Evaluating Tool Based Automated Malware Analysis Through Persistence Mechanism Detection

by

Matthew S Webb

B.S., Kansas State University, 2016

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Dr. Eugene Vasserman

Copyright

Abstract

Since 2014 there have been over 120 million new malicious programs registered every year. Due to the amount of new malware appearing every year, analysts have automated large sections of the malware reverse engineering process. Many automated analysis systems are created by re-implementing analysis techniques rather than automating existing tools that utilize the same techniques. New implementations take longer to create and do not have the same proven quality as a tool that evolved alongside malware for many years.

The goal of this study is to assess the efficiency and effectiveness of using existing tools for the application of automated malware analysis. This study focuses on the problem of discovering how malware persists on an infected system. Six tools are chosen based on their usefulness in manual analysis for revealing different persistence techniques employed by malware. The functions of these tools are automated in a fashion that emulates how they can be manually utilized, resulting in information about a tested sample. These six tools are tested against a collection of actual malware samples, pulled from malware families that are known for employing various persistence techniques. The findings are then scanned for indicators of persistence. The results of these tests are used to determine the smallest tool subset that discovers the largest range of persistence mechanisms. For each tool, implementation difficulty is compared to the number of indicators discovered to reveal the effectiveness of similar tools for future analysis applications.

The conclusion is that while the tools covered a wide range of persistence mechanisms, the standalone tools that were designed with scripting in mind were more effective than those with multiple system requirements or those with only a graphical interface. It was also discovered that the automation process limits functionality of some tools, as they are designed for analyst interaction. Regaining the tools' functionality lost from automation to use them for other reverse

engineering applications could be cumbersome and could require necessary implementation overhauls. Finally, the more successful tools were able to detect a broader range of techniques, while some less successful tools could only detect a portion of the same techniques. This study concludes that while an analysis system can be created by automating existing tools, the characteristics of the tools chosen impact the workload required to automate them. A well-documented tool that is controllable through a command line interface that offers many configuration options will require less work for an analyst to automate than a tool with little documentation that can only be controlled through a graphical interface.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

- OS – Operating System
- DLL – Dynamically Linked Library
- VM – Virtual Machine
- GUI – Graphical User Interface
- API – Application Programming Interface
- AESP – Autostart Extensibility Point
- AV – Anti Virus

Acknowledgements

This thesis would not have been possible without the help and support of the many individuals I am lucky enough to call friends or family. In particular, I would like to express my appreciation to the following:

First and foremost, my wonderful wife Katherine. You never stopped supporting and loving me, even as I traded our time together in the evenings for working in front of the computer. Your many meals and constant proofreading and assistance with grammar and sentences got me through this research, and meant more to me than I ever let on. I'm so glad I married you.

To my major professor, Eugene Vasserman, who tirelessly guided me from the development of my research through to the final defense. From discussions in your office to the many emails sent back and forth refining this thesis, you helped make this paper much better than it would have been otherwise.

Finally, to my family, Craig, Susan, Art, and Caitlyn for their late notice proofreading and editing. I'm lucky to have you in my life.

Chapter 1 - Introduction

This chapter provides an overview for the content of the thesis. The first section provides a cursory background of the problem, followed by the formalized problem statement, intended research objectives, and the significance of the research performed. The final section will define the scope of the work as well as implementation limitations.

Problem Background

Malware is code that has been created for malicious purposes. Malware analysts are the researchers tasked with studying malware in order to learn how it works and protect computers from being infected. A unique instance of malware that analysts study is called a malware sample.

In the past, malware analysts would manually reverse engineer malware samples using static and dynamic analysis to understand how malware worked and what malware was trying to accomplish. Information gathered by inspecting a malware sample without running it, therefore removing the risk of malware contamination, is called static analysis. Dynamic analysis involves executing malware in a controlled environment to study its behavior. Since dynamic analysis requires malware execution, special steps need to be taken to avoid malware proliferation.

As the analysis process develops, tools are created to simplify tasks and aid researchers. Analysts use the information gathered to create remedies for infected systems and identifying signatures to prevent future infection. This process can be time consuming, as malware often contains anti-reversal techniques that inhibit reverse engineering by an analyst.

As the world becomes more computerized, creating malware becomes more lucrative. This has led to a sharp increase in the amount of new malware create. Due to the time required to manually study a new sample, analysts are unable to keep up with the flood of new malware. To

combat this issue, analysts have turned to automating various parts of the reverse engineering process.

Automating certain tasks in the reversal process can save large amounts of time and work. Automation methods range from scripts that perform static analysis to extract information from a sample to automated sandboxes that perform dynamic analysis by running the malware in a contained environment and monitoring its actions. When the automated sandboxes perform their analysis, many choose to observe the malware sample through custom implementations of monitoring methods, rather than using tools created specifically for that purpose. This thesis examines the viability of automating existing reverse engineering tools to uncover the mechanisms that malware samples employ to gain persistence on an infected machine. Persistence mechanisms are well defined, which greatly aids the process of measuring the effectiveness of reverse engineering tools.

Problem Statement

With the increasing amount of malware, analysts must turn to automated analysis solutions in order to cope with the sheer quantity. Many current solutions are either closed source or employ custom implementations of common techniques to perform analysis. While custom implementations can work, they have the potential to miss edge cases that malware employs because of the relatively young age of the implementation. Malware analysts have developed tools that were created alongside evolving malware. If such tools can be fit into the automated analysis process, proven techniques can be employed against malware in rapid fashion. This thesis examines the possibility of automating these existing tools within the scope of detecting malware persistence on machines post-infection.

Research Objectives

1. Determine if automated tools can be used to accurately reveal malware persistence indicators.
2. Identify the best combination of tools to uncover the largest range of mechanisms.
3. Examine the difficulty of automating various kinds of tools.
4. Expand on information gathered to make judgments on using automated tools within other areas of malware analysis.

Significance of Research

If existing tested tools can be efficiently and effectively incorporated into an automated analysis process, analysts could save time and energy by automating a tool to perform a task, rather than developing a custom implementation to perform the same job. Additionally, since the incorporated tool has been widely used in the past and developed alongside existing malware, analysts have the advantage of knowing the limitation of its abilities. Rather than double-checking the effectiveness of a new implementation on the wide range of malware capabilities, analysts can focus on further developing a tool or shoring up the tool's existing limitations.

Given enough automated tools, analysts would then have access to a collection of analysis options that can be chosen and deployed to fit specific situations. This study examines the difficulty of automating various kinds of tools and discusses the potential workload of the automation process when choosing between different tools. Furthermore, the determination of the best combination of tools to utilize when detecting persistence mechanisms will also help researchers determine which tool added to their toolbox will most expand its effectiveness.

Limitations and Scope

The tools' ability to detect malware persistence indicators is used as a tool quality metric. Malware analysis is a cumbersome, subjective, and error-prone process, with many variables that frequently differ between tools and malware types. Limiting the scope of the analysis to the subset of persistence indicators reduces the ability to evaluate other benefits and trade-offs of tools, but in return provides a concrete, objective, and consistently testable metric against which to evaluate many tools with varying feature sets.

The chosen tools and sample set of malwares used for this research is limited to those that run on the Windows operating systems. Both malware and tools must be designed and compiled differently based on the OS they are intended for. Potential persistence techniques also change according to OS. Confining the analysis to one operating system reduces the range of tools and malware that can be tested, but provides a consistent testing environment and persistence technique set that allows for an accurate comparison of tool quality.

This research studies malware that does not utilize anti-reversal techniques that prevent VM analysis. Some anti-reversal techniques can sense the presence of a VM and will abort malware execution to prevent information being gained. Defeating anti-reversal techniques is a separate aspect of malware analysis; avoiding these forms of malware reduces the variability of experiments and, therefore, the time required to obtain concrete reproducible results, allowing for a more thorough analysis of the tools themselves. Malware that employs anti-reversal techniques that target specific tools or information gathering techniques were allowed in the sample set, since these techniques can show the tool limitations.

Chapter 2 - Background and Related Work

What is Malware?

Malware is a term used to describe software that fulfills the deliberately harmful intent of an attacker [26]. Malware was first created to demonstrate technical skills or for the creator's personal entertainment. For example, Animal, written in 1975 for the UNIVAC 1108 system, would create a copy of itself in every directory a user had access to while they played a twenty-questions-style guessing game. It would spread to other computers when tapes, the data storage medium of the time, were shared between users. Unlike modern malware, Animal was carefully written to avoid damaging existing files on a computer [36].

In 1982, a ninth grader named Rick Skrenta wrote the Elk Cloner virus on an Apple-II computer. Elk Cloner would display one of Skrenta's poems on every 50th computer boot, which entertained his friends, while still leaving the computer otherwise unharmed [46].

Today, profits are a major driving force for malware creation, supporting large underground economies [56]. Franklin et al studied an active underground economy that offered multiple goods or services, including bank or PayPal credentials that had been stolen from systems infected with malware. The potential losses from credit card fraud and financial account theft was calculated to exceed $93 million [10].

Another example of lucrative malware is ransomware. Ransomware encrypts personal files on a computer or smartphone, then demands money from the victim to decrypt the files. According to the Kaspersky Labs 2014 Security Bulletin, victims of these ransomware attacks reported a demand of anywhere from $200 to $500 to decrypt the affected files [35].

Malware Types

There are different types of malware, each categorized by specific capabilities and intentions. Some of the more common types are viruses, worms, and Trojan Horses. A piece of malware, known as a malware sample, can belong to multiple different types, depending on the way it spreads or based on its actions on a system. Below is a brief description of the various malware types.

Trojan Horses

A Trojan Horse, or Trojan for short, is the most commonly seen type of malware [9]. Trojans masquerade as normal software to trick users into installing it. Once installed, Trojans will execute a payload that are often spywares designed to steal a user's personal and financial information, or a backdoor that lets attackers access the system remotely.

