060425123645/http://www.offensivecomputing.net/files/active/0/vm.pdf. [Accessed 01 02 2019].

[9] T. Raffetsede, C. Kruege and E. Kirda, "Detecting System Emulators," Secure Systems Lab, Technical University of Vienna, Austria, Vienna, Austria.

[10 Liston, Tom; Skoudis, Ed;, "On the Cutting Edge:Thwarting Virtual MachineDetection," *SANS,* 2006.

[11 L. Zeltser, "Virtual Machine Detection in Malware via Commercial Tools," 18 01 2007. [Online]. Available: http://isc.sans.org/diary.html?storyid=1871&rss. [Accessed 01 02 2019].

[12 K. Zahn, "Case Study: 2012 DC3 DigitalForensic Challenge BasicMalware Analysis Exercise," 24 08 2013. [Online]. Available: https://www.sans.org/reading-room/whitepapers/malicious/case-study-2012-dc3-digital-forensic-challenge-basic-malware-analysis-exercise-34330. [Accessed 01 02 2019].

[13 Sikorski, Michael; Honig, Andrew; Lawler, Stephen;, Practical Malware Analysis, San Francisco, CA: No Starch Press, 2012, pp. 1 - 802.

[14 Barham, P., Dragovic, B., Fraser K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A., "Xen and the Art of Virtualization," 2003.

[15 J. Rutkowska, "Stealth Malware Taxonomy," 11 2006. [Online]. Available: blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf. [Accessed 01 02 2019].

[16 A. Sanabria, "Malware Analysis: Environment Design and Artitecture," 18 01 2007. [Online]. Available: https://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841. [Accessed 01 02 2019].

[17 K. Fiscus, "Base64 Can Get You Pwned," SANS Institute, 2011.

[18 M. B, "Malware Monday: Obfuscation," 19 12 2016. [Online]. Available: https://medium.com/@bromiley/malware-monday-obfuscation-f65239146db0. [Accessed 24 02 2019].

[19 D. Stevens, "Decoding malware via simple statistical analysis," Didier Stevens Labs, 30 08 2017. [Online]. Available: https://blog.nviso.be/2017/08/30/decoding-malware-via-simple-statistical-analysis/. [Accessed 24 02 2019].

[20 D. Stevens, "XORSearch & XORStrings," Didier Stevens Labs, 01 2007. [Online]. Available: https://blog.didierstevens.com/programs/xorsearch/. [Accessed 24 02 2019].

[21 Dilshan Keragala, "Detecting Malware and SandboxEvasion Techniques," SANS Institute, January 16, 2016.

[22 Ferrie, Peter, "The “Ultimate”Anti-Debugging Reference," 5 4 2011. [Online]. Available: https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf. [Accessed 1 12 2018].

Appendices

Appendix A

Specifications of Host's Hardware and Software, VM and VME installation and configuration

A.1 Hardware specification of single PC lab

Processor:
Intel Core i7-3930K CPU 3.20GHz, 3200 Mhz, 6 Cores, 12 Logical Processors
Supporting:
Intel 64 architecture
Intel HT Technology
Intel VT-d
Intel VT-x
Intel VT-x with EPT
Doe not support:
Intel vPro Technology[88]
Physical Memory (RAM):
32.0 GB
Hard Disk Drive for Host:
120 GB SSD
Hard Disk Drive for Virtual Machines Storage
2 x 3TB on software RAID 1.

A.2 Software specification of single PC lab

Host's OS:Windows 10 Pro x64
VME software: VMware Workstation 14 Pro

A.3 VM Configuration

Note that, all software and configurations written in this section are my personal additions based on Flare VM[89].

[88] It is preferred to have this feature, but on the current case was not available. Luckily the malware does not exploit Intel's Virtualization.
[89] FLARE VM - a fully customizable, Windows-based security distribution for malware analysis and incident response. A downloadable configuration script is provided to assist cyber security analysts in creating handy and versatile toolboxes for malware analysis environments. It provides a convenient interface for them to obtain a useful set of analysis tools directly from their original sources.

A.4 OS installation

For malware analysis, OS may vary, some malwares may only work on certain OS, so it would be better to have several of them. In the case under analysis, Windows 7 Pro x64 have been chosen. Any customized Virtual Machine in a Windows installation requires numerous tweaks and tools to aid analysis. Unfortunately trying to maintain a custom VM like this is very laborious: tools frequently get out of date and it is hard to change or add new things. There is also a constant fear that if the VM gets corrupted it would be super tedious to replicate all of the settings and tools that are being built up. To address this and many related challenges, a standardized (but easily customizable) Windows-based security distribution called FLARE VM will be used.

A.5 Windows SDK and Framework

Install windows SDK and .Net Framework 4, which also installs WinDBG. (source url: https://www.microsoft.com/en-us/download/details.aspx?id=8279).

A.6 Virtual Machine Environment Installation and configuration

1. Install VMware in your main operating system.
2. Install a new fresh Windows 7 Pro x64 version of your choice and update it.
3. Install VMware Tools addition.
4. Download, install and configure required software, via url. More specifically, the deployment of the FLARE VM environment can be done by visiting the following URL in Internet Explorer: *https://github.com/fireeye/flare-vm/*

A.7 FlareVM Installation Script

1. Decompress the FLARE VM repository to a directory of your choosing.
2. Start a new session of PowerShell with escalated privileges. FLARE VM attempts to install additional software and modify system settings; therefore, escalated privileges are required for installation.
3. Within PowerShell, change directory to the location where you have decompressed the FLARE VM repository.
4. Enable unrestricted execution policy for PowerShell by executing the following command and answering “Y” when prompted by PowerShell: *Set-ExecutionPolicy unrestricted*
5. Execute the install.ps1 installation script: *.\install.ps1*.
6. You will be prompted to enter the current user’s password. FLARE VM needs the current user’s password to automatically login after a reboot when installing. Optionally, you can specify the current user’s password bypassing the “-password <current_user_password>” at the command line. The rest of the installation process is fully automated. Depending upon your internet speed the entire installation may take up to one hour to finish. The VM also reboots multiple times due to the numerous software installations’ requirements. Once the installation completes, the PowerShell prompt remains open waiting for you to hit any key before exiting. After completing

the installation, you will be presented with the following desktop environment: (SCREENSHOT FROM FLARE VM HOME SCREEN)

7. At this point power off the VM, switch the VM networking mode to Host-Only, and then take a snapshot to save a clean state of your analysis VM.

A.8 Installed Tools with FlareVm [90]

Android
- dex2jar
- apktool

Debuggers
- flare-qdb
- scdbg
- OllyDbg + OllyDump + OllyDumpEx
- OllyDbg2 + OllyDumpEx
- x64dbg
- WinDbg + OllyDumpex + pykd
- Decompilers
- RetDec
- Delphi
- Interactive Delphi Reconstructor (IDR)

Disassemblers
- IDA Free (5.0 & 7.0)
- Binary Ninja Demo
- radare2
- Cutter

.Net
- de4dot
- Dot Net String Decoder (DNSD)
- dnSpy
- DotPeek
- ILSpy
- RunDotNetDll

Flash
- FFDec

Forensic
- Volatility

Hex Editors
- FileInsight
- HxD
- 010 Editor

Java

- JD-GUI
- Bytecode-Viewer

Networking

- FakeNet-NG
- ncat
- nmap
- Wireshark

Office

- Offvis
- OfficeMalScanner

PDF

- PDFiD
- PDFParser
- PDFStreamDumper

PE

- PEiD
- ExplorerSuite (CFF Explorer)
- PEview
- DIE
- PeStudio
- PEBear
- ResourceHacker
- LordPE

Pentest

- MetaSploit
- Windows binaries from Kali Linux

Text Editors

- SublimeText3
- Notepad++
- Vim

Visual Basic

- VBDecompiler

Web

- BurpSuite Free Edition

Utilities

- FLOSS
- HashCalc
- HashMyFiles
- Checksum
- 7zip
- Far Manager
- Putty
- Wget
- RawCap
- UPX

RegShot
Process Hacker
Sysinternals Suite
API Monitor
SpyStudio
Shellcode Launcher
Cygwin
Unxutils
Malcode Analyst Pack (MAP)
XORSearch
XORStrings
Yara
CyberChef

KernelModeDriverLoader

Python, Modules, Tools

Py2ExeDecompiler
Python 2.7
hexdump
pefile
winappdbg
pycryptodome
vivisect
capstone-windows
unicorn
oletools
unpy2exe
uncompyle6
Python 3
unpy2exe
uncompyle6

Other[91]

VC Redistributable Modules (2005, 2008, 2010, 2012, 2013, 2015, 2017)
.Net versions 4.6.2 and 4.7.1
Practical Malware Analysis Labs
Google Chrome
Cmder Mini

A.9 Staying up to date

Type the following command to update all of the packages to the most recent version:
cup all

---

[91] For the live updated list of features please check the online blog on the source url: https://www.fireeye.com/blog/threat-research/2018/11/flare-vm-update.html

A.10 Extra useful tools

In addition to Flare VM toolset, some useful tools have been installed manually, to have a complete gamma tool.

A.11 RDG packer detector

Download and extract RDG packer detector to C:\Tools\RDG (Source url: http://www.rdgsoft.net/). When you run it for first time, it tries to setup context menu which I choose yes. If you do so, you'll be able to right-click on binaries and let RDG scan it easily.

CFF Explorer

Download and install CFF Explorer. Run CFF Explorer, go to Settings and click Enable shell extensions.

Ollydbg plugins

Download and extract Ollydbg to C:\Tools\Olly (source url: http://www.ollydbg.de/odbg110.zip). Use this as Ollydbg.ini which will have nice theme (provided by jacob@reddit.com) and then install the following Ollydbg plugins:

- Olly advanced (source url: https://tuts4you.com/e107_plugins/download/download.php?view.75)
- Olly breakpoint manager (source url: https://tuts4you.com/e107_plugins/download/download.php?view.76)
- OllyBonE (source url: https://tuts4you.com/e107_plugins/download/download.php?view.85)
- OllyDumpEx (source url: https://tuts4you.com/e107_plugins/download/download.php?view.3451)
- OdbgScript (source url: https://sourceforge.net/projects/odbgscript/files/English%20Version/)
- StrongOD (source url: https://tuts4you.com/e107_plugins/download/download.php?view.2028)
- Ultra String Reference (source url: https://tuts4you.com/e107_plugins/download/download.php?view.107)
- CopyHexCode (source url: https://tuts4you.com/e107_plugins/download/download.php?view.3581)
- Multiline Ultimate Assemble (source url: https://tuts4you.com/e107_plugins/download/download.php?view.2805)
- ImportStudio (source url: https://tuts4you.com/e107_plugins/download/download.php?view.3438)

At last goto Options -> Just in time debugging and make Ollydbg just-in-time debugger.

Handle

Download and install Handle (source url: https://download.sysinternals.com/files/Handle.zip). Handle is a utility that displays information about open handles for any process in the system. You can use it to see the programs that have a file open, or to see the object types and names of all the handles of a program. Runs only via terminal.

DebugView

Download and install DebugView (source url: https://download.sysinternals.com/files/DebugView.zip). DebugView is an application that lets you monitor debug output on your local system, or any computer on the network that you can reach via TCP/IP. It is capable of displaying both kernel-mode and Win32 debug output, so you don't need a debugger to catch the debug output your applications or device drivers generate, nor do you need to modify your applications or drivers to use non-standard debug output APIs.

Autoruns for Windows

Download and install Autoruns for Windows (source url: https://download.sysinternals.com/files/Autoruns.zip). This utility, which has the most comprehensive knowledge of auto-starting locations of any startup monitor, shows you what programs are configured to run during system bootup or login, and when you start various built-in Windows applications like Internet Explorer, Explorer and media players. These programs and drivers include ones in your startup folder, Run, RunOnce, and other Registry keys. Autoruns reports Explorer shell extensions, toolbars, browser helper objects, Winlogon notifications, auto-start services, and much more. Autoruns goes way beyond other autostart utilities.

Dependency Walker

Download and install Dependency Walker (source url: http://www.dependencywalker.com/). Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module (exe, dll, ocx, sys, etc.) and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. Another view displays the minimum set of required files, along with detailed information about each file including a full path to the file, base address, version numbers, machine type, debug information, and more.

A.12 Snapshotting

At this point power off the VM, switch the VM networking mode to Host-Only, and then take a second snapshot to save a clean state of your analysis VM.

Appendix B
StealthyTools.reg

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\Use
rData\S-1-5-18\Products\43F974C0D0E8C1C4D9CA1C70A1C60570\InstallProperties]
"LocalPackage"="C:\\Windows\\Installer\\124ec.msi"
"AuthorizedCDFPrefix"=""
"Comments"="Build "
"Contact"=""
"DisplayVersion"="8.1.30629.3138"
"HelpLink"=""
"HelpTelephone"=""
"InstallDate"="20170205"
"InstallLocation"="C:\\Program Files\\VMware\\VMware Tools\\"
"InstallSource"="C:\\Users\\Admin\\AppData\\Local\\Temp\\{0C479F34-8E0D-4C1C-9DAC-
C1071A6C5007}~setup\\"
"ModifyPath"=hex(2):4d,00,73,00,69,00,45,00,78,00,65,00,63,00,2e,00,65,00,78,\
  00,65,00,20,00,2f,00,49,00,7b,00,30,00,43,00,34,00,37,00,39,00,46,00,33,00,\
  34,00,2d,00,38,00,45,00,30,00,44,00,2d,00,34,00,43,00,31,00,43,00,2d,00,39,\
  00,44,00,41,00,43,00,2d,00,43,00,31,00,30,00,37,00,31,00,41,00,36,00,43,00,\
  35,00,30,00,30,00,37,00,7d,00,00,00
"Publisher"="Microsoft Corporation"
"Readme"=""
"Size"=""
"EstimatedSize"=dword:0001685f
"UninstallString"=hex(2):4d,00,73,00,69,00,45,00,78,00,65,00,63,00,2e,00,65,00,\
  78,00,65,00,20,00,2f,00,49,00,7b,00,30,00,43,00,34,00,37,00,39,00,46,00,33,\
  00,34,00,2d,00,38,00,45,00,30,00,44,00,2d,00,34,00,43,00,31,00,43,00,2d,00,\
  39,00,44,00,41,00,43,00,2d,00,43,00,31,00,30,00,37,00,31,00,41,00,36,00,43,\
  00,35,00,30,00,30,00,37,00,7d,00,00,00
"URLInfoAbout"=""
"URLUpdateInfo"=""
"VersionMajor"=dword:0000000a
"VersionMinor"=dword:00000000
"WindowsInstaller"=dword:00000001
"Version"=dword:0a00000a
"Language"=dword:00000409
"DisplayName"="Microsoft Visual C++ 2005 Redistributable - x86 8.1.30629.3138"
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\{0
C479F34-8E0D-4C1C-9DAC-C1071A6C5007}]
"AuthorizedCDFPrefix"=""
"Comments"="Build "
"Contact"=""
"DisplayVersion"="8.1.30629.3138"
```

Windows Registry Editor Version 5.00

"HelpLink"=""
"HelpTelephone"=""
"InstallDate"="20170205"
"InstallLocation"="C:\\Program Files\\VMware\\VMware Tools\\"
"InstallSource"="C:\\Users\\Admin\\AppData\\Local\\Temp\\{0C479F34-8E0D-4C1C-9DAC-C1071A6C5007}~setup\\"
"ModifyPath"=hex(2):4d,00,73,00,69,00,45,00,78,00,65,00,63,00,2e,00,65,00,78,\
  00,65,00,20,00,2f,00,49,00,7b,00,30,00,43,00,34,00,37,00,39,00,46,00,33,00,\
  34,00,2d,00,38,00,45,00,30,00,44,00,2d,00,34,00,43,00,31,00,43,00,2d,00,39,\
  00,44,00,41,00,43,00,2d,00,43,00,31,00,30,00,37,00,31,00,41,00,36,00,43,00,\
  35,00,30,00,30,00,37,00,7d,00,00,00
"Publisher"="Microsoft Corporation"
"Readme"=""
"Size"=""
"EstimatedSize"=dword:0001685f
"UninstallString"=hex(2):4d,00,73,00,69,00,45,00,78,00,65,00,63,00,2e,00,65,00,\
  78,00,65,00,20,00,2f,00,49,00,7b,00,30,00,43,00,34,00,37,00,39,00,46,00,33,\
  00,34,00,2d,00,38,00,45,00,30,00,44,00,2d,00,34,00,43,00,31,00,43,00,2d,00,\
  39,00,44,00,41,00,43,00,2d,00,43,00,31,00,30,00,37,00,31,00,41,00,36,00,43,\
  00,35,00,30,00,30,00,37,00,7d,00,00,00
"URLInfoAbout"=""
"URLUpdateInfo"=""
"VersionMajor"=dword:0000000a
"VersionMinor"=dword:00000000
"WindowsInstaller"=dword:00000001
"Version"=dword:0a00000a
"Language"=dword:00000409
"DisplayName"="Microsoft Visual C++ 2005 Redistributable - x86 8.1.30629.3138"
 	
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Installer\Products\43F974C0D0E8C1C4D9CA1C70A1C60570]
"ProductName"="Microsoft Visual C++ 2005 Redistributable - x86 8.1.30629.3138"
"PackageCode"="769916177BF4A6642B24C24DE19F5D48"
"Language"=dword:00000409
"Version"=dword:0a00000a
"Assignment"=dword:00000001
"AdvertiseFlags"=dword:00000184
"ProductIcon"="C:\\Windows\\Installer\\{0C479F34-8E0D-4C1C-9DAC-C1071A6C5007}"
"InstanceType"=dword:00000000
"AuthorizedLUAApp"=dword:00000000
"DeploymentFlags"=dword:00000003
"Clients"=hex(7):3a,00,00,00,00,00
 	
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Installer\Products\43F974C0D0E8C1C4D9CA1C70A1C60570\SourceList]

