Analyzing Victim Process Behaviors Post Code Injection

Spyridon Mesaretzidis

M.Sc. Thesis
Octomber 2024

Supervisors:
Prof. dr. ir. A. Continella
PhD Candidate J.A.L. Starink

Semantics, Cybersecurity & Services Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Amongst other methods, malware uses code injection to propagate itself. Like any other technology method, new methods frequently arise. These advances lend themselves to new strains, which are accompanied by a lack of accurate detection mechanisms as well as a lack of understanding regarding the behavior of the infected processes that take post code injection. In this research, we examine the behavior of processes that are targets of code injection malware post-exploitation. To this end, We utilize the Virus Total dataset. We divide the data set according to the type of code injection utilized to determine which actions are taken by the infected processes after code injection. Subsequently, we determine their common action targets and extract any IP geo-location information from the observed network traffic. We observed heightened counts of behavioral metrics with operations revolving around the file system exhibiting more distinct behavior. Behavior revolving around IP and UDP processes did not exhibit any profound findings.

Contents

Chapter 1

Introduction

Amongst attackers that try to exploit software systems, a noticeable consideration in the eyes of security professionals is malware strains. The term ”malware strains” is used to denote software that is malicious and intends to cause harm to a target. This can be achieved in several ways, either by massively encrypting files and requiring ransom, causing resource drains on the system, or by simply deleting the contents of the hard drive. Most often, the driving force behind the creation of malware is the financial benefit of either the malware developer, the malware distributor, or both [1].

The phenomenon of malware is not novel, and the first sample - the Morris Worm - appeared in 1988, appropriately named after its creator Robert Morris [2]. Since this occasion, the number of malicious applications that have seen the light of day has increased, with their numbers constantly rising every year [3]. As is to be expected, several organizations formed with the express purpose of developing defensive software solutions, with the explicit aim of detecting infections on the target machines.

This type of software has led malware authors towards the development of methods that would allow their creations to infect a target and act virtually undetected for as long as possible. Among numerous techniques, one that stands out is *code injection*.

As denoted by the name of the technique, code injection is the practice of an application injecting code into another application. The goal is to manipulate the target application into executing actions that were not originally intended [4], [5]. This in turn shifts the responsibility of portraying malicious behavior to the injected process, potentially rendering malicious code benign during the scanning process of defensive software systems.

Infestations of malware have been numerous. In 2022, 5.5 billion malware attacks were detected [6]. Meanwhile, a 2021 study [7] on malware suggested that more than 11.15% of the examined dataset contained malware samples that used code injection techniques. A subset of the aforementioned dataset samples utilize novel injection mechanisms, for which we currently have no automated detection methods.

To the best of our knowledge, this is the only known modern academic effect that focuses on code injection. Previously released frameworks and tools focus on different aspects and techniques of dynamic analysis. All of these techniques primarily focus on determining the behavior that a malware sample undertook. For malware samples that make use of code injection, however, most of this behavior is missed since a target processes is used in order to perform any target actions. The aforementioned techniques include Function Call Analysis, Execution Control, Flow Tracking, Tracing, and Side-Channel Analysis [8].

Function Call Analysis examines the functions that are called by an executable to determine the specifics of its behavior. This can be achieved by analyzing the address space utilized by an executable, as it is done with TTAnalyze [9] and Mal-TRAK [10].

Execution Control regards halting the execution of a malware sample under analysis to examine its state. A notable tool that utilizes this method is Cobra [11]. Cobra bases itself on the concept of *debugging* to offer the analyst the ability to create sets of entry and exit breakpoints in the sample that is analyzed. The enclosed code is subsequently analyzed by Cobra offering fine-grained information about its functioning.

Flow Tracking regards tracking the flow of information within a malware sample. Tools utilizing the technique label binary data to indicate their origin. Once a malicious executable makes use of said data, any affected memory regions become tainted as well. Once a piece of tainted data reaches a specified region of code in memory, the corresponding data flow is analyzed to expose how a malware sample interacts with the operating system and the user. Vigilante [12] uses this technique to detect untrusted code that originated from the network.

Tracing involves the collection of information after the code has been executed. This can be achieved through Volatile Memory Analysis through the examination of memory dump files post malware execution or through Network Tracing, where the connections initiated by malware are analyzed to uncover *Command and Control* servers. LiveDM [13] utilizes memory analysis to detect the allocation of new kernel memory regions and thus track where a malware sample has installed itself. It subsequently performs an analysis of the sample's binary code. TrumanBox [14] on the other hand utilizes network tracing to track outgoing connections initiated by a malware sample and either emulate a response or simply forward the request.

Side-channel analysis regards the analysis of physical components of devices to ascertain whether an infection has taken place. Such analyses utilize power consumption, EM emission, and CPU event data to name a few. This method is primarily utilized for malware which target devices like PCI cards, medicals devices, micro-controllers etc. A notable example would be NumChecker [15]. The tool monitors hardware registers responsible for monitoring hardware-related activities, pre and post system call execution. Running on a system without any external connectivity, allows the tool to create a set of patterns for normal behavior. Once an unknown binary is executed a different execution pattern would be detected.

Examining these techniques leads to the discovery of flaws. Function Call Analysis, Execution Control, Flow Tracking, and Tracing would simply indicate that code injection has taken place while missing any behavior that would be observed post code injection. Side-channel analysis would only be able to detect that an infection has taken place without indicating its type (code injection). That is assuming that the malware has not managed to obfuscate itself.

This, in turn, limits how reliable our current dynamic analysis tools can be considered since they by default miss any behavior that stems from code injection. This can be supported by the findings of the aforementioned modern academic efforts performed by Starink et al. [7]. The author's study determined that 11.15% of the analyzed samples performed code injection. In reality however, this number is likely higher since the study itself focuses on a specific set of code injection techniques while being unable to detect some of them.

This means that there is behavior that even the most modern of tools are bound to miss. It is also unclear what kind of behavior is being missed. The results we obtained serve to support this argument.

Due to this, our research aims to answer several objectives. We first aim to distinguish whether or not significantly differing levels of activity are exhibited by targets of code injection pre and post-exploitation. Our aim here would be to determine the levels of behavior of code injection malware that is currently being missed by the presently available frameworks that do not monitor target processes of code injection.

Subsequently, we aim to understand whether code injection malware exhibits specific types of activity (e.g. network operations) more so than others, what this activity would be, and its degree.

Having gained a sufficient comprehension of our previous aim, we will examine whether specific processes are preferred over others regarding code injection malware. We will also examine whether any actions performed by the infected processes feature a preference over a specific set of targets.

We start by examining the behaviors observed by victim processes pre and post code injection. We categorize these behaviors based on the code injection type of each sample observed within our dataset, presenting and analyzing the most prevalent behaviors. The behaviors we examine include but are not limited to, Registry

Operations, File Operations, and Network Operations.

We discuss the deviations observed in the behavior of samples belonging to different code injection types, while also touching upon the most preferred targets encountered during the operation of the infected processes. The targets themselves vary based on the behavior itself.

We focus on the top five operations, determined based on the number of operations that were observed over the whole dataset. We proceed with examining the corresponding top 5 targets per code injection type. As a final piece of information, we examine the network traffic observed by each sample. We collect a list of IP Addresses and URLs. We process these artifacts obtaining and resolving any aliases through the use of DNS Records to provide geolocation information regarding the uniquely observed IPs per code injection type.

Chapter 2

Thesis Structure

The remainder of this paper is organized as follows. We begin by providing relevant background information, offering definitions of key concepts touched upon throughout the document, and surveying various use cases of code injection in chapter 3.

In chapter 4 we proceed by outlining our research methodology, offering a high-level overview of the processes that we followed denoting key information while arguing for our design decisions. Continuing with chapter 5 we discuss in detail how the experiment was conducted by analyzing the underlying infrastructure and other key details necessary to reproduce the experiment. We follow with chapter 6 where the results of our analysis are presented and discussed.

We move forth with discussing the limitations which we experienced in chapter 8 while we provide our conclusions in chapter 11. We close this document by presenting work undertaken by other researchers in chapter 10.

Chapter 3

Background

The focus of this research is to determine the behaviors of infected processes post code injection. To have a deeper understanding of the subject at hand, we need to initially explore what code injection entails, an outline of the injection techniques under study, as well as provide key information regarding the technologies which made this effort possible. In this section, we thus briefly outline legitimate and illegitimate uses for code injection, provide a short description of the techniques we focused on, as well as a concrete description of the technologies that made our efforts possible.

3.1 Code Injection Use Cases

Code injection is the practice of inserting code into a *victim* process by manipulating the memory space of the victim in one way or another [16]. This primarily takes place by enacting Processes Injection. A technique used by malware to evade defense measures and enact privilege escalation to achieve persistence and stealth of operations during cyberattacks. This results in a victim process becoming the target of the malware, aiming to make the target execute code on behalf of the malware, altering its originally intended functionality.

The practice of injecting code into a target has its own set of fruitful uses [17]. The most notable case would be the process of debugging an application by using breakpoints. This intends to halt an application's execution for an interested party to observe its state. Another well-known use would be the equivalent of method overloading in C implemented by naming a developer-defined function as a LibC function, followed by loading the respective code as a library [18].

The goals of malware developers making use of this technique differ. This goal can range from financial benefit by requiring ransom for encrypted files, cyber-espionage, and activism [19]. They would aim to manipulate a benign application into running arbitrary code for their purposes. Several techniques are currently known, with them being Classic DLL Injection, AppInit DLL Injection, and Thread Hijacking, just to name a few. [20]

Taxonomies that classify the currently known methods exist and are based on whether the sample indiscriminately infects processes, whether the injector process concurrently executes the malicious code alongside the victim, or whether the victim process is simply rendered unusable [21]. Code injection attacks are very damaging since they can be performed in a seamless manner, present difficulties during detection, and act as stepping stones towards other types of cyberattacks. The consequences can thus be severe leading to data breaches, financial losses, denial of service, botnet creation, etc [22].

3.2 Fundamental Concepts

Several technologies and concepts will be mentioned throughout this document. These concepts revolve around malware analysis and the process that was followed to collect all of the necessary experimental data. These technologies and concepts are worthy of mention and explanation since such an action can assure that the reader of this work will be able to concretely comprehend the discussed topics.

- **Static Analysis:** Static analysis refers to a technique that aims to examine the code and properties of a malicious file foregoing the need to execute it. There are several methods that fall under this definition, like reverse engineering (examining a disassembled malware binary), code signatures(byte patterns found in a malware sample), and AI approaches. AI Approaches mostly attempt to classify a sample based on previously encountered byte patterns found within well-researched malware samples. Static analysis is used mainly to classify malware samples into known families and extrapolate information on potential behaviors.

- **Dynamic Analysis:** The action of running a malware sample within an isolated environment determines the actions it performs and collects metrics of its functioning. These metrics include calls made to the Native OS API, runtime of the sample, IP/HTTP addresses reached, etc. Other metrics can be provided (e.g. number of spawned processes) but their availability depends on the underlying system used. Dynamic Analysis is generally preferred since it offers insight into a malware sample's behavior by executing the sample within a controlled environment.

- **Virtual Machine:** A simulation of a physical machine, equipped with its virtual hardware which supports an OS. Utilized for various purposes (e.g. hosting applications on production environments). For malware analysis, they are used to monitor the execution of malware samples.

- **Hypervisor:** A piece of software that is responsible for the creation, control, observation, and deletion of virtual machines(VMs).
- **Sandbox:** An isolated system based on a virtual machine can be used to individually examine the behavior of a malware strain.
- **Syscall Monitoring:** Monitoring of the system calls that a malicious executable under examination makes with the explicit purpose of pinpointing key characteristics of its behavior. Syscall Monitoring is the primary capability offered by several sandbox solutions like CAPE [23] and Drakvuf [24]. This information is important to a malware analyst since it offers insight into how a malware sample operates.
- **File Tracing:** Tracing of file accesses performed by processes. This information is used by malware analysts to obtain insight into malware sample's behavior.
- **Network Monitoring:** Tracing of network connections which may have been initiated by trojans, malware, or malware-infected processes.
- **Obfuscation:** The act of making the code of a program hard to understand by humans as well as computers without changing how the program works. Encryption, compression, and encoding are some of the most common obfuscation methods used by malware authors, frequently utilizing a mix of techniques.
- **Packing:** Used by malware authors to make executables more difficult to detect and analyze. Packed malware samples are a subset of obfuscated malware samples in which the malware sample is compressed.
- **Target Process:** A process running within an operating system infected by malware which is the target of a code injection attack by the aforementioned malware. The process can either be a service or application that came preinstalled with the system or it could have been installed by an administrator or user.

Chapter 4

Research Methodology

The overall process that we followed has been diagrammatically depicted in Figure 4.1. This chapter is devoted to describing the research methodology that we followed during the experiment. The tables referenced throughout this section are placed at the end. We begin by stressing the need for dynamic analysis. Malware developers are known for obfuscating and packing their samples to hinder static analysis techniques. This means that to gain a comprehensive insight into the functioning of the sample contained within our dataset, we need to use dynamic analysis methods.

This dynamic analysis system needs to be able to produce a list of all the system calls that are performed by an executable under analysis, which we denote as the *event stream*. This capability is important to us since it would allow us to accurately determine a malware sample's actions. Meanwhile, the process that was the target of code injection is referred to as *target process*. In light of this, the event stream generated by the malware strain under analysis is denoted as the *sample event stream* while the event stream generated by the target process is denoted as *target process event stream*.

During this experiment, we are not required to examine the totality of the event streams, as we are only interested in the call which resulted in code injection being performed as well as the subset of the target process event stream post code injection. We refer to the time that the code injection took place as the *injection timestamp* while the event stream that was generated by the target process before the code injection is named *pre-injection event stream*. Subsequently, we denote the part of the event stream that was generated by the target process post code injection as the *post-injection event stream*.

4.1 Methodology Overview

To determine what actions are taken by the samples in our dataset we need to run each sample through a Dynamic Analysis Sandbox. Within the sandbox, we install several applications that are known to be the targets of code injection for the samples in our dataset. After each successful run, we extract the event streams of the target applications. In the meantime, we also collect information regarding the behavior of the sample under analysis, provided by our underlying analysis system. Moreover, we proceed with saving the corresponding network trace for further analysis.

Having collected the event streams, we need to determine the injection timestamp. The injection timestamp is used during processing of the event stream to filter any actions that took place before code injection. It is also used to determine the function call that consolidates the code injection process. We perform these steps to filter the event streams of the target applications since we are only interested in behavior observed post-code injection while wanting to maintain a good level of accuracy of our results.

Having obtained the event stream of each sample run alongside the injection timestamp, we proceed with determining the code injection target per sample. This task is achieved through the use of two methods. By querying Virus Total API and by matching the target process PID to the arguments of the call that performed code injection. Code injection malware uses a process PID to target the process they wish to infect. Therefore, matching the arguments of the call responsible for code injection to a PID of an application running within our sandbox is a straightforward process. We utilized both methods since we do part, some of the Virus Total API responses do not include information on target applications.

Having obtained the event stream of each target application we extract the unique actions which all processes performed. We subsequently divide these actions into groups to increase their granularity which will in turn aid us in statistically processing our data.

Having grouped the actions we processed them by extracting the corresponding statistics. Consequently, we examine the target process's targets to ascertain with accuracy the actions that a target process performs post code-injection. As a final step, we process the extracted network traces to obtain IP geolocation information.