Spyware

Spyware is malware that is designed to monitor and harvest users' activity without their knowledge. They are commonly spread through software vulnerabilities or via Trojans. Spyware can monitor computer activity, and covertly take screenshots of the user's desktop [7]. Keyloggers are a common form of spyware that will collect keys pressed to steal account information and login data, such as usernames and passwords.

Rootkits

Rootkits are designed to conceal their presence from a user. Once installed, a rootkit will work deep in the background to tamper with operating system structures or system calls. Doing this allows a rootkit to eliminate traces of itself by hiding processes or files, allowing it to evade programs that might try to detect it. While hidden on a system, a rootkit commonly works as a backdoor for an attacker, giving the attacker remote access or letting other malware onto the

system. Many malware samples employ rootkit techniques to hide on a system they have infected [9].

Viruses

Viruses are designed to replicate and spread by hiding within normal computer programs or files. A program that a virus has infected is known as the virus' host. When the host is executed, the virus is also executed, usually attempting to infect as many other files as it can. It is important to note that because of this execution and spreading method, a virus cannot run independently and requires a host to function properly [19]. Viruses also require an outside actor such as a user to execute the host program. Viruses will commonly disable system defenses, steal sensitive data, and can even destroy a system under certain conditions.

Worms

A worm is like a virus in its design to self-replicate and spread, but a worm does not need a host to survive. Also unlike a virus, worms do not require human interaction to spread and replicate. While viruses tend to spread through files on a computer system, worms spread through network connections with the goal of infecting as many different systems as possible [25]. This is normally accomplished through email blasts or instant messages. Worms have been used to create botnets that can send spam mail en masse, or perform Distributed Denial of Service attacks (DDoS).

Ransomware

Ransomware is a form of malware that has grown in popularity recently. After infecting a system, ransomware will deny a user access to the system, demanding money in exchange for system access. Denying access can take the form of locking up the system or encrypting files on the system's data storage device to hold data hostage. Unfortunately, there is no guarantee that

paying the demanded funds will restore data or system access. Ransomware often spreads like a worm, arriving via a downloaded file or a system vulnerability [20].

Malware Families

As mentioned previously, a sample can exhibit multiple traits and so be classified as many different malware types. To better define them, malware is grouped into families based on the traits they exhibit and techniques they employ. While the categorization methods and naming conventions varies by Anti-Virus or research group, this study focuses on one categorization system for the sake of clarity [4].

MITRE's ATT&CK [25] knowledge model's names and groupings are used in this research to reference the samples tested and the families they belong to. When the ATT&CK database cannot provide information, the TrendMicro's Threat Encyclopedia [48] is used.

Persistence Techniques

After successful infection, many kinds of malware will attempt to gain a foothold on a victim machine, so they can survive computer reboots and continue to infect a system. A technique that gains persistence allows malware code to execute through regular system actions, or allows an adversary access to a system. When removing malware from an infected computer, if even a single persistence method is missed, the removal will fail and the malware will not be cleaned from the system. If a piece of malware is not fully removed, then it may re-infect the target machine and continue spreading.

There are several different persistence techniques identified that malware can employ. Many persistence techniques work by modifying the Windows Registry.

Registry Manipulation

The Windows Registry is a collection of configuration settings for Windows operating systems. The Registry stores information such as program settings, user preferences, and OS settings. The Registry is made up of multiple Registry *hives*, which act like top-level folders that categorize stored information. Inside each hive are Registry *keys*, which act like subfolders and contain other keys and Registry *values*. These Registry values are where the settings and specific instructions are stored [17].

Technique Categories

Based on their basic approach to persistence, the techniques researched in this thesis can be grouped into the following categories:

- User Login Execution
- System Startup Execution
- Dynamic Linked Library (DLL) Injection
- Execution Hijacking
- Adversary Backdoors

The individual techniques are covered in more detail in Appendix A.

User Login Execution

Windows has multiple programs that run automatically at log-in, without being intentionally started by a user. These programs are defined by various Registry keys called Autostart Extensibility Points (AESPs) [41]. When a user logs into a computer, these programs are triggered. If malware can add itself to one of the AESPs, it will be executed by the system during user log-in.

System Startup Execution

Malware using System Startup Execution achieve persistence by executing upon system startup. When a system starts, the OS is first initialized. After initialization, the OS runs a set of processes that perform necessary background system functions called services [21]. These services are also defined by AESPs in the Registry. To make a piece of malware execute during startup, these techniques directly insert the malware's code into the OS initialization process or modify an AESP and make the OS think the malware is an important service.

Dynamic Linked Library Injection

Dynamic Linked Libraries (DLLs) are shared libraries that contains functions and information used by other programs. To access the information in a DLL, a process must first load the DLL into its memory space. DLLs can optionally specify an entry-point function, which is called by the system whenever the DLL is loaded or unloaded. The entry-point function is usually used to perform creation or cleanup tasks [55].

Through DLL Injection, malware can force a process to load and execute an unintended DLL to gain system persistence. System settings can be changed to load a malicious DLL into certain important processes, or to load the DLL into every program that fits certain conditions. Malware can also trick the system into loading the wrong DLL into a process through careful placement and naming of malicious files in the file system.

Execution Hijacking

Malware can hijack the regular execution of a process in order to get itself executed. The most straightforward methods involve replacing an executable on the file system with itself. Every time the original program should be executed, the malware is executed instead. This can be dangerous because system instabilities can occur if the original application never launches.

Changing file references or default applications can also redirect execution from an intended program to malware. Once running, malware executes the hijacked program as a new process to hide evidence of the execution hijack. Often, it is difficult to notice the subtle execution redirection.

Adversary Backdoors

Certain techniques are more concerned with enabling outside access, rather than having constantly running malware. These techniques can grant an adversary remote access and control of a system when the correct conditions are met. A new account on the system with preconfigured credentials can lie dormant until an attacker decides to log in remotely. Subtler than a new account, certain executables that are rarely used can be overwritten so that pressing the right key combination at the login screen will grant system access without having to login.

Malware Analysis

To combat the various forms of malware, special researchers known as malware analysts are alerted. Malware encountered is almost always in a compiled binary format. The compilation process results in a loss of information from the original source code format, which makes analysis more complicated. The job of the analyst is to reverse engineer a malware binary and learn how it works. The analysist then determines methods to identify the malware on a system, and the steps required to safely remove it. To achieve those goals, researches must attempt to learn the goals of a malware sample, as well as how the malware acts on a computer.

Static Analysis

Static analysis involves analyzing malware without execution. Because the malware is never run, researchers do not have to fear infection by a sample. A cryptographic hash of the binary is a common method of program identification, and can be used to determine if other analysts have studied a particular sample previously. Examining the metadata of a binary can provide

information such as compilation time or imported and exported functions, which can often provide clues to its purpose. Specific text or words can appear in a binary as readable strings. Extracting these strings is also a common method of drawing conclusions about a sample. Many tools have been created to perform these actions and help analysts with static analysis.

Obfuscation techniques can be used on a binary to make static analysis less effective. Programs known as packers use various encryption or encoding algorithms to modify the original executable to scramble the extractable information. A decoding section is added to the binary, which is responsible for undoing this packing. When the packed binary is run, the decoding section first unpacks the original executable, then passes execution to the original program's start location. Advances in obfuscation techniques have revealed the limitations of static analysis and proven that other analysis forms are required to fully study malware [27].

Dynamic Analysis

Dynamic analysis is the solution to this problem. Dynamic analysis is the process of executing malware within a controlled environment to allow the study of its behavior. Since malware must unpack itself to execute, dynamic analysis evades the limitations faced by static analysis. There are two basic approaches to dynamic analysis: performing a system difference test between two points in time during malware execution and monitoring malware run-time behavior.

To examine the system differences created by malware, the sample is executed for a period of time and modifications are determined by comparing the current system state to the initial state. Researchers start with a clean machine. Then, they gather information about the file system and registry states. The malware is then executed and allowed to run, and the process is repeated to produce the final state for comparison. One issue with this method is that advanced malware can remove traces of its execution while it runs, so the final state lacks important information.

Monitoring malware behavior during run-time can solve this issue. Monitoring file changes can reveal executables created or changed by the malware. Changes to the Registry reveal edited keys and values that malware might use for persistence. Process monitoring through application programming interface (API) or system call hooks can trace all actions done by a process in a system, and reveal the internal structure of a sample. Network activity shows all communication the sample attempts to make, such as reaching out to command servers for instructions, or the exfiltration of stolen data [46].

To aide in dynamic analysis, tools have been created that help analysts monitor the complex internals of computer systems. These tools simplify the monitoring process so analysts can determine exactly what behaviors a malware sample exhibits.

Virtual Machines

To provide the execution environment, analysts commonly employ Virtual Machines (VM). A VM is run on a physical computer like a regular program but acts as "an efficient, isolated duplicate of a real computer machine" [30]. The physical computer is known as the host machine, while any VMs running on it are called guest machines. These guest machines create controllable environments that behave just like real machines [29]. Figure 1 provides a visualization of a VM running as an application on a host machine.