```
Windows Registry Editor Version 5.00
"PackageName"="VMware Tools64.msi"
"LastUsedSource"=hex(2):6e,00,3b,00,31,00,3b,00,43,00,3a,00,5c,00,55,00,73,00,\
  65,00,72,00,73,00,5c,00,41,00,64,00,6d,00,69,00,6e,00,5c,00,41,00,70,00,70,\
  00,44,00,61,00,74,00,61,00,5c,00,4c,00,6f,00,63,00,61,00,6c,00,5c,00,54,00,\
  65,00,6d,00,70,00,5c,00,7b,00,30,00,43,00,34,00,37,00,39,00,46,00,33,00,34,\
  00,2d,00,38,00,45,00,30,00,44,00,2d,00,34,00,43,00,31,00,43,00,2d,00,39,00,\
  44,00,41,00,43,00,2d,00,43,00,31,00,30,00,37,00,31,00,41,00,36,00,43,00,35,\
  00,30,00,30,00,37,00,7d,00,7e,00,73,00,65,00,74,00,75,00,70,00,5c,00,00,00

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Installer\Products\43F974C0D0E8C1C4
D9CA1C70A1C60570\SourceList\Media]
"1"=";"
"2"=";"
"3"=";"
"4"=";"
"5"=";"
"6"=";"
"7"=";"
"8"=";"
"9"=";"
"10"=";"
"11"=";"
"12"=";"
"13"=";"
"14"=";"
"15"=";"
"17"=";"
"18"=";"
"19"=";"
"20"=";"
"21"=";"
"22"=";"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\Installer\Products\43F974C0D0E8C1C4
D9CA1C70A1C60570\SourceList\Net]
"1"=hex(2):43,00,3a,00,5c,00,55,00,73,00,65,00,72,00,73,00,5c,00,41,00,64,00,\
  6d,00,69,00,6e,00,5c,00,41,00,70,00,70,00,44,00,61,00,74,00,61,00,5c,00,4c,\
  00,6f,00,63,00,61,00,6c,00,5c,00,54,00,65,00,6d,00,70,00,5c,00,7b,00,30,00,\
  43,00,34,00,37,00,39,00,46,00,33,00,34,00,2d,00,38,00,45,00,30,00,44,00,2d,\
```

Table 9: StealthyTools.reg on Attached zipped files

Appendix C
Registry Renames on VMware PowerShell script

```
$path = Get-ChildItem HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\
$results = $path | foreach-object {get-ItemProperty $_.pspath} | where {$_.DisplayName -match "VMware"} | where {$_.Publisher -match "VMware,"}
foreach ($result in $results){
$line = $result.pspath
set-ItemProperty -path $line DisplayName -value "MyWare"
set-ItemProperty -path $line Publisher -value "MyWare, Inc"
}
```

Table 10: registry Renames on VMware PowerShell script.ps1, Attached in zipped files

Appendix D
VirusTotal Results

21 engines detected this file

21 / 67

SHA-256	6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92
File name	malware.exe
File size	490 KB
Last analysis	2018-07-27 12:24:38 UTC

Detection | Details | Behavior | Community 1

Basic Properties

MD5	01fd682d16dfe26e180f4c7cd74cfb62
SHA-1	597fa0573848d44441f20b09c8b41169c132942d
Authentihash	2030c161cd04707bb49cb44e115793b674f7a79746226a3739f25f75be72f16a
Imphash	56c77f3392fa475e9972c5e94099a10c
File Type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 32-bit
SSDeep	6144:io+9eAJFquIHpGji+3WH7bhcKRhRGgqhcaCCytMujx3aST6/AKePRz6sBdwdhq02:P+QuIJei+EhcKR9R3aSXjPGPBmn
TRiD	Win32 Dynamic Link Library (generic) (38.3%) Win32 Executable (generic) (26.2%) OS/2 Executable (generic) (11.8%) Generic Win/DOS Executable (11.6%) DOS Executable Generic (11.6%)
File Size	490 KB

Tags

peexe

History

Creation Time	2014-10-14 08:18:51
First Submission	2014-11-18 08:30:31
Last Submission	2018-07-27 12:24:38
Last Analysis	2018-07-27 12:24:38

File Names

malware.exe
file-7706740_exe
6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92

Portable Executable Info

Header

Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2014-10-14 08:18:51
Entry Point	4768
Contained Sections	8

Sections

Name	Virtual Address	Virtual Size	Raw Size	Entropy	MD5
.text	4096	461640	461824	6.1	0b8cc6de10f7599a080799dc88261b21
.data	466944	600	1024	1.1	fa3a6789ad95497d492e3f04ef4c542c
.rdata	471040	27648	27648	5.25	2c75a15b7835393fcd86e041e7419e63

Figure 142: VirusTotal Results - 1

.bss	507904	27520	0	0	d41d8cd98f00b204e9800998ecf8427e
.idata	536576	3288	3584	5.01	cb9ca5b2eb102a48e266d1a379482165
.CRT	540672	24	512	0.12	f26044af392c5594ad34576aca15d1db
.tls	544768	32	512	0.22	b79dfdf69cb172a8497793b5d97c5214

Imports

ADVAPI32.DLL
- CryptAcquireContextW
- GetUserNameA
- CryptReleaseContext
- CryptGenRandom
- RegOpenKeyExA

KERNEL32.dll
- GetLastError
- EnterCriticalSection
- ReleaseMutex
- WaitForSingleObject
- IsDebuggerPresent
- ExitProcess
- TlsAlloc
- VirtualProtect
- DeleteCriticalSection
- GetAtomNameA
- AddAtomA
- FindAtomA
- TlsGetValue
- MultiByteToWideChar
- GetProcAddress
- GetComputerNameExA
- CreateMutexA
- IsDBCSLeadByteEx
- CreateSemaphoreA
- WideCharToMultiByte
- TlsFree
- GetModuleHandleA
- FindFirstFileA
- InterlockedExchange
- SetUnhandledExceptionFilter
- CloseHandle
- FindNextFileA
- ReleaseSemaphore
- InitializeCriticalSection
- VirtualQuery
- FindClose
- InterlockedDecrement
- Sleep
- TlsSetValue
- GetCurrentThreadId
- LeaveCriticalSection
- SetLastError
- InterlockedIncrement

SHELL32.DLL
- SHGetSpecialFolderPathA

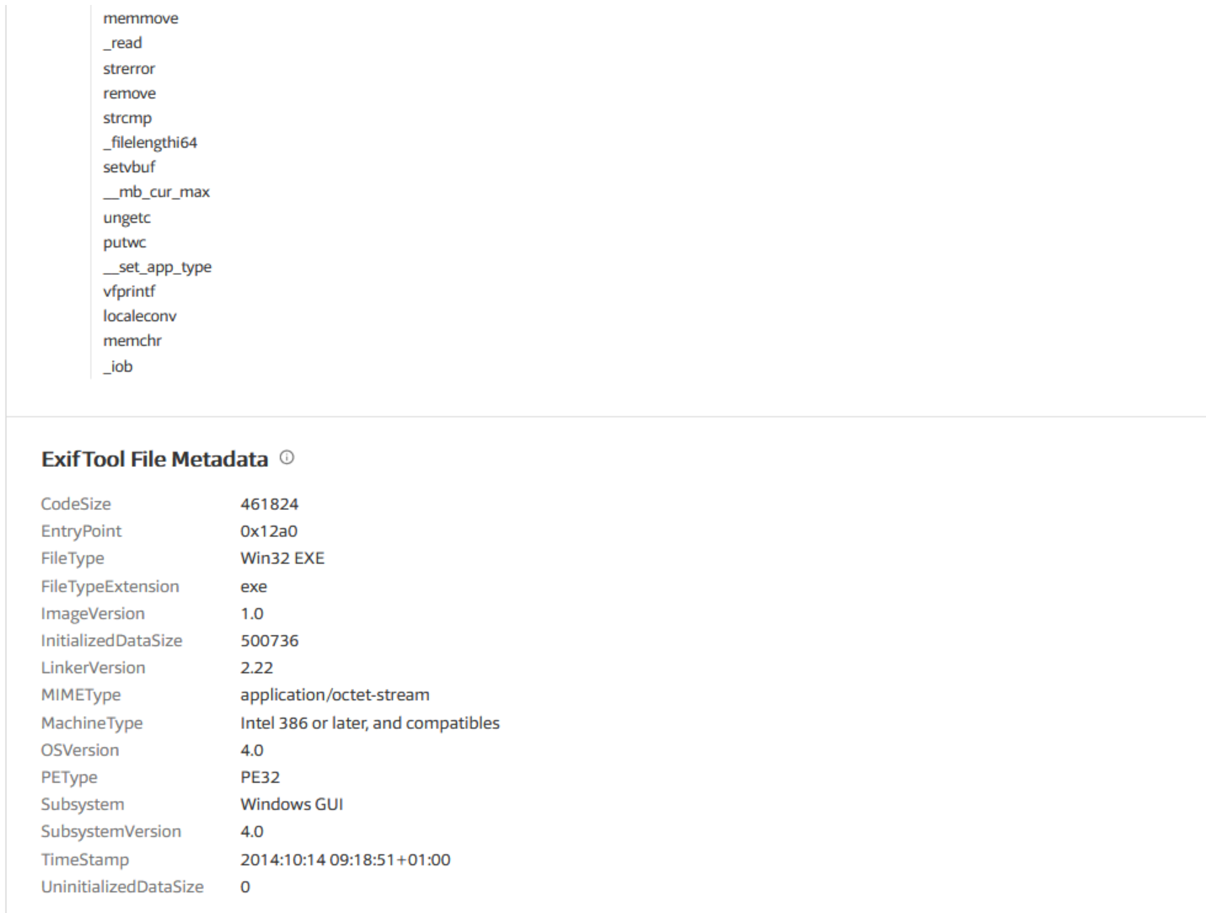
WSOCK32.DLL
- WSAStartup
- gethostbyname
- gethostname

msvcrt.dll
- __p__fmode
- malloc
- getc
- srand
- __p__environ

Figure 143: VirusTotal Results - 2

__p__environ
fgetc
realloc
fread
fclose
wcsftime
ungetwc
wcsxfrm
atexit
abort
_setmode
getwc
fflush
fopen
strlen
towupper
_cexit
fputc
iswctype
_errno
strtod
fwrite
fgetpos
strftime
_onexit
wcslen
fputs
exit
sprintf
putc
memcmp
strxfrm
rand
fsetpos
towlower
strchr
memset
_fdopen
wcscoll
time
free
getenv
setlocale
signal
atoi
_fstati64
__getmainargs
calloc
_write
strcoll
memcpy
_lseeki64
memmove
_read
strerror
remove
strcmp
_filelengthi64
setvbuf
__mb_cur_max
ungetc
putwc
__set_app_type
vfprintf
localeconv
memchr
_iob

Figure 144: VirusTotal Results - 3



memmove
_read
strerror
remove
strcmp
_filelengthi64
setvbuf
__mb_cur_max
ungetc
putwc
__set_app_type
vfprintf
localeconv
memchr
_iob

ExifTool File Metadata

CodeSize	461824
EntryPoint	0x12a0
FileType	Win32 EXE
FileTypeExtension	exe
ImageVersion	1.0
InitializedDataSize	500736
LinkerVersion	2.22
MIMEType	application/octet-stream
MachineType	Intel 386 or later, and compatibles
OSVersion	4.0
PEType	PE32
Subsystem	Windows GUI
SubsystemVersion	4.0
TimeStamp	2014:10:14 09:18:51+01:00
UninitializedDataSize	0

Figure 145: VirusTotal Results – 4

HybridAnalysis results

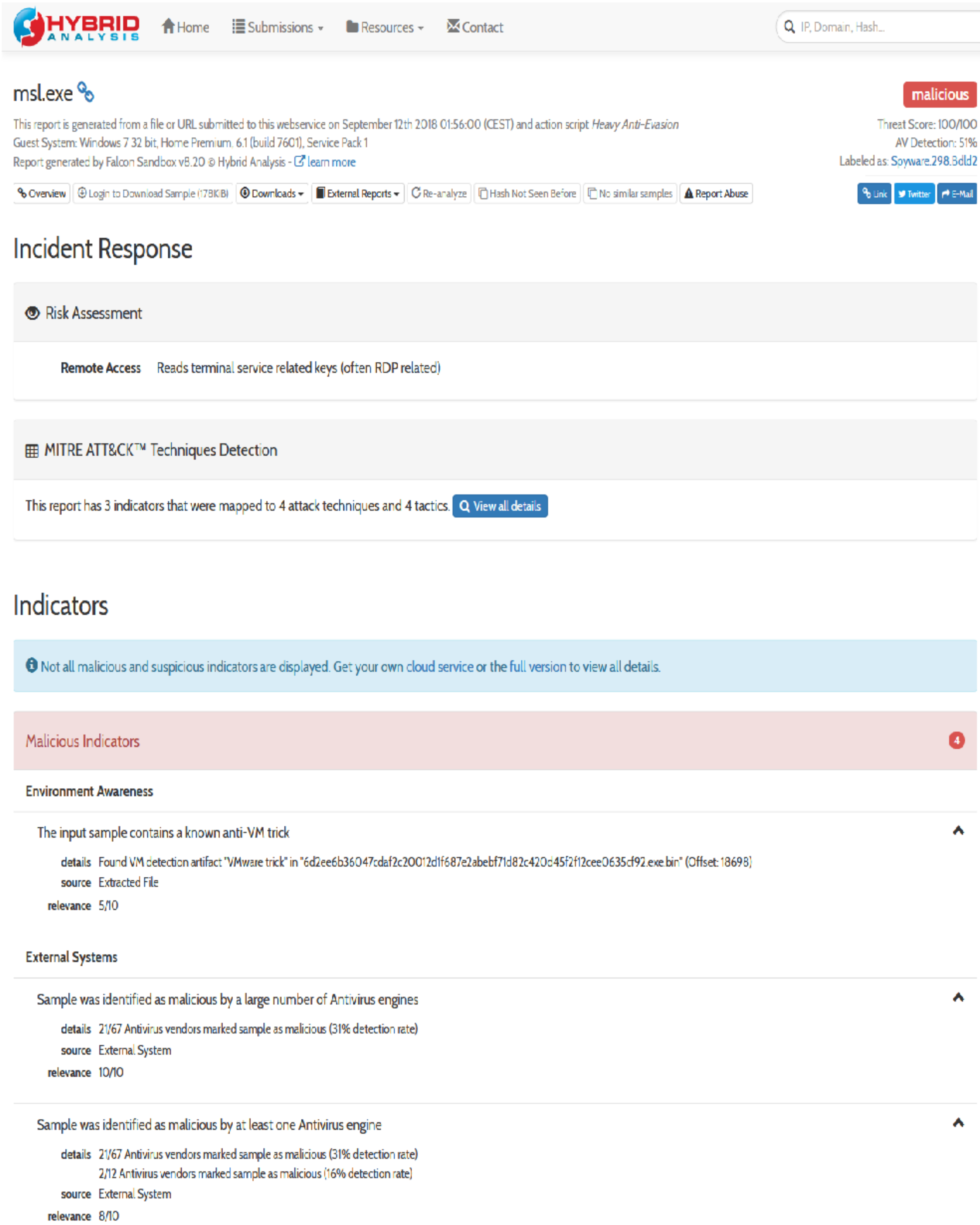


Figure 146: HybridAnalysis Results – 1

Suspicious Indicators 7

Environment Awareness

Contains ability to measure performance

details rdtsc from 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe (PID: 3756) (Show Stream)
source Hybrid Analysis Technology
relevance 10/10

Contains ability to query CPU information

details cpuid from 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe (PID: 3756) (Show Stream)
source Hybrid Analysis Technology
relevance 10/10
ATT&CK ID T1082 (Show technique in the MITRE ATT&CK™ matrix)

Remote Access Related

Reads terminal service related keys (often RDP related)

details "<Input Sample>" (Path: "HKLM\SYSTEM\CONTROLSET001\CONTROL\TERMINAL SERVER"; Key: "TSUSERENABLED")
source Registry Access
relevance 10/10
ATT&CK ID T1076 (Show technique in the MITRE ATT&CK™ matrix)

Unusual Characteristics

Creates a mutant that has been seen in the context of malware

details "gcc-shmem-tdm2-use_fc_key"
"gcc-shmem-tdm2-fc_key"
"gcc-shmem-tdm2-sjlj_once"
source Created Mutant
relevance 6/10

Found TLS callbacks

details "6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe.bin" has a TLS callback with entrypoint at 0x41a310
"6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe.bin" has a TLS callback with entrypoint at 0x41a2c0
source Static Parser
relevance 10/10
ATT&CK ID T1055 (Show technique in the MITRE ATT&CK™ matrix)

Imports suspicious APIs

details GetUserNameA
RegOpenKeyExA
IsDebuggerPresent
VirtualProtect
GetProcAddress
GetComputerNameExA
GetModuleHandleA
FindFirstFileA
FindNextFileA
Sleep
WSAStartup
source Static Parser
relevance 1/10

Figure 147: HybridAnalysis Results – 2

PE file contains unusual section name

details "6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe.bin" has a section named ".eh_fram"
"6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe.bin" has a section named ".CRT"
source Static Parser
relevance 10/10

Informative

Anti-Reverse Engineering

Contains ability to register a top-level exception handler (often used as anti-debugging trick)

details SetUnhandledExceptionFilter@KERNEL32.DLL from 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe (PID: 3756) (Show Stream)
SetUnhandledExceptionFilter@KERNEL32.DLL from 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92.exe (PID: 3756) (Show Stream)
source Hybrid Analysis Technology
relevance 1/10

PE file contains zero-size sections

details Raw size of "bss" is zero
source Static Parser
relevance 10/10

General

Creates mutants

details "\Sessions\1\BaseNamedObjects\gcc-shmem-tdm2-use_fc_key"
"\Sessions\1\BaseNamedObjects\gcc-shmem-tdm2-sjlj_once"
"\Sessions\1\BaseNamedObjects\gcc-shmem-tdm2-fc_key"
"gcc-shmem-tdm2-use_fc_key"
"gcc-shmem-tdm2-fc_key"
"gcc-shmem-tdm2-sjlj_once"
source Created Mutant
relevance 3/10

Network Related

Found potential URL in binary/memory

details Heuristic match: "tL<EtH<.tD"
source String
relevance 10/10

Figure 148: HybridAnalysis Results – 3

File Details

All Details: Off

msl.exe

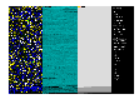
Filename	msl.exe
Size	490KiB (501760 bytes)
Type	peexe executable
Description	PE32 executable (GUI) Intel 80386 (stripped to external PDB), for MS Windows
Architecture	WINDOWS
SHA256	6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92

Resources

Icon

Visualization

Input File (PortEx)



Classification (TrID)

- 43.4% (.DLL) Win32 Dynamic Link Library (generic)
- 29.8% (.EXE) Win32 Executable (generic)
- 13.2% (.EXE) Generic Win/DOS Executable
- 13.2% (.EXE) DOS Executable Generic
- 0.2% (.VXD) VXD Driver

File Sections

Name	Entropy	Virtual Address	Virtual Size	Raw Size	MD5
.text	6.10374954224	0x1000	0x70b48	0x70c00	0b8cc6de10f7599a080799dc85261b21
.data	1.09623992959	0x72000	0x258	0x400	fa3a6789ad95497d492e3f04ef4c542c
.rdata	5.24681103496	0x73000	0x6c00	0x6c00	2c75a15b7835393fcd86e041e7419e63
.eh_fram	4.74049852252	0x7a000	0x14f8	0x1600	f834ed6184729a99174882df3a2b34ed
.bss	0	0x7c000	0x6b80	0x0	d41d8cd98f00b204e9800998ecf8427e
.idata	5.01426353705	0x83000	0xcd8	0xe00	cb9ca5b2eb102a48e256d1a379482165
.CRT	0.118369631259	0x84000	0x18	0x200	f26044af392c5594ad34576aca15d1db
.tls	0.22482003451	0x85000	0x20	0x200	b79dfdf69cb172a8497793b5d97c5214

File Imports

ADVAPI32.DLL | KERNEL32.dll | msvcrt.dll | SHELL32.DLL | WSOCK32.DLL

- CryptAcquireContextW
- CryptGenRandom
- CryptReleaseContext
- GetUserNameA
- RegOpenKeyExA

Figure 149: HybridAnalysis Results – 4

Appendix E
Win32 Portable Executable File Format

Section Names

Name	Description
.text	The default code section.
.data	The default read/write data section. Global variables typically go here.
.rdata	The default read-only data section. String literals and C++/COM vtables are examples of items put into .rdata.
.idata	The imports table. It has become common practice (either explicitly, or via linker default behavior) to merge the .idata section into another section, typically .rdata. By default, the linker only merges the .idata section into another section when creating a release mode executable.
.edata	The exports table. When creating an executable that exports APIs or data, the linker creates an .EXP file. The .EXP file contains an .edata section that's added into the final executable. Like the .idata section, the .edata section is often found merged into the .text or .rdata sections.
.rsrc	The resources. This section is read-only. However, it should not be named anything other than .rsrc, and should not be merged into other sections.
.bss	Uninitialized data. Rarely found in executables created with recent linkers. Instead, the VirtualSize of the executable's .data section is expanded to make enough room for uninitialized data.
.crt	Data added for supporting the C++ runtime (CRT). A good example is the function pointers that are used to call the constructors and destructors of static C++ objects. See the January 2001 Under The Hood column for details on this.
.tls	Data for supporting thread local storage variables declared with __declspec(thread). This includes the initial value of the data, as well as additional variables needed by the runtime.
.reloc	The base relocations in an executable. Base relocations are generally only needed for DLLs and not EXEs. In release mode, the linker doesn't emit base relocations for EXE files. Relocations can be removed when linking with the /FIXED switch.
.sdata	"Short" read/write data that can be addressed relative to the global pointer. Used for the IA-64 and other architectures that use a global pointer register. Regular-sized global variables on the IA-64 will go in this section.
.srdata	"Short" read-only data that can be addressed relative to the global pointer. Used on the IA-64 and other architectures that use a global pointer register.
.pdata	The exception table. Contains an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures, which are CPU-specific. Pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION slot in the DataDirectory. Used for architectures with table-based exception handling, such as the IA-64. The only architecture that doesn't use table-based exception handling is the x86.
.debug$S	Codeview format symbols in the OBJ file. This is a stream of variable-length CodeView format symbol records.
.debug$T	Codeview format type records in the OBJ file. This is a stream of variable-length CodeView format type records.
.debug$P	Found in the OBJ file when using precompiled headers.
.drectve	Contains linker directives and is only found in OBJs. Directives are ASCII strings that could be passed on the linker command line. For instance: `-defaultlib:LIBC` Directives are separated by a space character.
.didat	Delayload import data. Found in executables built in nonrelease mode. In release mode, the delayload data is merged into another section.

IMAGE_EXPORT_DIRECTORY Structure Members

Size	Member	Description
DWORD	Characteristics	Flags for the exports. Currently, none are defined.
DWORD	TimeDateStamp	The time/date that the exports were created. This field has the same definition as the IMAGE_NT_HEADERS.FileHeader. TimeDateStamp (number of seconds since 1/1/1970 GMT).
WORD	MajorVersion	The major version number of the exports. Not used, and set to 0.
WORD	MinorVersion	The minor version number of the exports. Not used, and set to 0.
DWORD	Name	A relative virtual address (RVA) to an ASCII string with the DLL name associated with these exports (for example, KERNEL32.DLL).
DWORD	Base	This field contains the starting ordinal value to be used for this executable's exports. Normally, this value is 1, but it's not required to be so. When looking up an export by ordinal, the value of this field is subtracted from the ordinal, with the result used as a zero-based index into the Export Address Table (EAT).
DWORD	NumberOfFunctions	The number of entries in the EAT. Note that some entries may be 0, indicating that no code/data is exported with that ordinal value.

Figure 150: Section Names & IMAGE_EXPORT_DIRECTORY Structure Members

DWORD	NumberOfNames	The number of entries in the Export Names Table (ENT). This value will always be less than or equal to the NumberOf-Functions field. It will be less when there are symbols exported by ordinal only. It can also be less if there are numeric gaps in the assigned ordinals. This field is also the size of the export ordinal table (below).
DWORD	AddressOfFunctions	The RVA of the EAT. The EAT is an array of RVAs. Each nonzero RVA in the array corresponds to an exported symbol.
DWORD	AddressOfNames	The RVA of the ENT. The ENT is an array of RVAs to ASCII strings. Each ASCII string corresponds to a symbol exported by name. This table is sorted so that the ASCII strings are in order. This allows the loader to do a binary search when looking for an exported symbol. The sorting of the names is binary (like the C++ RTL strcmp function provides), rather than a locale-specific alphabetic ordering.
DWORD	AddressOfNameOrdinals	The RVA of the export ordinal table. This table is an array of WORDs. This table maps an array index from the ENT into the corresponding export address table entry.

KERNEL32 Exports