There is one important limitation that we need to denote, however. The determination of the timestamp required the utilization of execution traces obtained using the Drakvuf sandbox [25]. In the meantime, collecting the event streams of the available malware samples post-code injection was achieved using the CAPE Sandbox [23]. More information on why this choice was made, alongside any potential impact on the experimental results can be found in section 4.13.

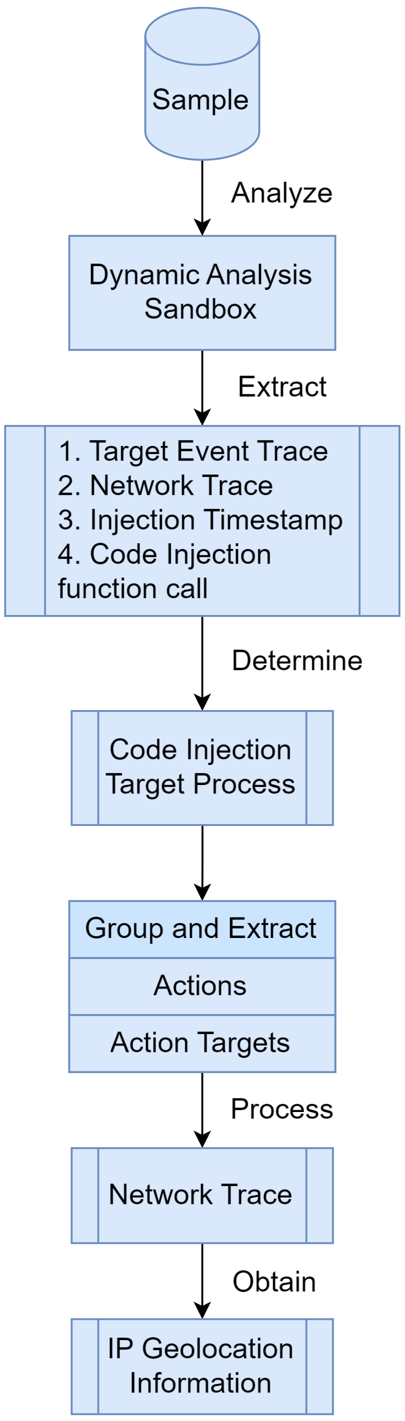

Figure 4.1: Methodology

4.2 Selection Of Software to Install

A machine used by an average Windows user is expected to have several applications installed. These applications can belong to a different type. These types include Document Processors, Browsers, File Exchange Applications, Instant Messaging applications, Email Clients as well as Programming Languages and frameworks which are frequently required by modern applications. Moreover, code injection malware will seek to find an application that it can infect. Based on this fact we install a selection of applications into the sandbox.

4.3 Obtaining the Code Injection Timestamp

Obtaining the code injection timestamp is paramount to this investigation, as it allows us to determine from which point onward we need to examine the post-injection event stream. Though code injection is performed using multiple system calls, we are only interested in the system call which concretely signifies that code injection has been performed.

The injection timestamp can be determined by obtaining the system call of the sample that injected the code into the target process. One of the methods which can be used is reverse engineering. Such an endeavor though would be very time-consuming and error-prone considering any obfuscation and packing mechanisms that the malware developer has put in place. In addition, any executable can feature several system calls which could be utilized to perform code injection thus leaving too much room for error and speculation. Moreover, malware samples are known to behave differently based on their environment [26] suggesting that a different set of calls could be observed based on the environment in which the malware sample was analyzed. This would in turn introduce further complexities and inaccuracies.

We thus obtained the code injection timestamp through an automated solution like the one developed by Starink et al. [7] which leverages a sample execution trace. This solution provided the timestamp of the system call used for code injection. The timestamp is required to understand from which point onward the Process Monitor event trace is to be examined. This allows us to remove any extraneous execution artifacts and thus obtain greater accuracy.

4.4 Injection Timestamp Utilization

A target process that has not yet been injected is expected to behave in a manner determined by its developers, as well as any possible user actions. A target process that has undergone code injection, however, will exhibit behaviors that encompass actions imposed by the injection code. If we were to examine the event stream of a target process that has not undergone code injection and the event stream of a target process that has undergone code injection, we would observe notable differences unless the injected code is stealthy or remains dormant. The injection timestamp is used during our analysis to determine from which point onward we can start examining the behavior and actions of a process having undergone code injection. This step is important since it allows us increase the accuracy of our results.

4.5 Obtaining the Target Process Event Stream

To be able to determine the action of applications that are the targets of code injection, we need a sandbox equipped with the capability of extracting the target process event stream. The event stream would contain a list of actions that the target process performed during the duration of our analysis. Each action of said event stream should also contain the corresponding targets of each action. Obtaining both the list of actions that the target processes performed as well as their corresponding targets would allow us to determine with specificity which behaviors are observed by the target processes post code injection.

4.6 Determining Code Injection Targets

For code injection to take place, a corresponding process should be installed and running in the sandbox used. However, said processes should initially be determined. Determining these processes is paramount to our endeavors since it allows us to pinpoint which target's post-exploitation event stream we are to examine. Being aware of the targets of code injection also allows us to obtain concrete results. Not performing this step would require us to determine the targets by examining all running processes and subsequently obtaining a list of potential candidates based on the differences observed in their event streams. This approach is not very trustworthy however since we cannot exclude the possibility that a target process may exhibit behavior during the analysis process which we have not managed to capture beforehand. This would in turn tag a process running in the sandbox as a target process, without this being the case.

We applied two methods to determine the target process. The first method would be to leverage information from the Premium version of the Virus Total API, which returns a report containing information on the target process that Virus Total managed to uncover. For the samples where such information was not available, we applied a second method.

We obtained a ProcMon process trace from a sandbox run without any sample loaded and designated this as the baseline. We continue by obtaining a ProcMon trace after analyzing a sample. We compare these two traces and pinpoint which process showcased deviations from the baseline, past the known code injection point. Any further calculations were performed using the data obtained from this process.

4.7 Injection Timestamp Accuracy

Since we are using timestamps obtained in Drakvuf while studying the behavior of the target process of code injection, we expect differences in the exact sequence of system calls performed by the sample under analysis. We do however expect that the same function call would be used to perform the injection.

In order to account for that effect we pinpoint the exact call and corresponding arguments that resulted in code injection from the obtained Drakvuf traces. We locate the timestamp of the same call in the obtained CAPE reports of ten samples.

Per sample we obtain the observed time difference between the Drakvuf system call trace and the CAPE system call trace. We perform this process manually for 10 samples and obtain their average. We factor in this difference in any subsequent operations and metrics. Since this fact can be seen as an impacting factor of our study, we have discussed any further impact in 8.1.1.

4.8 Trimming the Post-Injection Target Process Event Stream

The behavior of a modern application can differ based on several factors (e.g., having automatic updates enabled). This presents challenges regarding accurately determining the actions of the target process post code injection since artifacts from the target process' normal execution would be introduced. To our knowledge, there is currently no literature on how to exclude such artifacts. We have thus resorted to obtaining the ten execution traces of the processes listed in table 5.1 during various times to account for this fact, as well as any underlying Windows services.

Having collected these traces we are able to determine artifacts that were part of the normal execution of applications. This allows us to exclude said artifacts from our analyses. The rationale that guides our selection of installed applications in our sandbox that are not native Windows applications or services has been outlined in section 4.6.

4.8.1 Removing Extraneous Artifacts

Analyzing the totality of our dataset would result in several extraneous artifacts within the collected event traces. These artifacts would stem from the normal execution of applications and services that were not affected by code injection malware. Said artifacts would introduce noise during our analysis and we therefore took steps to programmatically remove a number of them.

4.9 Determining Target's Post-Injection Actions

After each successful sandbox run we obtain the event stream of the malware sample under analysis as well as the Process Monitor trace of the Windows installation underlying our Sandbox. After pinpointing the target through the process outlined in the section 4.6 we remove any artifacts that we have identified using the method outlined in the section 4.8 while also excluding and process actions that take place before the code injection timestamp obtained through the actions described in section 4.3. This results in a Process Monitor trace which lacks any artifacts that would originate from normal usage of the underlying Windows OS. Having obtained this cleaned-up trace for each of our samples we extract the actions of the target process alongside their targets.

4.10 Results Processing

The data we collected have been processed to extract counts and percentages of the top 5 most prominent observations without delving deeply into statistical analysis since such an action would surpass the goals of our research.

We offer Figure 4.2 to showcase the sources of our data as well as the chain of processing that they underwent. We begin by running the execution traces through the *Code Injection Detector* to obtain the code injection type and the corresponding timestamp. In the meantime, run our dataset samples. For each sample, we obtain the Process Monitor Event Streams and the CAPE report. We use the Event Streams to determine the behavior of a process post code injection. In the meantime we use the CAPE reports to process any triggered behavior signatures and obtain all the observed IP, UDP, TCP and Domain(URL) addresses.

From the event stream, we extract the actions undertaken by the sample post code-injection alongside the action targets. From the CAPE report, we extract any triggered signatures while obtaining the corresponding network trace. Having collected this information we determine the behavior of each sample. We subsequently examine its network trace to extract IP geolocation information.

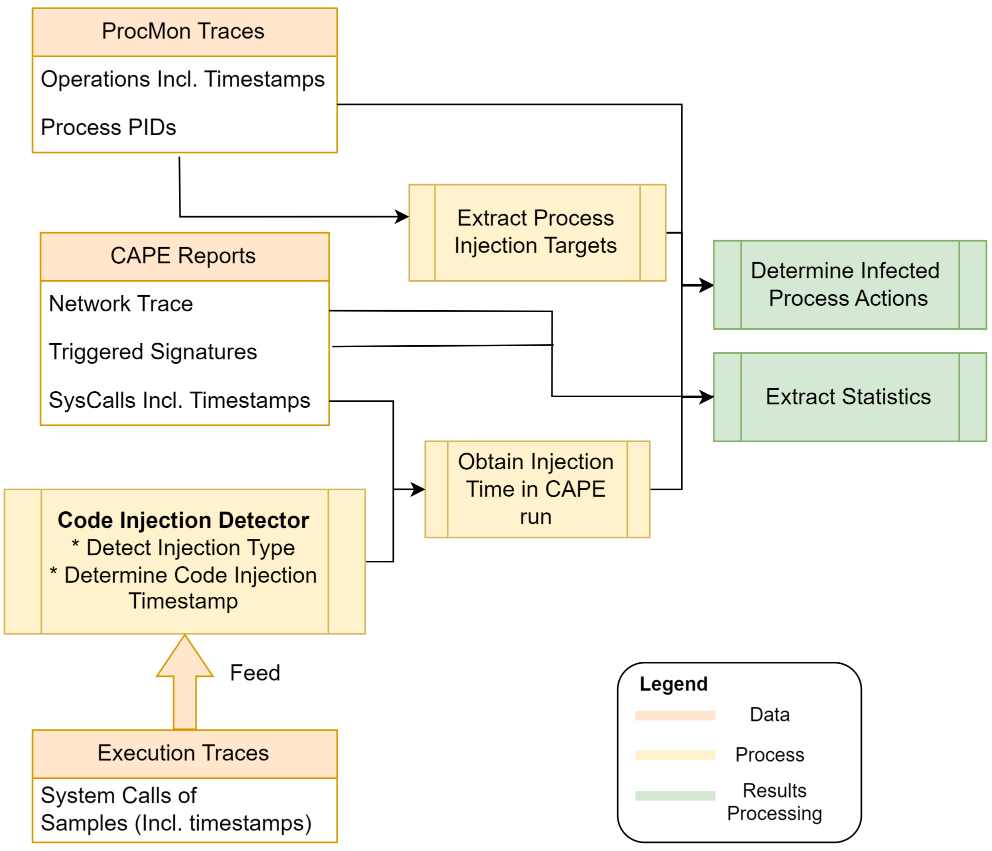
ProcMon Traces
Operations Incl. Timestamps
Process PIDs
Extract Process Injection Targets
CAPE Reports
Network Trace
Triggered Signatures
SysCalls Incl. Timestamps
Determine Infected Process Actions
Extract Statistics
Obtain Injection Time in CAPE run
Code Injection Detector
* Detect Injection Type
* Determine Code Injection Timestamp
Feed
Execution Traces
System Calls of Samples (Incl. timestamps)
Legend
Data
Process
Results Processing

Figure 4.2: Results Processing

4.11 Experimental Metrics

There are several metrics and data that can assist us in examining how much behavior is been neglected during dynamic analysis of code injection malware. We want to initially investigate the list of processes that were the target of code injection. These processes can either be native Windows applications and services or software components that were not native to the Windows OS and were installed by us.

We are similarly interested in the list of target processes per code injection type. This list can help us understand in greater detail if there are any notable differences between the targets of code injection malware and whether specific interest should be paid to a set of applications during analysis.

The actions of the processes that were the targets of code injection are of particular interest to us. This metric can accurately determine the set of actions that were observed after code injection has taken place. This set serves as an indicator of the behavior that is currently missed by dynamic analysis tools and frameworks during the analysis of code injection malware.

As a supplementary set of information, we are also interested in the targets of the malware samples we analyzed, per code injection type. These targets exist to denote the fact that not only should processes fall under the examination of dynamic analysis tools, but also the various components of the underlying system.

4.12 IP And Domain Addresses Metrics

Malware is known to reach Command and Control servers for several operations. In light of this, we are interested in understanding if any of the analyzed samples reached out to any servers. This is accomplished by examining if any of the samples reached any IP or Domain addresses.

4.13 CAPE and Drakvuf Sandboxes

As mentioned in the beginning of this chapter, the injection timestamp was determined using traces obtained from the Drakvuf sandbox, while the event streams that allowed us to examine the behavior of code injection malware were obtained using the CAPE sandbox. There are several reasons for this action. This section offers a set of methodological reasons for this alongside any potential impact and consequences. Any limitations are discussed in section 8.1.

Our key research objective is to indicate the level of activity that code injection malware exhibits post-code injection, and possibly quantify it. A code injection malware sample however is nothing more than an executable. And just like any executable, some behavior is expected before the main functionality is performed. This behavior can be attributed to several potential actions like process detection (for code injection) or environment detection (for evasion). Therefore, a means by which to be able to determine behavior pre-code-injection is required to gain the ability to quantify any behavior observed post-code injection.

Out of the two sandboxes, only CAPE offers this capability. CAPE uses a large number of author and community-issued signatures that examine the system calls performed by the sample under analysis to determine high-level behaviors (e.g. ransomware behaviors). These signatures can be used to collect information as to whether or not there is any noteworthy activity taking part before code injection being performed Drakvuf on the other hand, offers several plugins that offer insight into the purely technical execution details for a sample like the system calls it performed.

If we were to accurately determine what kind of actions were performed by each sample, we would need to map a set of system calls alongside their arguments to a set of high-level behaviors. Such an action however could have been met with low accuracy and could constitute a study on its own. Furthermore, utilizing Drakvuf's plugins would not allow us to peer into the code injection target's functioning since the plugins only target the sample under analysis.

This would lead us to install a different solution to monitor the code injection target processes' action. Considering how the Windows OS is supported both by CAPE as well as Drakvuf, said the solution could have been installed in either OS. If we were to prefer Drakvuf over CAPE, the only thing we would achieve would be a minor improvement in the accuracy observed regarding the determination of the time injection timestamp and potentially less evasive sample behavior. Since this fact alongside our choice of sandbox can be seen as a limitation we have expanded upon it in section 8.1 alongside other limiting factors.