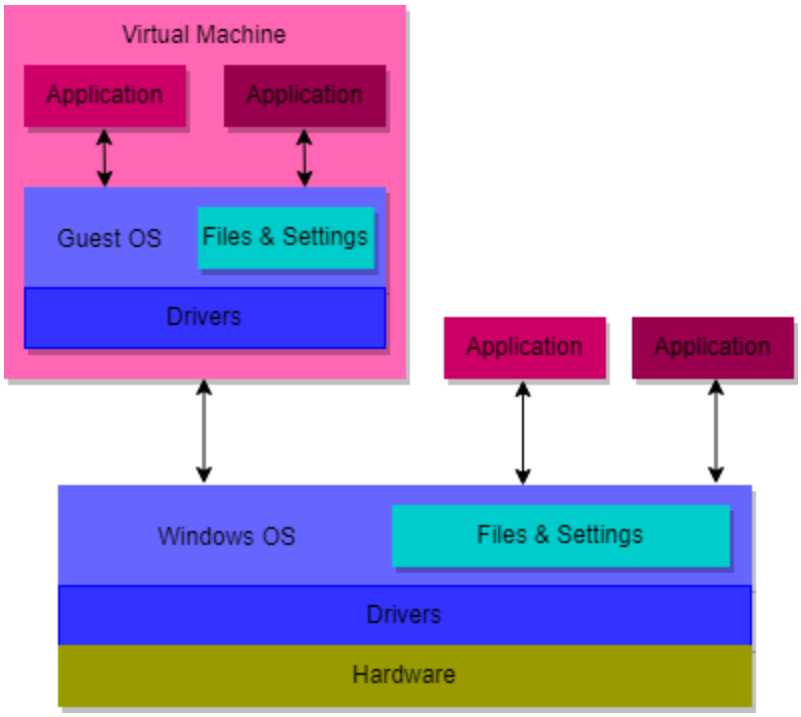

Figure 1: Virtual Machines simulate a real computer in a contained environment

Since VMs are isolated, any actions happening inside a guest machine will stay within the guest machine and not affect the host. VMs are also capable of taking and saving snapshots of their current state. At any time, a VM can revert to one of these snapshots, completely erasing any changes that happened since that snapshot was taken. These two qualities make for a perfect malware testing ground.

Analysts can run a sample on the guest machine, and study its actions without the risk of spreading the malware to other systems. When they are done, the analysts can revert the machine to a snapshot from before the malware was executed, and once again have a completely clean system.

Analysis Issues

Once a malware sample is identifiable and a process to remove it exists, the author of the malware will lose the income source or control that it provided. Because a piece of malware loses

its effectiveness once it is researched and identifiable, malware authors employ multiple methods to make analysis complicated and slow down the reverse engineering process. These methods are known as anti-reversal techniques.

In the past, malware reversal was performed manually by an analyst, which can be time consuming. Anti-reversal techniques can make manual analysis take even longer. This leads to excessive amounts of analysis required, often without enough time or manpower to complete the reversal. Additionally, the amount of malware has increased greatly in the last few years, creating an even bigger challenge for analysts attempting to combat malware.

Increase in Malware

The increase in profitability from producing malware has led to a sharp increase in new malware over the last few years. To gain an understanding of just how much new malware is being created and spread, researchers at the Norwegian University of Science and Technology conducted a test in 2009. For their test, multiple Windows 7 computers with different fully updated anti-virus software were exposed to multiple risky websites [49].

After two weeks of exposure to these sites, the computers were shut down for one month. This month-long period allowed anti-virus (AV) companies to process the latest malware threats and generate signatures and solutions. When the computers were turned back on, an updated virus scan revealed 124 newly found instances of malware on the systems, indicating that in those two weeks, the computer systems were hit with 124 newly created forms of malware the original AV software failed to detect.

There has been an explosion of new malware affecting computers since the 2009 study. According to AV-TEST, there were under 20 million new malware samples recorded during 2010. Figure 2 shows how since 2014, AV-TEST has reported over 120 million new malware samples

appearing every year [3]. With new attacks emerging at a staggering rate, analysists do not have the time to spend days manually reversing each new malware specimen. One solution is for the malware analysis process to become more automated.

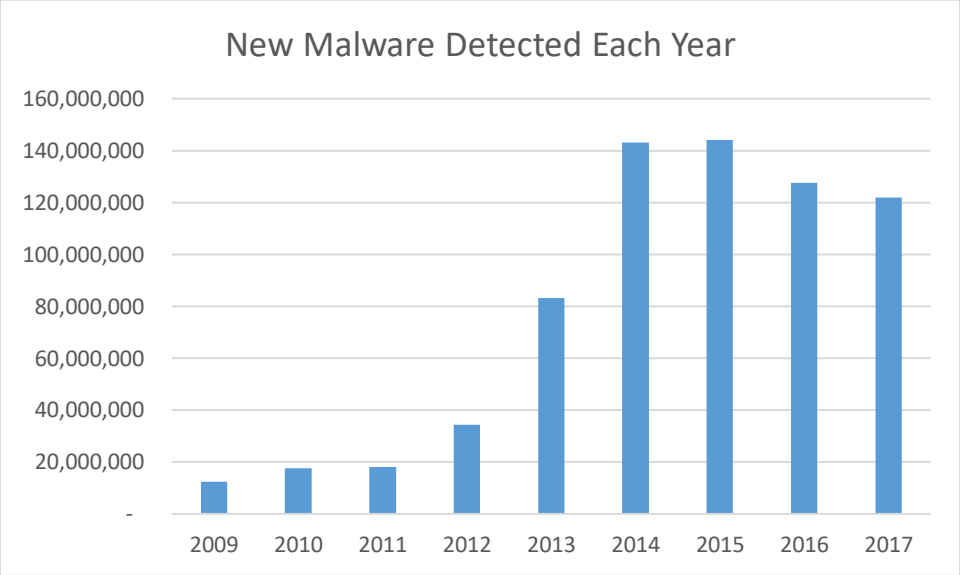

Figure 2: The amount of new malware discovered has sharply increased in the last 5 years

Automated Analysis

By automating time consuming portions or repetitive aspects of malware analysis, analysts can direct their focus towards more challenging or intensive portions of malware reversal and the entire process can be expedited. Sandboxes offer carefully designed environments where general malware analysis is performed. Other tools have been created which automate specific aspects of the analysis process. Existing tools have also been automated to perform a classification analysis of unknown binaries.

Sandboxes

A sandbox is similar to a VM in that it provides a restricted environment on a system that can securely run code. While in a sandbox, a program runs as though it has access to the entire system as normal, but cannot affect anything outside of the sandbox [42]. This is done by copying

the applicable parts of the system to the sandbox and discarding everything upon program completion. In this way, all system changes made in a sandbox only exist inside of the sandbox, and no changes are permanent. Figure 3 provides a visualization of an application running within a sandbox on a windows system.

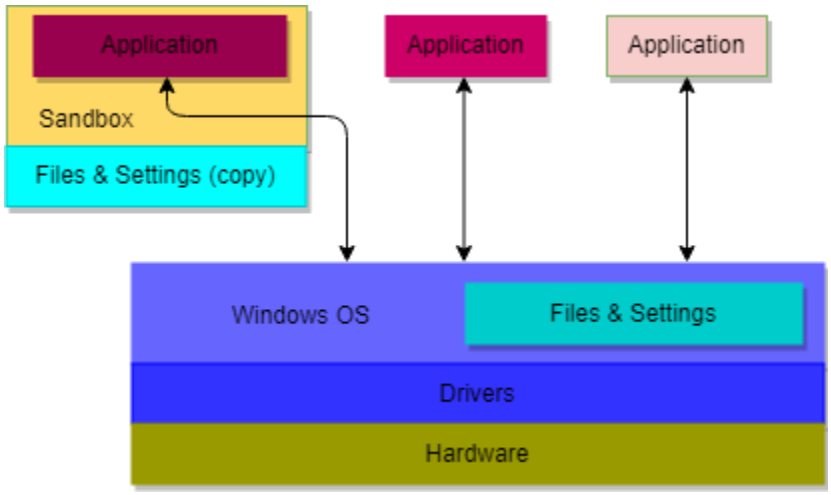

Figure 3: A sandbox provides an execution environment with an ephemeral system configuration

Sandboxes are not the same as VMs. They do not emulate an entire system, making them much less resource intensive than a VM. Because of their ephemeral nature, preserving changes or extracting files from a sandbox becomes a complex task [29].

Norman SandBox was one of the first analysis sandboxes. Its purpose was to monitor malware behavior and worms and viruses that spread through email or network shares. To create the analysis environment, creators of Norman SandBox reimplemented the core Windows system from the ground up [29].

CWSandbox hooks API and system functions to capture the behavior of malware that interacts with a system's file system or Registry [54]. To monitor these interactions, CWSandbox performs DLL Injection on all the processes that need to be monitored by API function hooks.

These hooks redirect certain API calls to CWSandbox, which analyzes the parameters of the call before continuing execution in the original API function.

Cuckoo is an open source sandbox that sports many customizable features, including API call tracing, network traffic capturing, and memory analysis [8]. A researcher can download Cuckoo and use the provided documentation to configure the exact tests desired.

Sandboxes generally use custom implementations of monitoring methods, instead of using tools that have been created and developed for the same purpose.

Automated Tasks

Rather than attempt a fully automated analysis, research has been done to automate certain key aspects of the analysis process. PolyUnpack employs both static and dynamic techniques to defeat packing methods, and extract hidden-code bodies from obfuscated malware [38].

TTAnalyze [5] and its successor Anubis [8] were designed to monitor Windows system and API calls. TTAnalyze and Anubis employed Qemu [6], an open-source PC emulator, to execute a sample in an emulated Windows XP environment and monitor its actions.

Argos [31] uses dynamic taint analysis and forensic code injection correlated with network traces in an x86 emulator to produce fingerprints for new types of system attacks.

Existing Tool Automation

Rather than reimplement existing techniques, some sources have done analysis work by automating one or more existing tools, and working with their generated output.

Tain, Islam and Batten [47] utilize the API call hooking tool HookMe [13] to record runtime behaviors of malware. They then used the Weka [23] collection of machine learning algorithms to differentiate malicious programs from non-malicious programs with over 97 percent accuracy.

CMB [33] is a locally deployable malware analysis framework built upon the open source sandboxes Cuckoo and Malheur [37]. It uses dynamic analysis functions of the provided sandboxes and clustering to compare new malware samples to existing malwares and create a comparison behavior report.

Anderson, Storlie, and Lane [2] combine six different data sources with machine learning to classify malware. Of these data sources, disassembled code is generated by IDA Pro [16], dynamic instruction traces are collected with the Intel Pin program [23], and packing information was collected with PEiD [1].

Chapter 3 - Tools Tested

This chapter provides information on the six tools chosen for research. A brief description is given of what each tool does, as well as the reason that a given tool was chosen.

