SCALABLE, AUTOMATIC MALWARE ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Allison Sommers

In Partial Fulfillment
of the Requirements for the Degree
Computer Science in the
School of College of Computation

Georgia Institute of Technology
May 2018

SCALABLE, AUTOMATIC MALWARE ANALYSIS

ACKNOWLEDGEMENTS

I wish to thank everyone in the Astrolavos Lab at Georgia Tech, for their collective spirit. I would especially like to thank Omar Alwari and Morgan Mango, as without them and their major contributions to our project, this paper would not exist.

I also thank Manos Antonakakis, for his guidance as my advisor.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

In the realm of this computing age, malware is becoming steadily more prevalent. With the amount of malware samples taken from the wild increasing, malware analysis is becoming increasingly necessary. However, the necessary step of malware analysis is not straightforward, and is often made intentionally more difficult by malware authors. Dynamic sandboxes, often used to analyze wild malware samples, have been used for years as a trusted and necessary component for malware analysis.

We expand on the traditional approach to malware analysis by creating a system to provide autonomous, automated assistance for analyzing malware samples. Utilizing virtual machine technology, open-source memory forensics software, and custom scripts in our system, we built our system with the goal of speeding up memory forensics during malware analysis.

CHAPTER 1

INTRODUCTION

Literature Review

In this computing age, malware is becoming steadily more prevalent. With the amount of malware samples taken from the wild increasing, malware analysis – specifically, memory forensics for malware analysis – is becoming increasingly necessary. However, malware analysis is not a simple matter, as the malware binaries are usually packed and it is not always obvious what the malware is doing to the system it has infected.

Malware analysis in general is made difficult in part by code obfuscation methods, which cause the malware binary file to be more difficult to reverse engineer. When these malware binaries are packed, meaning that they encrypt important components and only decrypt them for use, it presents challenge to the analysis of the binary. With over half of malware binaries captured exhibiting packing [1], this is obviously a prevalent problem for reverse engineering malware.

To combat the issue of packing, reverse engineers need to unpack the encrypted elements. The methods usually involve using pre-built malware unpackers or custom software. Unfortunately, most malware unpackers each only work on specific malware encryption strategies [4, 5] – they will fail for novel and unique encryption methods used by more inventive authors. To unpack a wider variety of malware binaries, a more adaptable decryption tool proposed by Raber allows for unpack a much larger group of malware binaries [7]. However, analyzing malware binaries still requires that the malware successfully infect and run on a machine, which provides a new set of challenges.

When malware binaries are run, they must be run in a controlled and secure environment. To achieve this end, albeit focusing on dynamic analysis of malware rather than memory forensics, virtual machine setups have been proposed [2, 3, 6], for dynamic malware analysis and possibly extracting the unpacked version of the binaries from memory, after the malware can run and decrypt itself. This method relies on memory forensics tools [8] to analyze the changes malware has made to the infected computer.

Automating Memory Forensics Analysis

Our goal in this thesis is to propose a system to assist in automating a portion of the malware analysis, specifically by building KVM sandboxes to infect with malware and then dissect with open-source memory forensics tools. We want our system to be able to take a malware binary, run the malware binary on a KVM sandbox, run scripts to analyze the volatile memory of the infected KVM, and to give the analyst the results from those scripts. Ideally, we want the output of our system to tell the analyst concisely tell the analyst what changes to the memory are from malware.

CHAPTER 2

METHODS AND MATERIALS

System-Setup

Building on running Virtual Machine sandboxes, we designed a system for running multiple automatic malware runs on multiple VMs simultaneously. The host, Ubuntu server, utilizes KVMs, a type of Virtual Machine, to run our guest Virtual Machines. Our KVM acts as our hypervisor, VIRSH the manager of the KVMs, and CUCKOO as our live sandbox environment. The relationship is explained through diagram below.