Chapter 5

Implementation

This chapter is devoted to the discussion of the experimental setting that was used. The physical machine utilized was equipped with Ubuntu 22.04 while the machine was connected to an isolated WiFi network.

We opted for this option since we desired to provide the sandbox with Internet access to adequately simulate a real machine and collect all of the network artifacts generated from analyzing our samples for further reanalysis.

We were provided with the capability to directly select which hardware components the VM underlying our sandbox would contain, and thus opted-in for components that were released during 2017 while providing them with realistic UUID values to increase the fidelity of our setup in the eyes of our samples. To aid the reader's understanding we have provided Figure 5.1 which showcases how the machine was set up. The figure showcases how we analyzed the VirusTotal Dataset through CAPE Sandbox. CAPE The Sandbox itself was equipped with Process Monitor, and the CAPE agent - a Python script used to run and monitor the sample under examination. We performed this action since CAPE is fulfilling the role of the typical sandbox that uses process-specific monitoring. This means that only the malware sample was observed by CAPE. On the other hand Process Monitor allows us to observe all of the running processes, including the target of code injection. As a final step in the figure we obtain a CAPE report, alongside any triggered CAPE signatures, the Network Trace that was captured during the run, the list of System Calls the sample performed, as well an event trace generated by the Process Monitor.

The sandbox itself was equipped with Windows 10 21H2 as per the suggestions of CAPE's authors to maximize compatibility while disabling any automatic update, User Account Control, and firewall services since these could interfere with our measurements.

We similarly disable automatic updates of the software detailed in Table 5.1 while ensuring that processes of the aforementioned applications have spawned and are running.

Before loading any samples for analysis, we generate several usage artifacts on the OS as well as the application level. On the OS level, we load the machine with various images and files having manipulated their creation and modification dates setting them sometime in the past. We perform this action to make these artifacts look as realistic as possible. We take the extra step of populating our nonimage files (e.g..xls files) with data and give them names that are expected to be found in a real user's machine (e.g. 2016 Tax Return.xlsx). Furthermore, we utilize a directory structure that would resemble a real machine, in an attempt to circumvent any further checks that the malware sample under analysis would perform.

To obtain the actions performed by the target process, we have opted in for a solution. This solution is Microsoft's Process Monitor [27] which is part of System Internals. The application monitors system processes both on the OS and on the user level, monitoring a variety of information. This information includes Process IDs, process actions as well as process action targets. The actions of a process include, but are not limited to, Registry Operations, File System Operations, and Network Operations.

After we run the data set samples we collect all execution artifacts which include the Process Monitor traces, the corresponding CAPE reports, as well as the code injection timestamp from the code injection detector, as also shown in Figure 4.2. The timestamp is used to ascertain from which point onward we should consider examining the Process Monitor traces.

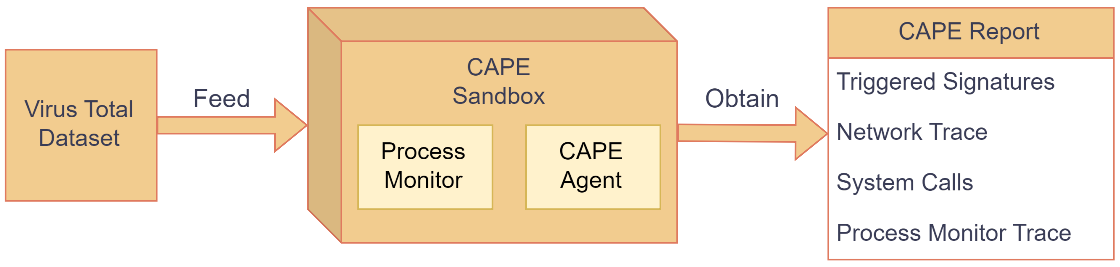

Figure 5.1: Experimental Setup

5.1 Selection Of Software to Install

For the remaining software, we utilized the VirusTotal API to gain insight into the applications that our samples were targeting. The responses we collected from the API contained the set of software applications into which our dataset was injecting code. Having obtained the resulting set, we moved forth towards installing the software which is listed in Table 5.1. From this table, two entries that stand out are the Python 3.10 programming language as well as the .NET 4.5 Framework. Python was required to run CAPE's agent. The .NET 4.5 Framework was required to ensure that our samples could run within the sandbox.

Installed Software
Firefox
Chrome
Opera
Libre Office
Open Office
Outlook 2010
WinZip 32
Win Rar
QBit Torrent
VS Code
F.lux
Discord
Glary Utilities
Lightshot
Python 3.10
.NET 4.5

Table 5.1: Installed Sandbox Software

5.2 Process Monitor Action Grouping

A Process Monitor event trace contains the process name, along with the process action, as it is shown in the above subsection. This representation of a process's behavior is finer than the portrayal of the system calls that a process performed. However, it is still coarse enough to not allow fine statistical processing. We have thus extracted the set of all unique Process Monitor actions we have observed from all analyses of our dataset, and have aggregated them into groups.

Though the tool's granularity in terms of process actions is finer than the traditional system calls, it is still not on a high enough level to allow us to extract and analyze information. Therefore after collecting all of the ProcMon actions generated by all of the target processes during our experiments, we have aggregated them together into groups to better facilitate results processing.

The aggregation process was based on the fact that each observed Process Monitor action name was prepended by a stem which indicated part of its functionality. For example, the action named RegFlushKey indicates that the action writes all the attributes of the specified open registry key into the registry. We have thus grouped actions that begin with a common stem into a group. The only exception to this rule was the *File Operations* and *Process Operations* groups. This occurred because the actions of these groups were not prepended by a stem but were either postpended or exhibited names composed of two words. Notable examples are the *CloseFile* and the *Thread Exit* actions.

The groups themselves along with their associated actions have been listed in Tables 5.3 through 5.9. The unique actions that were observed during our analyses have been listed under Table 5.2, while we also offer the definitions of these actions which have been extrapolated from the name of each action as well as it's observed targets.

ProcMon Action Definitions	
CreateFileMapping	Create an in-memory mapping of the contents of a file.
FileSystemControl	A operation on the file system.
RegFlushKey	Writes all the attributes of the specified open registry key into the registry.
RegUnloadKey	Removes a section of the registry that was loaded using the reg load operation.
QueryAllInformationFile	Get various kinds of information about a file object.
QueryAttributeInformationVolume	Retrieve information about the file system and volume associated with the specified root directory.
QueryAttributeTagFile	Retrieve file attributes and tags.
QueryBasicInformationFile	Get basic kinds of information about a file object, like it's name and size.
QueryDeviceRelations	Retrieve device relation (property keys) as defined by the Unified Device Property Model.
QueryDirectory	Query for directory information.
QueryEAFile	Query Extended Attributes Metadata.

Continued on next page

Table 5.2 – continued from previous page

QueryFileInternalInformationFile	Query the file that contains information on the system itself.
QueryFullSizeInformationVolume	Query information about a system volume.
QueryInformationVolume	Query information about a system volume.
QueryNameInformationFile	Query the system information file for a name.
QueryNetworkOpenInformationFile	Query the system information file for network information.
QueryNormalizedNameInformationFile	Query the system information file for a normalized name.
QueryObjectIdInformationVolume	Query for the Object ID of a file or folder.
QueryOpen	Query for open files.
QueryPositionInformationFile	Query for the position of the information file.
QueryRemoteProtocolInformation	Query for information for Remote Desktop Protocol.
QuerySecurityFile	Query for a security file.
QuerySizeInformationVolume	Query size information for a volume.
QueryStandardInformationFile	Query information for a standard file.
QueryStreamInformationFile	Query information for a stream file.
RegQueryKeySecurity	Query registry key associated with security.
SetAllocationInformationFile	Set the allocation information file.
UnlockFileSingle	Unlock a region in an open file. Unlocking a region enables other processes to access the region.
QueryEaInformationFile	Query Extended Attributes Information File.
QueryDeviceInformationVolume	Query a device information file for information regarding a volume.

Continued on next page

Table 5.2 – continued from previous page

QueryNetworkPhysicalNameInformationFile	Query an information file for the physical name for a network.

Table 5.2: Definitions of the Process Monitor Actions That Were Observed Within All Of The Malware Sample Execution Traces.

File Operations	
CloseFile	CreateFile
CreateFileMapping	DeviceIoControl
FileSystemControl	FlushBuffersFile
LockFile	NotifyChangeDirectory
ReadFile	UnlockFileSingle
FileSystemControl	FlushBuffersFile
WriteFile	

Table 5.3: *Process Monitor Actions Which Belong to The File Operations Aggregated Actions Group.*

Process Operations	
Load Image	Process Create
Process Exit	Process Profiling
Process Start	Thread Create
Thread Exit	

Table 5.4: *Process Monitor Actions Which Belong to The Process Operations Aggregated Actions Group.*

Query Operations	
QueryDeviceInformationVolume	QueryNetworkPhysicalNameInformationFile
QueryAllInformationFile	QueryAttributeInformationVolume
QueryAttributeTagFile	QueryBasicInformationFile
QueryBasicInformationFile	QueryDeviceRelations
QueryDirectory	QueryEAFile
QueryFileInternalInformationFile	QueryFullSizeInformationVolume
QueryInformationVolume	QueryNameInformationFile
QueryNetworkOpenInformationFile	QueryNormalizedNameInformationFile
QueryObjectIdInformationVolume	QueryOpen
QueryPositionInformationFile	QueryRemoteProtocolInformation
QuerySecurityFile	QuerySizeInformationVolume
QueryStandardInformationFile	QueryStreamInformationFile

Table 5.5: *Process Monitor Actions Which Belong to The Query Operations Aggregated Actions Group.*

Registry Operations	
RegFlushKey	RegUnloadKey
RegCloseKey	RegCreateKey
RegDeleteKey	RegDeleteValue
RegEnumKey	RegEnumValue
RegLoadKey	RegOpenKey
RegQueryKey	RegQueryKeySecurity
RegQueryMultipleValueKey	RegQueryValue
RegSetInfoKey	RegSetKeySecurity
RegSetValue	

Table 5.6: *Process Monitor Actions Which Belong to The Registry Operations Aggregated Actions Group.*

Set Information File Operations	
SetAllocationInformationFile	SetBasicInformationFile
SetDispositionInformationFile	SetDispositionInformationEx
SetEndOfFileInformationFile	SetPositionInformationFile
SetRenameInformationFile	SetSecurityFile
SetStorageReservedIdInformation	

Table 5.7: *Process Monitor Actions Which Belong to The Set Information File Operations Aggregated Actions Group.*

TCP Operations	
TCP Accept	TCP Connect
TCP Disconnect	TCP Receive
TCP Reconnect	TCP Retransmit
TCP Send	TCP TCPCopy

Table 5.8: *Process Monitor Actions Which Belong to The TCP Operations Aggregated Actions Group.*

UDP Operations	
UDP Receive	UDP Send

Table 5.9: *Process Monitor Actions Which Belong to The UDP Operations Aggregated Actions Group.*

5.3 Removing Extraneous Artifacts

From our runs, we obtained several execution traces. These execution traces also contained several actions that were generated during normal application execution. We therefore need to take action to reduce the number of these artifacts present within the execution traces of the malware sample we analyze. We resorted to obtaining the ten execution traces of the processes listed in Table 5.1 at various times to account for this fact, as well as any underlying Windows services.

This collection did not take place during random runs of our sandbox while having its underlying OS boot from the start but after spawning our sandbox from a saved OS state. This was achieved because our sandbox of choice (CAPE) utilizes KVM [28] to offer a virtual machine within which to examine our samples while requiring us to save the machine state once we are done setting it up for it to be reused for each analysis. This allowed us to spawn instances of our installed applications before saving the snapshot, thus preserving their Process IDs during each sandbox run alongside the Process IDs of each running Windows service.

Having a fixed set of Process IDs, we were able to obtain a collection of tuples for each running process and service with each tuple of the form:

$$(Process\ Name,\ Process\ ID,\ Process\ Action,\ Target)$$

With *Process Action* denoting a Process Monitor Action, and *Target* determining the Action's target (e.g a file if a File Operation is performed). An example tuple follows bellow:

$$(SVCHost, 1332, RegCloseKey, HKCU)$$

This process allowed us to remove several extraneous artifacts from our traces. We are rather confident in the completeness of this method since we have disabled any form of automated updates on both the OS and the application level.

5.4 Determining Code Injection Targets

In Chapter 4 we identified two methods to recognize the code injection target processes. The first method regarded using the Virus Total Academic Dataset to obtain information on which process was the target. This method produced results only for a subset of our dataset, leaving us oblivious to the targets of the remaining samples.

We were thus required to obtain information on the targets of the rest of the samples under examination. For this purpose, the sandbox that we utilized was equipped with several applications listed in the table 5.1 and collected their Process IDs through the use of Process Monitor. We have divided the installed software into categories which are listed in the same table. For the samples that we were unable to detect which application they targeted after having obtained their injection timestamps, we used a different method. We extracted which system call was responsible for code injection from their corresponding event streams. This call was the one that matched the timestamp which was provided by the injection detector we utilized.

By performing this action we were able to obtain the set of unique system calls which were used for code injection. Examining their corresponding documentation we extrapolated which arguments were used to specify which was the PID of the target processes. Armed with this knowledge at hand we were able to determine which process was the target of code injection by cross-referencing their PIDs from a Process Monitor trace we obtained from the sandbox, while not having any samples under analysis.

5.5 Sample Behavioral Information

CAPE offers its users the ability to obtain information on the behavior of a malware sample based a number of signatures. Said signatures examine information that was extracted during the analysis of a malware sample. This information includes but is not limited to the system call trace that the sample generated, any IP addresses observed, and file system objects accessed. The overwhelming majority of the signatures has been created by the malware analysis community.

Due to the high number of signatures which accompanied CAPE (more than 400) we have once aggregated those signatures into groups. Each signature that was aggregate represented a very specific malware characteristic. A distinct example would be the signature which was responsible for detecting whether a loaded sample was a banking Trojan which attempted to steal login credentials. We were therefore able to aggregate these signatures together into distinct high level groups based on common characteristics.

5.6 Experimental Metrics

In this section, we describe the processes followed to obtain the various metrics from our measurements. We provide the reasoning behind our metrics and how they were obtained while offering code listings and formulas where applicable. We begin by portraying Listing 5.1. The listing provides an understanding of how we calculated the percentages which we portray in Results section 6.

```
def getPercentage(occurancesCount: int, total: int):
    return float(
        "{:.2f}".format(
                ( occurancesCount / total ) * 100
            )
        )
```

Listing 5.1: Function Used To Obtain Percentages.

5.6.1 Targeted Processes List

Each Sample Under Examination (SUE) targets an application to perform code injection. Considering the source of our dataset we can safely assume that all if not the majority of the samples target consumer applications and services found within

Windows systems. From the list of services and applications running in our sandbox, we can extract a list of the targets that were encountered. We obtain this metric by associating each running application and service with a PID, and subsequently querying the cross referencing this list with the known target process of each SUE. The resulting set contains the unique target processes which were observed.

5.6.2 Target Processes per Code Injection Type

Because each malware sample target has an application as its target we developed an interest in obtaining the percentage of samples that target each unique code injection target process per code injection type.