FLOSS

The FireEye Labs Obfuscated String Solver (FLOSS) [12] is a static analysis tool designed by FireEye Labs to extract strings from a binary file. FLOSS is a self-contained executable that requires no installation. These extracted strings commonly contain DLLs imported and functions used, but they can also contain registry keys or file system locations. Because most malware is obfuscated in some way in its original form, FLOSS uses control flow analysis and heuristics to find and emulate decoder functions within a sample. This allows FLOSS to extract human readable strings from locations in a binary that would be otherwise hidden.

FLOSS uses the malware binary as input to produce a file of extracted strings. The extracted strings are scanned for persistence technique indicators to determine any methods employed by malware. Since the malware is never run, the results found by FLOSS cannot be completely trusted on their own. While an indicator string found in a binary is usually used for a persistence technique, it does not have to be. For example, scanning the file containing the list of indicators would flag all the techniques, although the file uses no such techniques.

FLOSS was chosen for testing because it has a chance of catching a wide range of techniques and provides insight into the automation of static analysis tools. FLOSS is unique because it performs static analysis on a malware sample, meaning the malware does not need to be executed for the analysis to occur. While static analysis is not always as effective as dynamic analysis, it is by far the safer method. Additionally, FLOSS can be controlled through the command line, which aids the automation process.

Autorunsc

Autorunsc [39] is a Windows Sysinternals tool created by Mark Russinovich. Autorunsc is self-contained, requiring no installation. It checks a large list of Autostart Extensibility Points (AESPs) in the windows system and Registry to show what programs will be run automatically upon system boot or user login. Autorunsc can record all the AESPs that launch a program and save them to a file. To check for malware persistence, a system difference test is performed with Autorunsc's findings.

Autorunsc is first executed on a clean system, to determine a baseline. After running malware, Autorunsc is run again to produce the second system state. These two states are compared using an external difference tool to reveal any new auto-start programs.

Autorunsc was chosen for testing because many persistence mechanisms rely on the locations that Autorunsc checks. Autorunsc provides good coverage of User Login and System Startup Execution methods. Autorunsc can be controlled through the command line, just like FLOSS.

Regshot

Regshot [45] is a self-contained, open-source tool designed for system difference testing. Like Autorunsc, Regshot records Registry and file system states in a series of snapshots. These snapshots are used to perform a system difference test which uncovers malware effects. Unlike Autorunsc, Regshot records the state of the entire Registry and file system, not just locations defined by a list of AESPs. While this creates more information that must be sorted through, it also means that no important change will be missed by an incomplete AESP list. The file system states can locate executables or DLLs that have been modified by malware. Regshot is also designed to perform the state comparison itself, without requiring an outside tool.

After starting, Regshot takes a snapshot of a clean system. After Regshot finishes taking the snapshot, malware is executed and allowed to make changes. Regshot then takes its second snapshot and compares them, producing a difference report. The difference report is scanned for persistence technique indicators to determine any methods employed by malware.

Regshot was chosen because it can record any AESPs missed by Autorunsc, and for its comprehensive file system checking. Regshot provides decent coverage of all persistence categories, but does so by returning a large amount of information. This in turn places the burden of technique identification on the exact configuration of the indicator scanner.

Regshot only works through a Graphical User Interface (GUI), and cannot be controlled via the command line. This makes the script driven interaction required during an automated run much more troublesome. To solve this, a GUI scripting python module called pywinauto was used to simulate tool interactions via a python script. This script acts as a human pressing buttons to control Regshot. While this method works, it makes the implementation process more intricate and time consuming.

Capture-BAT

Capture-BAT [18] is an open-source, run-time system monitoring tool developed by a group of researchers in New Zealand. Capture-BAT uses kernel callback functions to perform system and behavior monitoring. The kernel callbacks allow Capture-BAT to be notified of important system changes in real time, as well as Registry key addition or removal. It receives notifications about processes creation and destruction, or file reads and writes [43]. When a callback is triggered, Capture-BAT logs information about the event, including the triggering process, action performed, and location affected. Because these callbacks can trigger many notifications, Capture-BAT has a built-in filtering mechanism in the form of exclusion lists for

monitoring the Registry, file system, and processes. The exclusion lists are built using regular expressions and can be configured by researchers. Persistence techniques require malware to make changes to the Registry or file system, meaning that any mechanism employed should be caught by Capture-BAT. Kernel callbacks work at high privilege level in the system, so only malware that directly modifies the kernel can circumvent the monitoring process.

To monitor malware, Capture-BAT is started on a clean system, and told to redirect its output to a report file. It then monitors malware as it executes, filling the report file with captured information. The report is then scanned for persistence technique indicators to determine any methods employed by malware.

Capture-BAT was chosen because as a run-time monitoring tool, it provides different insights than state difference tools like Regshot do. Capture-BAT covers most mechanisms, and can be used against malware that removes evidence, unlike Regshot or Autorunsc. Capture-BAT can copy files that malware tries to delete into a separate folder, an important functionality in identifying malware processes. While that capability does not aid this research, it would be very helpful to other forms of malware analysis. To utilize Capture-BAT, it must be installed on the system prior to tests, making it less portable than other tools. Like some of the other tools being tested, Capture-BAT can be controlled through the command line.

Procmon

Process Monitor (Procmon) [40] is another Windows Sysinternals tool created by Mark Russinovich. Unlike Autorunsc, it is a monitoring tool that shows real-time file system, Registry and process activity [41]. Procmon captures system operations in detail, and offers a configurable filtering system that allows an analyst to quickly focus on specific operations. The exact method Procmon uses to capture system events is not shared by the author. Procmon also offers the option

to start system monitoring at boot time, which can help catch malware that executes on system start and employ rootkit evasion tactics.

To monitor malware, Procmon is started with a preconfigured filter on a clean system. It then monitors the system as the malware executes, generating filtered event information. Procmon then exports the event log to an XML file for further analysis. The XML log is simplified and summarized with a PowerShell script [14] into a final report. The report is then scanned for persistence technique indicators to determine any methods employed by malware.

While Procmon is similar to Capture-BAT, it is an entirely self-contained program and is equipped with a more powerful filtering interface, allowing for convenient information parsing. Procmon also covers most mechanisms, but provides more data about captured events than Capture-BAT. Procmon is designed to be used with other Sysinternal tools to create a more comprehensive understanding of system workings without much extra work and can be controlled through the command line.

Volatility

Volatility [53] is not like the others on this list. Volatility is an open source framework that gathers data through memory analysis. It is not run on the VM, instead running on the host where it can access the virtual memory file of the VM. This allows it to access the VM's non-volatile memory and Random-Access Memory (RAM), where the running instance of a malware sample lives during execution. Memory analysis usually focuses on transient information on a system, not on the non-volatile memory that persistence techniques must change if malware is to survive a system reboot. Many of the functions and capabilities of Volatility do not apply to this research. Because Volatility can access the non-volatile memory of the VM, it can be used to determine

some persistence techniques. Volatility can examine the Master Boot Record (MBR) and get information on system services or the file system.

To use Volatility, the VM is started and Volatility is used to examine and save the states of the MBR, configured services, and symbolic links in the file system. Once these are saved, the malware is executed. Volatility then reexamines the states of the components checked previously and saves them to produce the second set of states. An external difference tool compares the before and after state of each component, and saves them to a combined report. The report is then scanned for persistence technique indicators to determine any methods employed by the malware.

Volatility was chosen for testing because of its unique capabilities in memory analysis. No other tool in the list can examine the state of the MBR. Volatility's ability to directly read from structures in memory allow it to bypass rootkit techniques that might try to hide persistence mechanisms.

Tool Detection Capabilities

It is important to note that no tool listed here can detect all of the persistence techniques alone. This is by design, and directly supports the overall motivation of combining these tools to produce a wider range of persistence mechanism coverage (and, in general, allow for a more comprehensive malware analysis than a single tool used alone).

Table 1 lists each tool along with the persistence techniques it is able to detect when used as intended. This does not reflect how likely a tool is to detect these indicators. Tools that can identify indicators of techniques with confidence are marked with a **green checkmark**.

Table 1: Techniques the tested tools are capable of detecting

	Regshot	Capture-BAT	Procmon	Floss	Autorunsc	Volatility
Accessibility Features	✓	✓	✓	✓		
Account Creation	✓	✓	✓	✓		
AppCert DLL Injection	✓	✓	✓	✓	✓	
AppInit DLL Injection	✓	✓	✓	✓	✓	
Authentication Package Injection	✓	✓	✓	✓	✓	
Boot Sector Modification						✓
Browser Helper Objects	✓	✓	✓	✓	✓	
COM Hijacking	✓	✓	✓	✓		
DLL Search Order Hijacking	✓	✓	✓	✓	✓	
Executable Path Interception	✓	✓	✓	✓		
File Association Manipulation	✓	✓	✓	✓	✓	
File System Permission Weakness	✓	✓	✓			
IFEO Injection	✓	✓	✓	✓	✓	

Logon Script Creation	✓	✓	✓	✓	✓	
New Service	✓	✓	✓			✓
Port Monitor Manipulation	✓	✓	✓	✓	✓	
Run Key Injection	✓	✓	✓	✓	✓	
Screensaver Hijack	✓	✓	✓	✓		
Service Modification	✓	✓	✓	✓	✓	✓
Shortcut Modification		✓	✓			✓
Start Folder Injection	✓	✓	✓	✓	✓	
Winlogon Helper DLL Injection	✓	✓	✓	✓	✓	
Winsock Providers	✓	✓	✓	✓	✓	

Chapter 4 - Method

This section discusses the criteria for selecting the malware samples, as well as the environment chosen for testing. It outlines how the tests were set up to examine the malware samples with the chosen tools and covers how tool success was determined.

