AUTOMATED MALWARE ANALYSIS USING SANDBOX

Project report submitted to the Amrita Vishwa Vidyapeetham University in partial fulfillment of the requirement for the Degree of

BACHELOR of TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by

PARUCHURI RAKESH
AM.EN.U4CSE13044
SAKHAMURI VIJAY PAVAN KUMAR
AM.EN.U4CSE13052

श्रद्धावान् लभते ज्ञानम्

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)
AMRITAPURI CAMPUS

KOLLAM -690525

MAY 2017

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)

Amritapuri Campus

Kollam -690525

BONAFIDE CERTIFICATE

This is to certify that the project report entitled **"AUTOMATED MALWARE ANALYSIS USING SANDBOX"** submitted by **Paruchuri Rakesh (AM.EN.U4CSE13044), Sakhamuri Vijay Pavan Kumar (AM.EN.U4CSE13052)**, in partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science and Engineering from Amrita Vishwa Vidyapeetham, is a bonafide record of the work carried out by his/her under my guidance and supervision at Amrita School of Engineering, Amritapuri during Semester 8 of the academic year 2013-2017.

(SUMESH K J)	(SIJI RANI)
Project Guide	Project Coordinator

Dr. M R Kaimal	External Examiner
Chairperson,	
Dept. of Computer Science & Engineering	

Place : Amritapuri
Date : 05-06-2017

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)

Amritapuri Campus

Kollam -690525

DECLARATION

We, **Paruchuri Rakesh(AM.EN.U4CSE13044), Sakhamuri Vijay Pavan Kumar, (AM.EN.U4CSE13052)**, hereby declare that this project entitled **"AUTOMATED MALWARE ANALYSIS USING SANDBOX"** is a record of the original work done by me under the guidance of **SUMESH K J**, Dept. of Computer Science and Engineering, Amrita Vishwa Vidyapeetham , that this work has not formed the basis for any degree/diploma/associationship/fellowship or similar awards to any candidate in any university to the best of my knowledge.

Place : Amritapuri

Date : 05-06-2017

Signature of the student Signature of the Project Guide

Acknowledgements

I offer my sincere pranams to the lotus feet of Mata Amritanandamayi Devi, fondly called AMMA.

Firstly, I am thankful to Dr. M. R. Kaimal, Chairperson, Dept of CSE, Amrita school of Engineering, Amritapuri and Dr. Jyothisha J. Nair, Vice Chairperson, Computer Science, School of Engineering, Amritapuri.

I would like to thank Amrita Vishwa Vidyapeetham and place on record my thankfulness to Mr. Sumesh K J for providing me with the great mentorship. He have been very supportive throughout my tenure there and have provided me with work which has been extremely challenging and informative at the same time. This would not have materialised if not for their friendly demeanour and continuous words of encouragement.

Last but not the least, I would like to thank and dedicate this endeavour of mine to our parents who were constantly at my side, through thick and thin.

Abstract

Number of devices in the software and corporate industry are running Windows due to its flexibility and availability of different types of applications. This has made Windows platform the target for malware attacks. The attacks that are happening is because of the incapability of the firewalls and defenders. So to improve the capabilities of these applications we have to analyze malware and keep on updating them to work efficiently and detect malwares and there is a need to develop this system in a automated way, so that it decreases the human effort. Hence we have developed a sandbox as a research project written in python, which automatically collects, analyzes and reports on the run time indicators of Windows malware. It allows one to inspect the malware before execution, during execution, and after execution (post-mortem analysis) by performing static, dynamic and memory analysis using many of the open source tools. It determines the malware's process activity, interaction with the file system, network, it also performs memory analysis and stores the analyzed information for later analysis. Since our sandbox relies on open source tools, it's easy for any security analyst to setup a personal sandbox to perform malware analysis.

AUTOMATED MALWARE ANALYSIS USING SANDBOX

Project report submitted to the Amrita Vishwa Vidyapeetham University in partial fulfillment of the requirement for the Degree of

BACHELOR of TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by

PARUCHURI RAKESH
AM.EN.U4CSE13044
SAKHAMURI VIJAY PAVAN KUMAR
AM.EN.U4CSE13052

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)
AMRITAPURI CAMPUS

KOLLAM -690525

MAY 2017

Contents

List of Figures

Background

0.1 Malware

Malware is a malicious software computer program designed to damage computers or steal confidential data without the users consent. The term malware includes viruses, worms, Trojan Horses, rootkits, spyware, keyloggers and many more.

0.1.1 Why is Malware Analysis?

Malware is a piece of software which causes harm to a computer system without the owner's consent. Viruses, Trojans, worms, backdoors, rootkits and spyware which are considered as malwares. With new malware attacks making news every day and compromising companies network and critical infrastructures around the world by injecting malwares through fishing or some other technique, malware analysis is important for anyone who responds to such incidents.Malware analysis is the process of understanding the behaviour and characteristics of malware, how to detect and eliminate it. There are many reasons why we would want to analyze a malware.

Determine the nature and purpose of the malware i.e whether the malware is stealing confidential data, damaging infrastructure etc..

Detect network and host based activities to cure and prevent future infections.

0.1.2 Types of Malwares

This section gives a brief overview of the different classes of malware programs that have been observed in a while. The following paragraphs are solely intended to familiarize the reader with the terminology that we will be using in the remainder of this work. Furthermore, these classes are not mutually exclusive. That is, specific malware instances may exhibit the characteristics of multiple classes at the same time. A more detailed discussion of malicious code in general can be found for example in Skoudis and Zeltser [2003], or Szor [2005].