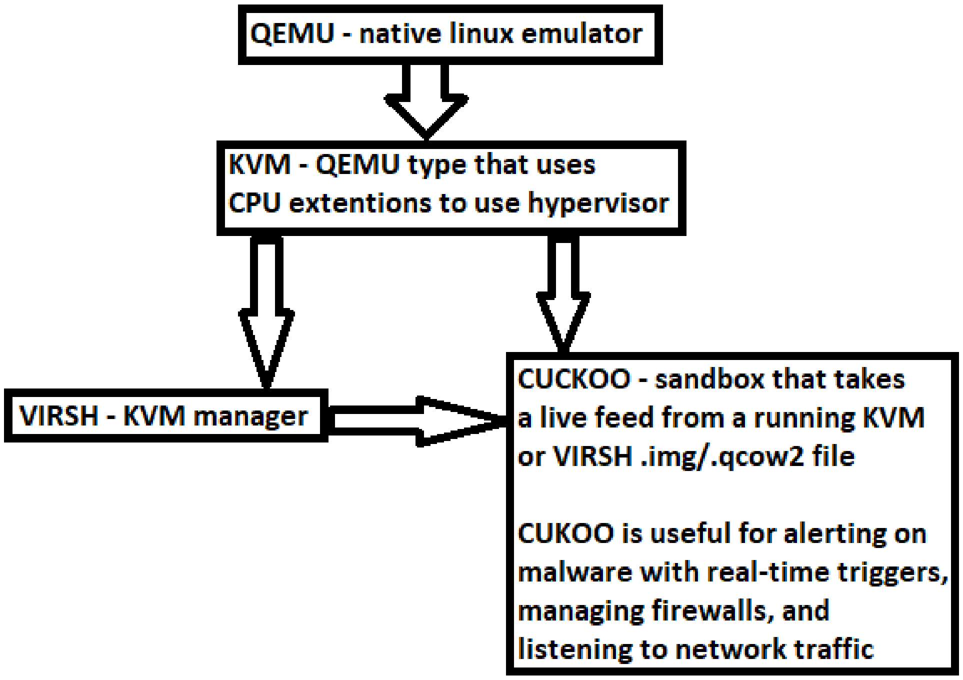
QEMU - native linux emulator
KVM - QEMU type that uses
CPU extentions to use hypervisor
VIRSH - KVM manager
CUCKOO - sandbox that takes
a live feed from a running KVM
or VIRSH .img/.qcow2 file
CUKOO is useful for alerting on
malware with real-time triggers,
managing firewalls, and
listening to network traffic

Figure 1: Hypervisor Setup

Our system gives us the ability to run potentially malicious programs inside the guests from the host, without risking contamination from the guest: as our guest VMs are a different type of operating system and the host's memory is protected through the hypervisor. Furthermore, we use Cuckoo to trigger in real time on alerts and gives further capabilities to the system. One of Cuckoo's such capabilities include Virsh, a KVM management tool, which we utilize in a multitude of ways. Firstly, we use Virsh to create a fresh install on a KVM, which functions as our clean image. Secondly, use Virsh to create copies of this clean image, to infect these KVMs. Finally, Virsh efficiently produces memory dumps for our analysis from the infected KVMs.

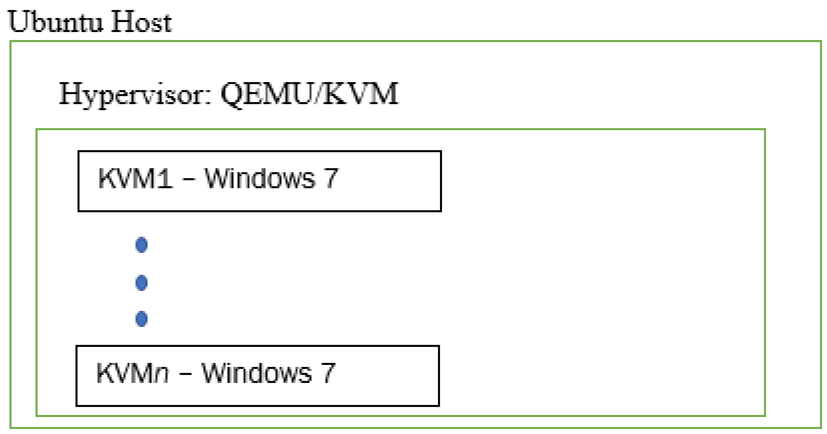

Figure 2: KVM Setup

The KVMs themselves are allocated approximately 4 GB of RAM and 25 GB of memory for the virtual hard disk. For our initial prototype, we opt to use Windows 7 on all our KVMs, as this is the version of Windows which works best for our system setup. Our tools for memory forensics, Sleuthkit and Volatility, are compatible with Windows 7 and can also be used on Windows 10.

Volatile Memory Analysis Using Volatility.

In the effort to gather accurate information on what the malware is doing to the system post-infection, it is necessary to analyze the volatile memory of the KVM. Volatility is one of the only open source projects that provides volatile memory analysis, and it is often considered the best at this type of analysis. [8] For these reasons, we highly recommend utilizing Volatility with a system like ours – especially as Volatility only requires a memory dump and the appropriate commands to return useful information. For our volatile memory analysis, we are most interested in the following: created processes, terminated processes, created files, command line history, modified security identifiers, code injection, and registry keys.