To obtain this metric we determine the code injection target for each sample per code injection type and obtain the count of samples that targeted each unique application. We subsequently obtain the sum of all targeted applications and the top 5 targeted applications before generating the percentages per code injection type. The Listing 5.3 offers sample Python code as to how this has been achieved.

```
def getTargetPercsPerInjType(codeInjectionTypes):
    for each injectionType in codeInjectionTypes:
        sumOfTargets = injectionType.getSumOfTargets()

        # Get the top 5 application targets and their
        # hit counts.
        top5TargetsDict = injectionType.getTop5Targets()

        percentagesDict = {}
        for target in top5TargetsDict:
            percentagesDict[target] = \
                getPercentage(top5TargetsDict,
                    sumOfTargets)
    return sumOfTargets
```

Listing 5.2: Function Used To Obtain The Target Processes And Their Percentages Per Code Injection Type.

```
def getTargetPercsPerInjType(codeInjectionTypes):
    for each injectionType in codeInjectionTypes:
        sumOfTargets = injectionType.getSumOfTargets()

        # Get the top 5 application targets and their
        # hit counts.
        top5TargetsDict = injectionType.getTop5Targets()

        percentagesDict = {}
        for target in top5TargetsDict:
            percentagesDict[target] = \
                getPercentage(top5TargetsDict,
                    sumOfTargets)
    return sumOfTargets
```

Listing 5.3: Function Used To Obtain The Target Processes And Their Percentages Per Code Injection Type.

5.6.3 Target Process Actions per Code Injection Type

To obtain this metric we collect the count of the aggregated actions that each analyzed sample performed. We proceed with summing each count for each sample, per code injection injection type. We move forth with calculating the total sum to obtain the percentages per code injection type as shown in Listing 5.4.

```
def getTargetActionsPercsPerInjType(actionsOfTargets):
    # Get all observed Injection Type and set them
    # as keys.
    percsDict = dict.fromKeys(actionsOfTargets.injTypes)

    # Get total of actions per injection type per action.
    sumsDict = getTotalPerInjType(actionsOfTargets)

    for target in actionsOfTargets:
        injType = target.injectionType
        targetActionsLst = target.getListOfActions()

        # Count of actions per injection type.
        actionCountsDict = getActionCounts(targetActionsLst)

        # Add to the corresponding dict
        # entry the count of actions.
        getPercsForInjType(percsDict, injType, target)
    return percsDict
```

Listing 5.4: Function Used To Get The Target Process Actions per Code Injection Type.

5.6.4 Sample Targets per Operation Type Code Injection Type

To calculate the counts and percentages found within this section we collect the targets of each operation per malware sample. We divide each target into sets based on the aggregated action. For example, a Registry Operation which had the *HKCU* registry key as its target has been in the Registry Operations set. Whenever we encounter a target a second time, we simply increment its count by 1. After this process has been completed we obtain the corresponding percentages and plot our results. An example of how this processing took place has been provided in Listing 5.5.

```
def getTargetCountsPercentages(targetsDict, actionGroupsList):
    results = dict.fromKeys(actionGroupsList)

    for target in targetsDict.keys():
        # Get type of operation. E.g. Registry
        opType = target.getOperation()

        # Get responsible sample injType
        injType = target.getSampleName.getInjType()

        # Count the occurances of the target.
        if target not in results[opType]:
            results[opType][target] = 1
        else:
            results[opType][target] += 1

    return results
```

Listing 5.5: Function Used To Get The Sample Targets per Operation Type Code Injection Type.

5.7 IP And Domain Addresses Metrics

After each successful sandbox run, CAPE created a report that contained the list of IP, TCP, and UDP addresses (henceforth denoted as IP addresses) alongside the domains that were observed during the analysis. From these lists, we compose a set of unique IP addresses and a set of unique domains. We use both of these sets and lists to produce metrics and statistics. Before pursuing this goal, however, we collect the domains and IP addresses from ten sandbox runs without a sample loaded for analysis. We collect the set of unique domains and IP addresses and we exclude their values from subsequent analyses.

For the lists of non-unique domains and IP addresses, we obtain the number of samples that initiated such communications. We utilize these lists, alongside the list of successfully processed samples to extract metrics. These metrics include the total number of reached IP and domain addresses and the average number of IP/domain addresses per sample alongside others.

For each unique domain, we obtain its corresponding DNS record. From the DNS queries that returned a response, we extract any aliases that may be associated with the domain. The unique aliases are added to the domain set. Having obtained this information we query the DNS records again for any IPs which are associated with said domains. Not all domains appear to be still active, with many of them having no IP associated with them.

From the domains that still have an associated IP address, we obtain any unique values and add them to the existing IP set, avoiding any duplicates. We also performed a reverse DNS search to obtain any URLs from the set of unique IP addresses. We performed this step to ensure that we have collected a wealth of data. And results were added to the set of unique domains. From the resulting set, obtain IP-geolocation information from an online service. The service's responses provided for each IP the country as well as the city with which it could be associated.

Chapter 6

Evaluation and Experimental Results

This chapter is devoted to discussing the experimental results which were obtained during our experiments. We offer several lists and figures that allow us to delve into the actions observed per code injection type. Our goal would be to understand which are the most prominent actions observed within the malware samples featuring a specific code injection type, most used action targets, and differences. The sections that correspond to the analysis of our results are accompanied by tables. Said tables have been provided in Appendix 12.

6.1 General Dataset Statistics

Our analysis produced several general statistics. Table 12.1 details the number of code injection samples that our code injection analyzer was able to detect and the number of code injection samples that we were able to process. The samples themselves belong to the 2017 VirusTotal Academic dataset. Examining the table we observe that the original dataset contained 957 samples. The table also details that there were 44 unique running processes. These were the processes and services which were running within the Windows sandbox. After using the injection detector [7] we obtain the code injection method which each sample utilized. The injection detector itself utilized a system call trace, recorded from the DRAKVUF sandbox. The corresponding traces themselves, were provided by Starink et al. [7]. The number of number of samples performed on each observed injection type are presented in Table 12.3 alongside their corresponding percentage.

We observe that the samples that utilize the *COM Hijack DLL Injection* code injection method, dominate the dataset, representing 57.26% of the total number of samples. They are followed by the samples which perform *Process Hollowing*. These samples however represent only 14.31% of the total dataset samples. The spread exhibited by the rest of the samples is not as great as the top two entries, having *AppInit DLL Injection* and *Image File Execution Options* stand at 6.89% of the total dataset. A notable observation is the fact that our dataset only contains one *Classic DLL Injection* sample, representing 0.1% of the total dataset.

These counts and percentages, however, do not represent the number of samples that we utilized in our experiment. For a sample to be considered viable for analysis it needs to have undergone a successful analysis (no errors and a successfully generated CAPE report). The sample also needs to have resulted in a Process Monitor event trace, which was not malformed and which we would thus be able to process. These requirements meant that a subset of our original dataset was processed during our analysis. We therefore offer Table 12.5 which details the number of samples belonging to a specific code injection type that were analyzed.

We begin by denoting the difference observed in the total number of samples. The initial dataset exhibits 957 samples, while the number of samples that were suitable for analysis was 778. This difference of 179 samples amounts to a difference of 18.7%. Exploring the differences between the two datasets further, we observe that the samples that perform *COM Hijack Injection* remain the most abundant reaching 55.1%. They once again make up the majority of the dataset, exhibiting a minor reduction of 2.16% regarding the percentage of the dataset they represent.

The same observation can be made for the remaining samples, observing that the percentile differences from the initial dataset remain minimal. The only notable case is the one of *Classic DLL Injection*, where the percentage has remained unchanged. This would be because there is only one sample in both the initial dataset and in the dataset which has been used for analysis.

6.2 Observed Malware Families

To gain a firmer grasp of our results, we need to gain a firmer grasp of our dataset. For this reason, we retrieved information regarding the malware families observed within our dataset. We have performed this step by making use of the AVClass project [29].

Observing the distribution of each observed malware family as detailed in table 12.6 reveals two interesting statistics. We first observe that the overwhelming majority of samples belong to the *dinwod* and *berbew* families (155 and 106 samples respectively). Samples that belong in the*cosmu* and *tinba* follow alongside samples that have been classified as *singletons*. We observe that those three families feature minor spreads between the number of samples that can be attributed to them. They however portray a difference of almost 100 samples in comparison to the first (top) two families.

From these top 5 entries, dinwod and berbew can be classified as trojans based on the work of O'Shaughnessy and Breitinger [30]. Similarly, an export of the known malware families found within OTX [31] regards strains of the tinbu family as trojans, while strains of the cosmu family are considered as banking trojans by the malware analysis community [32]. As for the malware samples that fall under the *singleton* family, these are considered singular instances of a malware sample, without any connections to any other families.

Examining Tables 12.7 through 12.13 reveals that malware samples that utilize a common injection method appear to belong to a specific set of malware families. These sets do not intersect between different types of code injection. The only exception is the *singleton* family which is encountered in almost all types of code injection, except that of *AppInit DLL Injection* and *Image File Execution Options*. These results serve to signify the fact that there are unique malware strains, each utilizing a code injection method that they potentially find more preferable.

In the meantime examining the Tables 12.7 all through 12.13 reveals that malware samples that belong in the code injection method of ”COM Hijack DLL Injection” attribute the most to these counts. This phenomenon can be attributed to the high number of samples contained in our dataset which perform this type of code injection, numbering four hundred and twenty-six, making up about 55.1 % of the examined and processed samples.

6.3 Triggered CAPE Signatures

CAPE offers its users the ability to examine any signatures that were triggered during the analysis of a malware sample. Each signature checks whether the system call trace of a sample under examination matches several conditions. These conditions included any files accessed, the sequence of system calls, or their arguments.

We observed that no signatures were triggered by any of the analyzed samples. We attribute this finding to the fact that each CAPE signature examines the system call trace generated by a sample under analysis. Therefore, if a sample uses a target process to perform any malicious actions, then the sample would not trigger any signatures. We also observed that none of the samples triggered any of the aggregated signatures that were responsible for detecting code injection.

6.4 Targeted Processes List

This section is devoted to discussing each unique observed targeted process. Our findings can be found under Tables 12.14 and 12.15. We have denoted processes that were the targets of code injection as *targeted processes*, while the processes that were not targets of code injected we name as *not targeted processes*.

In a similar fashion processes that were native installations of the underlying Windows 10 OS (that is they came preinstalled) have been denoted are *native processes*. Applications that we installed have been denoted as *not native processes* and have been listed in Table 12.14.

Examining the resulting list we observe several processes that we installed were the targets of code injection and have been listed in the table 12.14. Examining these two tables, we initially observe that not all non-native applications were the targets of code injection, but only a subset of them. By cross-referencing their category by using Table 5.1 found under chapter 4 we can extract some interesting statistics.

We observe that 100% of the applications that fall under the *Browser* category have been the targets of code injection, followed by 60% of software that falls under the *Utilities* category. We similarly examine the native targeted processes, by initially investigating how they are used by the Windows OS before moving in to categorize them in table 12.15.

Counting the instances of categories from the aforementioned table we observe that 10 out of the 31 ($\approx 13\%$) unique native targeted applications belong to the *Process and System Management* category. Applications that belong to the *System Security* category follow with 19% (6 out of 31) while applications that fall under the *UI Management* category make up almost 13% (4 out of 31) of the applications.

This allows us to theorize that code injection malware that falls under the examined types of code injection have a clear preference for the type of software they aim to infect. Moreover, even though we examine 700 hundred samples we observe that there is not a large selection of native targeted processes. This can, in turn, imply the existence of a specific set of attack vectors, or a collection of post-injection actions which require privileges attainable only by injecting code into the target process.

6.5 Target Processes per Code Injection Type

Each malware sample targets an application with the explicit purpose of injecting code. This process can be of interest to us since it can offer information as to the overall behavior of the malware samples in our dataset. We have collected and plotted the Top 5 targets alongside the percentage of the *Other* targets which is the sum of the remaining targets. The set of Top 5 targets was determined by the total number of operations that targeted a unique process. To communicate our findings we have provided Figure 6.1 alongside the corresponding statistics in Table 12.16.

In the figure, we have denoted one of our findings as *firefox.exe/explorer.exe*. This is because these processes featured the same counts and percentages and we have subsequently chosen to depict them both without skewing the overall format of the figure.

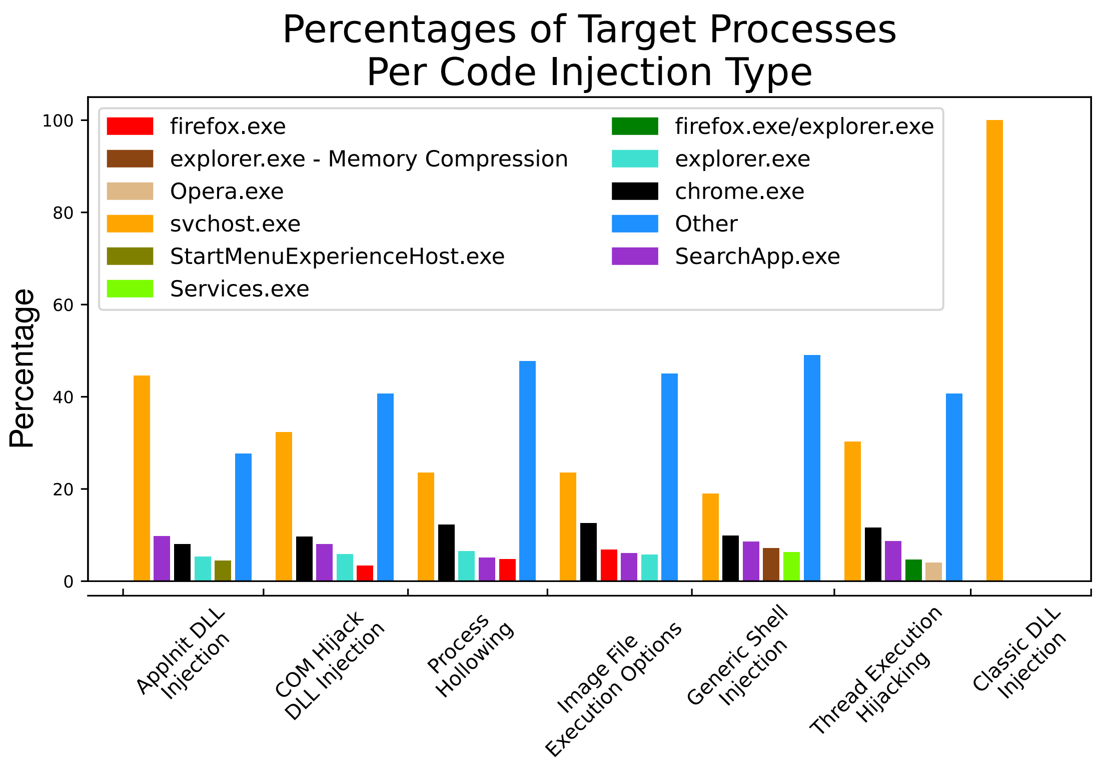

Figure 6.1: Top 5 Injected Processes Per Code Injection Type(Including ”Other”).

From the graph, we can see that from the Top 5 code injection targets *svchost.exe* is the primary target of code injection surpassing the second most targeted processes by an average of 17.91%. We identify the notable case of *AppInit DLL Injection* where the *svchost.exe* process surpasses even the percentage of *Other* processes.

Consulting with Figure 6.1 and the percentages found in Table 12.16 we conclude that all samples exhibit a strong preference towards targeting the *svchost.exe* process regardless of the code injection type being utilized. This is an interesting result, as it indicates a strong preference for a specific native process.

We can offer several hypotheses as to why this preference occurs. This can happen because the application is bound to be found within a system. Secondly, it is the target of code injection malware because, as a shared service Windows process, it is accompanied by a set of permissions that allow code injection malware to perform operations without hindrance.