0.1.2.1 Worm

Prevalent in networked environments, such as the Internet, Spaord [1989] defines a worm as a program that can run independently and can propagate a fully working version of itself to other machines. This reproduction is the characteristic behavior of a worm. The Morris Worm [Spaord 1989] is the first publicly known instance of a program that exposes worm-like behavior on the Inter-net. More recently, in July 2001, the Code Red worm infected thousands (359,000) of hosts on the Internet during the first day after its release [Moore et al. 2002]. Today, the Storm worm and others are used to create botnets that are rented out by the bot masters to send spam emails or perform distributed denial of service attacks (DDOS) [Kanich et al. 2008], where multiple worm infected computers try to exhaust the system resources or the available network bandwith of a target in a coordinated manner.

0.1.2.2 Virus

A virus is a piece of code that adds itself to other programs, including operating systems. It cannot run independently - it requires that its host program be run to activate it. [Spaord 1989] As with worms, viruses usually propagate themselves by infecting every vulnerable host they can find. By infecting not only local files but also les on a shared le server, viruses can spread to other computers as well.

0.1.2.3 Trojan Horse

Software that pretends to be useful, but performs malicious actions in the background, is called a Trojan horse. While a Trojan horse can disguise itself as any legitimate program, frequently, they pretend to be useful screen-savers, browser plug-ins, or downloadable games. Once installed, their malicious part might download additional malware, modify system settings, or infect other les on the system.

0.1.2.4 Spyware

Software that retrieves sensitive information from a victims system and transfers this information to the attacker is denoted as spyware. Information that might be interesting for the attacker include accounts for computer systems or bank account credentials, a history of visited web pages, and contents of documents and emails.

0.2 Infection Vectors

This section gives an overview of the infection vectors that are commonly used by malicious software to infect a victims system. Brief examples are used to illustrate how these infections work and how malware used them in the past.

Exploiting Vulnerable Services over the Network

Network services running on a server provide shared resources and services to clients in a network. For example, a DNS service provides the capabilities of resolving host names to IP addresses, a file server provides shared storage on the network. Many commodity o the shelf operating systems come with a variety of network services that are already installed and running. Vulnerabilities in such services might allow an attacker to execute her code on the machine that is providing the service. Large installation bases of services that share the same vulnerability (e.g., [Microsoft Corporation 2008]) pave the way for automatic exploitation. Thus, such conditions allow malicious software to infect accessible systems automatically. This characteristic makes network service exploitation the preferred method for infection by worms. Moreover, services that provide system access to remote users, and authenticate these users with passwords (e.g., ssh, administrative web interfaces, etc.), are frequently exposed to so-called dictionary attacks. Such an attack iteratively tries to log into a system using passwords stored in a dictionary.

0.2.0.1 Drive-by downloads

Drive-by downloads usually target a victims web browser. By exploiting a vulnerability in the web browser application, a drive-by download is able to fetch malicious code from the web and subsequently execute it on the victims machine. This usually happens without further interaction with the user. In contrast to exploiting vulnerabilities in network services in which push-based infection schemes are dominant, drive-by downloads follow a pull-based scheme. That is, the connections are initiated by the client as it is actively requesting the malicious contents. Therefore, rewalls that protect network services from unauthorized access cannot mitigate the threat of drive-by attacks. Currently, two different techniques are observed in the wild that might lead to a drive-by infection:

API Misuse:

- If a certain API allows for downloading an arbitrary file from the Internet, and another API provides the functionality of executing a random le on the local machine, the combination of these two APIs can lead to a drive-by infection [Microsoft Corporation 2006]. The widespread usage of browser plug-ins usually gives attackers a huge portfolio of APIs that they might use and combine for their nefarious purposes in unintended ways.

Exploiting Web Browser Vulnerabilities:

- This attack vector is identical to the case of exploitable network services. Moreover, as described in and Daniel et al. [2008] the availability of client-side scripting languages, such as Javascript or VBScript, provide the attacker with additional means to successfully launch an attack.

Malware Analysis

Today, signatures for anti-virus toolkits are created manually. Prior to writing a signature, an analyst must know if an unknown sample poses a threat to the users. Different malware analysis techniques allow the analyst to quickly and in detail understand the risk and intention of a given sample. This insight allows the analyst to react to new trends in malware development or existing detection techniques to mitigate the threat coming from that sample. The desire of analysts to understand the behavior of a given sample, and the opposing intention of malware authors, to disguise their creations malicious intents, leads to an arms race between those two parties. As analysis tools and techniques become more elaborate, attackers come up with evasion techniques to prevent their malware from being analyzed. Such techniques cover self modifying or dynamically generated code, as well as approaches that detect the presence of an instrumented analysis environment, thus, allowing the malware to conceal or inhibit its malicious behavior. Before we elaborate on possible evasion mechanisms, the following sections present an overview of applicable program analysis techniques that are used today to analyze malicious code. The process of analyzing a given program during execution is called dynamic analysis, while static analysis refers to all techniques that analyze a program by inspecting it.