For seeing the processes of a system, Volatility has several commands that go through and enumerate the processes of the memory dump – we use pstree and psxview. Malware will often try to hide itself in process enumeration through renaming itself and updating the lists for process enumeration within the system, which makes automating this segment of the analysis tricky. The methods for retrieving files from the volatile memory work in a similar fashion, as Volatility's command filescan completes the job. Likewise, to obtain command line history, Volatility has two commands, cmdscan and consoles, to achieve this – the first simply tells us what commands were entered into the command line, while the second offers what the input and output of those commands were. Thus, with a few simple commands from Volatility, it is easy to extract this information with an automated system – saving an analyst time from manually doing so.

However, finding security identifiers and code injection is less concrete, and there is always the risk that a code injection method that Volatility does not detect has been used. To get security identifiers, DLLs, and kernel objects (such as drivers), we use the following Volatility commands: dlllist, getsids, svcscan, driverscan, ssdt, privs, envars, modules, and modscan. Because the Windows operating system is bloated, these results return a very large amount of data – and most of it is not relevant to the malware.

Thus, it is necessary to compare the results of the non-infected KVM to the memory dump of the infected KVM, where the two memory snapshots have been taken within the closest time proximity possible – so that the unnecessary information can be more easily filtered from the memory snapshot of the infected machine, as it is similar to the snapshot from before it was infected.

Finally, registry keys can be analyzed with Volatility, and is sometimes necessary for memory forensics analysis. However, there is no efficient way simply dump all the registry keys – for analyzing a registry key, you must have its memory address and use one of Volatility's commands to extract the data from that memory address. We used a recursive algorithm to list all the registry keys in the volatile memory, but this is an imperfect and inelegant solution.

In all, our system for automating memory analysis with Volatility is efficient – running in approximately 6 minutes on a memory snapshot less than 4 GB. However, the registry key enumeration can takes several hours, and often outputs registry values more than once due to its recursive structure.

Network Traffic Analysis Using Cuckoo.

Cuckoo has a multitude of capabilities that make it useful for automated malware analysis. For our purposes in analysis, we use Cuckoo to analyze network traffic. In the future, we also plan on expanding our system's capabilities utilizing more of Cuckoo's functionality; Cuckoo has plugins for Volatility, for analyzing volatile memory forensics, which we would like to see added to this system's capabilities.

Disk Memory Analysis with Sleuthkit.

Sleuthkit is one of the best open source platforms for disk memory analysis; it is trusted for analyzing memory by law enforcement. [8] It is capable of efficiently analyzing a hard drive for data, but we use it specifically for analyzing the virtual hard drive for our KVM. This can be done by virtually mounting the virtual hard drive, then using Sleuthkit on the mounted partition.

Because we have a copy of the machine before it was infected and a copy of the machine after it's infection, we can simplify the disk forensics. When we already know the file system of the machine before the malware ran on it, we know we are most interested in: the timeline of events happening on the disk, created and deleted files by the malware, and registry keys. Sleuthkit has tools for recovering deleted files (tsk_recover is the method we chose) and the tool tsk_gettimes for a timeline of events happening on the disk. Thus, we can get all the deleted files on the disk, and we can obtain a timeline for everything that happened to the disk since the moment our malware sample started to run.

Another component that Sleuthkit can do is extract registry key information – however, this is not entirely clear in the documentation how to do this through command line commands. The user interface for Sleuthkit, Autopsy, allows an analyst to extract registry key information with ease, but we were unable to repeat this with terminal commands on our Linux host system.

Once we have these results, it’s simply a matter of comparing the clean copy of the machine with the infected copy of the machine. The time this will take depends on the size of the KVM and the RAM of the host machine, but usually will take less than an hour for KVMs with disk space less than 50 GB.

CHAPTER 3

THE EXPERIMENT

For our experiment, we used two samples of Pony Loader malware. Specifically, we used two samples with the SHA1 hashes b6ab338d8058e2555a573f46afbd016b and 6db434088b32a974a557eacd0d5b9b60. These can be found online, through searching for their hashes.

Our system, once we have decided on a malware sample to analyze, uses a pipeline as described in Chapter 2. The system can create a new KVM to infect with the malware sample – there are a multitude of different methods for this. For our purposes, specifically used a Simple Python HTTPS Server to download the samples onto the KVM and infect the machine. Cuckoo, monitoring the network traffic of the KVM, will record this initial infection, but it is clearly not relevant for the analysis as we know the time and reason for the connections.