Malware Selection

To perform the tests, the MITRE ATT&CK database [25] was used to find malware families that employed the tested persistence techniques. When the ATT&CK database did not list any malware families for a technique, TrendMicro's Threat Encyclopedia [48] was used to find applicable malware. VirusShare.com [50], a repository of malware for research purposes, was used to gather samples identified as belonging to these families. For the tests, 33 samples were collected from VirusShare, based on the criteria below:

1. The malware family was not documented as using VM detection mechanisms
2. They belong to malware families that are known for persistence techniques
3. Together, their families employ a broad range of persistence techniques

First, malware with VM detection anti-reversal techniques were not used because they might prevent the malware from executing, rendering it useless without work outside the defined scope of this thesis. Second, a broad range of techniques allowed for a thorough testing of the capabilities of the tools chosen. Last, testing the tools against more persistence techniques also enabled a meticulous evaluation when determining which set of tools provides for the best coverage.

Environment

All tool and malware tests were performed in a VM running on VMWare Workstation 14 Pro, version 14.0.0 [52]. Performing the tests in a VM kept the malware contained and allowed for

state snapshots. The snapshots allowed an analyst to run malware and perform analysis, then reset the machine to its clean state, before the malware was executed.

The testing VM ran Windows XP SP3, and was updated to Security Update KB2892075. A Windows XP system made an ideal malware testing ground because it is a common OS for legacy systems, and most malware can run on it.

VMWare's vmrun tool [51] enabled the VM to be controlled through the command line. To use vmrun on a VM, it must have VMware Tools installed. The testing VM had VMware Tools version 9.9.2.2496486 installed. Scripting vmrun through Python [32] on the host allowed automation of the VM snapshot utilities. Python 2.7.10 (64-bit) was installed on the host machine.

Automation of some tools in the VM was controlled by Python scripts. Python 2.7.13150 was installed on the VM. To automate GUI interaction for Regshot, pywinauto [15] was installed on the VM, which required the Python packages "pyWin32" [43], "comtypes" [11], and "six" [33].

Capture-BAT had to be installed onto the guest system in order to run on the VM. It also required Microsoft Visual C++ 2005 Redistributable Package or later to be installed. The VM had Microsoft Visual C++ 2008 Redistributable Package 9.0.30723.6161 installed.

Test Overview

To determine the effectiveness of each tool, the tools were used in a series of tests to analyze multiple different samples. Each test involved one tool and one malware sample, and went through the following stages:

1. Pre-Run Setup
2. Execute Malware
3. Extract Data
4. Generate Report

Pre-Run Setup

When analyzing a piece of malware, it was crucial to perform work on a clean system. If a system was already infected with malware, analysis results may be incomplete or misattributed. Therefore, before each tool performed its analysis, the guest was restored to the state of a clean snapshot.

After a clean environment was assured, the tool and malware were prepared for execution. To prevent tools and malware samples from influencing each other during tests, they were stored on the host machine until they were needed for testing. The malware samples were stored within zipped files to prevent execution outside of the testing environment. Once a tool was run on a malware sample, both the tool and the zipped sample were copied to the VM. The malware was then extracted from the zip file, which made it executable again.

Finally, the tool performed any required analysis setup steps. Common steps included recording information about the VM's clean state, or listening to events or system calls. Once the tool had finished its setup, the next stage began.

Execute Malware

Once the tool and testing environment were ready, the malware sample was executed to initiate infection. During execution, the sample performed actions and made changes to the VM just as it would infect a real machine. During this stage, some of the tools monitored and recorded the malware's actions to gather information about the sample.

The malware was allowed to run for 3 minutes. This number was derived from the studies of Kreibich et al [22], who found that many samples required around 3 minutes to display their full capabilities. Since some forms of malware run in the background and gather data, this time limit was not enough to finish execution (if execution ever actually ends). While this may not be

enough time for full execution, it was usually sufficient time for a sample to employ its persistence techniques. After the three-minute period, the test moved to the third stage.

Extract Data

During this stage, the tools performed all final actions required to complete their analysis. For example, tools that took a snapshot of the clean system state compared it to the current state. Tools then compiled their findings and generated an output of the data they gathered. This output was then pulled from the VM back onto the host machine to avoid destruction when the VM is reset.

Generate Report

Before generating the final report from a tool, the data extracted from the VM had to be cleansed of irrelevant information. Because there was a lot of activity happening within a system, data that did not pertain to the malware sample might have been gathered by the tool. By cleaning the gathered data before generating the final report, it made it simpler to interpret the results. The data clean up occurred in three steps:

1. Remove Standard and Known Good Artifacts
2. Highlight Known Bad Keywords
3. Final Formatting

Remove Standard and Known Good Artifacts

Unhelpful information can easily fill a generated report since a system running normally generates exorbitant quantities of data. This clutter can hide the important information an analyst needs. To remove the unhelpful information, a baseline was created.

During this stage, each tool was run without an instance of malware, and its output was stored. This was then removed from the results obtained after the malware execution. By removing

the information present in the baseline, the analyst was able to focus on the effects generated specifically by the malware.

To clean up the information further, the analyst identified and removed information known to be unrelated to malware.

Highlight Known Bad Keywords

The analyst then scanned the Registry keys and system locations that are crucial to malware persistence and highlighted these in the report. With this, an analyst was able to quickly locate items of interest.

Final Formatting

After the data was cleaned, environmental information was added to the report, which correlated the tool and its findings to the sample analyzed. This final step was to aid the analyst if multiple files were being compared, or if future analysis work needed to be performed on the final reports.

Test Results

To determine a tool's capability to detect a persistence mechanism, its generated output was parsed for known persistence indicators using a regex scanner for string pattern matching. For each test, a tool's data was used to generate one of three conclusions for the existence of each persistence mechanism: *confident*, *possible*, and *not detected*.

A *confident* result indicated that an artifact relating to a specific persistence technique was found, and there was a high likelihood that the malware analyzed employed that technique. *Confident* results were generated when changes to specific Registry keys or file system locations were detected.

A *possible* result indicated that an artifact relating to one or more techniques was found, but there was not enough evidence to say with certainty. These results were commonly generated for Non-Registry manipulation techniques when an executable or file had changed in the system, but the exact persistence mechanism could not be determined.

A *not detected* result meant that for the given technique, there were no correlating artifacts present that indicated its existence. As most malware samples will only employ a few mechanisms at most, this was a common result.

Chapter 5 - Results

Tool Findings

The results of the tests were compiled to produce a list of the different techniques discovered by each tool, provided in the table below. Symbols are used in the table to make the results easier to read. A **green check-mark** indicates the *confident* result, a **yellow question mark** stands for the *possible* result, and an **empty** box indicates the *not detected* result.

An empty row indicates one of two possibilities, no tool could discover the corresponding technique, or the technique was not utilized by the malware in the sample set. Because the malware in the sample lacks an official analysis which would provide a ground truth for its behavior, undiscovered behavior and unutilized behavior cannot be differentiated.

Table 2: Persistence indicators found by the tested tools. If the tool detects the technique in any of the samples, it is marked with a checkmark in this table.

	Regshot	Capture-BAT	Procmon	Floss	Autorunsc	Volatility
Accessibility Features						
Account Creation	✓	✓	✓			
AppCert DLL Injection						
AppInit DLL Injection						
Authentication Package Injection						
Boot Sector Modification						✓

Browser Helper Objects	✓	✓	✓		✓	
COM Hijacking	✓	✓	✓			
DLL Search Order Hijacking	?	?	?	?		
Executable Path Interception						
File Association Manipulation	✓	✓	✓			
File System Permission Weakness	✓	✓	✓			
IFEO Injection						
Logon Script Creation	✓	✓	✓		✓	
New Service						✓
Port Monitor Manipulation						
Run Key Injection	✓	✓	✓	?	✓	
Screensaver Hijack						
Service Modification	✓	✓	✓		?	
Shortcut Modification	?	?	?		?	✓
Start Folder Injection	✓	✓	✓	✓	✓	
Winlogon Helper DLL Injection	✓	✓	✓	✓	✓	
Winsock Providers	✓		✓		✓	

Coverage Set

Regshot, Procmon, and Capture-BAT displayed a similar performance, however, Capture-BAT could not detect Winsock Providers, making Regshot or Procmon the best tool choice. Volatility provided the next most diverse technique coverage, detecting two techniques not discovered by the other tested tools, and gave confident feedback for one technique the other tools could not confirm. Procmon or Regshot combined with Volatility provides the best coverage with the least number of tools.

Redundant Tools

After analyzing the individual tests, we can conclude that some tools are rendered redundant – they can be removed without decreasing the level of coverage. An example of redundant tools can be seen in Table 3, which shows the detected techniques for a sample from the WhiskyAlfa malware family. For this sample, Procmon rendered the results of all tools except for Volatility redundant.

Table 3: Techniques detected for sample from WhiskyAlfa family

	Regshot	Capture-BAT	Procmon	Floss	Autorunsc	Volatility
File System Permission Weakness		✓	✓			
New Service						✓
Run Key Injection	✓	✓	✓		✓	
Shortcut Modification						✓
Winlogon Helper DLL Injection			✓			

When consulting the results from all 33 samples, it was found that Procmon consistently detected the same, if not more, results than both Floss and Autorunsc, rendering them redundant. Capture-BAT and Regshot both provide unique detections in different tests, but only one tool is required when combined with Procmon. Regshot is not needed as long as Capture-BAT and Procmon are used together, and Capture-BAT is not needed if Regshot and Procmon are used together. Capture-BAT provides extra detections more often than Regshot does, making Capture-BAT the preferred tool of the two. A true coverage set of techniques displayed by each sample only requires Procmon, Capture-BAT, and Volatility.

Implementation Difficulties

There were many challenges during the process of automating the testing tools. The lack of documentation for one tool impacted the time and effort required to understand its capabilities. Many of the tools required the use of different methods to start execution. Output from tools had to be gathered in various ways as well because of how the tools were designed. After output was gathered across the range of tools, searching for indicators using only one method was impossible because of varying formats and the different implications of output.

Existing Documentation

Documentation can save an analyst the time of discovering tool capabilities through trial and error. For this research, different forms of documentation provided varying amounts of assistance for the automation process.