Examining the aforementioned figure further, we uncover another trend. We observe that malware samples that utilize a different code injection type seem to prefer a common set of processes that they infect. We name these preferences as *attack profile* while presenting table 12.17.

The table itself features groups of code injection types, their preferred attack profile, and which processes belong to said profiles. Each attack profile is comprised of 4 target processes. The number was chosen to limit the number of attack profiles to a degree that can be easily utilized. Our work presents 3 different attack profiles. More attack profiles can be created by either reducing or increasing the number of target processes that are to be included.

Grouping target processes of malware in such a manner can assist in the categorization of malware samples in large organizations whenever a large number of machines are being targeted.

6.6 Target Process Actions per Code Injection Type

In this section, we discuss which target process actions are performed by malware samples belonging to a specific code injection type. The first subsection is devoted to discussing the details of how we processed the acquired data. After utilizing the method discussed in the previous subsection, we proceed towards summing the occurrences of these actions and pictorially present their corresponding percentages in Figure 6.3. The numerical figures of each code injection type can be found in Tables 12.19 through 12.25.

From the aforementioned resulting figure, we observe that the examined injected target processes perform a disproportionate amount of Registry Operations post code injection, followed by File, Process, and Query Operations in that order. Examining Tables 12.19 through 12.25 that the highest percentage of Registry Operations reached 85.48% and the lowest reaching 60.53%. As for File Operations, the highest observed percentage was 29.81% and the lowest was 9.14%. Process operations on the other hand reached 8.66% at its highest and 4.06% at its lowest.

We would subsequently be unable to ascertain whether these metrics themselves could be considered as an indicator of malware infection without supplementary material. Thus to circumvent this issue we examine the aggregated actions observed within the sandbox while having no malware sample loaded for analysis. We offer a pictorial depiction of our findings in Figure 6.2 and a numerical description of our findings in Table 12.18.

We observe that the majority of operations belong to the File Operations category followed by Registry Operations. Query and Process operations feature a small difference while their spread is quite noticeable from the first two columns in the figure. The remaining operations (Query, Set Information File, TCP, and UDP) have

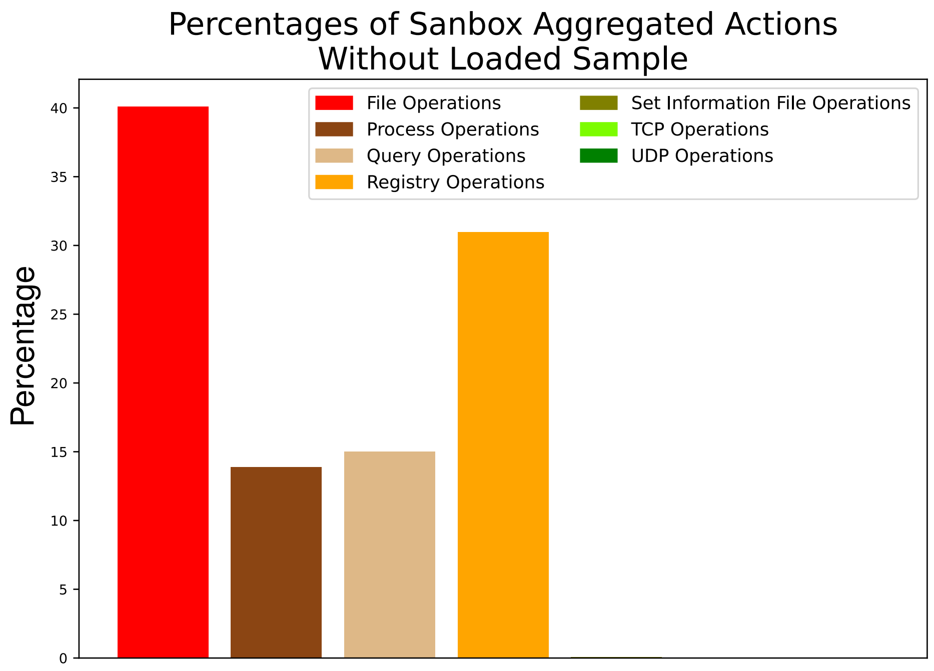

Figure 6.2: Percentages Of Sandbox Aggregated Actions With No Sample Loaded.

not been pictorially depicted due to their low percentages. Their percentages and counts are noted in the aforementioned tables.

6.7 Sample Targets per Operation Type per Code Injection Type

In this section, we examine the targets (e.g. Registry Keys, files, file system objects) of the malware sample found in our dataset. We have grouped our findings per operation. We have subsequently grouped our findings by code injection type. We have performed the grouping in this manner because it allows us to view the differences observed between injection types as well as aggregated actions.

Having examined the actions collectively taken by the samples under examination, and taking into consideration the impact that samples have on actions observed in our sandbox, the only step remaining in understanding what actions are taken during post-infection is to collect information on the targets that are observed post code injection.

Considering the nature of the results we seek to examine in the section of the report, we are devoting a subsection to each aggregated Process Monitor Action, to gain insight into the preferred targets of malware per aggregated action. In each

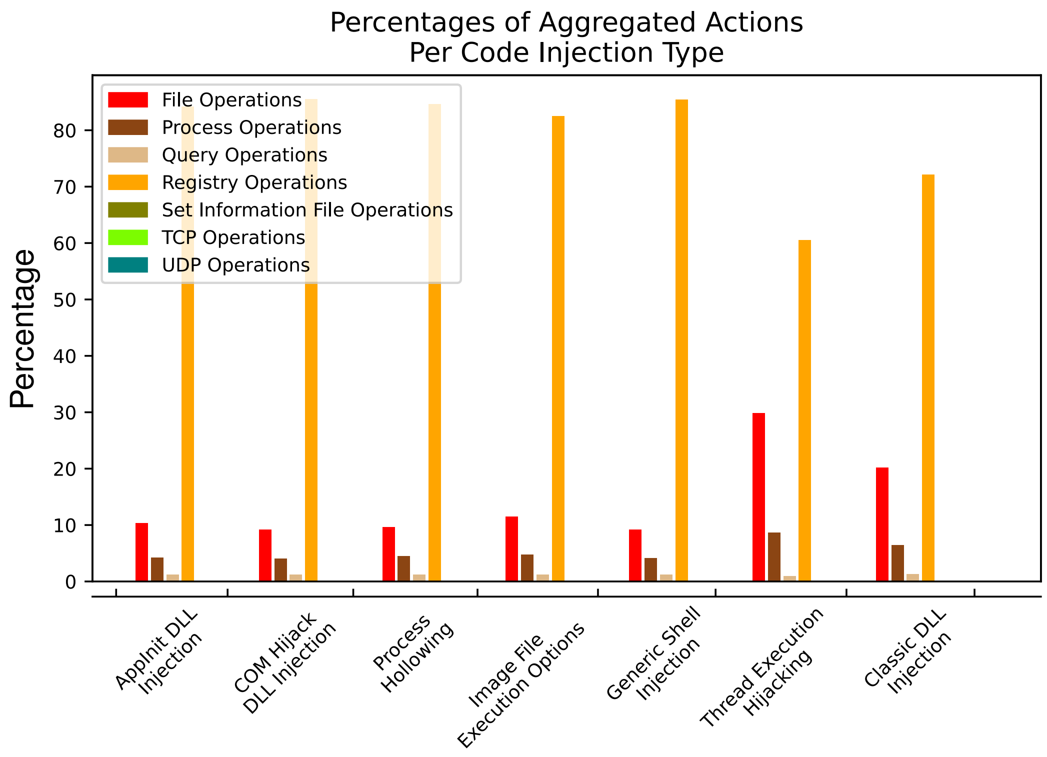

Figure 6.3: Percentages Of Process Actions Per Code Injection Type.

subsection, we offer a corresponding figure alongside a table. The figures diagrammatically depict the Top 5 most preferred targets of each operation per code injection type. The corresponding tables offer the percentages and counts of the Top 5 most preferred action targets per code injection type.

Both the figures as well as the tables provide a target under the name of *Other*. In the figures, this target is used to denote the percentage of all the other operation targets that were encountered. In the corresponding tables, this target is additionally accompanied by the sum of all encountered targets.

6.7.1 File Operations

By examining Figure 6.4 we observe a trend. The most preferred targets of applications that have undergone code injection share a common set of most preferred targets. We subsequently observed that these preferences do not only span across said targets but also across target percentages. This trend becomes apparent while we examine Table 12.26. The only observed difference is in the counts of each target. This is an interesting observation considering that the corresponding percentages portray a maximum spread of 7.86% and a minimum spread of 1.94%. It is interesting to note that all of the targets appear to be Windows files, instead of any application file. It is currently unclear whether these targets are accessed during normal system operation or if this behavior is purely malware-specific.

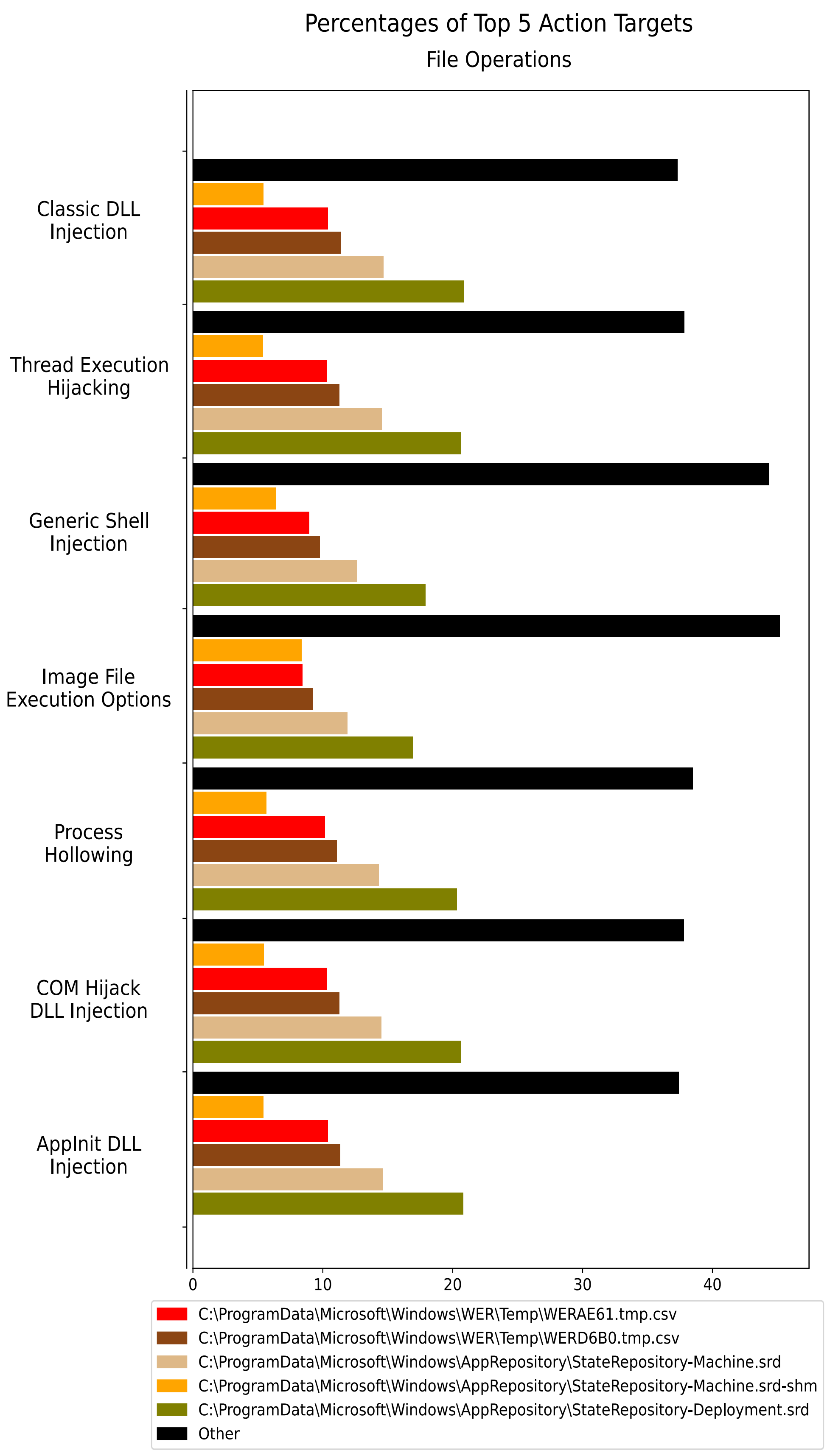

Figure 6.4: File Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.2 Process Operations

Examining Figure 6.5 and the corresponding Table 12.27 we observe that samples of our dataset do not feature a heavy preference towards a specific target. We once more observe that the list of Top 5 targets across code injection types remains the same. Their percentages however are under 0.5%, with the overwhelming majority of targets ($> 98\%$) belonging to the *Other* category. The maximum spread between percentages of the same that observed was 0.78% and the minimum 0.04%.

This may indicate that each sample target prefers its own set of targets upon which to operate, without exhibiting a large overlap between samples. In the set of Top 5 targets, however, we observe two interesting findings. We therefore first observe that two specific temporary CSV files are preferred by the samples we analyzed. These are the *WERD6B0.tmp.csv* and the *WERAE61.tmp.csv* files, which are part of Windows Error Reporting. The remaining three targets belong to *AppRepository* and are used during the installation process of applications.

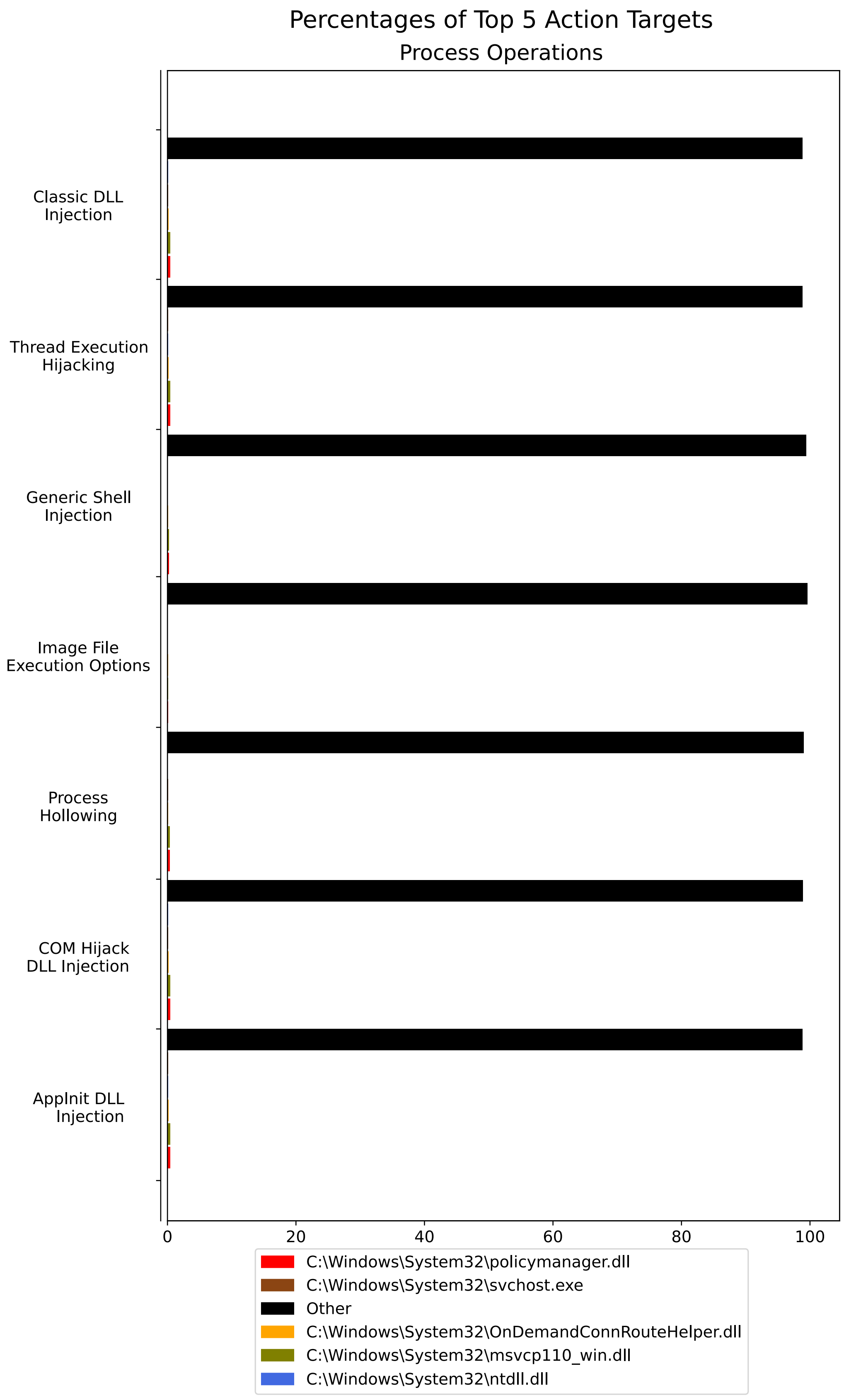

Figure 6.5: Process Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.3 Query Operations

For Query operations, we observe similarities and differences with the previously examined operation types as shown in Figure 6.6 and Table 12.28. We therefore observe that there exists a common set of Top 5 operation targets that span across the examined code injection types. The percentages of these targets do not feature major differences between types of code injection, with the greatest observed spread being $4.86\%0$ and the smallest 0.68%.