```
exports table:
  Name:              KERNEL32.dll
  Characteristics: 00000000
  TimeDateStamp:   3B7DDFD8 -> Fri Aug 17 23:24:08 2001
  Version:           0.00
  Ordinal base:    00000001
  # of functions:  000003A0
  # of Names:        000003A0

  Entry Pt  Ordn  Name
  00012ADA     1  ActivateActCtx
  000082C2     2  AddAtomA
•••remainder of exports omitted
```

IMAGE_IMPORT_DESCRIPTOR Structure

Size	Member	Description
DWORD	OriginalFirstThunk	This field is badly named. It contains the RVA of the Import Name Table (INT). This is an array of IMAGE_THUNK_DATA structures. This field is set to 0 to indicate the end of the array of IMAGE_IMPORT_DESCRIPTORs.
DWORD	TimeDateStamp	This is 0 if this executable is not bound against the imported DLL. When binding in the old style (see the section on Binding), this field contains the time/date stamp (number of seconds since 1/1/1970 GMT) when the binding occurred. When binding in the new style, this field is set to -1.
DWORD	ForwarderChain	This is the Index of the first forwarded API. Set to -1 if no forwarders. Only used for old-style binding, which could not handle forwarded APIs efficiently.
DWORD	Name	The RVA of the ASCII string with the name of the imported DLL.
DWORD	FirstThunk	Contains the RVA of the Import Address Table (IAT). This is array of IMAGE_THUNK_DATA structures.

ImgDelayDescr Structure

Size	Member	Description
DWORD	grAttrs	The attributes for this structure. Currently, the only flag defined is dlattrRva (1), indicating that the address fields in the structure should be treated as RVAs, rather than virtual addresses.
RVA	rvaDLLName	An RVA to a string with the name of the imported DLL. This string is passed to LoadLibrary.
RVA	rvaHmod	An RVA to an HMODULE-sized memory location. When the Delayloaded DLL is brought into memory, its HMODULE is stored at this location.
RVA	rvaIAT	An RVA to the Import Address Table for this DLL. This is the same format as a regular IAT.
RVA	rvaINT	An RVA to the Import Name Table for this DLL. This is the same format as a regular INT.
RVA	rvaBoundIAT	An RVA of the optional bound IAT. An RVA to a bound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently, this copy of the IAT is not actually bound, but this feature may be added in future versions of the BIND program.
RVA	rvaUnloadIAT	An RVA of the optional copy of the original IAT. An RVA to an unbound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently always set to 0.
DWORD	dwTimeStamp	The date/time stamp of the delayload imported DLL. Normally set to 0.

Resources from ADVAPI32.DLL

```
Resources (RVA: 6B000)
ResDir (0) Entries:03 (Named:01, ID:02) TimeDate:00000000
    ———————————————————————————————
    ResDir (MOFDATA) Entries:01 (Named:01, ID:00) TimeDate:00000000
```

Figure 151: Kernel32 Exports, IMAGE_IMPORT_DESCRIPTOR Structure, ImgDelayDescr Structure, Resources from ADVAPI32.DLL