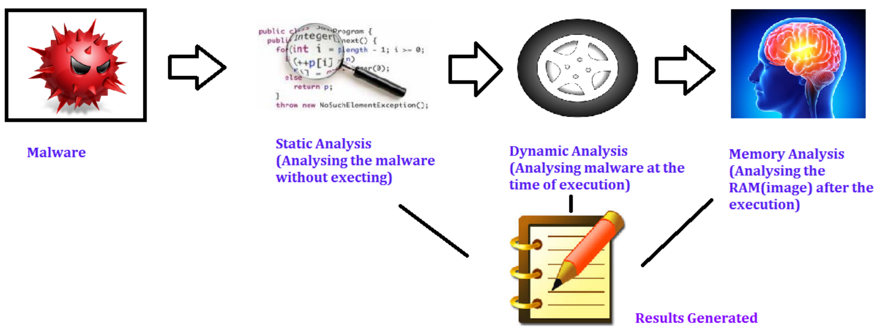

Malware
Static Analysis (Analysing the malware without execting)
Dynamic Analysis (Analysing malware at the time of execution)
Memory Analysis (Analysing the RAM(image) after the execution)
Results Generated

Figure 1: Analysis Chart

0.3 Types of Malware Analysis

In order to understand the characteristics of the malware three types of analysis can be performed they are:

- **Static Analysis**
- **Dynamic Analysis**
- **Memory Analysis**

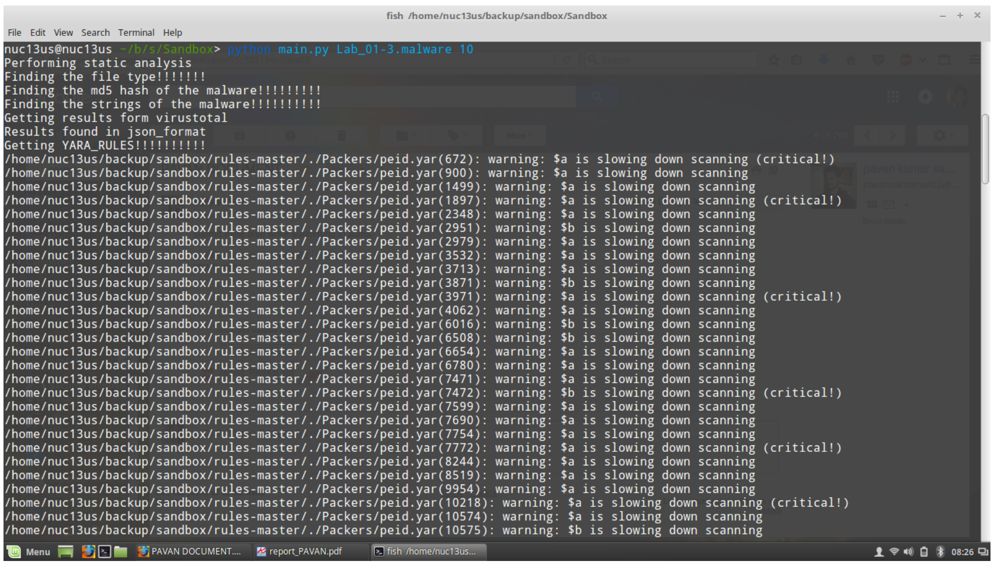
In most cases static and dynamic analysis will yield sufficient results however Memory analysis helps in determining hidden artifacts, rootkit and stealth malware capabilities.

Related Work

While these analysis methods are proven, malware authors introduced new evasion techniques to render them unsuccessful. Obfuscators, Cryptors and Packers are just some of the tools attackers have in their arsenal to evade static analysis. These tools mutate, obfuscate and pack code sections to look different from any other sample, modify the entry point to make it hard to reverse engineer, insert anti-debugging tricks to make it challenging to debug and destroy import tables to blur the sample intentions. These techniques pushed malware researchers towards dynamic analysis, that is, instead of fighting with the sample statically, they simply run it in a controlled environment and observe its execution. Unfortunately, malware authors prevailed and introduced new tricks in that field as well. Sleeping (running out the clock), detecting the underlying virtual/analysis machine and user-interaction which forces a human to click on dialogs or move the mouse are just some of the evasion techniques malware employ today to dodge dynamic analysis. In general, during dynamic analysis the concept of what you see is what you get applies. If the malware feels comfortable on the machine, it will execute its malicious logic, if not, it may mislead the researcher and act benign. With dynamic analysis, code coverage may depend on input given to the sample throughout its execution. While analyzed in an automated system, this input is minimal.

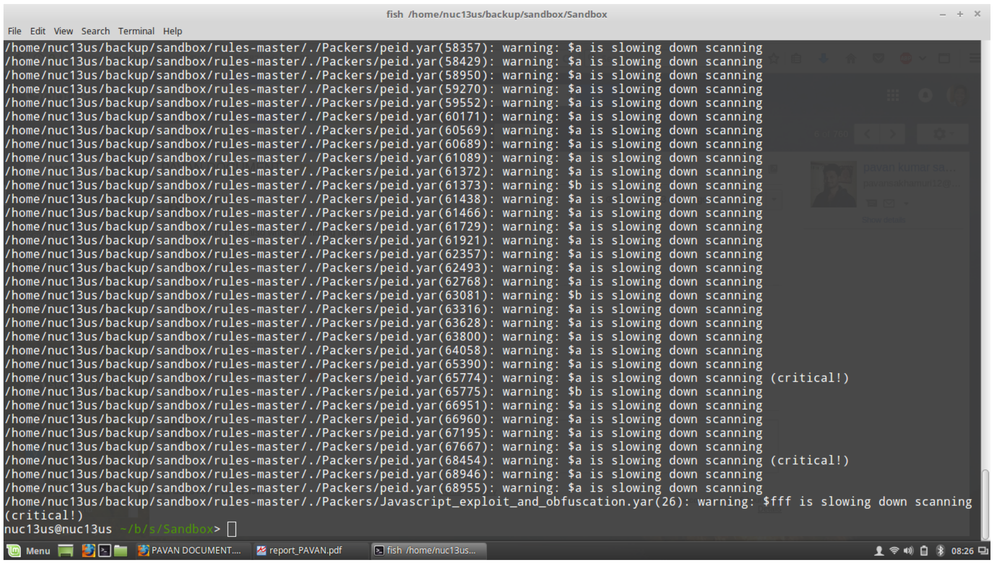
Static Analysis

Static analysis is a code analysis method in which a program is debugged that is examining the code without executing it. In malware analysis it consists of examining the executable file without viewing the actual instructions. Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signatures. Basic static analysis is straightforward and can be quick, but its largely ineffective against sophisticated malware, and it can miss important behaviors.