The individual percentage of each operation target remains relatively low, having *Other* portray the greatest percentage regardless of code injection type. We once again observe that the Top 5 targets are Windows file system objects, while the observed spreads are once again low. These spreads showcase a maximum of 4.86% and a minimum of 0.68%.

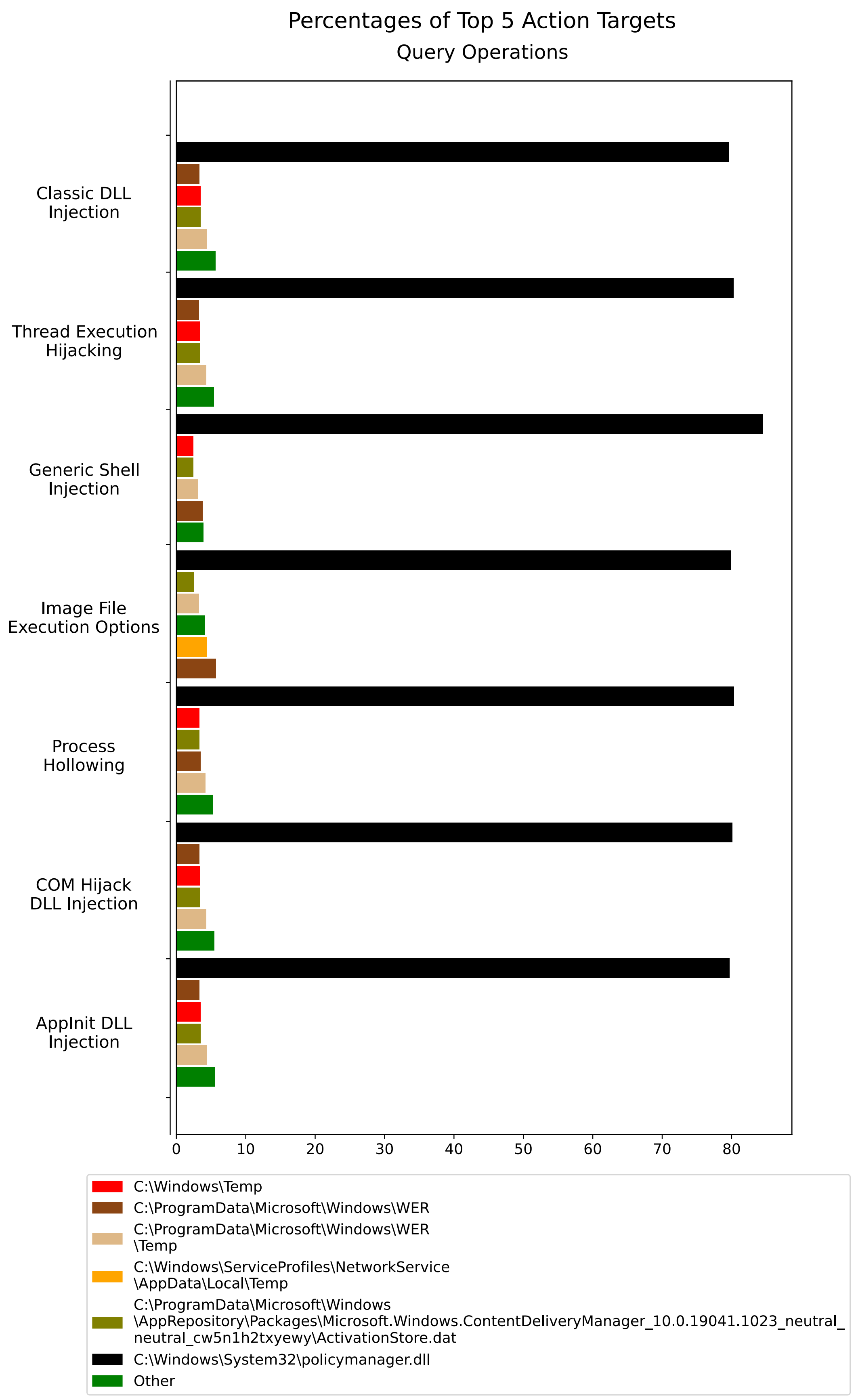

Figure 6.6: Query Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.4 Registry Operations

The previously observed trend continues with targets of Registry Operations as shown in Figure 6.7 and Table 12.29. We once again observe a common set of Top 5 targets, which are shared between the different injection types. There spreads observed between percentages of the same target of different injection types are once again small. The maximum observed spread is 3.69% with the smallest being 0.13%. The *Other* target still holds the majority reaching a maximum of 75.39% and a minimum of 71.7%.

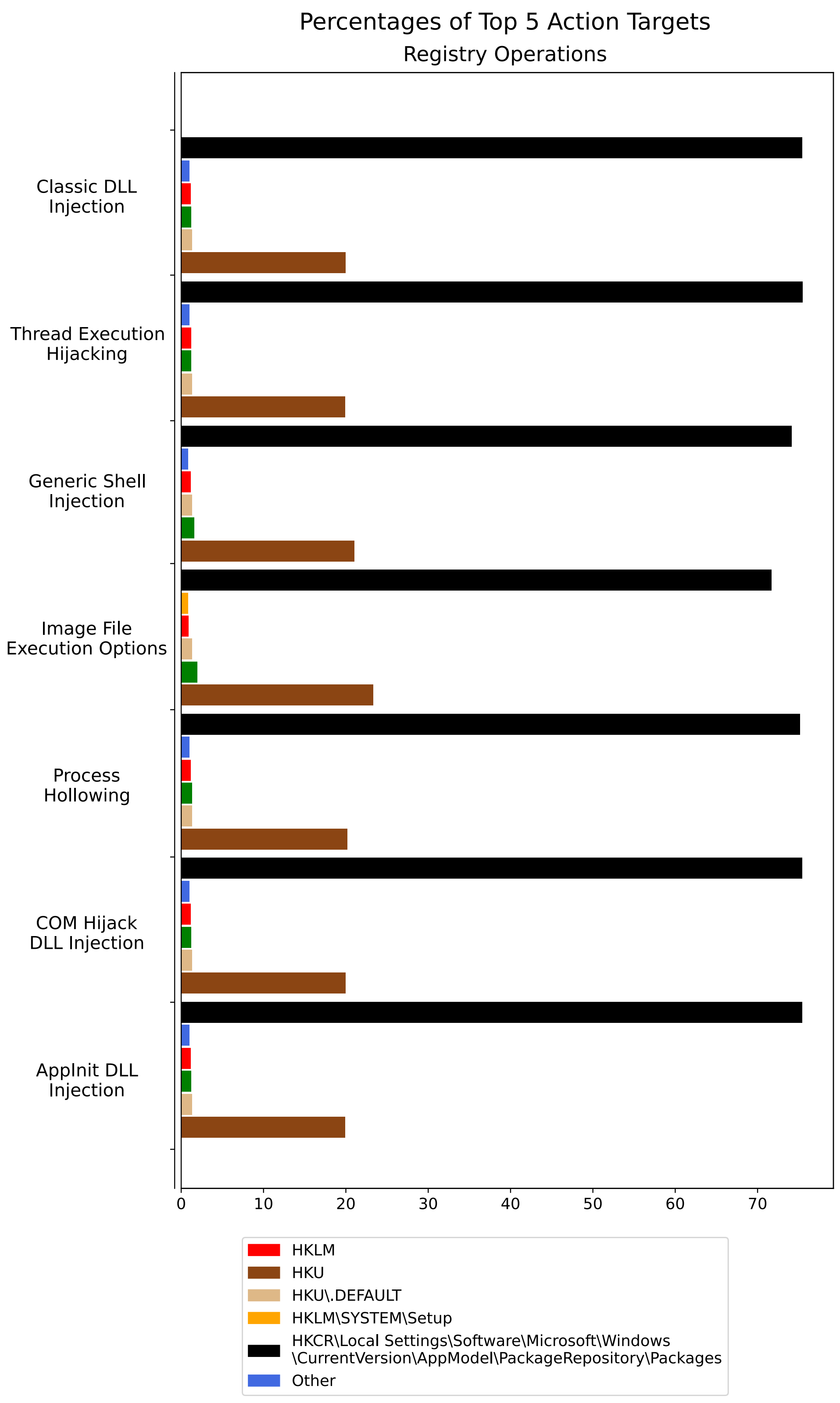

Figure 6.7: Registry Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.5 Set Information File Operations

The targets of this type of operation exhibit an interesting trend shown in Figure 6.8 and Table 12.30. Even though the counts of each target are different their percentages remain the same. We attribute this phenomenon to two causes. First, our processing of the results has rounded minor differences between the percentages of the targets. Secondly, we attribute most of these operations to operations legitimately performed by the underlying OS, its services, and operations.

This attribution however does not fully account for the differences we observe in counts of each target per code injection type. We therefore conclude that even though these figures have been influenced by the native operations of Windows, these operations still do not account for all of the behavior observed under this operation type.

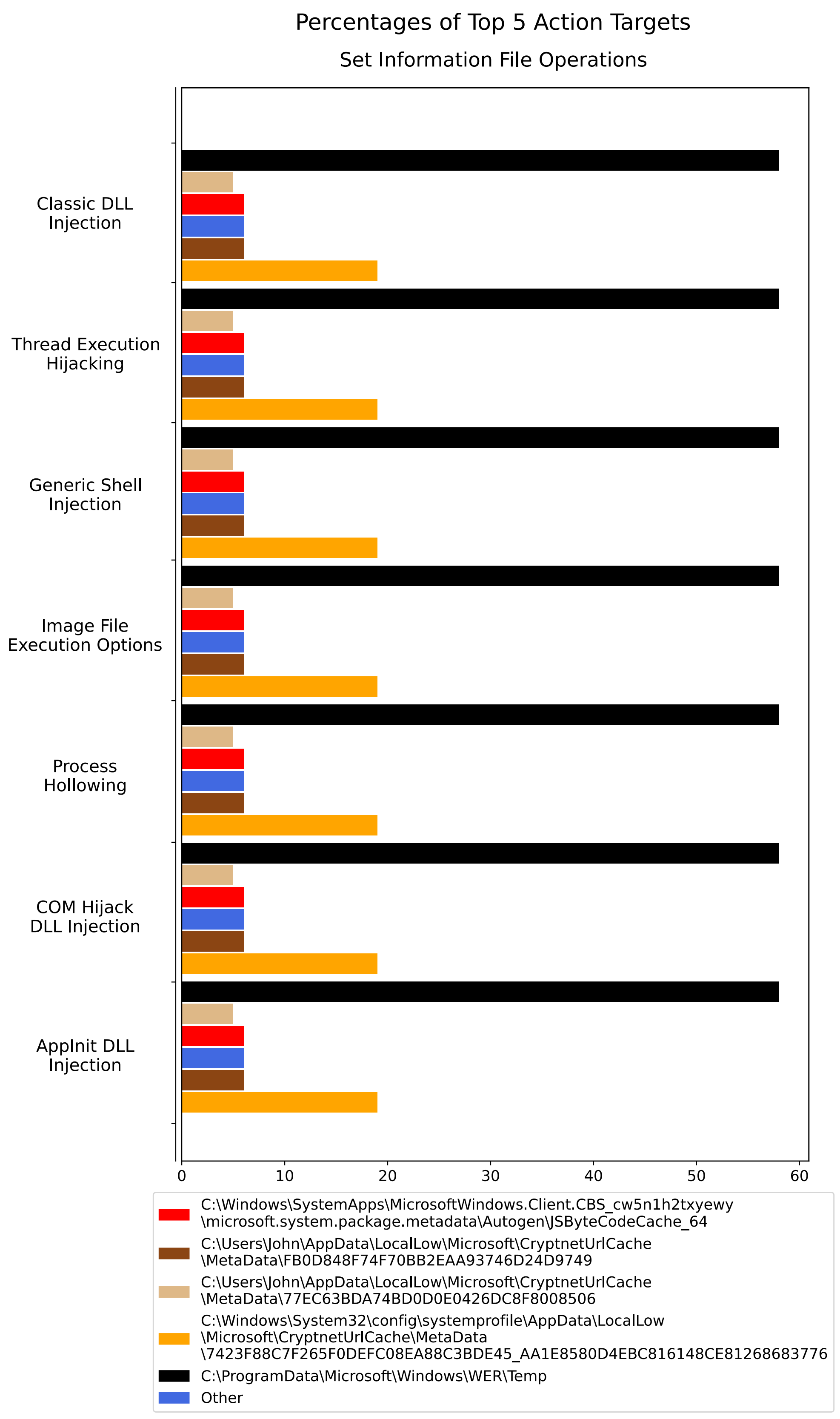

Figure 6.8: Set Information File Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.6 TCP Operations

The percentages of the targets of TCP operations have been pictorially depicted in Figure 6.9, while exact figures have been provided in Table 12.31. Within the table, we observe that the set of targets remains constant across all code injection types, with small spreads between percentages of different types. The maximum observed spread is 0.62%, while the minimum is 0.13%.

A combination of trends observed in previous operations. Maximum spread 0.62%, minimum 0.13%.