```
        ResDir (MOFRESOURCENAME) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000128
            DataRVA: 6B6F0  DataSize: 190F5  CodePage: 0
    -------------------------------
    ResDir (STRING) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (C36) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000138
            DataRVA: 6B1B0  DataSize: 0053C  CodePage: 0
    -------------------------------
    ResDir (RCDATA) Entries:01 (Named:00, ID:01) TimeDate:00000000
        ResDir (66) Entries:01 (Named:00, ID:01) TimeDate:00000000
            ID: 00000409  DataEntryOffs: 00000148
            DataRVA: 85908  DataSize: 0005C  CodePage: 0
```

Fields of IMAGE_DEBUG_DIRECTORY

Size	Member	Description
DWORD	Characteristics	Unused and set to 0.
DWORD	TimeDateStamp	The time/date stamp of this debug information (number of seconds since 1/1/1970, GMT).
WORD	MajorVersion	The major version of this debug information. Unused.
WORD	MinorVersion	The minor version of this debug information. Unused.

DWORD	Type	The type of the debug information. The following types are the most commonly encountered:
		`IMAGE_DEBUG_TYPE_COFF`
		`IMAGE_DEBUG_TYPE_CODEVIEW // Including PDB files`
		`IMAGE_DEBUG_TYPE_FPO // Frame pointer omission`
		`IMAGE_DEBUG_TYPE_MISC // IMAGE_DEBUG_MISC`
		`IMAGE_DEBUG_TYPE_OMAP_TO_SRC`
		`IMAGE_DEBUG_TYPE_OMAP_FROM_SRC`
		`IMAGE_DEBUG_TYPE_BORLAND // Borland format`

DWORD	SizeOfData	The size of the debug data in this file. Doesn't count the size of external debug files such as .PDBs.
DWORD	AddressOfRawData	The RVA of the debug data, when mapped into memory. Set to 0 if the debug data isn't mapped in.
DWORD	PointerToRawData	The file offset of the debug data (not an RVA).

IMAGE_COR20_HEADER Structure

Type	Member	Description
DWORD	cb	Size of the header in bytes.
WORD	MajorRuntimeVersion	The minimum version of the runtime required to run this program. For the first release of .NET, this value is 2.
WORD	MinorRuntimeVersion	The minor portion of the version. Currently 0.
IMAGE_DATA_DIRECTORY	MetaData	The RVA to the metadata tables.

DWORD	Flags	Flag values containing attributes for this image. These values are currently defined as:
		`COMIMAGE_FLAGS_ILONLY // Image contains only IL code that`
		`// is not required to run on a specific CPU.`
		`COMIMAGE_FLAGS_32BITREQUIRED // Only runs in 32-bit processes.`
		`COMIMAGE_FLAGS_IL_LIBRARY`
		`STRONGNAMESIGNED // Image is signed with hash data`
		`COMIMAGE_FLAGS_TRACKDEBUGDATA // Causes the JIT/runtime to`
		`// keep debug information`
		`// around for methods.`

DWORD	EntryPointToken	Token for the MethodDef of the entry point for the image. The .NET runtime calls this method to begin managed execution in the file.
IMAGE_DATA_DIRECTORY	Resources	The RVA and size of the .NET resources.
IMAGE_DATA_DIRECTORY	StrongNameSignature	The RVA of the strong name hash data.
IMAGE_DATA_DIRECTORY	CodeManagerTable	The RVA of the code manager table. A code manager contains the code required to obtain the state of a running program (such as tracing the stack and track GC references).
IMAGE_DATA_DIRECTORY	VTableFixups	The RVA of an array of function pointers that need fixups. This is for support of unmanaged C++ vtables.
IMAGE_DATA_DIRECTORY	ExportAddressTableJumps	The RVA to an array of RVAs where export JMP thunks are written. These thunks allow managed methods to be exported so that unmanaged code can call them.
IMAGE_DATA_DIRECTORY	ManagedNativeHeader	For internal use of the .NET runtime in memory. Set to 0 in the executable.

Figure 152: Fields of IMAGE_DEBUG_DIRECTORY, IMAGE_COR20_HEADER Structure

IMAGE_TLS_DIRECTORY Structure

Size	Member	Description
DWORD	StartAddressOfRawData	The beginning address of a range of memory used to initialize a new thread's TLS data in memory.
DWORD	EndAddressOfRawData	The ending address of the range of memory used to initialize a new thread's TLS data in memory.
DWORD	AddressOfIndex	When the executable is brought into memory and a .tls section is present, the loader allocates a TLS handle via TlsAlloc. It stores the handle at the address given by this field. The runtime library uses this index to locate the thread local data.
DWORD	AddressOfCallBacks	Address of an array of PIMAGE_TLS_CALLBACK function pointers. When a thread is created or destroyed, each function in the list is called. The end of the list is indicated by a pointer-sized variable set to 0. In normal Visual C++ executables, this list is empty.
DWORD	SizeOfZeroFill	The size in bytes of the initialization data, beyond the initialized data delimited by the StartAddressOfRawData and EndAddressOfRawData fields. All per-thread data after this range is initialized to 0.
DWORD	Characteristics	Reserved. Currently set to 0.

Figure 153: IMAGE_TLS_DIRECTORY Structure

Appendix F
List of Imports

0048327C	CryptAcquireContextW	ADVAPI32
00483280	CryptGenRandom	ADVAPI32
00483284	CryptReleaseContext	ADVAPI32
00483288	GetUserNameA	ADVAPI32
0048328C	RegOpenKeyExA	ADVAPI32
00483294	AddAtomA	KERNEL32
00483298	CloseHandle	KERNEL32
0048329C	CreateMutexA	KERNEL32
004832A0	CreateSemaphoreA	KERNEL32
004832A4	DeleteCriticalSection	KERNEL32
004832A8	EnterCriticalSection	KERNEL32
004832AC	ExitProcess	KERNEL32
004832B0	FindAtomA	KERNEL32
004832B4	FindClose	KERNEL32
004832B8	FindFirstFileA	KERNEL32
004832BC	FindNextFileA	KERNEL32
004832C0	GetAtomNameA	KERNEL32
004832C4	GetComputerNameExA	KERNEL32
004832C8	GetCurrentThreadId	KERNEL32
004832CC	GetLastError	KERNEL32
004832D0	GetModuleHandleA	KERNEL32
004832D4	GetProcAddress	KERNEL32
004832D8	InitializeCriticalSection	KERNEL32
004832DC	InterlockedDecrement	KERNEL32
004832E0	InterlockedExchange	KERNEL32
004832E4	InterlockedIncrement	KERNEL32
004832E8	IsDBCSLeadByteEx	KERNEL32
004832EC	IsDebuggerPresent	KERNEL32
004832F0	LeaveCriticalSection	KERNEL32
004832F4	MultiByteToWideChar	KERNEL32
004832F8	ReleaseMutex	KERNEL32
004832FC	ReleaseSemaphore	KERNEL32
00483300	SetLastError	KERNEL32
00483304	SetUnhandledExceptionFilter	KERNEL32
00483308	Sleep	KERNEL32
0048330C	TlsAlloc	KERNEL32
00483310	TlsFree	KERNEL32
00483314	TlsGetValue	KERNEL32
00483318	TlsSetValue	KERNEL32
0048331C	VirtualProtect	KERNEL32
00483320	VirtualQuery	KERNEL32
00483324	WaitForSingleObject	KERNEL32
00483328	WideCharToMultiByte	KERNEL32
00483330	_fdopen	msvcrt
00483334	_read	msvcrt
00483338	_write	msvcrt
00483340	__getmainargs	msvcrt
00483344	__mb_cur_max	msvcrt
00483348	__p__environ	msvcrt
0048334C	__p__fmode	msvcrt

0048327C	CryptAcquireContextW	ADVAPI32
00483350	__set_app_type	msvcrt
00483354	_cexit	msvcrt
00483358	_errno	msvcrt
0048335C	_filelengthi64	msvcrt
00483360	_fstati64	msvcrt
00483364	_iob	msvcrt
00483368	_lseeki64	msvcrt
0048336C	_onexit	msvcrt
00483370	_setmode	msvcrt
00483374	abort	msvcrt
00483378	atexit	msvcrt
0048337C	atoi	msvcrt
00483380	calloc	msvcrt
00483384	exit	msvcrt
00483388	fclose	msvcrt
0048338C	fflush	msvcrt
00483390	fgetc	msvcrt
00483394	fgetpos	msvcrt
00483398	fopen	msvcrt
0048339C	fputc	msvcrt
004833A0	fputs	msvcrt
004833A4	fread	msvcrt
004833A8	free	msvcrt
004833AC	fsetpos	msvcrt
004833B0	fwrite	msvcrt
004833B4	getc	msvcrt
004833B8	getenv	msvcrt
004833BC	getwc	msvcrt
004833C0	iswctype	msvcrt
004833C4	localeconv	msvcrt
004833C8	malloc	msvcrt
004833CC	memchr	msvcrt
004833D0	memcmp	msvcrt
004833D4	memcpy	msvcrt
004833D8	memmove	msvcrt
004833DC	memset	msvcrt
004833E0	putc	msvcrt
004833E4	putwc	msvcrt
004833E8	rand	msvcrt
004833EC	realloc	msvcrt
004833F0	remove	msvcrt
004833F4	setlocale	msvcrt
004833F8	setvbuf	msvcrt
004833FC	signal	msvcrt
00483400	sprintf	msvcrt
00483404	srand	msvcrt
00483408	strchr	msvcrt
0048340C	strcmp	msvcrt
00483410	strcoll	msvcrt
00483414	strerror	msvcrt
00483418	strftime	msvcrt
0048341C	strlen	msvcrt

0048327C	CryptAcquireContextW	ADVAPI32
00483420	strtod	msvcrt
00483424	strxfrm	msvcrt
00483428	time	msvcrt
0048342C	towlower	msvcrt
00483430	towupper	msvcrt
00483434	ungetc	msvcrt
00483438	ungetwc	msvcrt
0048343C	vfprintf	msvcrt
00483440	wcscoll	msvcrt
00483444	wcsftime	msvcrt
00483448	wcslen	msvcrt
0048344C	wcsxfrm	msvcrt
00483454	SHGetSpecialFolderPathA	SHELL32
0048345C	WSAStartup	WSOCK32
00483460	gethostbyname	WSOCK32
00483464	gethostname	WSOCK32

Appendix G
PortExAnalyzer PE file report

```
Report For rtms.exe
*******************

file size 0x7a800
full path C:\Users\Windows7Flare\Downloads\rtms.exe

Section Table
*************
                         1. .text        2. .data        3. .rdata       4. .eh_fram
------------------------------------------------------------------------------------------
Entropy                  6.10            1.10            5.25            4.74
Pointer To Raw Data      0x400           0x71000         0x71400         0x78000
Size Of Raw Data         0x70c00         0x400           0x6c00          0x1600
Physical End             0x71000         0x71400         0x78000         0x79600
Virtual Address          0x1000          0x72000         0x73000         0x7a000
Virtual Size             0x70b48         0x258           0x6c00          0x14f8
-> actual virtual size   0x71000         0x1000          0x7000          0x2000
Pointer To Relocations   0x0             0x0             0x0             0x0
Number Of Relocations    0x0             0x0             0x0             0x0
Pointer To Line Numbers  0x0             0x0             0x0             0x0
Number Of Line Numbers   0x0             0x0             0x0             0x0
Code                     x
Initialized Data         x               x               x               x
Align 1 Byte             x                                               x
Align 2 Bytes                            x               x               x
Align 4 Bytes            x               x               x               x
Align 8 Bytes            x               x               x
Align 16 Bytes           x               x               x               x
Align 32 Bytes           x               x               x               x
Align 64 Bytes           x               x               x               x
Align 256 Bytes          x                                               x
Align 512 Bytes                          x               x               x
Align 1024 Bytes         x               x               x               x
Align 2048 Bytes         x               x               x
Align 4096 Bytes         x               x               x               x
Align 8192 Bytes         x               x               x               x
Execute                  x
Read                     x               x               x               x
Write                    x               x

                         5. .bss         6. .idata       7. .CRT         8. .tls
------------------------------------------------------------------------------------------
Entropy                  0.00            5.01            0.12            0.22
Pointer To Raw Data      0x79600         0x79600         0x7a400         0x7a600
Size Of Raw Data         0x0             0xe00           0x200           0x200
Physical End             0x79600         0x7a400         0x7a600         0x7a800
Virtual Address          0x7c000         0x83000         0x84000         0x85000
Virtual Size             0x6b80          0xcd8           0x18            0x20
-> actual virtual size   0x7000          0x1000          0x1000          0x1000
Pointer To Relocations   0x0             0x0             0x0             0x0
Number Of Relocations    0x0             0x0             0x0             0x0
Pointer To Line Numbers  0x0             0x0             0x0             0x0
Number Of Line Numbers   0x0             0x0             0x0             0x0
Initialized Data                         x               x               x
Uninitialized Data       x
Align 1 Byte                             x               x               x
Align 2 Bytes            x               x               x               x
Align 4 Bytes            x               x               x               x
Align 8 Bytes            x
Align 16 Bytes           x               x               x               x
Align 32 Bytes           x               x               x               x
Align 64 Bytes           x               x               x               x
Align 256 Bytes                          x               x               x
Align 512 Bytes          x               x               x               x
Align 1024 Bytes         x               x               x               x
Align 2048 Bytes         x
```

```
Report For rtms.exe
Align 4096 Bytes        x               x               x               x
Align 8192 Bytes        x               x               x               x
Read                    x               x               x               x
Write                   x               x               x               x

MSDOS Header
************

description                           value          file offset
---------------------------------------------------------------------
signature word                        0x5a4d         0x0
last page size                        0x90           0x2
file pages                            0x3            0x4
relocation items                      0x0            0x6
header paragraphs                     0x4            0x8
minimum number of paragraphs allocated 0x0           0xa
maximum number of paragraphs allocated 0xffff        0xc
initial SS value                      0x0            0xe
initial SP value                      0xb8           0x10
complemented checksum                 0x0            0x12
initial IP value                      0x0            0x14
pre-relocated initial CS value        0x0            0x16
relocation table offset               0x40           0x18
overlay number                        0x0            0x1a
Reserved word 0x1c                    0x0            0x1c
Reserved word 0x1e                    0x0            0x1e
Reserved word 0x20                    0x0            0x20
Reserved word 0x22                    0x0            0x22
OEM identifier                        0x0            0x24
OEM information                       0x0            0x26
Reserved word 0x28                    0x0            0x28
Reserved word 0x2a                    0x0            0x2a
Reserved word 0x2c                    0x0            0x2c
Reserved word 0x2f                    0x0            0x2e
Reserved word 0x30                    0x0            0x30
Reserved word 0x32                    0x0            0x32
Reserved word 0x34                    0x0            0x34
Reserved word 0x36                    0x0            0x36
Reserved word 0x38                    0x0            0x38
Reserved word 0x3a                    0x0            0x3a
PE signature offset                   0x80           0x3c

COFF File Header
****************

time date stamp  Oct 14, 2014 11:18:51 AM
machine type     Intel 386 or later processors and compatible processors
characteristics  * Image only, Windows CE, and Windows NT and later.
                 * Image only.
                 * COFF line numbers have been removed. DEPRECATED
                 * COFF symbol table entries for local symbols have been removed. DEPRECATED
                 * Machine is based on a 32-bit-word architecture.
                 * Debugging is removed from the image file.

description                         value          file offset
-------------------------------------------------------------------
machine type                        0x14c          0x84
number of sections                  0x8            0x86
time date stamp                     0x543cdc6b     0x88
pointer to symbol table (deprecated) 0x0           0x8c
number of symbols (deprecated)      0x0            0x90
size of optional header             0xe0           0x94
characteristics                     0x30f          0x96

Optional Header
***************

Magic Number: PE32, normal executable file

Entry Point is in section 1 with name .text
```

```
Report For rtms.exe
No DLL Characteristics
Subsystem:           The Windows graphical user interface (GUI) subsystem

standard field                       value            file offset
-----------------------------------------------------------------------
magic number                         0x10b            0x98
major linker version                 0x2              0x9a
minor linker version                 0x16             0x9b
size of code                         0x70c00          0x9c
size of initialized data             0x7a400          0xa0
size of unitialized data             0x0              0xa4
address of entry point               0x12a0           0xa8
address of base of code              0x1000           0xac
address of base of data              0x72000          0xb0

windows field                        value            file offset
-----------------------------------------------------------------------
image base                           0x400000         0xb4
section alignment in bytes           0x1000           0xb8
file alignment in bytes              0x200            0xbc
major operating system version       0x4              0xc0
minor operating system version       0x0              0xc2
major image version                  0x1              0xc4
minor image version                  0x0              0xc6
major subsystem version              0x4              0xc8
minor subsystem version              0x0              0xca
win32 version value (reserved)       0x0              0xcc
size of image in bytes               0x86000          0xd0
size of headers                      0x400            0xd4
checksum                             0x7bae2          0xd8
subsystem                            0x2              0xdc
dll characteristics                  0x0              0xde
size of stack reserve                0x200000         0xe0
size of stack commit                 0x1000           0xe4
size of heap reserve                 0x100000         0xe8
size of heap commit                  0x1000           0xec
loader flags (reserved)              0x0              0xf0
number of rva and sizes              0x10             0xf4

data directory          rva              -> offset        size             in section       file
offset
-----------------------------------------------------------------------------------------------
--------------
import table           0x83000          0x79600          0xcd8            6 .idata         0x100
TLS table              0x85000          0x7a600          0x18             8 .tls           0x140
IAT                    0x8327c          0x7987c          0x1f0            6 .idata         0x158

Imports
*******

ADVAPI32.DLL
------------
[Registry]
rva: 0x8309c, va: 0x48308c, hint: 413, name: RegOpenKeyExA -> Opens the specified registry key.

[System Information]
rva: 0x83098, va: 0x48308c, hint: 245, name: GetUserNameA -> Retrieves the user name of the
current thread.

[Cryptography Functions] <Key Generation/Exchange>
rva: 0x83090, va: 0x48308c, hint: 110, name: CryptGenRandom -> Generates random data.

[Cryptography Functions] <Service Provider>
rva: 0x8308c, va: 0x48308c, hint: 94, name: CryptAcquireContextW -> Acquires a handle to the
current user's key container within a particular CSP.
rva: 0x83094, va: 0x48308c, hint: 120, name: CryptReleaseContext -> Releases the handle acquired
by the CryptAcquireContext function.
```

```
Report For rtms.exe
KERNEL32.dll
------------
[Error Handling]
rva: 0x830dc, va: 0x4830a4, hint: 510, name: GetLastError -> Retrieves the calling thread's last-error code value.
rva: 0x83110, va: 0x4830a4, hint: 1091, name: SetLastError -> Sets the last-error code for the calling thread.

[Memory Management] <Virtual Memory>
rva: 0x8312c, va: 0x4830a4, hint: 1213, name: VirtualProtect -> Changes the access protection on a region of committed pages in the virtual address space of the calling process.
rva: 0x83130, va: 0x4830a4, hint: 1215, name: VirtualQuery -> Provides information about a range of pages in the virtual address space of the calling process.

[Dynamic-Link Library]
rva: 0x830e0, va: 0x4830a4, hint: 529, name: GetModuleHandleA -> Retrieves a module handle for the specified module.
rva: 0x830e4, va: 0x4830a4, hint: 577, name: GetProcAddress -> Retrieves the address of an exported function or variable from the specified DLL.

[Synchronization] <Interlocked>
rva: 0x830ec, va: 0x4830a4, hint: 743, name: InterlockedDecrement -> Decrements (decreases by one) the value of the specified 32-bit variable as an atomic operation.
rva: 0x830f0, va: 0x4830a4, hint: 744, name: InterlockedExchange -> Sets a 32-bit variable to the specified value as an atomic operation.
rva: 0x830f4, va: 0x4830a4, hint: 747, name: InterlockedIncrement -> Increments (increases by one) the value of the specified 32-bit variable as an atomic operation.