The preferred documentation was in the form of a book or a research paper. This documentation was very robust, and was the most helpful. Autorunsc and Procmon were documented in a book that provided their configuration options, full abilities, and usage examples.

Capture-BAT was accompanied by a research paper detailing its capabilities, uses, and how it worked on a system.

Open-source was the next preferred form of documentation. Volatility and FLOSS had documentation on their GitHub pages, which explained the command line options and explained how the tools worked.

The least preferred form of documentation were text files that accompanied the tool. Regshot only had a README.txt file with limited information. This text file provided little insight into how the tool worked, and the most basic instructions to run it. Regshot's ability to read registry keys were not possible without diving into source code, and the only information concerning command line interfaces came from comments from users who wondered why there wasn't one. Regshot was severely limited for this reason.

Execution Differences

Across the six different tools, four different methods were required to execute them and prepare them for testing - using vmrun, starting a process with python on the host, starting a process with python on the guest, and starting a process with a pywinauto python script on the guest.

Vmrun was the preferred method to begin program execution. Vmrun has the ability to accept an executable and a simple argument list, which then runs inside the VM. Of the six tools, Capture-BAT and Procmon were able to use this method. Vmrun is the most basic option for scripting execution in a VM. All other methods of invocation within the VM were built using its functionality.

Vmrun could not be used to run Volatility, because Volatility ran on the host machine. Python was used to run Volatility as a sub process of the automation framework.

Some of the tools required a more complicated invocation using output redirection to be used properly. Vmrun is unable to handle these command characters, resulting in improper execution on the VM. To solve this, the tool execution and file redirection were implemented via a python sub process as well. Because execution needed to occur on the VM, the python script was saved into a separate file which was loaded onto the VM with the malware and the tool. Finally, the python script was executed with vmrun. FLOSS and Autorunsc required this method of initiating execution.

The final tool, Regshot, was capable of being launched with vmrun, but required additional setup requiring GUI interaction. The additional setup required pywinauto to interact with the active window, so loading a python script and starting Regshot with pywinauto became the best solution. After starting the process, pywinauto could finish preparing Regshot for testing.

Output Generation Differences

Because not all the tools were created with each other or automation for a framework in mind, the tools' methods of delivering results also varied. Specifying where to output information and in what format from all the tools required different four methods. The preferred output was strings in a text file, with a configured filename to make extracting the file from the VM straightforward and uniform.

The most uncomplicated tool took output options as command line arguments upon execution. Capture-BAT was the only tool with this capability, allowing users to specify where tool output should be saved.

Procmon could also accept an output file as a command line argument, but not when first running. When Procmon was first executed, it took an argument of where it should log its recorded events. There was no filtering of this log, and it was only readable by Procmon. The original

process had to be stopped, and a second process run with the explicit purpose of reading and filtering the stored log to convert it into a usable format. Procmon supports log conversion into PML, XML, or CSV formats, but to achieve the desired data format, the CSV format was chosen and converted into a text file after extraction from the VM.

Many tools assumed a human operator, and sent the output to stdout, where it would be displayed on the original interactive terminal. FLOSS, Autorunsc, and Volatility required their output be redirected to a file so additional analysis could occur.

Regshot required a user to interact with its GUI to specify where its results should be saved. This was accomplished in the setup stage of Regshot with pywinauto, but also required more interaction than most of the other tools.

Output Format Differences

Once output was generated and extracted from the VM, it had to be interpreted to determine persistence mechanisms present. This was accomplished with a regex scan of the output to locate key strings for most of the tools. The different organization styles of extracted information made the scanning step more complicated.

While FLOSS provided a string dump, Regshot output a list of registry and file locations organized under sections defined by the action done on that location. The section that a scanned string is read from is important because registry key creation in a location might indicate a persistence mechanism, while key deletion might mean normal system activity.

```
----------------------------------
Keys added: 13
----------------------------------
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\InprocServer32
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\ProgID
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\TypeLib
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\VersionIndependentProgID
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\1.0
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\1.0\0
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\1.0\0\win32
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\1.0\FLAGS
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\1.0\HELPDIR
HKU\S-1-5-21-329068152-2147245803-1177238915-500\Software\ASProtect
HKU\S-1-5-21-329068152-2147245803-1177238915-500\Software\ASProtect\SpecData

----------------------------------
Values added: 14
----------------------------------
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\: "Jeracapa Decage class"
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\InprocServer32\: "C:\Program Files\Common
Files\Microsoft Shared\Speech\sapi.dll"
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\ProgID\: "SAPI.SpObjectTokenEnum.1"
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\TypeLib\:
"{5242F540-17BD-BC4E-AE10-E2A663FE68F7}"
HKLM\SOFTWARE\Classes\CLSID\{90F88B48-B814-48D1-5C85-5E33E9987597}\VersionIndependentProgID\:
"SAPI.SpObjectTokenEnum"
HKLM\SOFTWARE\Classes\TypeLib\{5242F540-17BD-BC4E-AE10-E2A663FE68F7}\: (NULL!)
```

Figure 4: Regshot organizes its information under headers

Once Procmon's CSV output was converted to regular text, information was laid out in a manner similar to Capture-BAT. Both tools provide an action and the target of the action in a horizontal form, but in different orders.

```
Procmon:
"lsass.exe","RegSetValue","HKLM\SAM\SAM\Domains\Account\Users\000001F4\F"

CaptureBAT:
"registry","SetValueKey","C:\WINDOWS\system32\lsass.exe","HKLM\SAM\SAM\Domains\Account\Users\000001F4\F"
```

Figure 5: Procmon and Capture-BAT represent the same registry change event in different layouts

Both Volatility's and Autorunsc's output had to be compared after extraction to create a state difference. This led to either a few registry or file locations as output, or no output if there was no state change. Because of this, any output was considered significant. While standard string scanning still worked for Autorunsc, Volatility required a separate check by the analyst.

Tool Evaluation

Table 4 provides a final look at the tools tested. The type of analysis provided by the tool is shown, along with the number of indicators types found throughout the study. The indicators are compared to the number of indicators found by any tool. Finally, the major implementation difficulties associated with automating each tool are given.

Table 4: Final tool evaluation

Tool Name	Analysis Type	Indicators	Implementation Difficulties
FLOSS	Static Strings Dump	4/15	Required Output Redirect
Autorunsc	Dynamic State Compare	9/15	Required Output Redirect Manual State Comparison
Regshot	Dynamic State Compare	13/15	GUI Only Sparse Documentation
Capture-BAT	Dynamic Behavior Monitor	12/15	Required Installation Required Installed Packages
Procmon	Dynamic Behavior Monitor	13/15	Limited Output Formats
Volatility	Memory Analysis State Compare	3/15	Required Output Redirect Manual State Comparison

Analysis of Results

The implementation difficulties analyzed were derived from the author's own experiences, and are thus subjective results. However, this subjectivity does not change the inherent characteristics of various tools, only the degree that they might impact different individuals.

Tool Characteristics to Seek

The tools with the smallest workload to automate had four specific characteristics - they were controllable through the command line, had many configurable options, were well documented, and were self-contained programs.

Command Line Interface

First, tools that were controllable through the command line were the simplest to work and interact with as an automated process. These tools could be run and configured in a straightforward manner, and did not require intermediate steps such as execution from a python script.

Configuration Options

Second, tools with many configurable options were more flexible than the others. Options to look for include output location, output format, or filtering options. Options that configure the nature of the task performed by the tool are also helpful, and provide the potential for that tool to be used for other application with a minimal amount of code revisions.

Thorough Documentation

Tools that had a command line interface with many options were more flexible and required less work to automate, but only when the tool came with good documentation. Documentation can save an analyst the time of discovering tool capabilities through trial and error. Good documentation will also help an analyst determine if a tool will provide coverage for limitations in their existing toolset.

Self-Contained

The final trait of a straightforward to automate tool was one designed as a self-contained program, as they can simply be placed into the VM and executed. This characteristic is helpful and simplifies the automation process. While this is a preferred characteristic for a tool, an analyst can still automate tools that have requirements concerning development packs on the system or installation before running. Tools with outside requirements simply require more work, such as changing the base configuration of execution environment.

Tool Characteristics to Avoid

Analysts looking for a tool to automate should do their best to avoid those that only use a graphical user interface or do not have much documentation. These two characteristics can make the automation process require much more work.

Graphical User Interface

The GUI tool, Regshot, required the most additional implementation steps and the largest amount of debugging. Automating Regshot required trial and error when scripting to see how the it interacted with the system and the malware. Automation scripts for GUI tools needed to be customized for each tool. This increased the time and effort required from an analyst.

Sparse Documentation

Tools without comprehensive or thorough documentation required extra time and energy to automate. The ways a tool worked must be manually explored by an analyst before automation can occur. Sparse documentation also made it a demanding task for the analyst to identify how a tool would work in their existing tool-set. Even if a prospective tool seems powerful or helpful, caution should be exercised if there is not adequate documentation to perform the automation process.

Automated Tools in Multiple Malware Analysis Areas

Tools that had been automated for one area of malware analysis could be used for other analysis applications as well. The framework created for the initial automation saved time and effort for the analyst looking to repurpose a tool. The tool itself determined the amount of functionality lost during the initial automation process, increasing the revisions required to repurpose that tool.

Existing Tool Framework

To automate a tool for an analysis task, the tool had to be understood, and methods to control the tool and extract its output had to be implemented. Much of this work, such as understanding the tool and extracting impute, only needed to occur once. To use a tool with different configurations, the methods to control it needed to be changed, but not reimplemented. This existing groundwork could save an analyst time and effort when using a tool for a different application within malware analysis.

Loss of Functionality

For some tools however, the automation process required preset configurations or workflows that limited the functionality of the tool. An example of this was preset filters for tools like Procmon or Capture-BAT. These filters drastically reduced the information that had to be handled by secondary analysis by omitting data considered extraneous to the task, but doing so required a pre-configured filter file that must be copied to the VM. Changing the filter to focus on different aspects of malware behavior required a new filter manually configured by an analyst. Procmon could dynamically change its filtering options, but only through its GUI.