fish /home/nuc13us/backup/sandbox/Sandbox
File Edit View Search Terminal Help
Menu
PAVAN DOCUMENT....
report_PAVAN.pdf
fish /home/nuc13us...
08:26

```
nuc13us@nuc13us ~/b/s/Sandbox> python main.py Lab_01-3.malware 10
Performing static analysis
Finding the file type!!!!!!!
Finding the md5 hash of the malware!!!!!!!!!
Finding the strings of the malware!!!!!!!!!!
Getting results form virustotal
Results found in json_format
Getting YARA_RULES!!!!!!!!!!
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(672): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(900): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(1499): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(1897): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(2348): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(2951): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(2979): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(3532): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(3713): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(3871): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(3971): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(4062): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(6016): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(6508): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(6654): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(6780): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7471): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7472): warning: $b is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7599): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7690): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7754): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(7772): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(8244): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(8519): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(9954): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(10218): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(10574): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(10575): warning: $b is slowing down scanning
```

Figure 2: Static Analysis 1


fish /home/nuc13us/backup/sandbox/Sandbox
File Edit View Search Terminal Help
Menu
PAVAN DOCUMENT....
report_PAVAN.pdf
fish /home/nuc13us...
08:26

```
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(58357): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(58429): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(58950): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(59270): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(59552): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(60171): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(60569): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(60689): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61089): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61372): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61373): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61438): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61466): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61729): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(61921): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(62357): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(62493): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(62768): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(63081): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(63316): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(63628): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(63800): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(64058): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(65390): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(65774): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(65775): warning: $b is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(66951): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(66960): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(67195): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(67667): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(68454): warning: $a is slowing down scanning (critical!)
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(68946): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/peid.yar(68955): warning: $a is slowing down scanning
/home/nuc13us/backup/sandbox/rules-master/./Packers/Javascript_exploit_and_obfuscation.yar(26): warning: $fff is slowing down scanning
(critical!)
nuc13us@nuc13us ~/b/s/Sandbox>
```

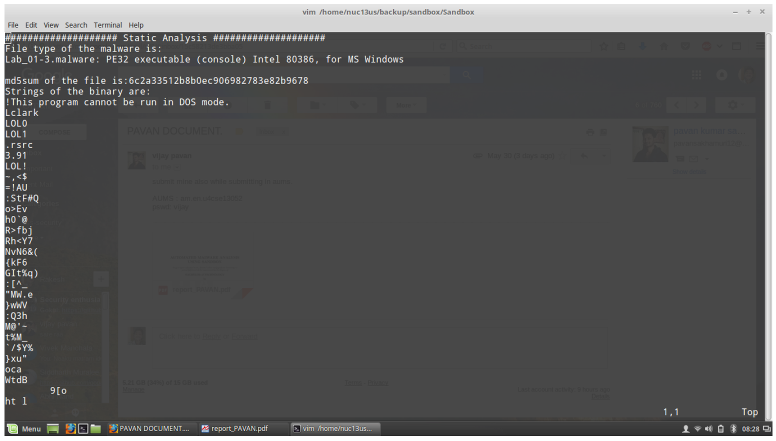
Figure 3: Static Analysis 2

0.4 Determining the File Type

Determining the file type can also help you understand the type of environment the malware is targeted towards, for example if the file type is EXE format which is a standard binary file format for Windows like file systems, then it can be concluded that the malware is targeted towards a Windows flavored systems.

0.5 Determining the Cryptographic Hash

Cryptographic Hash values like MD5 and SHA1 can serve as a unique identifier for the file throughout the course of analysis. Malware, after executing can copy itself to a different location or drop another piece of malware, cryptographic hash can help you determine whether the newly copied/dropped sample is same as the original sample or a different one. With this information we can determine if malware analysis needs to be performed on a single sample or multiple samples. Cryptographic hash can also be submitted to online antivirus scanners like VirusTotal to determine if it has been previously detected by any of the AV vendors.Cryptographic hash can also be used to search for the specific malware sample on the internet.


vim /home/nuc13us/backup/sandbox/Sandbox
File Edit View Search Terminal Help
1,1 Top
Menu
PAVAN DOCUMENT....
report_PAVAN.pdf
vim /home/nuc13us...

```
#################### Static Analysis ####################
File type of the malware is:
Lab_01-3.malware: PE32 executable (console) Intel 80386, for MS Windows

md5sum of the file is:6c2a33512b8b0ec906982783e82b9678
Strings of the binary are:
!This program cannot be run in DOS mode.
Lclark
LOLO
LOL1
.rsrc
3.91
LOL!
~,<$
=!AU
:StF#Q
o>Ev
h0`@
R>fbj
Rh<Y7
NvN6&(
{kF6
GIt%q)
:[^_
"MW.e
}wWV
:Q3h
M@'~
t%M_
`/$Y%
}xu"
oca
WtdB
        9[o
ht l
```

Figure 4: Static Analysis 3

0.6 Strings search

Strings are plain text ASCII and UNICODE characters embedded within a file. Strings search give clues about the functionality and commands associated with a malicious file. Although strings do not provide complete picture of the function and capability of a file, they can yield information like file names, URL, domain names, ip address, attack commands etc.