[Structured Exception Handling]
rva: 0x83114, va: 0x4830a4, hint: 1140, name: SetUnhandledExceptionFilter -> Enables an application to supersede the top-level exception handler of each thread and process.

[Synchronization] <Mutex>
rva: 0x830ac, va: 0x4830a4, hint: 154, name: CreateMutexA -> Creates or opens a named or unnamed mutex object.
rva: 0x83108, va: 0x4830a4, hint: 974, name: ReleaseMutex -> Releases ownership of the specified mutex object.

[Debugging]
rva: 0x830fc, va: 0x4830a4, hint: 764, name: IsDebuggerPresent -> Determines whether the calling process is being debugged by a user-mode debugger.

[Synchronization] <Wait>
rva: 0x83134, va: 0x4830a4, hint: 1223, name: WaitForSingleObject -> Waits until the specified object is in the signaled state or the time-out interval elapses.

[Process and Thread] <Process>
rva: 0x830bc, va: 0x4830a4, hint: 279, name: ExitProcess -> Ends the calling process and all its threads.

[Process and Thread] <Thread>
rva: 0x830d8, va: 0x4830a4, hint: 451, name: GetCurrentThreadId -> Retrieves the thread identifier of the calling thread.
rva: 0x83118, va: 0x4830a4, hint: 1152, name: Sleep -> Suspends the execution of the current thread for a specified interval.
rva: 0x8311c, va: 0x4830a4, hint: 1171, name: TlsAlloc -> Allocates a thread local storage (TLS) index.
rva: 0x83120, va: 0x4830a4, hint: 1172, name: TlsFree -> Releases a TLS index.
rva: 0x83124, va: 0x4830a4, hint: 1173, name: TlsGetValue -> Retrieves the value in the calling thread's TLS slot for a specified TLS index.
rva: 0x83128, va: 0x4830a4, hint: 1174, name: TlsSetValue -> Stores a value in the calling thread's TLS slot for a specified TLS index.

[File Management]
rva: 0x830c4, va: 0x4830a4, hint: 300, name: FindClose -> Closes a file search handle opened by the FindFirstFile, FindFirstFileEx, FindFirstFileNameW, FindFirstFileNameTransactedW, FindFirstFileTransacted, FindFirstStreamTransactedW, or FindFirstStreamW functions.
rva: 0x830c8, va: 0x4830a4, hint: 304, name: FindFirstFileA -> Searches a directory for a file or subdirectory with a name that matches a specific name (or partial name if wildcards are used).
```

```
Report For rtms.exe
rva: 0x830cc, va: 0x4830a4, hint: 321, name: FindNextFileA -> Continues a file search from a
previous call to the FindFirstFile, FindFirstFileEx, or FindFirstFileTransacted functions.

[Atom]
rva: 0x830a4, va: 0x4830a4, hint: 3, name: AddAtomA -> no description
rva: 0x830c0, va: 0x4830a4, hint: 298, name: FindAtomA -> no description
rva: 0x830d0, va: 0x4830a4, hint: 363, name: GetAtomNameA -> no description

[System Information]
rva: 0x830d4, va: 0x4830a4, hint: 395, name: GetComputerNameExA -> Retrieves the NetBIOS or DNS
name of the local computer.

[Synchronization] <Critical section>
rva: 0x830b4, va: 0x4830a4, hint: 207, name: DeleteCriticalSection -> Releases all resources
used by an unowned critical section object.
rva: 0x830b8, va: 0x4830a4, hint: 236, name: EnterCriticalSection -> Waits for ownership of the
specified critical section object.
rva: 0x830e8, va: 0x4830a4, hint: 734, name: InitializeCriticalSection -> Initializes a critical
section object.
rva: 0x83100, va: 0x4830a4, hint: 814, name: LeaveCriticalSection -> Releases ownership of the
specified critical section object.

[Unicode and Character Set]
rva: 0x830f8, va: 0x4830a4, hint: 763, name: IsDBCSLeadByteEx -> Determines if a specified
character is potentially a lead byte.
rva: 0x83104, va: 0x4830a4, hint: 860, name: MultiByteToWideChar -> Maps a character string to
a UTF-16 (wide character) string.
rva: 0x83138, va: 0x4830a4, hint: 1247, name: WideCharToMultiByte -> Maps a UTF-16 (wide
character) string to a new character string.

[Handle and Object]
rva: 0x830a8, va: 0x4830a4, hint: 82, name: CloseHandle -> Closes an open object handle.

[Synchronization] <Semaphore>
rva: 0x830b0, va: 0x4830a4, hint: 169, name: CreateSemaphoreA -> Creates or opens a named or
unnamed semaphore object.
rva: 0x8310c, va: 0x4830a4, hint: 978, name: ReleaseSemaphore -> Increases the count of the
specified semaphore object by a specified amount.


msvcrt.dll
----------
[Other]
rva: 0x83140, va: 0x483140, hint: 23, name: _fdopen
rva: 0x83144, va: 0x483140, hint: 64, name: _read
rva: 0x83148, va: 0x483140, hint: 109, name: _write


msvcrt.dll
----------
[Other]
rva: 0x83150, va: 0x483150, hint: 55, name: __getmainargs
rva: 0x83154, va: 0x483150, hint: 65, name: __mb_cur_max
rva: 0x83158, va: 0x483150, hint: 77, name: __p__environ
rva: 0x8315c, va: 0x483150, hint: 79, name: __p__fmode
rva: 0x83160, va: 0x483150, hint: 99, name: __set_app_type
rva: 0x83164, va: 0x483150, hint: 147, name: _cexit
rva: 0x83168, va: 0x483150, hint: 182, name: _errno
rva: 0x8316c, va: 0x483150, hint: 203, name: _filelengthi64
rva: 0x83170, va: 0x483150, hint: 224, name: _fstati64
rva: 0x83174, va: 0x483150, hint: 266, name: _iob
rva: 0x83178, va: 0x483150, hint: 317, name: _lseeki64
rva: 0x8317c, va: 0x483150, hint: 383, name: _onexit
rva: 0x83180, va: 0x483150, hint: 426, name: _setmode
rva: 0x83184, va: 0x483150, hint: 583, name: abort
rva: 0x83188, va: 0x483150, hint: 590, name: atexit
rva: 0x8318c, va: 0x483150, hint: 592, name: atoi
rva: 0x83190, va: 0x483150, hint: 595, name: calloc
rva: 0x83194, va: 0x483150, hint: 604, name: exit
```

```
Report For rtms.exe
rva: 0x83198, va: 0x483150, hint: 607, name: fclose
rva: 0x8319c, va: 0x483150, hint: 610, name: fflush
rva: 0x831a0, va: 0x483150, hint: 611, name: fgetc
rva: 0x831a4, va: 0x483150, hint: 612, name: fgetpos
rva: 0x831a8, va: 0x483150, hint: 618, name: fopen
rva: 0x831ac, va: 0x483150, hint: 620, name: fputc
rva: 0x831b0, va: 0x483150, hint: 621, name: fputs
rva: 0x831b4, va: 0x483150, hint: 624, name: fread
rva: 0x831b8, va: 0x483150, hint: 625, name: free
rva: 0x831bc, va: 0x483150, hint: 630, name: fsetpos
rva: 0x831c0, va: 0x483150, hint: 633, name: fwrite
rva: 0x831c4, va: 0x483150, hint: 635, name: getc
rva: 0x831c8, va: 0x483150, hint: 637, name: getenv
rva: 0x831cc, va: 0x483150, hint: 639, name: getwc
rva: 0x831d0, va: 0x483150, hint: 658, name: iswctype
rva: 0x831d4, va: 0x483150, hint: 671, name: localeconv
rva: 0x831d8, va: 0x483150, hint: 676, name: malloc
rva: 0x831dc, va: 0x483150, hint: 680, name: memchr
rva: 0x831e0, va: 0x483150, hint: 681, name: memcmp
rva: 0x831e4, va: 0x483150, hint: 682, name: memcpy
rva: 0x831e8, va: 0x483150, hint: 683, name: memmove
rva: 0x831ec, va: 0x483150, hint: 684, name: memset
rva: 0x831f0, va: 0x483150, hint: 690, name: putc
rva: 0x831f4, va: 0x483150, hint: 693, name: putwc
rva: 0x831f8, va: 0x483150, hint: 697, name: rand
rva: 0x831fc, va: 0x483150, hint: 698, name: realloc
rva: 0x83200, va: 0x483150, hint: 699, name: remove
rva: 0x83204, va: 0x483150, hint: 704, name: setlocale
rva: 0x83208, va: 0x483150, hint: 705, name: setvbuf
rva: 0x8320c, va: 0x483150, hint: 706, name: signal
rva: 0x83210, va: 0x483150, hint: 709, name: sprintf
rva: 0x83214, va: 0x483150, hint: 711, name: srand
rva: 0x83218, va: 0x483150, hint: 714, name: strchr
rva: 0x8321c, va: 0x483150, hint: 715, name: strcmp
rva: 0x83220, va: 0x483150, hint: 716, name: strcoll
rva: 0x83224, va: 0x483150, hint: 719, name: strerror
rva: 0x83228, va: 0x483150, hint: 720, name: strftime
rva: 0x8322c, va: 0x483150, hint: 721, name: strlen
rva: 0x83230, va: 0x483150, hint: 729, name: strtod
rva: 0x83234, va: 0x483150, hint: 733, name: strxfrm
rva: 0x83238, va: 0x483150, hint: 739, name: time
rva: 0x8323c, va: 0x483150, hint: 744, name: towlower
rva: 0x83240, va: 0x483150, hint: 745, name: towupper
rva: 0x83244, va: 0x483150, hint: 746, name: ungetc
rva: 0x83248, va: 0x483150, hint: 747, name: ungetwc
rva: 0x8324c, va: 0x483150, hint: 748, name: vfprintf
rva: 0x83250, va: 0x483150, hint: 757, name: wcscoll
rva: 0x83254, va: 0x483150, hint: 760, name: wcsftime
rva: 0x83258, va: 0x483150, hint: 761, name: wcslen
rva: 0x8325c, va: 0x483150, hint: 774, name: wcsxfrm


SHELL32.DLL
-----------
[Deprecated Shell APIs]
rva: 0x83264, va: 0x483264, hint: 106, name: SHGetSpecialFolderPathA -> SHGetSpecialFolderPath
is not supported. Instead, use ShGetFolderPath.


WSOCK32.DLL
-----------
[Winsock]
rva: 0x8326c, va: 0x48326c, hint: 31, name: WSAStartup -> Initiates use of WS2_32.DLL by a
process.
rva: 0x83270, va: 0x48326c, hint: 41, name: gethostbyname -> Retrieves host information
corresponding to a host name from a host database. Deprecated: use getaddrinfo instead.
rva: 0x83274, va: 0x48326c, hint: 42, name: gethostname -> Retrieves the standard host name for
the local computer.
```

```
Report For rtms.exe
Anomalies
*********

* Deprecated Characteristic in COFF File Header: IMAGE_FILE_LINE_NUMS_STRIPPED
* Deprecated Characteristic in COFF File Header: IMAGE_FILE_LOCAL_SYMS_STRIPPED
* COFF Header: Time date stamp is in the future
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_1BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_8BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_256BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_2048BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 1 with name .text: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_8BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_2048BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 2 with name .data: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_8BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_2048BYTES characteristic is only valid for object files
```

```
Report For rtms.exe
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 3 with name .rdata: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_1BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_256BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 4 with name .eh_fram: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_8BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_2048BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 5 with name .bss: POINTER_TO_RAW_DATA must be 0 for sections with only uninitialized data, but is: 497152
* Section Header 5 with name .bss: SIZE_OF_RAW_DATA is 0
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_1BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_256BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
```

```
Report For rtms.exe
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 6 with name .idata: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_1BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_256BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 7 with name .CRT: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_1BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_2BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_4BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_16BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_32BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_64BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_256BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_512BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_1024BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_4096BYTES characteristic is only valid for object files
* Section Header 8 with name .tls: IMAGE_SCN_ALIGN_8192BYTES characteristic is only valid for object files
* Section name is unusual: .eh_fram
* Section name is unusual: .CRT
* Section 5 with name .bss (range: 497152--497152) physically overlaps with section .idata with number 6 (range: 497152--500736)
* Section 1 with name .text has write and execute characteristics.
* Entry point is in writeable section 1 with name .text
* Section Header 1 with name .text has unusual characteristics, that shouldn't be there: Initialized Data, Align 1 Byte, Align 4 Bytes, Align 8 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 256 Bytes, Align 1024 Bytes, Align 2048 Bytes, Align 4096 Bytes, Align 8192 Bytes, Write
* Section Header 2 with name .data has unusual characteristics, that shouldn't be there: Align 2 Bytes, Align 4 Bytes, Align 8 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 512 Bytes, Align 1024 Bytes, Align 2048 Bytes, Align 4096 Bytes, Align 8192 Bytes
* Section Header 3 with name .rdata has unusual characteristics, that shouldn't be there: Align 2 Bytes, Align 4 Bytes, Align 8 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 512 Bytes, Align 1024 Bytes, Align 2048 Bytes, Align 4096 Bytes, Align 8192 Bytes
* Section Header 5 with name .bss has unusual characteristics, that shouldn't be there: Align 2 Bytes, Align 4 Bytes, Align 8 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 512 Bytes, Align 1024 Bytes, Align 2048 Bytes, Align 4096 Bytes, Align 8192 Bytes
```

```
Report For rtms.exe
* Section Header 6 with name .idata has unusual characteristics, that shouldn't be there: Align
1 Byte, Align 2 Bytes, Align 4 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 256
Bytes, Align 512 Bytes, Align 1024 Bytes, Align 4096 Bytes, Align 8192 Bytes
* Section Header 8 with name .tls has unusual characteristics, that shouldn't be there: Align 1
Byte, Align 2 Bytes, Align 4 Bytes, Align 16 Bytes, Align 32 Bytes, Align 64 Bytes, Align 256
Bytes, Align 512 Bytes, Align 1024 Bytes, Align 4096 Bytes, Align 8192 Bytes
* Import function typical for code injection: VirtualProtect may set PAGE_EXECUTE flag for memory
region

Hashes
******

MD5:    01fd682d16dfe26e180f4c7cd74cfb62
SHA256: 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92

Section      Type      Hash Value
-------------------------------------------------------------------------------------------
1. .text     MD5       0b8cc6de10f7599a080799dc88261b21
             SHA256    79e1284fa66133c50da8d4fdb7fe21d274b32c910eccaace9803654739dd91a4
2. .data     MD5       fa3a6789ad95497d492e3f04ef4c542c
             SHA256    32ab69ef87d8f1ee2100380c3a5b3728de65f892a5b3cc6f1c7fd1f888cfd35a
3. .rdata    MD5       2c75a15b7835393fcd86e041e7419e63
             SHA256    50d2b44107b1c53d832dbb3f7de656cc1fd871084f80c00c81d3d9ea6c113819
4. .eh_fram  MD5       f834ed6184729a99174882df3a2b34ed
             SHA256    b872e0625e488b5fc46397502a401dda3733aa6941d91b54b908c8a11d6d03a9
5. .bss      MD5
             SHA256
6. .idata    MD5       cb9ca5b2eb102a48e266d1a379482165
             SHA256    58904b76a7cf526ed9762b7f5cda81e5f736bca6237dbdd703fa92d526b766b4
7. .CRT      MD5       f26044af392c5594ad34576aca15d1db
             SHA256    c2d9414c1b11bfddf9b8ecc61ad5eac4df86a503cb520ce210a77fb51797d5a3
8. .tls      MD5       b79dfdf69cb172a8497793b5d97c5214
             SHA256    763567f0cbcb1844c227829aad3d41e9cb39442093acdc8218354b0ea20a828a