Next, our system creates memory snapshots of the KVM after infection in time increments of one minute, two minutes, and then three minutes. These memory snapshots are then passed to a bash script, which will run the Volatility commands we selected as useful in Chapter 2. The system can then compare the results of the volatile memory analysis with the known volatile memory of the KVM before the malware infected it.

The second script that runs deals with the virtual hard disk: first, it converts the KVM’s virtual hard disk from a .qcow2 format to a .img format, virtually mounts it, and then uses Sleuthkit to perform memory forensics on the virtual hard disk.

At the end of this process, the system outputs the results of the memory and volatile memory analysis, as well as the network traffic. This will give the analyst the list of connections made by the KVM that are potentially malicious network traffic.

CHAPTER 4

RESULTS

From the sample malware we tested the system on, we were able to see some success from our system, but we also saw a multitude of its failures. We were successfully able to extract network information from Cuckoo, to give a timeline of the network traffic, and we successfully extracted volatile memory artifacts utilizing Volatility in an efficient manner. While the system succeeded in extracting this information, it is not easy for an analyst to parse – the Volatility script alone outputted a text file of over 4kB to parse, which is not fit our goal for the system returning concise information to assist analysts.

The Volatility script was able to output useful information for most of the components we targeted, finishing in under twenty minutes. However, our script failed to terminate quickly when enumerating the registry keys. We manually terminated the Volatility script after it continued running for several hours; we found that the recursive enumeration caused several registry keys to be outputted more than once, showing us that this was an inefficient approach to the problem. We expected that the recursively enumerating the registry keys was the most time intensive part of the Volatility script, but this shows that this manner of extracting registry key information is not ideal for automated memory forensics due to the slow run time and inefficient method for listing the registry keys.

In addition, our Sleuthkit script failed to run on several occasions due to a mounting error for the virtual hard disk. Our method for mounting the virtual hard drive, such that the KVM’s memory could be analyzed by Sleuthkit, often failed in mounting the hard disk. As the hard disk was not mounted, the tools from Sleuthkit could not run and extract the information we wanted from the hard disk. These errors are easy to fix with human involvement but are not currently fixed in our automated system.

Thus, the main problems with the system is its sensitivity to errors and too much unnecessary data from the Volatility script. The system is not selective enough to pick out what is truly activity from malware, and what is simply expected from the Windows operating system itself. We stand by the basis of the system, but not in the finesse of the system – thus, we succeeded in creating this system to help assisting analysts to reverse engineer malware, but we failed in making the system useful to the analyst.

Future Work.

First, we would want the system to become useful to the analyst, and this would involve making the system more capable of autonomously handling its errors. In addition, we would want the output from the system to be more useful for the analyst; instead of a massive amount of text to sort through from the volatile memory analysis, we want the system to be able to present the analyst with only relevant information. One possible example of this would be to add a list of whitelisted processes, such as schvhost.exe, for the system to use to narrow down the list of potentially malicious processes running in the system.

REFERENCES

[1] Bustamante P, Packing a Punch. PandaResearch Blog, February 2007, [online] Available: http://research.pandasecurity.com/Packing-a-punch/.

[2] Payne, B. D., Carbone, M., Sharif, M., & Lee, W. (2008). Lares: An Architecture for Secure Active Monitoring Using Virtualization. 2008 IEEE Symposium on Security and Privacy (sp2008). doi:10.1109/sp.2008.24

[3] Spensky, C., Hu, H., & Leach, K. (2016). LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. *Proceedings 2016 Network and Distributed System Security Symposium*. doi:10.14722/ndss.2016.23121

[4] Unpackers. exetools.com, August 2002, [online]

Available: http://www.exetools.com/unpackers.htm.

[5] Unpacking Tools, Collaborative RCE Tool Library, 2013, [online] Available: http://www.woodman.com/collabrative/tools/index.php/Category:Unpacking_Tools.

[6] Royal, P, Halpin M, Dagon D, Edmonds R, Lee W. (2006) Automating the Hidden-Code Extraction of Unpack-Executing Malware. *IEEE Computer Society*, dl.acm.org/citation.cfm?id=1191885.

[7] Raber J. (2017) Columbo: High performance unpacking. *2017 IEE 24th International Conference*. http://ieeexplore.ieee.org/document/7884663/

[8] Ligh M, Adair S, Hartstein B, Richard M (2010). Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code. https://dl.acm.org/citation.cfm?id=1964877