0.7 File obfuscation (packers, cryptors) detection

Malware authors often use softwares like packers and cryptors to obfuscate the contents of the file in order to evade detection from anti-virus softwares and intrustion detection systems. This technique slows down the malware analysts from reverse engineering the code.

0.8 Submission to online Antivirus scanning services

This will help you determine if the malicious code signatures exist for the suspect file. The signature name for the specific file provides an excellent way to gain additional information about the file and capabilities. By visiting the respective antivirus vendor websites or searching for the signature in search engines can yield additional details about the suspect file. Such information may help in further investigation and reduce the analysis time of the malware specimen. VirusTotal (http://www.virustotal.com) is a popular web based malware scanning services.

Dynamic Analysis

Dynamic Analysis involves executing the malware in a separate environment and monitoring as it runs. Sometimes we cannot extract much of the capabilities of the malware in static analysis due to obfuscation, packing in such cases dynamic analysis is the best way to identify malware functionality. the steps in which dynamic analysis is performed is:

First we setup a separate environment for the malware, where we would be running and analyzing it then we copy the malware sample to the environment and then we execute it. The following steps are performed to analyze the malware.

0.9 Monitoring Process Activity

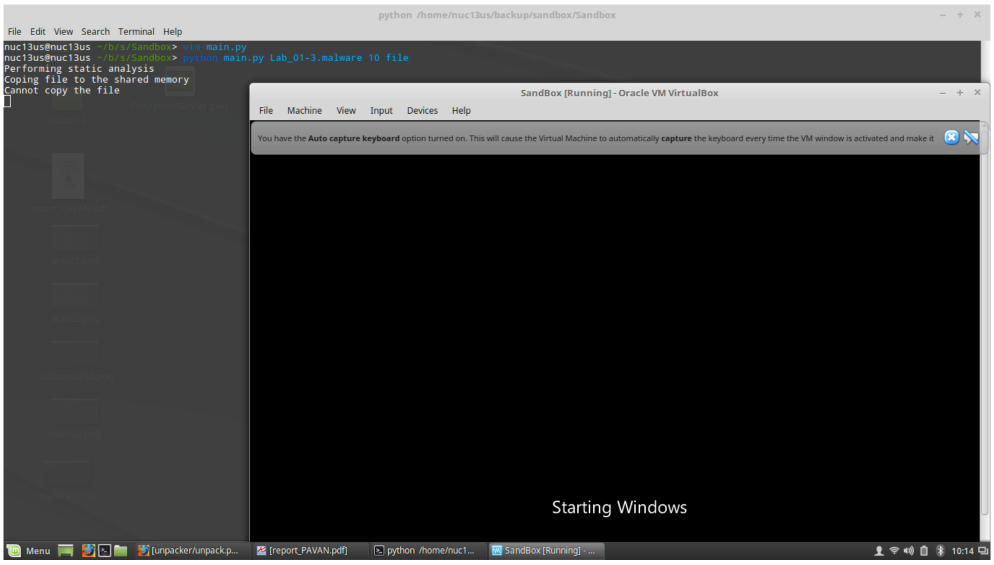
This involves executing the malicious program and examining the properties of the resulting process and other processes running on the infected system. This technique can reveal information about the process like process name, process id, child processes created, system path of the executable program, modules loaded by the suspect program.

0.10 Monitoring File System Activity

This involves examining the real time file system activity while the malware is running, this technique reveals information about the opened files, newly created files and deleted files as a result of executing the malware sample.

0.11 Monitoring Network Activity

In addition to monitoring the activity on the infected host system, monitoring the network traffic to and from the system during the course of running the malware sample is also important. This helps to identify the network capabilities of the specimen and will also allow us to determine the network based indicator which can then be used to create signatures on security devices like Intrusion Detection System.

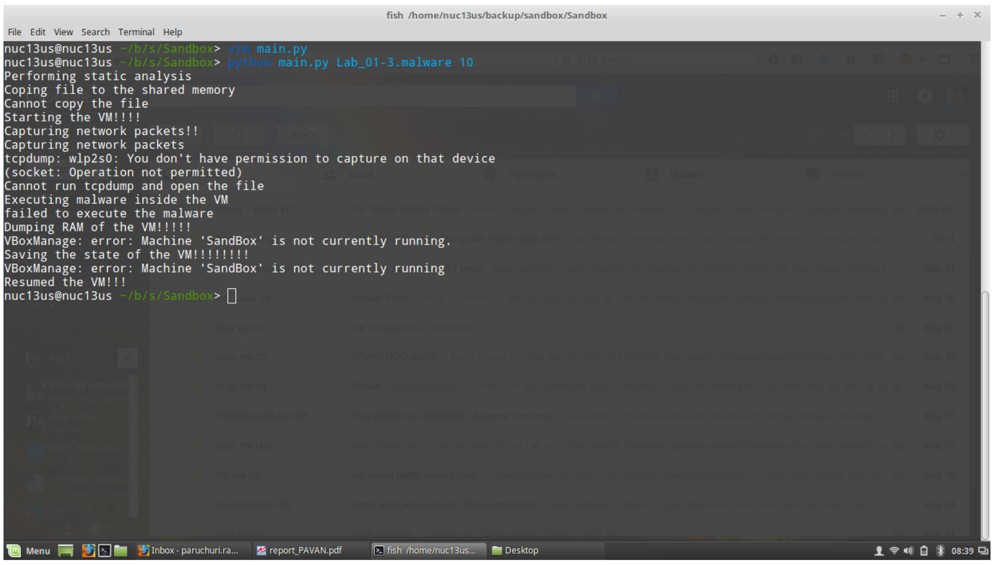


```
nuc13us@nuc13us ~/b/s/Sandbox> vim main.py
nuc13us@nuc13us ~/b/s/Sandbox> python main.py Lab_01-3.malware 10 file
Performing static analysis
Coping file to the shared memory
Cannot copy the file
```