--- end of report ---
```

I

Appendix H
BinText Strings list

```
File: malware.exe
MD5:  01fd682d16dfe26e180f4c7cd74cfb62
Size: 501760

Ascii Strings:
---------------------------------------------------------------------------
0000004D  !This program cannot be run in DOS mode.
000001C8  .rdata
000001EE  `@.eh_fram
00000216  0@.bss
00000240  .idata
.text:401551  DDDDDDDD'
.text:401624  ?.?1L)
.text:401671  DDDDDDD
.text:401721  33333333'
.text:402C37  8%n$y%
.text:403A11  """"""""
.text:403B25  ^d@g~h
.text:403B2C  #Zd@g~l
.text:403C1D  Xd@g~`
.text:403C60  w~l@g~`
.text:403CD4  'XdoGRf
.text:4055ED  D$4<f@
.text:405605  D$@IV@
.text:405685  D$4<f@
.text:405F04  D$4<f@
.text:405FFD  D$4<f@
.text:406355  p< t6v
.text:40635C  <@t$<Pt
.text:40665D  D$T<f@
.text:406BB4  D$D<f@
.text:406BCF  D$P@l@
.text:406D51  9t$0wa
.text:406DC9  D$4<f@
.text:406DE1  D$@zn@
.text:407245  D$4<f@
.text:407331  D$4<f@
.text:40741D  D$4<f@
.text:407509  D$4<f@
.text:4075F5  D$4<f@
.text:4076E1  D$4<f@
.text:4077CD  D$4<f@
.text:4077E5  D$@ux@
.text:4078B9  D$4<f@
.text:4078D1  D$@ay@
.text:4079A5  D$4<f@
.text:4079BD  D$@Mz@
.text:407A91  D$4<f@
.text:407AA9  D$@1{@
.text:407B61  D$4<f@
.text:407B79  D$@m|@
.text:407CB1  D$4<f@
.text:407E39  D$4<f@
.text:408194  \$,;\$4t3
.text:408343  D$D<f@
.text:4085BC  D$4<f@
.text:408781  D$4<f@
.text:409949  D$4<f@
.text:409AA5  D$4<f@
.text:40A129  D$D<f@
.text:40A434  D$D<f@
.text:40A764  D$D<f@
.text:40AA15  D$4<f@
.text:40AC8C  D$T<f@
.text:40B1F5  D$4<f@
```

```
File: malware.exe
.text:40B249  D$4<f@
.text:40B2F5  D$4<f@
.text:40B391  D$4<f@
.text:40B519  D$4<f@
.text:40B605  D$4<f@
.text:40B782  D$4<f@
.text:40B905  D$4<f@
.text:40BA4A  D$4<f@
.text:40BBA9  D$4<f@
.text:40BD5E  l$L9l$D
.text:40C6FC  S4Qf@u
.text:40C735  D$4<f@
.text:40C80D  D$4<f@
.text:40CA15  D$4<f@
.text:40CB0F  D$4<f@
.text:40CC2B  D$D<f@
.text:40CFE7  D$T<f@
.text:40D215  D$ 9D$,
.text:40D410  D$D<f@
.text:40D57E  \$$+\$
.text:40D7E4  D$D<f@
.text:40DC6B  D$D<f@
.text:40DE4E  t\;D$$
.text:40E0BC  D$D<f@
.text:40E4A4  D$D<f@
.text:40E67A  f9D$"t
.text:40E89C  D$D<f@
.text:40F45C  D$4<f@
.text:40F509  D$4<f@
.text:40F905  D$4<f@
.text:40FA19  D$4<f@
.text:40FB91  D$4<f@
.text:40FC5D  D$4<f@
.text:40FE99  D$4<f@
.text:40FF65  D$4<f@
.text:410031  D$4<f@
.text:41010D  D$4<f@
.text:410219  D$4<f@
.text:4103B5  D$4<f@
.text:41044D  D$4<f@
.text:4107AD  D$4<f@
.text:410959  D$4<f@
.text:410C41  D$4<f@
.text:410D01  D$4<f@
.text:410FE9  D$4<f@
.text:4110A9  D$4<f@
.text:411167  D$4<f@
.text:411216  D$\;D$pt
.text:4112FB  D$4<f@
.text:4113AA  D$\;D$pt
.text:4114A7  D$4<f@
.text:411558  D$\;D$pt
.text:411727  D$4<f@
.text:411A88  t$@;t$DsL
.text:411C35  uU;D$Ds?
.text:411D74  D$D<f@
.text:412080  D$D<f@
.text:412557  D$D<f@
.text:41256F  D$Pb&A
.text:4129CB  D$D<f@
.text:413020  ;\$dsf
.text:413117  sU;t$Tr
.text:41322D  D$4<f@
.text:41354F  8<VtL<Kuh
.text:413D16  <EtN<I|
.text:413D1E  <J~.<Lt6<Xu
.text:41412F  <rt!<Vt
.text:41417D  C ;C$}L
.text:414205  S ;S$}
.text:4149BE  tL<EtH<.tD
```

```
File: malware.exe
.text:4153DE  t<<Et8
.text:415873  <_t|<$
.text:41C58E  9l$Xv.
.text:41FA97  9D$tu7
.text:421136  )D$T)D$P)D$L
.text:4213D5  L$ )L$T
.text:4220C3  |$4+t$(
.text:422109  9D$Ds%
.text:4221C7  9D$Ds'
.text:422E6C  T$8+T$<
.text:42321F  t$ +\$,
.text:423E1D  D$ 9D$X
.text:42455D  D$4<f@
.text:424759  D$4<f@
.text:424771  D$@>HB
.text:424875  D$4<f@
.text:42488D  D$@ZIB
.text:4249A9  D$4<f@
.text:424AC5  D$4<f@
.text:424C11  D$4<f@
.text:424CE1  D$4<f@
.text:424CF9  D$@bMB
.text:424DC1  D$4<f@
.text:424EDD  D$4<f@
.text:425011  D$4<f@
.text:42512D  D$4<f@
.text:425279  D$4<f@
.text:425349  D$4<f@
.text:425EAE  D$ ;D$$r
.text:4263AE  ;\$4w,
.text:426566  ;\$DwD
.text:4265DC  ;\$$w2
.text:426641  ;\$$w9
.text:4266CB  ;t$4wK
.text:4269B2  D$ ;D$$r
.text:427021  ;\$$wE1
.text:427098  ;\$$w2
.text:4270FD  ;\$$w9
.text:427187  ;t$4wK
.text:427C9D  D$D<f@
.text:427E39  D$D<f@
.text:427FDD  D$D<f@
.text:42819D  D$D<f@
.text:42835D  D$D<f@
.text:428521  D$D<f@
.text:4289E7  D$D<f@
.text:428B82  D$D<f@
.text:428CCD  9D$$wW@
.text:428E99  D$D<f@
.text:428FBE  T$(9T$
.text:429059  D$D<f@
.text:4291A5  9D$ wU@
.text:429368  D$T<f@
.text:429520  T$ 9T$4
.text:4297FA  T$ 9T$4
.text:4298ED  T$ 9T$4
.text:429F24  D$T<f@
.text:42AAF4  D$T<f@
.text:42ACAC  T$ 9T$4
.text:42AF86  T$ 9T$4
.text:42B079  T$ 9T$4
.text:42B6B1  D$T<f@
.text:42B868  T$ f9T$8
.text:42BB4A  T$ f9T$8
.text:42BC46  T$ f9T$8
.text:42C27F  D$d<f@
.text:42C44C  D$(8H$
.text:42CA45  \$(8CJ
.text:42CA7D  D$(8H%
```

```
File: malware.exe
.text:42CA8E  D$(8P$
.text:42CAB3  D$(8PLt
.text:42CBE1  T$(8Z$
.text:42CFF3  D$d<f@
.text:42D1A4  D$08H$
.text:42D1E2  \$$9\$@
.text:42D1F2  D$ 9D$D
.text:42D55A  \$$9\$@
.text:42D56A  D$ 9D$D
.text:42D6B0  \$$9\$@
.text:42D779  \$08CJ
.text:42D7B1  D$08H%
.text:42D7C2  D$08P$
.text:42D7E7  D$08PLt
.text:42D915  T$08Z$
.text:42DD06  D$D<f@
.text:42F401  D$D<f@
.text:42F5AD  D$D<f@
.text:42F759  D$D<f@
.text:42FAB4  D$T<f@
.text:42FC78  T$ 9T$4
.text:42FFB4  L$,f9L$
.text:42FFCB  T$ 9T$4
.text:430108  T$ 9T$4
.text:430720  D$T<f@
.text:4313A0  D$T<f@
.text:431564  T$ 9T$4
.text:4318A0  L$,f9L$
.text:4318B7  T$ 9T$4
.text:4319F4  T$ 9T$4
.text:43200D  D$T<f@
.text:432028  D$`;,C
.text:4321D0  T$ f9T$8
.text:432510  L$0f9L$
.text:432527  T$ f9T$8
.text:43266C  T$ f9T$8
.text:432C87  D$d<f@
.text:432CA2  D$p_:C
.text:432E59  D$(f9P$
.text:43356A  \$(f9Cp
.text:4335A5  D$(f9P&
.text:4335B7  D$(f9P$
.text:4335DE  D$(f9Ptt
.text:43371E  \$(f9S$
.text:433AAB  D$d<f@
.text:433C65  D$0f9P$
.text:433CA4  D$$9D$@
.text:433CB4  T$ 9T$D
.text:434080  L$8f9L$
.text:434097  \$$9\$@
.text:4340A7  D$ 9D$D
.text:434230  \$$9\$@
.text:43433A  \$0f9Cp
.text:434375  D$0f9P&
.text:434387  D$0f9P$
.text:4343AE  D$0f9Ptt
.text:4344EE  \$0f9S$
.text:434861  D$T<f@
.text:43497B  D$(f9P&
.text:43498D  D$(f9P$
.text:434AF2  D$(f9P&
.text:434E5D  D$(f9P$
.text:434FA7  D$(f9Pr
.text:4350BB  L$(f9Ar
.text:4352B6  \$(f9K$
.text:4353F2  \$(f9Ar
.text:435425  D$(f9P&
.text:435433  D$(f9P$
.text:43552E  \$(f9K$
.text:4360A0  D$L<f@
```

```
File: malware.exe
.text:4360BB  D$X2bC
.text:436264  D$L<f@
.text:436428  D$L<f@
.text:4398C5  D$l9D$
.text:439919  L$l9L$ t
.text:439D4A  CG;} u
.text:43A17C  t6;|$Ds0
.text:43B447  D$l9D$
.text:43B4A9  L$l9L$
.text:43BCFA  t<;\$4s6
.text:43FD0B  D$T<f@
.text:43FEEC  D$D<f@
.text:442398  D$T<f@
.text:4425FC  D$D<f@
.text:442617  D$Pr'D
.text:442AD6  D$t<f@
.text:442B98  \$(9\$(
.text:442E04  L$,+L$4
.text:442E65  9D$,vG
.text:443202  D$t<f@
.text:4432C4  \$(9\$(
.text:443530  L$,+L$4
.text:443591  9D$,vG
.text:443C2A  D$t<f@
.text:443F7F  L$0+L$4
.text:444006  9D$0vG
.text:444356  D$t<f@
.text:4446C5  D$0+D$4
.text:44474A  9D$0vG
.text:444E6C  D$4<f@
.text:44506C  D$4<f@
.text:445311  D$4<f@
.text:445399  D$4<f@
.text:4453E5  D$4<f@
.text:4455EC  9T$0s&
.text:44561A  B9D$0w
.text:4456F0  ;\$8wb
.text:445D65  D$4<f@
.text:4460DE  D$D<f@
.text:4465CB  \$X+\$T
.text:446651  D$X9D$Tt
.text:4467AD  D$4<f@
.text:446B09  D$4<f@
.text:44707D  D$4<f@
.text:4471DD  D$4<f@
.text:44733E  D$4<f@
.text:44744E  D$4<f@
.text:4475F4  D$D<f@
.text:4477E8  D$D<f@
.text:4479DC  D$D<f@
.text:447BD0  D$D<f@
.text:447DC4  D$D<f@
.text:447FB8  D$D<f@
.text:4481AC  D$D<f@
.text:4483A0  D$D<f@
.text:448594  D$D<f@
.text:448788  D$D<f@
.text:44897C  D$D<f@
.text:448BE3  D$D<f@
.text:448EC7  D$D<f@
.text:449198  D$4<f@
.text:449384  D$4<f@
.text:449568  D$4<f@
.text:44972C  D$4<f@
.text:4498F8  D$4<f@
.text:449AD3  D$D<f@
.text:449CF5  D$T<f@
.text:449EF7  D$D<f@
.text:44A0E0  D$4<f@
```

```
File: malware.exe
.text:44A2BC  D$4<f@
.text:44A727  D$4<f@
.text:44A908  D$4<f@
.text:44AB25  D$4<f@
.text:44AC19  D$4<f@
.text:44AE24  D$4<f@
.text:44B034  D$D<f@
.text:44B24C  D$D<f@
.text:44B4C3  D$D<f@
.text:44B6D8  D$4<f@
.text:44B863  D$D<f@
.text:44B90E  D$p#D$t@u
.text:44BA39  D$T<f@
.text:44BAE0  D$p#D$t@u
.text:44BBF4  D$D<f@
.text:44BDAC  D$4<f@
.text:44C15B  D$T<f@
.text:44C452  D$T<f@
.text:44C746  D$T<f@
.text:44CA2B  D$T<f@
.text:44CD13  D$T<f@
.text:44CFFF  D$T<f@
.text:44D301  D$T<f@
.text:44D601  D$T<f@
.text:44D8E5  D$4<f@
.text:44D9CD  D$4<f@
.text:44DBAC  D$4<f@
.text:44E168  D$4<f@
.text:44E350  D$4<f@
.text:44E5DD  D$4<f@
.text:44E665  D$4<f@
.text:44E6B1  D$4<f@
.text:44E8A0  9L$@s*
.text:44E994  9l$Dw|
.text:44E99C  +l$D;l$HwZ
.text:44EF95  D$4<f@
.text:44F2A3  D$D<f@
.text:44F2FC  9L$$w~
.text:44F75F  \$X+\$T
.text:44F768  +T$P+T$T
.text:44F7F9  L$X9L$Tt
.text:44F92D  D$4<f@
.text:44FC8D  D$4<f@
.text:45048D  D$4<f@
.text:450545  D$4<f@
.text:4505FD  D$4<f@
.text:4506B5  D$4<f@
.text:45076D  D$4<f@
.text:450825  D$4<f@
.text:4508DD  D$4<f@
.text:450995  D$4<f@
.text:450A4D  D$4<f@
.text:450B05  D$4<f@
.text:450BBD  D$4<f@
.text:450C75  D$4<f@
.text:450D2D  D$4<f@
.text:450DE5  D$4<f@
.text:450E9D  D$4<f@
.text:450F55  D$4<f@
.text:45100D  D$4<f@
.text:4510C5  D$4<f@
.text:45117D  D$4<f@
.text:451235  D$4<f@
.text:4512ED  D$4<f@
.text:4513A5  D$4<f@
.text:45145D  D$4<f@
.text:451515  D$4<f@
.text:4515D5  D$4<f@
.text:451693  D$D<f@
.text:451825  D$4<f@
```

```
File: malware.exe
.text:4518E1  D$4<f@
.text:45199F  D$D<f@
.text:451B31  D$4<f@
.text:451C05  D$4<f@
.text:451CED  D$4<f@
.text:451DD5  D$4<f@
.text:451E93  D$D<f@
.text:452025  D$4<f@
.text:4520E1  D$4<f@
.text:4520F9  D$@e!E
.text:45219F  D$D<f@
.text:452331  D$4<f@
.text:452405  D$4<f@
.text:4524ED  D$4<f@
.text:452AE8  D$ #D$$@u
.text:452B4C  D$D<f@
.text:452B64  D$P>,E
.text:452DD5  D$ #D$$@
.text:4537C9  D$0#D$4@
.text:453922  P4R@tx
.text:453984  D$0#D$4@t
.text:453BE8  L$(;K`
.text:453D9D  T$(;S`
.text:453E66  L$(;K`
.text:453E85  D$4<f@
.text:454019  D$4<f@
.text:454031  D$@mAE
.text:4541AD  D$4<f@
.text:4541C5  D$@:BE
.text:454289  D$4<f@
.text:454359  D$4<f@
.text:454900  D$ #D$$@u
.text:454968  D$D<f@
.text:454980  D$PZJE
.text:454BF1  D$ #D$$@
.text:45553B  D$@#D$D@
.text:455682  P4Qf@u
.text:455704  D$0#D$4@
.text:4558D6  P4Uf@t
.text:455964  L$8;Kd
.text:455B2D  T$8;Sd
.text:455BF6  L$8;Kd
.text:455C15  D$4<f@
.text:455C2D  D$@m]E
.text:455DAD  D$4<f@
.text:455F45  D$4<f@
.text:456021  D$4<f@
.text:4560F1  D$4<f@
.text:456109  D$@raE
.text:4562B9  D$D<f@
.text:4562D1  D$PGdE
.text:4564D9  D$D<f@
.text:4564F1  D$PkfE
.text:4566FD  D$D<f@
.text:456715  D$P6hE
.text:4568C6  D$4<f@
.text:4568DE  D$@'jE
.text:456ABA  D$4<f@
.text:456CAE  D$4<f@
.text:456E4D  D$4<f@
.text:456E65  D$@(oE
.text:456F9D  D$4<f@
.text:456FB5  D$@lpE
.text:4570E1  D$4<f@
.text:45733D  D$D<f@
.text:457561  D$D<f@
.text:457785  D$D<f@
.text:457952  D$4<f@
.text:457B46  D$4<f@
```