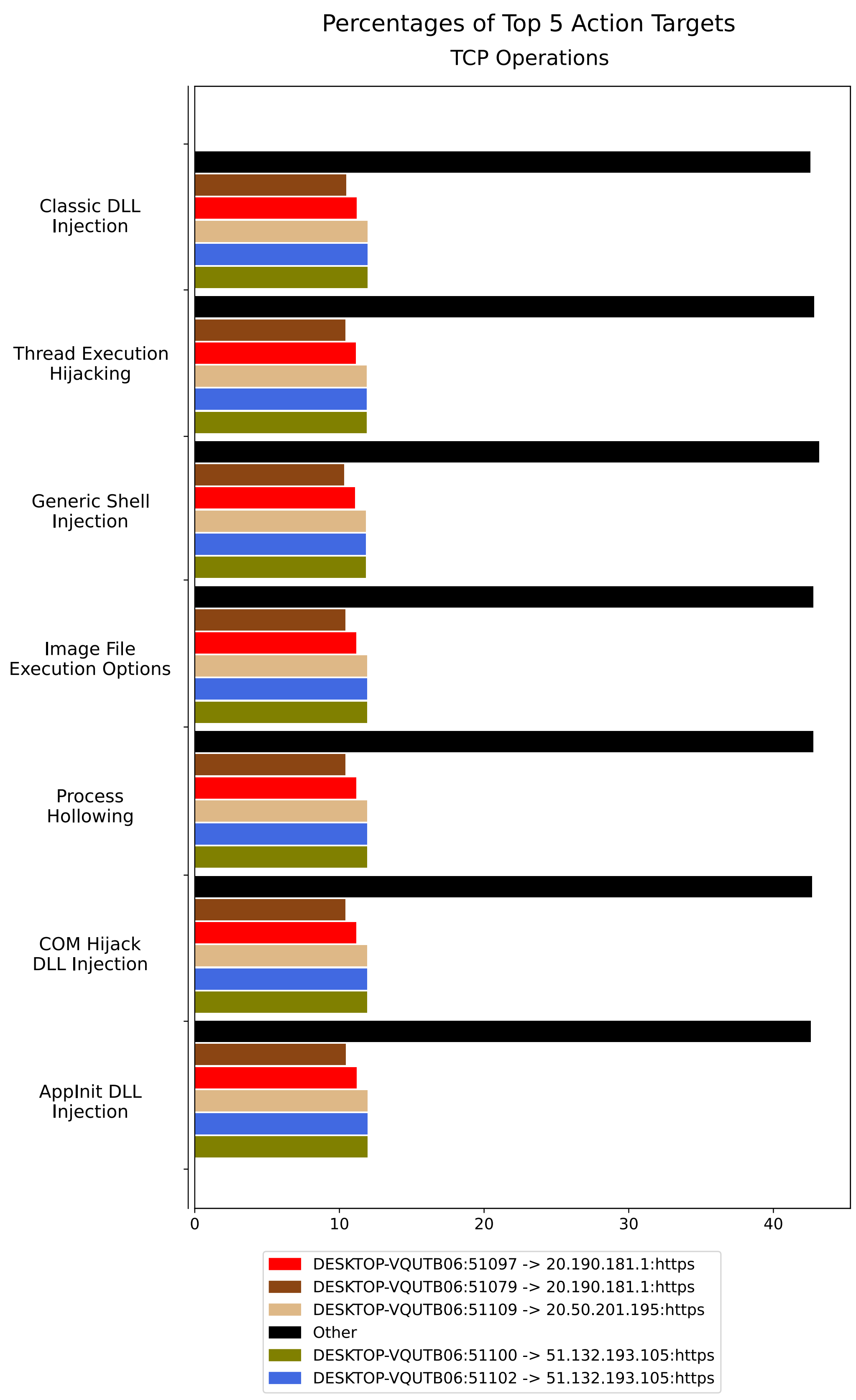

Figure 6.9: TCP Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.7.7 UDP Operations

Regarding the targets of UDP Operations, we once again observe small spreads between the percentages of the Top 5 targets. We denote that the maximum spread is 8.24% and the minimum 1.65% over a common set of targets of all injection types. These has been diagrammaticaly depicted in Figure 6.10 and numerically portrayed in Table 12.32.

Moreover, we observe that the percentages exhibited by the targets themselves remain common across all injection types with minor deviations found in the *Other* targets. These targets were discovered in the samples that performed *COM Hijack DLL Injection*, *Process Hollowing*, *Image File Execution Options* and *Generic Shell Injection*.

We note that even though the obtained percentages are common across the Top 5 targets, their total counts differ. This difference can be attributed to the differing number of samples per code injection type our dataset contains. We assume however that the common percentages can be attributed to commonalities observed between the network operations performed by the samples we analyzed. The operations could, in turn, be attributed to common frameworks utilized by the samples, like Gryphon [33]. This could explain our results, considering how all the background network noise has been removed either by filtering it or by disabling noisy services based on the suggestions of CAPE's authors.

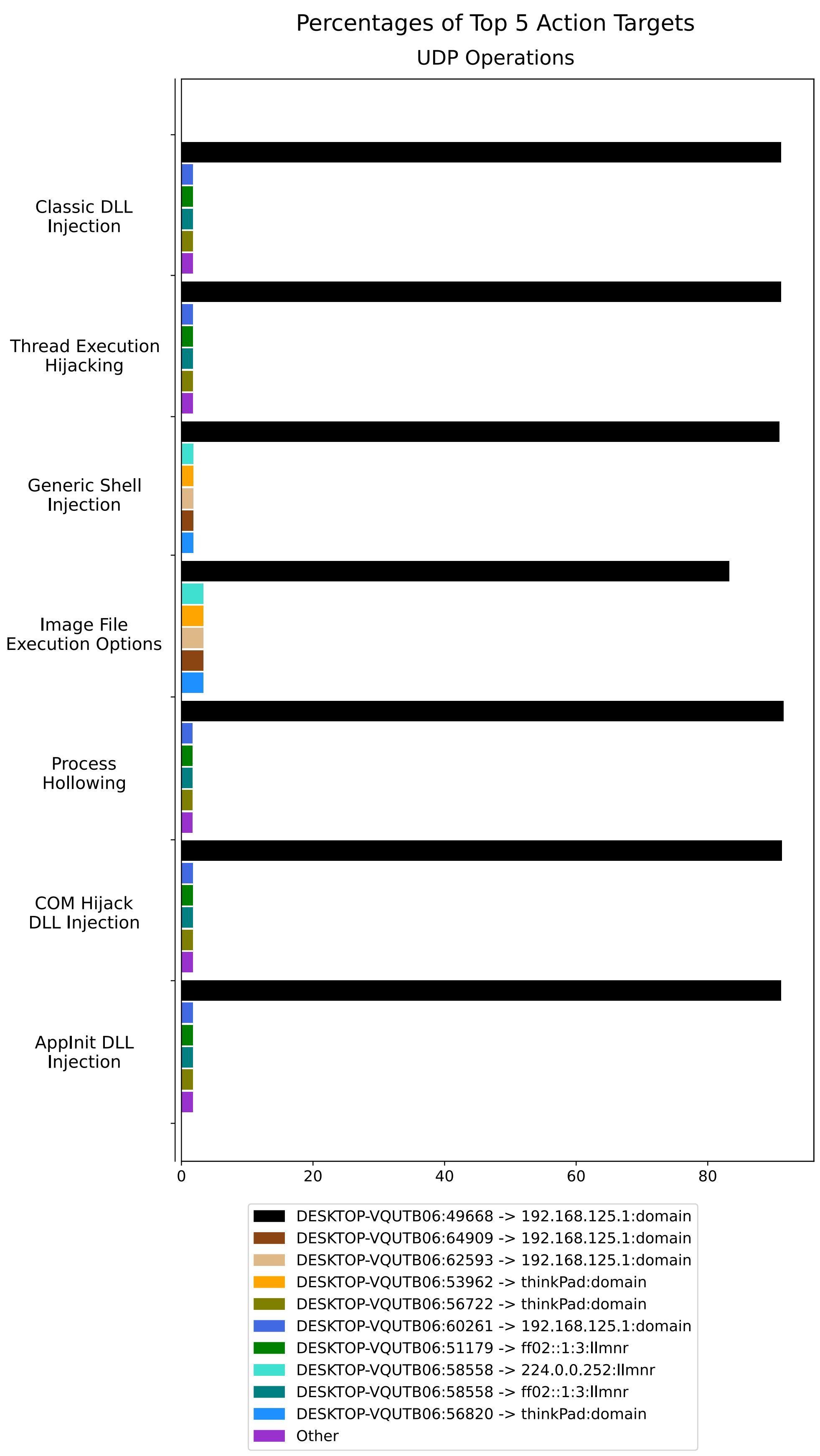

Figure 6.10: UDP Operations: Percentages of the Top 5 Malware Sample Operation Targets per Code Injection Type.

6.8 IP And Domain Addresses Metrics

Malware is also expected to perform network actions. These actions could range from connecting to a C2 server to attempting to connect to a botnet. From each CAPE we thus extract any URLs and IP addresses that were observed. We expect our Windows system to communicate with external services. Whether due to native Windows services or due to services introduced by installed software. We expect this behavior to take place even though automatic updates have been disabled and even though we have followed CAPE's documentation in disabling noisy Windows services. Due to this fact, we perform a dry run of the sandbox, analyzing a benign executable that performs no network operations. All the collected network artifacts were subsequently removed from our analysis.

We begin by collecting all of the URLs that were observed during the analysis of our samples. We move forth towards querying DNS information on each URL. The goal is to obtain CNames, NSNames, and Aliases for each URL to remove any duplicate entries. As expected, for a large number of URLs we obtain no results, having the DNS records expired. For the URLs that still show active DNS records, we collect all unique IP addresses from subsequent DNS queries. We subsequently move towards combining the resultant list with the list of observed TCP and UDP addresses to obtain geolocation information, like the corresponding country and city.

Examining Table 12.33 we observe several interesting metrics. We initially observed that the total number of unique IPs was 98, even though the number of samples that were successfully processed was 778. This large spread indicates that the servers the samples are reaching are common between a large number of samples. This is in line with the knowledge that their C2 servers are currently available as a service [34]. We also observe that though the count of unique IPs is 98 and the count of unique domain addresses is 71, the count of the unique observed countries is 10, while the count of unique cities is 39. These two metrics indicate that the servers are hosted in specific locations. Examining Table 12.34 which lists the unique countries, we discover that the majority of the samples reach servers in the USA. This indicates that any services which the samples contact may also utilize legitimate Infrastructure as a Service platforms.

To support our hypothesis, we also examine which cities the unique IPs map to. We list our findings in Table 12.35. We observe that the city to which most IPs mapped was the city of Secaucus located in the US. Performing research on whether or not the town hosts any data centers, reveals that this location alone is home to 11 data centers.

6.8.1 Metrics per Code Injection Type

Within Table 12.36 we can find various metrics per code injection type. These metrics include the number of Observed IP Addresses, the number of Observed Domain Addresses, the sum of both IP and Domain Addresses, the percentage of addresses which can be attributed to that specific code injection type, as well as the IP and Domain addresses per sample. These metrics aim to provide an understanding of the traffic uncovered for each injection type. These metrics do not give an accurate representation of the network behavior of the samples in our dataset but act as a simple indicator of what has transpired.

Examining each entry in the table, we observe distinct differences between the samples belonging to each injection type. We observe that even though *COM Hijack Injection* features the greatest number of samples (426) it does not feature the greatest number of observed addresses per sample. This type of injection portrays $47,04$ IP addresses and 1.7 Domain addresses per sample. The samples of *AppInit DLL Injection* on the other hand showcase $1,377.3$ IP addresses and 128 Domain Addresses reached per sample, surpassing all the other code injection types. Notable are the metrics of *Classic DLL Injection* due to their low counts. This is to be expected however since there was only one sample in our dataset utilizing this technique.

6.9 Summary and Key Takeaways

Within this section, we provided several metrics and results obtained from our study. Their purpose was to help us understand the objectives that were posed in the Introduction section of this study. We present a list of our key takeaways in Table 6.1.

We began our study by attempting to understand whether or not significantly differing levels of activity are exhibited by targets of code injection pre and post-exploitation. By reaching this objective we were also able to determine the levels of activity missed by currently available frameworks, that do not monitor the targets of code injection. By examining the aggregated actions per code injection type, we did indeed observe that there was a spike of activity observed within the sandbox, as shown in Figure 6.3. It is worth noting, that no CAPE signature was triggered during our analysis. We attribute this action to the fact that code injection malware does not perform any malicious activity, other than infecting any target processes.

By again referring to Figure 6.3 we observed that all samples regardless of code injection type, performed an overwhelming amount of Registry Operations. File Operations were also preferred, albeit featuring a large spread with Registry Operations. Process Operations showed a small level of activity while the rest of the operation types showed negligible results.

These results hint at a correlation with the malware families that were observed within our dataset. The majority of the samples belong to malware families which have been classified as trojans. This entails that behavior that aims to offer persistence to malicious software can be expected. This may entail a slight skewing of our results, towards behavioral patterns encountered by trojans.

We moved to examine whether specific processes are preferred over others regarding code injection malware. We observed a high preference for the *svchost* process and other Windows processes to avoid detection while taking advantage of the system-level privileges that it provides. As for *Firefox* and *Chrome*, we assume that they were targeted to have user credentials stolen from them. This behavior has already been noted in the work of Gregio et al. [35].

We finally investigated if any actions performed by the infected processes feature a preference over a specific set of targets. We observed that there was a preference for targets located under directories managed by the Windows OS, as well as for Registry keys. The directories `C:\Windows\System32\`, `C:\ProgramData\Microsoft\Windows` and `C:\Windows` appeared to be primarily affected. The registry keys *HKLM* and *HKU* were part of the primary targets of the examined samples. The samples could have attempted to gain persistence, remove evidence of their activity, or prevent the system from starting in a secure self-repairing mode [35].

Our results call for the creation of new tools that not only monitor the executable itself but also the file system, the registry, and the processes that are running within. Such tools could also monitor the system calls made by the sample under analysis. Once code injection has been recorded, they could start monitoring the aforementioned resources for any changes.

Key Research Takeaways	
Elevated activity observed post exploitation	Yes
Categories performing the most actions	Registry and File Operations
Preferred Target Processes	svchost.exe chrome.exe SearchApp.exe firefox.exe
	`C:\Windows\System32\` `C:\ProgramData\Microsoft\Windows`

Preferred Infected Process Targets Continued on next page

Table 6.1 – continued from previous page

	`C:\Windows`
	HKLM Registry Key
	HKU Registry Key

Table 6.1: Key Research Takeaways

Chapter 7

Results Discussion

Through our research, we aimed to answer several questions. We wanted to determine whether or not there is any level of activity that is being missed by code injection malware post exploitation. We moved forth to expand this question into whether or not we can observe a certain set of processes that are the preferred targets of code injection. The most common targets of these processes were also of interest since they provide more insight into what kind of activity would We expand this question into the specific level of activity that may have been exhibited, alongside the targets of said activity.

We thus started by examining whether or not code injection malware performed most of its activity post code injection. The activity we focused on and were interested in was any form of behavior that could be considered as malicious pre-exploitation. Though there are several papers that focus on determining program behavior [36]–[39], this body of work is not accompanied by practical implementations and wouldn't allow us to accurately pinpoint the nature of any malicious activity pre-exploitation.

We thus used CAPE's community-developed signatures that were checking whether or not the system calls performed by the sample under analysis matched any known signatures. Our results showed that none of the samples performed any malicious activity pre-exploitation. This however does not entail that no activity was performed whatsoever. The samples within our dataset could have for example analyzed our system for any artifacts that would indicate the use of a sandbox. Even though there were signatures that would be triggered if this were the case, we cannot exclude this possibility.