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically **capture** the keyboard every time the VM window is activated and make it

python /home/nuc13us/backup/sandbox/Sandbox
File Edit View Search Terminal Help
SandBox [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Starting Windows
Menu
[unpacker/unpack.p...
[report_PAVAN.pdf]
python /home/nuc1...
SandBox [Running] - ...
10:14

Figure 5: Dynamic Analysis 1


fish /home/nuc13us/backup/sandbox/Sandbox
File Edit View Search Terminal Help
Menu
Inbox - paruchuri.ra...
report_PAVAN.pdf
fish /home/nuc13us...
Desktop
08:39

```
nuc13us@nuc13us ~/b/s/Sandbox> vim main.py
nuc13us@nuc13us ~/b/s/Sandbox> python main.py Lab_01-3.malware 10
Performing static analysis
Coping file to the shared memory
Cannot copy the file
Starting the VM!!!!
Capturing network packets!!
Capturing network packets
tcpdump: wlp2s0: You don't have permission to capture on that device
(socket: Operation not permitted)
Cannot run tcpdump and open the file
Executing malware inside the VM
failed to execute the malware
Dumping RAM of the VM!!!!!
VBoxManage: error: Machine 'SandBox' is not currently running.
Saving the state of the VM!!!!!!!!
VBoxManage: error: Machine 'SandBox' is not currently running
Resumed the VM!!!
nuc13us@nuc13us ~/b/s/Sandbox>
```

Figure 6: Dynamic Analysis 2

Memory Analysis

Memory Forensics is the analysis of the memory image taken from the running computer. Memory forensics plays an important role in investigations and incident response. It can help in extracting forensics artifacts from a computers memory like running process, network connections, loaded modules etc. It can also help in unpacking, Rootkit detection and reverse engineering. When an organization is a victim of advanced malware infection, a quick response action is required to identify the indicators associated with that malware to remediate, establish better security controls and to prevent future ones from occurring. In this article you will learn to detect advance malware infection in memory using a technique called Memory Forensics and you will also learn to use Memory Forensic Toolkits such as Volatility to detect advanced malware with a real case scenario.

Analyzing the memory after executing the malware sample provides post-mortem perspective and helps in extracting forensics artifacts from a computer's memory like:

Running Processes

Network Connections

Loaded Modules

Code Injections

Hooking and Rootkit Capabilities

API Hooking

We would be using the following commands form volatility for analyzing the malware.

pslist - To list the processes of a system.

psscan - This can find processes that have been hidden or unlinked by a rootkit.

pstree - To view the process listing in tree form.

handles - To display the open handles in a process like files, registry keys, mutexes, threads.

privs - This plugin shows you which process privileges are present and enabled.

dlllist - To display a process's loaded DLLs, use the dlllist command.

hivescan - To view the list of kernel drivers loaded on the system.

kernel Modules - To view the list of kernel drivers loaded on the system.

kernel drivers - This can pick up previously unloaded drivers and drivers that have been hidden/unlinked by rootkits.

driverscan - To find DRIVER OBJECTs in physical memory using pool tag scanning.

bash history - This plugin finds structures known as COMMAND HISTORY.

connscan - To find TCPT OBJECT structures using pool tag scanning.

sockets - To detect listening sockets for any protocol (TCP, UDP, RAW, etc).

Automated Unpacking

A Behaviour Based Approach

Certain memory conditions have to be met before malware can unpack code and run it the memory has to be writeable to unpack code to it, and executable to be able to execute it. The question is, can we use Win32 API calls to detect malware creating these conditions, and subsequently not only detect and identify unpacked code, but also find the original entry point?

My honeynet was showing attacks, mostly from the same couple of class C networks, which were downloading malware from hosts on a different but consistent class C network. The file name of the downloaded file would change periodically, but the MD5 hash of the downloaded file seemed to change on a daily basis regardless of whether the file name changed or not.

This was suggesting that the attacks were mostly downloading the same malware, but that the malware was changing slightly on a daily basis in order to make it harder to identify later downloads as being the same as previously captured samples.

We wanted to test this theory, so We started reversing some of the samples and it wasnt long before we noticed the same pattern of behaviour:

Aquire a block of writeable, executable memory.

Unpack code to the newly allocated memory.

Transfer execution to the unpacked code in the newly allocated memory.

In order to be able to unpack code, malware needs a block of writeable memory. It has to write to that memory. It has to pass control to that block of memory. The first condition requires allocation of a block of memory with write permissions.

The second requires a write memory access to that block. The third requires that the block of memory have execute permissions and also for the processor to start fetching instructions from that block of memory.

We were going to discuss how a scriptable debugger can be used to automate detection of the above pattern of behavior. We first used PyDbg as the debugger, but switched to WinAppDbg as we added more analysis functionality, because we found that WinAppDbg made it easier to debug multiple processes at once.

0.12 Acquiring a block of writeable, executable memory

The first step in automating this process is to detect a VirtualAlloc() request for executable memory. This can be done using an API hook. Note that VirtualAlloc() calls VirtualAllocEx() with hProcess = 0xffffffff (-1), so I hooked VirtualAllocEx() (to support some further analysis functionality which isnt included here as this post is about automated unpacking).