```
File: malware.exe
.text:457D3A  D$4<f@
.text:457D52  D$@J~E
.text:457ED9  D$4<f@
.text:458029  D$4<f@
.text:45816D  D$4<f@
.text:4582D4  D$D<f@
.text:4584C8  D$D<f@
.text:4586BC  D$D<f@
.text:4588B0  D$D<f@
.text:458AA4  D$D<f@
.text:458C98  D$D<f@
.text:458E8C  D$D<f@
.text:459080  D$D<f@
.text:459274  D$D<f@
.text:459468  D$D<f@
.text:45965C  D$D<f@
.text:4598A8  D$D<f@
.text:4599B8  tbf9D$*t
.text:459B7C  D$D<f@
.text:459C4E  tyf9D$&t
.text:459C62  D$$f9D$&
.text:459E34  D$4<f@
.text:45A030  D$4<f@
.text:45A224  D$4<f@
.text:45A3EC  D$4<f@
.text:45A5B8  D$4<f@
.text:45A793  D$D<f@
.text:45A9B5  D$T<f@
.text:45ABB7  D$D<f@
.text:45ADA0  D$4<f@
.text:45AF80  D$4<f@
.text:45B408  D$4<f@
.text:45B5F8  D$4<f@
.text:45B819  D$4<f@
.text:45B90D  D$4<f@
.text:45BB1C  D$4<f@
.text:45BD2C  D$D<f@
.text:45BF44  D$D<f@
.text:45C1BC  D$D<f@
.text:45C3E0  D$4<f@
.text:45C56B  D$D<f@
.text:45C616  D$p#D$t@u
.text:45C741  D$T<f@
.text:45C7E8  D$p#D$t@u
.text:45C8FC  D$D<f@
.text:45CAB4  D$4<f@
.text:45CE63  D$T<f@
.text:45D136  D$T<f@
.text:45D406  D$T<f@
.text:45D6C7  D$T<f@
.text:45D98B  D$T<f@
.text:45DC53  D$T<f@
.text:45DF31  D$T<f@
.text:45E20D  D$T<f@
.text:45E4CD  D$4<f@
.text:45E5B9  D$4<f@
.text:45E798  D$4<f@
.text:45EBB9  D$D<f@
.text:45ED8D  D$D<f@
.text:45EF65  D$D<f@
.text:45F0E2  D$4<f@
.text:45F25A  D$4<f@
.text:45F3D6  D$4<f@
.text:45F4F9  D$4<f@
.text:45F641  D$4<f@
.text:45F77D  D$4<f@
.text:45F9AD  D$D<f@
.text:45FB85  D$D<f@
.text:45FD5D  D$D<f@
.text:45FEDA  D$4<f@
```

```
File: malware.exe
.text:460052  D$4<f@
.text:4601CE  D$4<f@
.text:4602F1  D$4<f@
.text:460439  D$4<f@
.text:460575  D$4<f@
.text:4606A9  D$4<f@
.text:46080D  D$4<f@
.text:46096E  D$4<f@
.text:460A7E  D$4<f@
.text:460D1D  D$D<f@
.text:460EE5  D$D<f@
.text:4610AD  D$D<f@
.text:461219  D$4<f@
.text:461389  D$4<f@
.text:4614FD  D$4<f@
.text:46161D  D$4<f@
.text:461755  D$4<f@
.text:461881  D$4<f@
.text:461AA1  D$D<f@
.text:461C69  D$D<f@
.text:461E35  D$D<f@
.text:461FA5  D$4<f@
.text:462115  D$4<f@
.text:46212D  D$@+"F
.text:462289  D$4<f@
.text:4622A1  D$@H#F
.text:4623A9  D$4<f@
.text:4623C1  D$@o$F
.text:4624E1  D$4<f@
.text:46260D  D$4<f@
.text:462733  D$4<f@
.text:462833  D$4<f@
.text:46296B  D$4<f@
.text:462983  D$@1*F
.text:462A6B  D$4<f@
.text:462A83  D$@1+F
.text:462BA3  D$4<f@
.text:462BBB  D$@y,F
.text:462CE3  D$4<f@
.text:462E5B  D$4<f@
.text:462E73  D$@1/F
.text:462F9B  D$4<f@
.text:462FB3  D$@q0F
.text:463329  E9l$4~)
.text:46339F  P4W@t/EF9l$4~'
.text:463485  D$4<f@
.text:4638E7  E9l$4~8
.text:46396B  P4Rf@t@E
.text:463976  9l$4~6
.text:463A69  D$4<f@
.text:463CAF  D$4<f@
.text:463CC7  D$@u=F
.text:463DAF  D$4<f@
.text:463DC7  D$@u>F
.text:463EE7  D$4<f@
.text:463FE7  D$4<f@
.text:46411F  D$4<f@
.text:464227  D$4<f@
.text:464367  D$4<f@
.text:46437F  D$@SDF
.text:4644BF  D$4<f@
.text:4659DB  D$4<f@
.text:465AEF  D$4<f@
.text:465C3B  D$4<f@
.text:465D4F  D$4<f@
.text:465D67  D$@)^F
.text:465E9B  D$4<f@
.text:465EB3  D$@u_F
.text:465FAF  D$4<f@
```

```
File: malware.exe
.text:4660FB  D$4<f@
.text:46620F  D$4<f@
.text:468D4B  L$D)L$@
.text:468FB7  P(Qf9G
.text:469079  D$4<f@
.text:46914D  D$4<f@
.text:469209  D$4<f@
.text:4692AD  D$4<f@
.text:469381  D$4<f@
.text:469455  D$4<f@
.text:469511  D$4<f@
.text:4695B5  D$4<f@
.text:4697F9  D$4<f@
.text:46989D  D$4<f@
.text:469941  D$4<f@
.text:4699E5  D$4<f@
.text:469AA1  D$4<f@
.text:469B45  D$4<f@
.text:469BE9  D$4<f@
.text:469C8D  D$4<f@
.text:469D31  D$4<f@
.text:469DD5  D$4<f@
.text:469E91  D$4<f@
.text:469F35  D$4<f@
.text:469FD9  D$4<f@
.text:46A089  D$4<f@
.text:46A139  D$4<f@
.text:46A1E9  D$4<f@
.text:46A299  D$4<f@
.text:46A349  D$4<f@
.text:46A3F9  D$4<f@
.text:46A4A9  D$4<f@
.text:46A559  D$4<f@
.text:46A609  D$4<f@
.text:46A6B9  D$4<f@
.text:46A769  D$4<f@
.text:46AB61  D$4<f@
.text:46AEF9  D$4<f@
.text:46B019  D$4<f@
.text:46B2FD  D$4<f@
.text:46B661  D$4<f@
.text:46B785  D$4<f@
.text:46BDDC  ;\$4t!
.text:46C18C  D$D<f@
.text:46C750  D$D<f@
.text:46CD19  D$4<f@
.text:46CE6D  D$4<f@
.text:46E6E8  D$D<f@
.text:46EDE1  D$4<f@
.text:46EEC5  D$4<f@
.text:46EFBD  D$4<f@
.text:46F066  D$4<f@
.text:46F170  D$4<f@
.text:46F43C  D$D<f@
.text:46F5C4  D$ 9D$
.text:46F7E0  D$4<f@
.text:46FA9F  D$D<f@
.rdata:473000  libgcj-13.dll
.rdata:47300E  _Jv_RegisterClasses
.rdata:4730B0  US@HOHOF!
.rdata:4730BA  HI@EZI^U,
.rdata:4730C4  [XTVC^XY7
.rdata:4730CE  16#671B
.rdata:4730D6  )(=!"4M
.rdata:473154  ]V[JAHL]\
.rdata:47318F  [WVL]VL
.rdata:4731D5  \QJ][LWJA
.rdata:4731F1  HJW_JYU
.rdata:4731FE  \][JAHLK
.rdata:47320B  ]V[JAHL]\
```

```
File: malware.exe
.rdata:473229  YzQLxZY\
.rdata:473238  3+*<;= 6*+0)&#*<00=*.+"*A;7;o
.rdata:473264  >G!98.)/2$89";418.""/8<908S)%)}
.rdata:473287  ),9,c/$#M
.rdata:4732B8  list::_M_check_equal_allocators
.rdata:473350  basic_string::at
.rdata:473361  basic_string::copy
.rdata:473374  basic_string::compare
.rdata:47338A  basic_string::_S_create
.rdata:4733A2  basic_string::assign
.rdata:4733B7  basic_string::_M_replace_aux
.rdata:4733D4  basic_string::replace
.rdata:4733EA  basic_string::insert
.rdata:4733FF  basic_string::erase
.rdata:473413  basic_string::append
.rdata:473428  basic_string::resize
.rdata:473440  basic_string::_S_construct null not valid
.rdata:47346A  basic_string::basic_string
.rdata:473485  basic_string::substr
.rdata:4734B4  basic_filebuf::xsgetn error reading the file
.rdata:4734E4  basic_filebuf::underflow codecvt::max_length() is not valid
.rdata:473520  basic_filebuf::underflow incomplete character in file
.rdata:473558  basic_filebuf::underflow invalid byte sequence in file
.rdata:473590  basic_filebuf::underflow error reading the file
.rdata:4735C0  basic_filebuf::_M_convert_to_external conversion error
.rdata:4735F8  basic_ios::clear
.rdata:47360C  std::future_error
.rdata:473620  ios_base::_M_grow_words is not valid
.rdata:473648  ios_base::_M_grow_words allocation failed
.rdata:473720  __gnu_cxx::__concurrence_lock_error
.rdata:473744  __gnu_cxx::__concurrence_unlock_error
.rdata:473840  __gnu_cxx::__concurrence_lock_error
.rdata:473864  __gnu_cxx::__concurrence_unlock_error
.rdata:47388C  locale::_S_normalize_category category not found
.rdata:4738F0  locale::_Impl::_M_replace_facet
.rdata:473934  std::exception
.rdata:473943  std::bad_exception
.rdata:473958  eh_globals
.rdata:473968  __gnu_cxx::__concurrence_lock_error
.rdata:47398C  __gnu_cxx::__concurrence_unlock_error
.rdata:4739B8  std::bad_alloc
.rdata:473A40  basic_string::at
.rdata:473A51  basic_string::copy
.rdata:473A64  basic_string::compare
.rdata:473A7A  basic_string::_S_create
.rdata:473A92  basic_string::assign
.rdata:473AA7  basic_string::_M_replace_aux
.rdata:473AC4  basic_string::replace
.rdata:473ADA  basic_string::insert
.rdata:473AEF  basic_string::erase
.rdata:473B03  basic_string::append
.rdata:473B18  basic_string::resize
.rdata:473B30  basic_string::_S_construct null not valid
.rdata:473B5A  basic_string::basic_string
.rdata:473B75  basic_string::substr
.rdata:473C80  %m/%d/%y
.rdata:473C8F  %H:%M:%S
.rdata:473FF4  %m/%d/%y
.rdata:474003  %H:%M:%S
.rdata:474120  __unexpected_handler_sh
.rdata:474138  __terminate_handler_sh
.rdata:474150  std::bad_cast
.rdata:474160  std::bad_typeid
.rdata:474180  generic
.rdata:474188  system
.rdata:474220  *N12_GLOBAL__N_122generic_error_categoryE
.rdata:474260  *N12_GLOBAL__N_121system_error_categoryE
.rdata:4742A0  future
.rdata:4742A7  Broken promise
```

```
File: malware.exe
.rdata:4742B6  Future already retrieved
.rdata:4742CF  Promise already satisfied
.rdata:4742E9  No associated state
.rdata:4742FD  Unknown error
.rdata:474360  *N12_GLOBAL__N_121future_error_categoryE
.rdata:4743A0  regex_error
.rdata:4743AC  __gnu_cxx::__concurrence_lock_error
.rdata:4743D0  __gnu_cxx::__concurrence_unlock_error
.rdata:474634  locale::facet::_S_create_c_locale name not valid
.rdata:47466C  LC_CTYPE
.rdata:474675  LC_NUMERIC
.rdata:474680  LC_TIME
.rdata:474688  LC_COLLATE
.rdata:474693  LC_MONETARY
.rdata:47469F  LC_MESSAGES
.rdata:474700  pure virtual method called
.rdata:47471C  deleted virtual method called
.rdata:474774  xdigit
.rdata:474788  -+xX0123456789abcdef0123456789ABCDEF
.rdata:4747AD  -+xX0123456789abcdefABCDEF
.rdata:4747C8  -0123456789
.rdata:474878  %m/%d/%y
.rdata:474881  %H:%M:%S
.rdata:474891  Sunday
.rdata:474898  Monday
.rdata:47489F  Tuesday
.rdata:4748A7  Wednesday
.rdata:4748B1  Thursday
.rdata:4748BA  Friday
.rdata:4748C1  Saturday
.rdata:4748E6  January
.rdata:4748EE  February
.rdata:474911  August
.rdata:474918  September
.rdata:474922  October
.rdata:47492A  November
.rdata:474933  December
.rdata:474B48  terminate called recursively
.rdata:474B68  terminate called after throwing an instance of '
.rdata:474B9C  terminate called without an active exception
.rdata:474BCA    what():
.rdata:474CFC  _GLOBAL_
.rdata:474D05  (anonymous namespace)
.rdata:474E34  string literal
.rdata:4752EB  JArray
.rdata:4752F5  vtable for
.rdata:475301  VTT for
.rdata:47530A  construction vtable for
.rdata:475328  typeinfo for
.rdata:475336  typeinfo name for
.rdata:475349  typeinfo fn for
.rdata:47535A  non-virtual thunk to
.rdata:475370  virtual thunk to
.rdata:475382  covariant return thunk to
.rdata:47539D  java Class for
.rdata:4753AD  guard variable for
.rdata:4753C1  reference temporary #
.rdata:4753DD  hidden alias for
.rdata:4753EF  transaction clone for
.rdata:475406  non-transaction clone for
.rdata:475427  _Accum
.rdata:47542E  _Fract
.rdata:475438  operator
.rdata:475441  operator
.rdata:475476  java resource
.rdata:475485  decltype (
.rdata:475499  {parm#
.rdata:4754A0  global constructors keyed to
.rdata:4754BE  global destructors keyed to
.rdata:4754DB  {lambda(
```