Scripting the use of a tool's GUI required a defined workflow for the tool to follow. Establishing this workflow also limited the functionality of a tool, when compared to its

capabilities when an analyst was using it interactively. Using a GUI tool for a different analysis application would require an analyst to create a new scripted workflow.

Conclusion

Automating a single tool to perform an analysis task is faster and requires less work than creating a custom implementation. The tools that require the least amount of work to automate have few execution requirements, can be controlled through a command line interface with a host of configurable options, and are well documented. These characteristics provide flexibility and reduce time required for automation. However, issues arise when automating multiple tools into an analysis framework.

Most existing tools have been created by separate groups and were not designed to be used together. They have different interactive interfaces, requiring the analyst to implement multiple methods of tool control. Furthermore, tools from multiple designers can generate output in different formats. This puts further strain on the analyst who must come up with ways to interpret the results.

Finally, many of these tools are designed to work interactively with an analyst, and through the automation process, capabilities are lost. Changing tool configurations without a GUI can require premade configuration files, or even changes to execution environment. Because of this, reusing a tool for multiple analysis approaches would necessitate implementation overhauls.

While an analysis framework can be created from existing tools, its implementation workload depends on key characteristics of the tools used. Even with tools that facilitate the automation process, using tools for multiple applications within malware analysis can require multiple manual configuration changes.

Bibliography

[1] Aldied. "PEiD." Aldied.com. https://www.aldeid.com/wiki/PEiD (accessed on December 8, 2017)

[2] Anderson, B., Storlie, C., and Lane, T. "Improving Malware Classification: Bridging the Static/Dynamic Gap." *AISec '12*. Raleigh, North Carolina: ACM, 2012.

[3] AV-Test. "Statistics: Malware." AV-Test: The Independent IT Security Institute. https://www.av-test.org/en/statistics/malware/ (accessed on December 6, 2017).

[4] Bailey, M., Oberheide, J., Anderson, J., Mao, Z. M., Jahanian, F., and Nazario, J. "Automated Classification and Analysis of Internet Malware." *RAID 10th International Symposium Proceedings*. Gold Coast, Australia: Springer-Verlag Berlin Heidelberg, 2007.

[5] Bayer, U., Moser, A., Kreugel, C., Kirda, E. "Dynamic Analysis of Malicious Code." *Journal of Computer Virology and Hacking Techniques*. France: Springer Verlag, 2006.

[6] Bellard, F. "QEMU, a Fast and Portable Dynamic Translator." *2005 USENIX Annual Technical Conference*. Berkeley, California: USINEX Association, 2005.

[7] Brunner, M. *Integrated Honeypot Based Malware Collection and Analysis*. Garching, Germany: Fraunhofer Research Institution for Applied and Integrated Security, 2012.

[8] Cuckoo Sandbox. "What is Cuckoo?" Cuckoo Sandbox: Automated Malware Analysis. https://cuckoosandbox.org/ (accessed February 1, 2018).

[9] Egele, M., Scholte, T., Kirda, E., and Kreugel, C. "A Survey on Automated Dynamic Malware Analysis Techniques and Tools." *ACM Computing Surveys* Vol. 44, No. 2 (2012). https://seclab.ccs.neu.edu/static/publications/acm2012survey.pdf.

[10] Frankin, J., Paxson, V., Perrig, A., and Savage, S. "An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants." *CCS '07*. Alexandria, Virginia: Center for Computational Sciences, 2007.

[11] GitHub. "Comtypes." GitHub.com. https://github.com/enthought/comtypes/releases (accessed on January 15, 2018).

[12] GitHub. "FLOSS: FireEye Labs Obfuscated String Solver." GitHub.com. https://github.com/fireeye/flare-floss (accessed on December 8, 2017).

[13] GitHub. "Hook Me." GitHub.com. https://github.com/NytroRST/HookMe (accessed on January 13, 2018).

[14] GitHub. "Process Monitor Analyze Software." GitHub.com https://github.com/MotiBa/ProcessMonitorAnalyzeMalware (accessed on December 8, 2018).

[15] GitHub. "Pywinauto." GitHub.com. https://github.com/pywinauto/pywinauto (accessed on January 15, 2018).

[16] Hex-Rays. "IDA: About." Hex-Rays.com. https://www.hex-rays.com/products/ida/index.shtml (accessed on December 8, 2017).

[17] Honeycutt, J. *Microsoft Windows Registry Guide 2nd Edition.* Redmond, Washington: Microsoft Press, 2005.

[18] Honeynet. "Capture-BAT Download Page." Honeynet.org. https://www.honeynet.org/node/315 (accessed on January 10, 2018).

[19] Idika, N., Mathur, A. *A Survey of Malware Detection Techniques*. West Lafayette, Indiana: Purdue University Press, 2007.

[20] Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., and Kirda, E. *Cutting the Gordian Knot: A Look into the Hood of Ransomware Attacks.* Milan, Italy (2015). https://wkr.io/publications/dimva2015ransomware.pdf.

[21] Kozierok, C. M. "Master Boot Record (MBR)." The PC Guide. http://www.pcguide.com/ref/hdd/file/structMBR-c.html (accessed on January 10, 2018).

[22] Kreibich, C., Weaver, N., Kanich, C., Cui, W., and Paxson, V. "GQ: Practical Containment for Measuring Modern Malware Systems." *IMC '11.* Berlin, Germany: ACM, 2011.

[23] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumenation." *PLDI '05.* Chicago, Illinois: ACM, 2005.

[24] Machine Learning Group at University of Waikato. "Weka 3: Data Mining Software in Java." Department of Computer Science at University of Waikato. https://www.cs.waikato.ac.nz/ml/weka/ (accessed on January 10, 2018).

[25] MITRE Corporation. "MITRE's Adversarial Tactics, Techniques, and Common Knowledge." https://attack.mitre.org/wiki/Main_Page (accessed January 15, 2018)

[26] Moore, D., Shannon, C., Voelker, G., and Savage, S. *Internet Quarantine: Requirements for Containing Self-Propagating Code.* San Diego, California: University of California Press, 2003

[27] Moser, A., Kruegel, C., and Kirda, E. "Exploring Multiple Execution Paths for Malware Analysis." *IEEE Symposium on Security and Privacy.* Vienna, Austria: Technical University Vienna, 2007.

[28] Moser, A., Kruegel, C., and Kirda, E. "Limits of Statistical Analysis for Malware Analysis." *Annual Computer Security Applications Conference*. Miami Beach, Florida: Technical University Vienna, 2007.

[29] "Norman Sandbox Analyzer." Norman Sandbox: Your Proactive IT Security Tool. http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf (accessed on February 5, 2018).

[30] Notenboom, L. "What's the Difference Between a Sandbox and a Virtual Machine?" Ask Leo. http://ask-leo.com/whats_the_difference_between_a_sandbox_and_a_virtual_machine.html (accessed on February 2, 2018).

[31] Popek, G. J. and Goldberg, R. P. "Formal Requirements for Virtualizable Third Generation Architectures." *Communications of the ACM Vol 17, No. 7.* New York City, New York: Association for Computer Machinery, Inc. 1974.

[32] Portokalidis, G., Slowinska, A., Bos, H. "Argos: An Emulator for Fingerprinting Zero-Day Attacks." *EuroSys '06.* Leuven, Belgium: ACM, 2006.

[31] Python Software Foundation. "Python." Python.org. https://www.python.org/ (accessed on January 15, 2018).

[33] Python Software Foundation. "Python: Six 1.11.0." Python.org. https://pypi.python.org/pypi/six (accessed on January 15, 2018).

[34] Qiao, Y., Yang, Y., He, J., Tang, C., and Liu, X. "CMB: Free, Automatic Malware Analysis Framework Using API Call Sequences." *Research Gate March 2014*. Berlin, Germany: Research Gate, 2014.

[35] Raiu, C. *Kaspersky Security Bulletin 2014*. Moscow, Russia: Kaspersky Lab, 2014.

[36] Rajesh, B., Janhardhan Reddy, Y. R., and Dillip Kumar Reddy, B. "A Survey Paper on Malicious Computer Worms." *IJARCST* Vol. 3, Issue 2 (April-June 2015). http://www.ijarcst.com/doc/vol3issue2/ver2/brajesh.pdf.

[37] Rieck, K., Trinius, P., Willems, C., and Holz, T. "Automatic Analysis of Malware Behavior using Machine Learning." *Journal of Computer Security 2011*. Amsterdam, Netherlands: IOS Press, 2011.

[38] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. *PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware.* Atlanta, Georgia: Georgia Institute of Technology.

[39] Russinovich, M. "Autoruns for Windows v13.82." Microsoft.com. https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns (accessed on February 23, 2018).

[40] Russinovich, M. "Process Monitor v3.50." Microsoft.com. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon (accessed February 22, 2018).

[41] Russinovich, M. and Margosis, A. *Troubleshooting with the Windows Sysinternals Tools*. Redmond, Washington: Microsoft Press, 2016.

[42] The Sandbox. "Understanding the Sandbox Concept of Malware Identification." The Sandbox: Understanding CyberForensics. http://cwsandbox.org/understand-the-sandbox-concept-of-malware-identification/ (accessed on January 10, 2018).

[43] Seifert, C., Steenson, R., Welch, I., Komisarczuk, P., and Endicott-Popovsky B. "Capture: A Tool for Behavioral Analysis of Applications and Documents." *The Digital Forensic Research Conference*. Pittsburg, Pennsylvania, 2007.

[44] SourceForge. "Python for Windows Extensions." SourceForge.net. https://sourceforge.net/projects/pywin32/files/pywin32/Build%20220/ (accessed on January 15, 2018).

[45] SourceForge. "Regshot." SourceForge.net. https://sourceforge.net/projects/regshot (accessed on January 15, 2018).

[46] Szor, P. *The Art of Computer Virus Research and Defense.* Boston, Massachusetts: Addison Wesley Professional, 2005.