To hook API calls in WinAppDbg, add them to the apiHooks associative array (dictionary/hash), which is indexed by library file name.

WinAppDbg calls the post VirtualAllocEx() hook callback function when the VirtualAllocEx() call returns. By hooking this call on exit rather than on entry, we have access to its return value, being the address of the allocated memory.

We need the address and size (passed as an argument to the VirtualAllocEx() call) of the allocated memory block for the next part determining when the unpacked code is written to the allocated block of memory.

0.13 The VirtualAllocEx() hook handler

The VirtualAllocEx() hook handler, post VirtualAllocEx(), and is called by WinAppDbg when the VirtualAllocEx() library call returns (or is about to return).
At this point, we know that VirtualAllocEx() was called and since it is about to return, we have access to its return value, being the address of the allocated memory.

If flProtect contained any of PAGE EXECUTE (0x10), PAGE EXECUTE READ

(0x20), PAGE EXECUTE READWRITE (0x40), or PAGE EXECUTE WRITECOPY (0x80), then the request was for executable memory which suggests that the malware is about to unpack (or otherwise obtain) code to store there.

Our plan is to catch the malware writing to the newly allocated memory, as that will help to identify the unpacking loop, and to also catch when the processor jumps to (starts executing instructions from) the newly allocated memory.

So the next step, is to set a memory breakpoint. Memory breakpoints (in this case) use guard pages to alert us when the allocated block of memory is accessed. In this case we want to know when it is written to, or you could say when we have new code on the block (and that should have generated a bad joke exception).

Memory breakpoints are created in WinAppDbg using the debugger instances watch buffer() method. Now be careful with this as we spent some time trying to figure out why my memory breakpoints werent working.
Eventually we examined the WinAppDbg source and realised that it is using watch buffer()s address and size parameters to (incorrectly) calculate an end address. This is because it needs to know which memory pages are affected by the breakpoint because guard pages can only be set on whole pages.

The problem arises because of the old fence-post problem. The size of a memory page (on 8086 processors) is 4096 bytes, however, these bytes are addressed (numbered) from 0 4095. That means that if you calculate which pages are affected by simply adding the size on to the base address of the first page, which WinAppDbg does, you get one too many pages.

In other words, WinAppDbgs watch buffer() functionality is calculating the end address incorrectly, by adding its address and size parameters. As an example, in the case of a one page guard page starting at address 0, we have an address of 0, and a size of 4096 (the size of a page). Adding them together gives you 4096, which is actually the start address of the second page. WinAppDbg thinks that it needs to create two guard pages, one for the page starting at address 0 to cover addresses 0 4095, and then a second page starting at address 4096 to cover address 4096.

This off-by-one error was causing my watch buffer() calls to fail. Just thinking about it, that makes me wonder if it will ignore access to the last byte of the range now that I have subtracted 1. That is, is it setting the range correctly, but calculating affected pages incorrectly theres something to test.

0.14 Unpack code to the newly allocated memory

After setting up guard pages using WinAppDbgs watch buffer(), WinAppDbg will call our guard page exception handler, guard page().

Guard page exceptions occur because a guard page was accessed. This access could be due to a memory read, memory write, or an instruction fetch (executing an instruction).

We dont care about memory reads, but if the access was due to a memory write then we want to attempt to find the unpacking loop. If the access was to execute instructions, then we want to log both the address, and the instruction that caused the processor to start executing instructions at that address.

Any write access is logged with the address and disassembly of the instruction that caused it. This is to log all the locations from which the allocated memory is written and make it easier to find the unpacking code in the original malware binary. Care is taken to make sure that we only log each instruction(s) once, as it (they) will more than likely be inside a loop and actually execute a number of times.

Since the instruction(s) that wrote to the newly allocated block of memory are more than likely inside some sort of unpacking loop, lets have some fun and attempt to find the bounds of the loop. To do this, unpack.py calls the WinAppDbg debugger instances start tracing() method to turn on single-step debugging for the thread that caused the guard page write exception.

Single-stepping causes the processor to issue a single-step exception after executing each instruction. WinAppDbg catches these exceptions and calls our single-step exception handler, single step()

0.15 Single-step exception handler

The single-step exception handler is used so that we can attempt to find the unpacking loop, and the address of the instruction that transfers control to the unpacked code.

Im not overly happy with my scripting for single step(), as Im not convinced that Ive arrived at the most elegant algorithm, and it isnt perfect. Under some circumstances it can fail and end up single-stepping through library functions. It is, however, the best that Ive come up with to date, so Im sticking with it for now.

The single-step exception handler attempts to find the bounds of the unpacking loop by assuming that the write instruction (that caused the guard page write exception) is in the middle of the loop. It watches the exception address, which is the address of the instruction that the processor just finished executing, which, under normal processing, should keep incrementing.

When it notices the exception address go backward, it assumes that the processor has reached the end of the loop body and is either looping back to re-evaluate the loop condition, or that it has just re-evaluated the loop condition and it is looping back for another iteration. Either way this lower address is stored as being the start of the loop. The previous execution address (that is the address before it dropped back) is stored as being the end of the loop, as that address obviously holds the instruction that caused the processor to jump back to the lower address.

Any subsequent instructions that are between these two addresses, that is, they are part of what we suspect is the unpacking loop, are then disassembled (if they havent already been disassembled remember, they are in a loop, and we only want to disassemble them once).

The final trick of the single-step handler is to remember the last two addresses that were executed. We need to know the second last address to log the address and instruction that transferred control to the unpacked code. The single-step debugging continues until we notice the processor executing instructions from the newly allocated memory.