```
File: malware.exe
.rdata:4754E7  {unnamed type#
.rdata:4754F6   [clone
.rdata:475630   restrict
.rdata:47563A   volatile
.rdata:475644   const
.rdata:47564E  complex
.rdata:475657  imaginary
.rdata:475666   __vector(
.rdata:475710  {default arg#
.rdata:47576C  signed char
.rdata:47577D  boolean
.rdata:47578F  double
.rdata:475796  long double
.rdata:4757A8  __float128
.rdata:4757B3  unsigned char
.rdata:4757C5  unsigned int
.rdata:4757D2  unsigned
.rdata:4757E0  unsigned long
.rdata:4757EE  __int128
.rdata:4757F7  unsigned __int128
.rdata:47580F  unsigned short
.rdata:475823  wchar_t
.rdata:47582B  long long
.rdata:475835  unsigned long long
.rdata:475848  decimal32
.rdata:475852  decimal64
.rdata:47585C  decimal128
.rdata:47586C  char16_t
.rdata:475875  char32_t
.rdata:47587E  decltype(nullptr)
.rdata:475B34  std::allocator
.rdata:475B43  allocator
.rdata:475B4D  std::basic_string
.rdata:475B5F  basic_string
.rdata:475B6C  std::string
.rdata:475B78  std::basic_string<char, std::char_traits<char>, std::allocator<char> >
.rdata:475BBF  std::istream
.rdata:475BCC  std::basic_istream<char, std::char_traits<char> >
.rdata:475BFE  basic_istream
.rdata:475C0C  std::ostream
.rdata:475C1C  std::basic_ostream<char, std::char_traits<char> >
.rdata:475C4E  basic_ostream
.rdata:475C5C  std::iostream
.rdata:475C6C  std::basic_iostream<char, std::char_traits<char> >
.rdata:475C9F  basic_iostream
.rdata:475D9A  alignof
.rdata:475DBC  delete[]
.rdata:475DCE  delete
.rdata:475E0C  operator""
.rdata:475EA4  sizeof
.rdata:475EBB  throw
.rdata:476284  Mingw runtime failure:
.rdata:47629C    VirtualQuery failed for %d bytes at address %p
.rdata:4762D0    Unknown pseudo relocation protocol version %d.
.rdata:476304    Unknown pseudo relocation bit size %d.
.rdata:476330  fc_static
.rdata:47633A  fc_key
.rdata:476341  use_fc_key
.rdata:47634C  sjlj_once
.rdata:476358  gcc-shmem-tdm2
.rdata:4763DC  xdigit
.rdata:476476  (null)
.rdata:47647D  PRINTF_EXPONENT_DIGITS
.rdata:476760  Infinity
.rdata:476920  ABCDEF
.rdata:476927  abcdef
.rdata:47692E  0123456789
.rdata:477560  N10__cxxabiv115__forced_unwindE
.rdata:477580  N10__cxxabiv117__class_type_infoE
```

```
File: malware.exe
.rdata:4775C0  N10__cxxabiv119__foreign_exceptionE
.rdata:477600  N10__cxxabiv120__si_class_type_infoE
.rdata:477640  N10__cxxabiv121__vmi_class_type_infoE
.rdata:477680  N9__gnu_cxx13stdio_filebufIcSt11char_traitsIcEEE
.rdata:4776C0  N9__gnu_cxx13stdio_filebufIwSt11char_traitsIwEEE
.rdata:477700  N9__gnu_cxx18stdio_sync_filebufIcSt11char_traitsIcEEE
.rdata:477740  N9__gnu_cxx18stdio_sync_filebufIwSt11char_traitsIwEEE
.rdata:477780  N9__gnu_cxx20recursive_init_errorE
.rdata:4777C0  N9__gnu_cxx24__concurrence_lock_errorE
.rdata:477800  N9__gnu_cxx26__concurrence_unlock_errorE
.rdata:477840  NSt6locale5facetE
.rdata:477854  NSt8ios_base7failureE
.rdata:477878  St10bad_typeid
.rdata:477888  St10ctype_base
.rdata:477898  St10money_base
.rdata:4778A8  St10moneypunctIcLb0EE
.rdata:4778C0  St10moneypunctIcLb1EE
.rdata:4778D8  St10moneypunctIwLb0EE
.rdata:4778F0  St10moneypunctIwLb1EE
.rdata:477908  St11__timepunctIcE
.rdata:47791C  St11__timepunctIwE
.rdata:477930  St11logic_error
.rdata:477940  St11range_error
.rdata:477950  St11regex_error
.rdata:477960  St12codecvt_base
.rdata:477974  St12ctype_bynameIcE
.rdata:477988  St12ctype_bynameIwE
.rdata:47799C  St12domain_error
.rdata:4779B0  St12future_error
.rdata:4779C4  St12length_error
.rdata:4779D8  St12out_of_range
.rdata:4779EC  St12system_error
.rdata:477A00  St13bad_exception
.rdata:477A20  St13basic_filebufIcSt11char_traitsIcEE
.rdata:477A60  St13basic_filebufIwSt11char_traitsIwEE
.rdata:477AA0  St13basic_fstreamIcSt11char_traitsIcEE
.rdata:477AE0  St13basic_fstreamIwSt11char_traitsIwEE
.rdata:477B20  St13basic_istreamIwSt11char_traitsIwEE
.rdata:477B60  St13basic_ostreamIwSt11char_traitsIwEE
.rdata:477BA0  St13messages_base
.rdata:477BB4  St13runtime_error
.rdata:477BE0  St14basic_ifstreamIcSt11char_traitsIcEE
.rdata:477C20  St14basic_ifstreamIwSt11char_traitsIwEE
.rdata:477C60  St14basic_iostreamIwSt11char_traitsIwEE
.rdata:477CA0  St14basic_ofstreamIcSt11char_traitsIcEE
.rdata:477CE0  St14basic_ofstreamIwSt11char_traitsIwEE
.rdata:477D20  St14codecvt_bynameIcciE
.rdata:477D38  St14codecvt_bynameIwciE
.rdata:477D50  St14collate_bynameIcE
.rdata:477D68  St14collate_bynameIwE
.rdata:477D80  St14error_category
.rdata:477D94  St14overflow_error
.rdata:477DC0  St15basic_streambufIcSt11char_traitsIcEE
.rdata:477E00  St15basic_streambufIwSt11char_traitsIwEE
.rdata:477E40  St15messages_bynameIcE
.rdata:477E58  St15messages_bynameIwE
.rdata:477E70  St15numpunct_bynameIcE
.rdata:477E88  St15numpunct_bynameIwE
.rdata:477EA0  St15time_get_bynameIcSt19istreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:477F00  St15time_get_bynameIwSt19istreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:477F60  St15time_put_bynameIcSt19ostreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:477FC0  St15time_put_bynameIwSt19ostreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:478020  St15underflow_error
.rdata:478034  St16__numpunct_cacheIcE
.rdata:47804C  St16__numpunct_cacheIwE
.rdata:478064  St16invalid_argument
.rdata:47807C  St17__timepunct_cacheIcE
.rdata:478098  St17__timepunct_cacheIwE
.rdata:4780B4  St17bad_function_call
.rdata:4780CC  St17moneypunct_bynameIcLb0EE
```

```
File: malware.exe
.rdata:4780EC  St17moneypunct_bynameIcLb1EE
.rdata:47810C  St17moneypunct_bynameIwLb0EE
.rdata:47812C  St17moneypunct_bynameIwLb1EE
.rdata:47814C  St18__moneypunct_cacheIcLb0EE
.rdata:47816C  St18__moneypunct_cacheIcLb1EE
.rdata:47818C  St18__moneypunct_cacheIwLb0EE
.rdata:4781AC  St18__moneypunct_cacheIwLb1EE
.rdata:4781CC  St21__ctype_abstract_baseIcE
.rdata:4781EC  St21__ctype_abstract_baseIwE
.rdata:478220  St23__codecvt_abstract_baseIcciE
.rdata:478260  St23__codecvt_abstract_baseIwciE
.rdata:4782A0  St5ctypeIcE
.rdata:4782AC  St5ctypeIwE
.rdata:4782B8  St7codecvtIcciE
.rdata:4782C8  St7codecvtIwciE
.rdata:4782D8  St7collateIcE
.rdata:4782E8  St7collateIwE
.rdata:478300  St7num_getIcSt19istreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:478340  St7num_getIwSt19istreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:478380  St7num_putIcSt19ostreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:4783C0  St7num_putIwSt19ostreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:478400  St8bad_cast
.rdata:47840C  St8ios_base
.rdata:478418  St8messagesIcE
.rdata:478428  St8messagesIwE
.rdata:478438  St8numpunctIcE
.rdata:478448  St8numpunctIwE
.rdata:478460  St8time_getIcSt19istreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:4784A0  St8time_getIwSt19istreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:4784E0  St8time_putIcSt19ostreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:478520  St8time_putIwSt19ostreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:478560  St9bad_alloc
.rdata:478580  St9basic_iosIcSt11char_traitsIcEE
.rdata:4785C0  St9basic_iosIwSt11char_traitsIwEE
.rdata:478600  St9exception
.rdata:478620  St9money_getIcSt19istreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:478660  St9money_getIwSt19istreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:4786A0  St9mon
.rdata:4786A6  ey_putIcSt19ostreambuf_iteratorIcSt11char_traitsIcEEE
.rdata:4786E0  St9money_putIwSt19ostreambuf_iteratorIwSt11char_traitsIwEEE
.rdata:478720  St9time_base
.rdata:478730  St9type_info
.idata:48346E  CryptAcquireContextW
.idata:483486  CryptGenRandom
.idata:483498  CryptReleaseContext
.idata:4834AE  GetUserNameA
.idata:4834BE  RegOpenKeyExA
.idata:4834CE  AddAtomA
.idata:4834DA  CloseHandle
.idata:4834E8  CreateMutexA
.idata:4834F8  CreateSemaphoreA
.idata:48350C  DeleteCriticalSection
.idata:483524  EnterCriticalSection
.idata:48353C  ExitProcess
.idata:48354A  FindAtomA
.idata:483556  FindClose
.idata:483562  FindFirstFileA
.idata:483574  FindNextFileA
.idata:483584  GetAtomNameA
.idata:483594  GetComputerNameExA
.idata:4835AA  GetCurrentThreadId
.idata:4835C0  GetLastError
.idata:4835D0  GetModuleHandleA
.idata:4835E4  GetProcAddress
.idata:4835F6  InitializeCriticalSection
.idata:483612  InterlockedDecrement
.idata:48362A  InterlockedExchange
.idata:483640  InterlockedIncrement
.idata:483658  IsDBCSLeadByteEx
```

```
File: malware.exe
.idata:48366C  IsDebuggerPresent
.idata:483680  LeaveCriticalSection
.idata:483698  MultiByteToWideChar
.idata:4836AE  ReleaseMutex
.idata:4836BE  ReleaseSemaphore
.idata:4836D2  SetLastError
.idata:4836E2  SetUnhandledExceptionFilter
.idata:483708  TlsAlloc
.idata:483714  TlsFree
.idata:48371E  TlsGetValue
.idata:48372C  TlsSetValue
.idata:48373A  VirtualProtect
.idata:48374C  VirtualQuery
.idata:48375C  WaitForSingleObject
.idata:483772  WideCharToMultiByte
.idata:483788  _fdopen
.idata:48379A  _write
.idata:4837A4  __getmainargs
.idata:4837B4  __mb_cur_max
.idata:4837C4  __p__environ
.idata:4837D4  __p__fmode
.idata:4837E2  __set_app_type
.idata:4837F4  _cexit
.idata:4837FE  _errno
.idata:483808  _filelengthi64
.idata:48381A  _fstati64
.idata:48382E  _lseeki64
.idata:48383A  _onexit
.idata:483844  _setmode
.idata:483858  atexit
.idata:48386A  calloc
.idata:48387C  fclose
.idata:483886  fflush
.idata:483898  fgetpos
.idata:4838CA  fsetpos
.idata:4838D4  fwrite
.idata:4838E6  getenv
.idata:4838F8  iswctype
.idata:483904  localeconv
.idata:483912  malloc
.idata:48391C  memchr
.idata:483926  memcmp
.idata:483930  memcpy
.idata:48393A  memmove
.idata:483944  memset
.idata:483966  realloc
.idata:483970  remove
.idata:48397A  setlocale
.idata:483986  setvbuf
.idata:483990  signal
.idata:48399A  sprintf
.idata:4839AC  strchr
.idata:4839B6  strcmp
.idata:4839C0  strcoll
.idata:4839CA  strerror
.idata:4839D6  strftime
.idata:4839E2  strlen
.idata:4839EC  strtod
.idata:4839F6  strxfrm
.idata:483A08  towlower
.idata:483A14  towupper
.idata:483A20  ungetc
.idata:483A2A  ungetwc
.idata:483A34  vfprintf
.idata:483A40  wcscoll
.idata:483A4A  wcsftime
.idata:483A56  wcslen
.idata:483A60  wcsxfrm
.idata:483A6A  SHGetSpecialFolderPathA
.idata:483A84  WSAStartup
```

```
File: malware.exe
.idata:483A92  gethostbyname
.idata:483AA2  gethostname
.idata:483AC4  ADVAPI32.DLL
.idata:483B6C  KERNEL32.dll
.idata:483B88  msvcrt.dll
.idata:483CA4  msvcrt.dll
.idata:483CB4  SHELL32.DLL
.idata:483CCC  WSOCK32.DLL


Unicode Strings:
---------------------------------------------------------------------------
.rdata:474460  XXXXXXXXXX
.rdata:474482  UUUUUUEEEEEEEEEEEEEEEEEEEEE
.rdata:4744C2  VVVVVVFFFFFFFFFFFFFFFFFFFF
.rdata:474966  c%m/%d/%y
.rdata:47497A  %H:%M:%S
.rdata:47499A  Sunday
.rdata:4749A8  Monday
.rdata:4749B6  Tuesday
.rdata:4749C6  Wednesday
.rdata:4749DA  Thursday
.rdata:4749EC  Friday
.rdata:4749FA  Saturday
.rdata:474A44  January
.rdata:474A54  February
.rdata:474A9A  August
.rdata:474AA8  September
.rdata:474ABC  October
.rdata:474ACC  November
.rdata:474ADE  December
.rdata:476466  f(null)
```

I

Appendix I

Anti-VM instructions detection with a python script

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))
antiVM = []
for i in heads:
    if (GetMnem(i) == "sidt" \
        or GetMnem(i) == "sgdt" \
        or GetMnem(i) == "sldt" \
        or GetMnem(i) == "smsw" \
        or GetMnem(i) == "str" \
        or GetMnem(i) == "in" \
        or GetMnem(i) == "cpuid"):
        antiVM.append(i)
print "Number of potential Anti-VM instructions: %d" % (len(antiVM))
for i in antiVM:
    SetColor(i, CIC_ITEM, 0x0000ff)
    Message("Anti-VM: %08x\n" % i)
```

Table 11: Highlighting potential Anti-VM instructions with a python script in IDA Pro

Attached zipped files

Attached zipped files provided

1. hybrid-Analysis results.pdf
2. report-0fb3e4c1b9fdbb05b7c429ddc854b204.pdf
3. 6d2ee6b36047cdaf2c20012d1f687e2abebf71d82c420d45f2f12cee0635cf92 _ ANY.RUN - Automated Malware Analysis Service.pdf
4. VirusTotal.pdf
5. VirusTotal-behaviour.pdf
6. VirusTotal-details.pdf
7. STARTEX RANSOMWARE FINAL DOCUMENTATION.pdf
8. Windows Functions for Malware Analysis.txt
9. VirusTotal_mlr.txt
10. RTSC SCRIPT.py
11. surface analysis report.txt
12. strings_mlr.txt
13. find anti-VM instructions.py
14. imports.txt
15. exports.txt
16. breakpoints.txt
17. install.ps1.txt
18. debugging_gdb_linux_vmware.pdf
19. sample files for rnsm.rar
20. inputlist_XorSearch.txt

21. VM config files.zip
22. Installed Tools Flare vm.txt
23. Boxstarter.WebLaunch.application
24. registry Renames on Vmware powershell script.ps1
25. base64dump.py
26. ProcessExporerStrings_Image.txt
27. ProcessExporerStrings_Memory.txt
28. COMPARISON OF 2 SHOTS.txt
29. proc mon Logfile.XML
30. PortExAnalyzer Results report.txt

Footnotes

---

[i] European Union Agency for Law Enforcement Cooperation (Europol), "THE INTERNET ORGANISED CRIME THREAT ASSESSMENT (IOCTA) 2017," Executive Director of Europol, ISBN 978-92-95200-80-7, pages 18-32, 2017, Source URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2017.

[ii] European Union Agency for Law Enforcement Cooperation (Europol), "THE INTERNET ORGANISED CRIME THREAT ASSESSMENT (IOCTA) 2018," Executive Director of Europol, ISBN 978-92-95200-94-4, pages 16-29, 2017, Source URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2018

[iii] European Union Agency for Law Enforcement Cooperation (Europol), "THE INTERNET ORGANISED CRIME THREAT ASSESSMENT (IOCTA) 2014," Executive Director of Europol, ISBN: 978-92-95078-96-3, pages 23-27, 2014, Source URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2014

[iv] European Union Agency for Law Enforcement Cooperation (Europol), "THE INTERNET ORGANISED CRIME THREAT ASSESSMENT (IOCTA) 2015," Executive Director of Europol, ISBN 978-92-95200-65-4, pages 18-27, 2015, Source URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2015

[v] European Union Agency for Law Enforcement Cooperation (Europol), "THE INTERNET ORGANISED CRIME THREAT ASSESSMENT (IOCTA) 2016," Executive Director of Europol, ISBN 978-92-95200-75-3, pages 17-23, 2016, Source URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2016