[47] Tian, R., Islam, R., Batten, L., and Versteeg, S. "Differentiating Malware from Cleanware Using Behavioral Analysis." *Malware 2010: Proceedings of the 5th International Conference on Malicious and Unwanted Software.* Piscataway, New Jersey: IEEE, 2010.

[48] Trend Micro Business. "Threat Encyclopedia." https://www.trendmicro.com/vinfo/us/threat-encyclopedia/ (accessed January 15, 2018).

[49] Vegge, H., Halvorsen, F. M., Nergard, R. W. "Where Only Fools Dare to Tread: An Empirical Study on the Prevalence of Zero-Day Malware." *Proceedings of Fourth International Conference on Internet Monitoring and Protection*. Venice, Italy: IEEE, 2009.

[50] VirusShare. VirusShare.com. https://virusshare.com/ (accessed on February 6, 2018).

[51] VMWare. *Using vmrun to Control Virtual Machines.* Palo Alto, California: VMWare, Inc, 2009.

[52] VMWare. "Workstation Player." VMWare.com. https://www.vmware.com/products/workstation-player.html (accessed on January 15, 2018).

[53] Volatility Foundation. "Volatility Foundation." VolatilityFoundation.org. http://www.volatilityfoundation.org/ (accessed on January 7, 2018).

[54] Willems, C., Holz, T., and Freiling, F. "Toward Automated Dynamic Malware Analysis Using CWSandbox." *IEEE Security and Privacy*. Washington, D.C.: IEEE Computer Society, 2011.

[55] Yosifovich, P., Ionescu, A., Russinovich, M. E., and Solomon, D. A. *Windows Internals Part 1 7th Edition*. Redmond: Washington: Microsoft Press, 2017.

[56] Zhuge, J., Holz, T., Song, C., Guo, J., Han, X., and Zou, W. *Studying Malicious Websites and the Underground Economy on the Chinese Web.* Riverside, California: University of California Riverside, 2008.

Appendix A - Malware Persistence Mechanisms

This appendix lists the various persistence mechanisms that the tested malware could utilize. A brief description of each mechanism gives an overview of how the method works, as well as what capability they provide to malware that employs it.

User Login Execution

Run Keys Injection

Executables pointed to by the Registry key *HKLM\Software\Microsoft\Windows\CurrentVersion\Run* will be executed with the account's permissions level. Adding a value that references a malicious program will cause it to execute when a user logs in, and can be used to gain persistence.

Start Folder Injection

Executables placed in the startup folders *C:\Users\<username>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup*, or *C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup* will be executed with the account's permissions level. Placing a malicious executable in one of these locations will cause it to execute when a user logs in, and can be used to gain persistence.

Logon Script Creation

Windows allows scripts to be run whenever a user or group of users log into a system. These scripts can be accessed with Group Policy Editor. Malware can edit these scripts to cause the OS to re-run the malware whenever a user logs in, which will give it persistence on a system.

System Startup Execution

Authentication Package

The Local Security Authority (LSA) loads Windows Authentication Package DLLs when a system starts up to provide support for login and security processes used by the OS. A binary specified by the *Authentication Packages* value in the Registry key *HKLM\SYSTEM\CurrentControlSet\Control\Lsa* will be executed at system start by the Local Security Authority. Malware can edit this value to cause a malicious binary to execute to gain persistence on a system.

Boot Sector Modification

The Master Boot Record (MBR) stores information about the hard disk, and is the first thing executed after the BIOS initializes. Control then passes to the Volume Boot Record (VBR) which loads the operating system into the active disk partition. Malware can modify and control the execution of the MBR and VBR so it executes on system startup.

Existing Service

When the OS boots up, it initiates programs called services, which perform background system functions. Windows service configuration information is stored in the Registry at *HKLM\SYSTEM\CurrentControlSet\Services*. Changing the binary path of a service to a malicious executable or enabling previously disabled services may both be ways that malware tries to gain persistence on a system.

New Service Creation

Like modifying an existing service, malware can create a new service to gain persistence on a machine. The new service is created by adding new entries to the Registry key *HKLM\Software\Microsoft\Windows NT\CurrentVersion\Run\Services*, and can be configured to execute at startup or interact with other existing services.

WinSock Providers

Windows Sockets is an extensible API on windows so third parties can add custom layers on top of existing networking protocols. Applications pointed to by values in the *HKLM\System\CurrentControlSet\ServicesWinSock2\Parameters\NameSpace* key are started as services upon system boot.

Dynamic Linked Library Injection

AppCert DLLs

DLLs specified by the AppCertDLLs value in the Registry key *HKLM\System\CurrentControlSet\Control\Session Manager* are loaded into each process that calls certain commonly used CreateProcess API functions. Malware can edit this list to cause malicious DLLs to be loaded and run in the context of any process that calls those API functions.

AppInit DLLs

DLLs specified by the AppInit_DLLs value in the Registry key *HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows* are loaded into every process that loads user32.dll. Since user32.dll is a very common library, this ends up being almost every process. Malware can edit this list to cause malicious DLLs to be loaded and run in the context of each of these processes.

DLL Search Order Hijacking

When loading a DLL into a process, the OS first locates the DLL by searching for it in various locations. The locations checked follows a specific order, and the OS will load the first instance of the DLL that it finds. If a DLL is in the third place that the OS would check, but malware places a malicious DLL with the same name in the second location that the OS would check, then the malicious DLL will be found first and loaded instead of the legitimate DLL.

Port Monitor Manipulation

Port monitors are DLLs that are loaded by the print spooler service (spoolsv.exe) on system boot. DLLs that will be loaded can be added via the AddMonitor API call, or by adding the DLL's path to the Registry at *HKLM\SYSTEM\CurrentControlSet\Control\Print\Monitors*. Malware can configure a malicious DLL to be loaded which will allow for persistence upon system reboots.

Winlogon Helper DLL

Winlogon performs actions on startup. By editing the Registry key *HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon*, malware can make a DLL load during startup.

Execution Hijacking

Browser Helper Objects

Internet Explorer was designed with multiple exposed interfaces that enable custom the use of custom toolbars, buttons, or menu items. These same interfaces can be used for malicious reasons like stealing data or launching malware when Internet Explorer is started by modifying values in the *HKLM\Software\Microsoft\Internet Explorer* Registry key.

Change Default File Association

The Registry keys *HKEY_CLASSES_ROOT\.<extension>* are used to store the default commands that should be executed when working with different types of files, such as text of PDF files. Malware can modify these values to execute various commands every time a certain kind of file is interacted with.

Component Object Model Hijacking

The Microsoft Component Object Model (COM) is a windows system that lets software interact with each other through the OS. Malware can replace a legitimate software object by modifying values in the Registry key *HKCU\Software\Classes\CLSID\<object_ID>\InprocServer32*. When the system tries to execute the replaced software object, the malware's code will be executed instead.

Shortcut Modification

Shortcuts are common ways to reference other files or programs on a system. Malware can edit existing shortcut paths so that a malicious program is executed along with the intended program when the shortcut is double clicked by a user. After the path has been edited, if the shortcut is utilized, the malware will have persistence on the system.

File System Permissions Weakness

If permissions on a file system are not correctly set, then malware might be able to find executables that it can edit and manipulate. If the executable found is configured to run on a schedule, then malware can change or replace the executable to gain persistence.

Image File Execution Options Injection

The Image File Execution Options (IFEO) allows developers to attach debuggers to programs. An executable can be set as the Debugger value in the Registry Key *HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options/<program>*. If this value is set for a given program, then the executable specified as the debugger will be run instead of the program, getting the original program's name as an argument. Malware can use this to make the system execute a malicious binary every time a regular process would be run, thus gaining persistence.

Screensaver Hijacking

The settings for a user's screensaver are stored in the Registry Key *HKCU\Control Panel\Desktop*. If malware edits the SCRNSAVE.exe value to contain a path to a malicious executable, then the malicious executable will be executed instead of the usual screensaver after every screen timeout.

Executable Path Interception

Path interception occurs when an executable is placed within the system in such a way that when a legitimate program gets called, the placed executable is run instead of the intended executable. This can occur because of unquoted paths, or because the PATH environment variable has been edited. Search order hijacking can also be used to trick the system into executing the wrong program. When this technique is used on legitimate executables that are called on a regular basis, system persistence can be obtained.

Adversary Triggered Execution

Create Account

Malware that gains control of a system can create new accounts with preset passwords. These new accounts can then be used by an adversary who knows the preset password to gain access to a system in the future.

Accessibility Features

The Windows Accessibility Features are designed to help users interact with their computer, such as the magnifying glass or visual notifications. Some of these features can be launched before a user has logged in by pressing a certain key combination. If malware modifies or replaces the binaries of these features, an adversary can execute the malicious code with SYSTEM privileges when the key combination is pressed.

Appendix B - Malware Families

Family	Technique
Zox, Derusbi, Rovnix	**Accessibility Features Modification**
Sofacy, Ardamax	**Component Object Model Hijacking**
BlackEnergy	**Using File System Permission Weaknesses** **New Service Creation** **Shortcut Modification**
Nemesis, WhiskyAlfa	**Boot Sector Modification**
Flame	**Authentication Package Injection** **Account Creation**
Gazer	**Screensaver Hijacking** **Winlogon Helper DLL** **Start Folder Injection** **Shortcut Modification**
Sednit	**Logon Script Creation** **Component Object Model Hijacking**
KRYPTK, Hupigon	**Image File Execution Options Injection**
Mis-Type, Pirpi	**Account Creation**
Neshta	**Default File Association Manipulation**
None in Database	**AppCert DLL Injection**
None in Database	**Port Monitor Manipulation**
Prikormka	**DLL Search Order Hijacking**
Sakula, PlugX	**Run Keys Injection** **New Service Creation**
winMM, Zegost, Sakto (agentB)	**Accessibility Features Modification** **Start Folder Injection** **Shortcut Modification**
Zegost	**New Service Creation**
None in Database	**AppInit DLL Injection**