The single-step debug exception occurs after an instruction has executed, and the exception address/return address is that of the next instruction to be executed. This means that when the single-step handler returns, control will resume with the next instruction. The guard page exception occurs upon access to the protected memory.

Consider the following sequence of instructions, but first, allow me to set the scene. Its a dark and dreary night, seems like nothings going right, and this code has just called VirtualAlloc() to request some memory of the executable variety. It has just finished unpacking and making a nice cosy nest for itself in the new block of memory, and is about to execute the following instructions.

The instruction at address 0x3c1530 is an unpacked instruction in the newly allocated block of memory, which the call edx instruction at 0x405153 is hoping to surprise us all by passing control to:

```
    ; edx contains 0x3c0000 at this point
0x40514d: adc edx,0x1530
0x405153: call edx ; transfer control to unpacked code
0x405155: add esp,0x0c
...
0x3c1530: push ebp
```

Here is what we believe happens. The processor executes the instruction at address 0x40514d and generates a single-step debug exception. WinAppDbg catches this exception and passes control to our single-step debug exception handler which goes by the imaginative name of single step() (the name is set by WinAppDbg).

The exception address, in the case of single-step debug exceptions, is the address to which the handler will return which, since we dont want to execute the same instruction again, will be the address of the next instruction to execute. In this case, that will be 0x405153.

Not wanting the call edx instruction to feel left out, and the fact that it is the next instruction probably has a lot to do with it, the processor executes the call edx instruction at 0x405153. Again, our single-step debug exception handler gets control but this time the exception address, being the address of the next instruction to execute, will be 0x3c1530 the destination of the call instruction.

So at the end of single step(), lasteip[0] will be 0x405153, lasteip[1] will be 0x3c1530.

When the processor attempts to fetch the instruction from address 0x3c1530, it triggers the guard page and our guard page handler, guard page(), is called. Since the single-step exception handler gets the address of the next instruction to be executed, our guard page() function sees 0x3c1530 as being the last address to execute. Hence we need to keep track of the last two addresses.

0.16 Transfer of execution to the unpacked code

Right, back to the guard page exception handler, guard page(), at E.1. If we look out for a guard page exception caused by an instruction fetch (instruction execution), then we can easily find the original entry point of the unpacked code.

The entry point will be the first address to cause a guard page execution exception, in the newly allocated memory. guard page() checks for this condition and if found, logs the entry point, and the address and disassembly of the instruction which transferred control to the new memory block.

After logging the above information, guard page() searches the list of executable memory blocks that were allocated by VirtualAllocEx() and recorded by post VirtualAllocEx(), until it finds the block of memory containing the address that the processor just jumped to. It deletes the memory breakpoint, stops the single-step debugging, resets the variables used to find the unpacking loop, and dumps the block of memory to a file.

Conclusion

Outcome of this work will provide useful and crucial assistance to forensic investigators in analyzing live memory dumps for use of possible malware. Tool was tested with 3 various samples and gave accurate results for all the samples. Automation, accuracy and user friendliness of the tool will help in speeding up the live forensics. Memory Analysis is becoming a great technique to analyze Malware. Unfortunately, the current approach has many limitations. In this paper, we proposed possible solutions to overcome these limitations and enhance the current solutions. Our main contribution is the Trigger-Based memory analysis approach for automatically taking memory dumps when interesting actions happen in the system during execution. This approach gives the security researcher invaluable information of what happens during the execution of the sample and not only what happened at the end. In combination with other techniques for memory analysis, this approach strengthens security and gives more valuable insight in the efforts to detect malicious activity

Future Scope

Though, the tool meets all the current requirements for automatic forensic malware analysis, there is still scope of enhancement. The tool can be further extended to include more aspects of malware analysis. The tool currently supports only existing profiles, so a provision can be made available in the tool to accommodate new profiles.

Last but not the least, a GUI to this automated process can make it possible to get results in couple of clicks

Tools Used by Our Sandbox

Our Sandbox relies on Open source tools to perform static, dynamic and memory analysis.

Custom python scripts

YARA-python

https://github.com/plusvic/yara

VirusTotal Public api

https://www.virustotal.com/en/documentation/public-api/

strings utility

http://linux.die.net/man/1/strings

Tcpdump

http://www.tcpdump.org/

Sysdig

http://www.sysdig.org/

Volatility memory forensics framework

http://www.volatilityfoundation.org/!releases/component 714

References

[1]Anubis. Analysis of unknown binaries. http://anubis.iseclab.org. Last accessed, May 2010.

[2] Avira Press Center. 2007. Avira warns: targeted malware attacks increasingly also threatening German companies. http://www.avira.com/en/security news/targeted attacks threatening companies.html. Last accessed, May 2010.

[3]Backes, M., Kopf, B., and Rybalchenko, A. 2009. Automatic discovery and quantication of information leaks. In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. IEEE Computer Society, Washington, DC, USA, 141153.

[4]Chen, H. and Wagner, D. 2002. MOPS: an infrastructure for examining security properties of software. In Proceedings of the 9th ACM conference on Computer and communications security (CCS). 235 244.

[5]Bayer, U., Moser, A., Krugel, C., and Kirda, E. 2006. Dynamic analysis of malicious code. Journal in Computer Virology 2, 1, 6777.

[6]H. Carvey. Windows Forensics Analysis.Syngress, 2007.

[7]B. M. An introduction to windows memory forensic. Technical report, July 2005. http://forensic.seccure.net/pdf/

[8]M.Anderberg. Cluster Analysis for Applications.Academic Press,Inc.,NewYork,NY,USA, 1973.

[9]Finding Advanced Malware Using Voltality by Monnappa Ka.