Having set a base level of activity (or lack thereof) we moved forth toward obtaining the set of actions performed by the target processes post exploitation. From the obtained data we learned that the infected processes seemed to perform a high level of File and Registry operations. Following were Query and Process Operations with a large spread from the first two. Information File and TCP/UDP operations featured negligible numbers.

We discussed that the high number of File and Registry Operations may have occurred due to the high number of samples that were labeled as trojans. These samples could have skewed the results that we received. It would be possible to accurately determine whether there is a correlation between malware family and observed behavior. This would however require a large enough dataset containing a uniform distribution of samples based on type (e.g. trojan).

We moved forth by acquiring the set of processes that were preferred by the samples in our dataset, as well as the most preferred targets of the aforementioned operations. This method proved itself very efficient regarding the determination of the type of behavior target processes exhibit while serving as an entry point and an incentive to more in-depth research.

Having determined that notable behavior was observed post-exploitation we moved forth towards investigating whether or not there were target processes that were preferred over others. From our analysis, we determined that browsers and native Windows processes were the most preferred targets (e.g. svchost). Malware is known to target Windows processes to cover their tracks and execute code with higher privileges. On the other hand, browsers can be used to mask network traffic, hide from firewalls, and steal credentials.

We subsequently obtained the most preferred targets of the target processes. We observed that a number of file system objects managed by the Windows OS alongside registry keys were common targets.

Having been armed with this knowledge, we could have taken our study a step further and extracted these files post analysis from our sandbox. These files could have been used for obfuscation, holding binary data. Once loaded into memory they could be run as an application.

As for the registry keys we could have obtained a snapshot of their values before and after a value is loaded. This would allow us to determine any changes that were made during the samples' analysis. We would also be able to gain deeper insight into the operations that fall under the Set Information File category, which currently do not feature an in-depth analysis.

In addition to the file system objects, processes appeared to be the targets of operations of targets of code injection. These actions require further analysis, since it indicates that malware may aim to propagate the infection into other targets. This could be for a number of reasons like evasion or further exploitation of the system. The aim could still however be to achieve an unknown goal. It would be intriguing to determine the nature of these processes, and from the obtained data attempt to determine whether or not these targets perform any notable activity.

Having gone through a discussion of our results, we reach a conclusion. The conclusion is that existing research efforts appear to miss considerable amounts of behavior exhibited by code injection malware. The missed behavior regards file system and registry objects, alongside other processes. This means that future research efforts need to focus towards the development of tools and frameworks that perform a holistic analysis on malware samples. This analysis should take place within a sandbox that monitors various elements of the virtualized machine. These elements should include file system objects, registry keys and of course the various processes running within the system. Further efforts could focus on using the data obtained from the aforementioned sources to construct high level behavioral descriptions of the sample under analysis.

Chapter 8

Limitations

In this section, we identify several limitations that we encountered during our experiments. The limitations that we encountered mostly focused on the accuracy and completeness of the results we were able to obtain. We begin by identifying hardware limitations that were present in our study. Having only one physical machine available which was not designed for hosting multiple virtual machines in parallel we could only make one CAPE instance.

This meant that we were unable to examine our dataset under different OS versions and different machine configurations since this would result in major increases during the data collection phase. Having only one physical machine available (which was intended for personal use) also meant that the sandbox that was utilized to obtain the event traces that were used to classify the available malware samples could not be used.

This is attributed to the fact that Drakvuf is a bare metal hypervisor, having its current version does not feature adequate support for dual boot systems. Even though it utilizes the Linux kernel it only offers an interface intended for running sandbox samples. This therefore prohibits utilizing the system on a personal physical machine.

We of course cannot disregard the fact that a large number of URLs for which we attempted to resolve into IP addresses did not offer any results. We attributed this phenomenon to dead hosts while supplementing this observation with the possibility that the examined samples could have altered their behavior due to this fact. Malware is known to utilize C2 Servers, colloquially known as Command and Control Servers [40]. Since we can argue that Wireshark could have been utilized to collect the network trace of the sandbox for further analysis, we designate the lack of network analysis as a limitation of our study.

These command centers are used to issue commands to malware samples and collect information. Having insight in mind it is safe to theorize that we were not able to capture the full range of actions of the malware samples we examined. The behavior that the available samples exhibited though can also have been altered by versions of the target processes available to them.

Malware tends to exploit flaws found within applications. Authors of target processes strive to patch their software once such flaws are uncovered and become known to them. Thus different versions of target processes can expose different attack vectors. Having only limited information as to which application each available sample targets, while having to uncover the target processes ourselves we cannot exclude the possibility that a number of the samples which we examined did not have available the target application version which they intended to infect. This could in turn have influenced our results.

8.1 CAPE and Drakvuf Sandboxes

A key limitation we must recognize in our study is the use of the Drakvuf syscall traces for determining the code injection timestamp, and the use of CAPE for determining the target of code injection and the corresponding actions post-code injection. This experimental design choice is expected to have an undetermined impact on the accuracy of the obtained results. This section is devoted to providing an in-depth understanding of this limitation. Similarly section 4.13 is dedicated to discussing the topic from a research methodology perspective.

Drakvuf as a sandbox utilizes a bare metal hypervisor making use of hardware virtualization to provide a sandboxing environment. In turn, this entails that the sandboxed OS and any process running within it can detect the underlying hardware used to host the sandbox itself. It is accompanied by a set of plugins that allow an in-depth examination of a malware sample. An example of the functionality exposed by the plugins is the ability to obtain any system calls, object creation and file manipulation actions performed by a sample. Running as a bare metal hypervisor however allows Drakvuf to be a stealthy analysis option, since any examined malware sample will not be able to detect the underlying sandbox. The drawback of this approach is the fact that this system can only run only Intel CPUs since it requires a specific set of virtualization features.

CAPE utilizes KVM to achieve virtualization. KVM uses the QEMU Emulator that supports a range of hardware. This entails that the system can be used under a large range of configurations, allowing an analyst to examine a sample under a carefully specified hardware configuration. If the suggestion of Rossow et al. [41] is to be followed, to uncover the full range of behavior a sample needs to be executed on different machines. This is further supported by Avllazagaj et al. [26] which discovered that a malware sample's behavior changes based on its execution environment. The feature of multiple configurations can thus be advantageous to an analyst if the full range of a malware sample's behavior is to be examined. Unlike Drakvuf, CAPE cannot peer into a VM's functioning. Therefore instead of plugins it makes use of an agent installed within the analysis machine while offering pattern-matching signatures and integration with other analysis tools (e.g. Yara).

8.1.1 Injection Timestamp Accuracy

As mentioned in 4.13 utilizing CAPE over Drakvuf will exhibit differences. These differences are expected to take place since we are running our samples within a different execution environment. We have obtained the same libraries and SDKs that were utilized by Starink [42] during his experiments. However, we can still locate differences in the version of Windows that were used between the two studies. Further differences can be expected in the underlying hardware that composed each sandbox host.

All of these facts paint the picture of a different execution environment. This entails that differences are expected to occur in the execution of each sample under analysis. What we expect is each sample performing the same type of code injection, with differences observed in the sequence of system calls utilized for this process. Even though these differences were accounted as discussed in 4.7, we still need to examine any potential impact on the outcome of our research.

The aforementioned section details how we obtained the average time difference between the system call traces of Drakvuf and CAPE and how we accounted for this within our measurements. As the section mentions however, we did not account for every sample's time difference. This entails that there would be some target process actions that either escaped examination or were considered part of the infected process actions. However are not concerned with this. This is because the average observed difference was just a few microseconds. Factoring in that the bulk of the behavior observed by the target processes took place post-code injection, and that our study's research questions have been adequately answered, we argue that accuracy was an existing though not a significant limitation.

8.1.2 Identified CAPE Sandbox Limitations

Having provided a basic high-level description of the two analysis systems we can move forth with discussing any possible limitations that we have identified. We also examine whether or not these limitations can affect our research and to which degree. These limitations mostly revolve around the possibility of CAPE being detected by a sample under analysis due to it's use of kernel-based virtualization The action of attempting to detect an analysis environment is by itself known as *fingerprinting* [43].

Drakvuf on the other hand which uses bare-metal virtualization which utilizes a bare-metal hypervisor and does not suffer from these issues. We discuss the solutions that we ourselves have implemented, and argue as to the degree that our research may have been affected.

CAPE requires an agent to be installed with the sandboxed OS. The agent itself is a Python script which spins up a local HTTP server. There are a number of ways that a sample under analysis can identify the said agent is running. First and foremost it can check whether or not there is a server listening on the URL where the agent is expected to be listening in. This face has already been accounted for by the authors of CAPE. This means that the agent does not respond to any requests made from the host itself. There are however other ways the agent can be detected.

The Windows OS requires the agent to be placed in a specific directory to run every time the underlying VM is created. This entails that a sample can check the contents of the directory for any Python scripts that match certain criteria. The first and foremost criterion would be to check if a script is named *agent.py*. As a first line of defense, we have given the script a random name. Provided that a script does exist, there are other conditions a sample can test for. It can for example check whether the hash of the file's contents match a known hash, whether the file contains a set of methods expected to be found within the agent, or if there is any suspicious HTTP traffic that the script performs. Considering this, we have taken action to safeguard against these checks by changing the name of the methods. This would change the hash of the contents and fail any checks performed on the methods existing within the file. We would however be unable to hide any observed HTTP packets intercepted by the sample under analysis.

The transmitted data however can be obfuscated, thus failing any checks performed by the sample under analysis. A sample however, could decide to cease its functioning if it detects HTTP traffic. Not having the capability to fully conceal the agent's HTTP traffic, a sample may choose to take evasive action. This evasive action can take a number of forms, from simply avoiding to infect a target process and remaining inactive to completely altering the behavior that it intends to exhibit once it has infected a target. We must however recognize that this method of sandbox detection is not the most robust. Firstly, as we discussed in the previous paragraph there are ways to obfuscate the existence of the agent. Secondly, the defensive option that we listed in the previous paragraph could be in place. Thirdly, provided that the aforementioned anti-evasion methods are in place, one cannot exclude the possibility that any observed HTTP traffic can be attributed to the machine interacting with other services in the same network.

Considering the above, we need to indeed recognize that the existence of the agent could act as a limitation to our experiments, provided that no defensive measures where taken. Considering however that defensive measures are in place, we cannot regard the agent as a limitation. Once could of course argue that the existence of the agent could be thoroughly hidden by creating a pipe between the sandbox and the host OS. This action however would introduce new emulated hardware components. These components could result in the sandbox being detected.

The sandbox could also be detected through the hardware implementation artifacts. These artifacts include abnormal values exhibited by the emulated hardware components like Device IDs or the size of any available storage mediums [44], [45]. Equipping the underlying VM with persistent storage and RAM that feature realistic storage values is a straightforward task, which only requires a simple mention.

Other defensive measures however, like setting appropriate Device IDs and component Serial Numbers require more attention. We therefore follow the instructions laid out by one of CAPE's authors [46]. These instructions provide a set of steps for configuring QEMU. These steps allowed us to set realistic values for any hardware components -obtained through a Google search- while hiding any VM Hypervisor artifacts that could be exposed through the VM. Moreover, following the aforementioned instructions has allowed us to also hide any abnormal processor flags set by QEMU, that would betray the existence of CAPE. We cannot however disregard the chance that any of the obtained samples may have queried for the existence of more miscellaneous components, like a CPU temperature sensor. CAPE's underlying hardware emulation provider, QEMU offers the capability to create such devices, which will be querying values from the actual underlying machine as shown here [47].

These actions may in the end not be enough to hide the existence of emulated hardware, since the exposed ACPI tables may contain abnormal values. We do not concern ourselves however with this possibility. This is because CAPE's documentation offers instructions on how to export the ACPI tables of the host machine, only for them to be utilized by QEMU and subsequently by the guest VM.

Based on this information we can argue that even though CAPE is using virtualization, the probability of our sandbox being detected by a sample under analysis should be low.

It is also possible that the VM's Hypervisor could be detected based on values of the Windows Registry [44], [48]. These values could contain key-value pairs that are either specific to the underlying virtualization technology or outright contain the name of the underlying virtualization system. Searching through the Windows Registry of the virtualized machine did not reveal any entries that would expose the fact that a sample is being analyzed.

One of course cannot disregard the practice of malware attempting to detect the presence of a debugger [43]. CAPE is accompanied by it's own debugger, which is designed in a manner which avoids the most commonly known debugger detection techniques [49]. The tool however exhibits the lack of proper NT API Hooking [45]. This is because CAPE's debugger incorrectly hooks NT API functions, utilizing an incorrect number of arguments. A sample checking for any such hooks would be able to detect the sandbox's presence. This can lead to a sample altering it's behavior in order to avoid detection. Not having utilized CAPE's debugger however, this limitation is not applicable in our case.

It would be possible for a sample to use *Time Based* techniques, having a number of them available. A sample can for example determine the difference between the time elapsed within a sandbox and the time elapsed between two NTP timestamps [50], or determine the system's uptime. Regarding evasion, malware samples can delay their execution for a few minutes to avoid being analyzed or base their activation on specific events.

Some type of time-based sandbox detection are easy to avoid and have been avoided in our experiments. Any kind of check that would check whether the uptime of the system would be longer than a few minutes (to account for the recently started sandbox) would succeed and trigger the sample's execution. This is because the virtual machine that comprises our sandbox is started from a saved machine state. This restores the machine to a state that it found itself in before it was saved and subsequently shut down. Techniques like querying an NTP server are not considered applicable in our case. This however does not entail that the sample cannot query an online service to the elapsed time, and compare the received response with the time elapsed within the sandbox. To counter this CAPE speeds up time within the VM counter any delays induced due to virtualization.

Regarding evasion, a malware sample may attempt to evade analysis by delaying its execution [44], [51]. The simplest technique that can be utilized to counter this technique would be for the sandbox to speed up the virtualized machine's time. Such an action however would leave the sandbox open to detection. This issue is not only encountered in CAPE but also Drakvuf. Therefore, the only counteraction would be to allow a sample to run for long intervals of time. Such a practice, however, presents itself with several challenges. A sample could for example remain benign until an unknown condition is being met, like being contacted by a C2 server. Or a sample may not be triggered at all due to unmet activation conditions.

As we have seen however in the work of Kp̈chler et. al. [52] malware exhibits most of its behavior within the first 5 minutes of execution. In light of this fact, we have foregone examining the samples for more than 5 minutes. Considering how the goal of our research was not to determine as to whether or not time based evasion was in effect, we cannot consider it's possibility as a limiting factor. This of course does not imply that no sample performed this technique, since we could not determine whether or not our sandbox was detected.

A sample under analysis may also attempt to verify as to whether or not the machine finds itself in a realistic environment. This fact has been accounted and countered for by creating user artifacts within the VM. These artifacts include but not limited to user files with spoofed metadata to indicate that they were created some time ago, various applications that were utilized before any experiments take place, and account credentials for social platforms saved in Chrome's password manager, accompanied by the corresponding Internet history.

We cannot however fully disregard the fact that different systems were used during the process of determining the code injection type, and of obtaining the actions of the target process of code injection. These different systems are bound to hold different user artifacts and configurations which may in turn affect the behavior our samples. Considering how both systems however have been equipped with a wealth of applications suitable for code injection, the expected change in behavior would mostly stem from any differences in OS version or available SDKs. We however have taken action to bridge those differences by installing the same SDKs that were available on the Drakvuf instance used to determine the code injection type.

Regardless of all the steps taken, we cannot exclude the possibility that any number of the samples in our dataset may successfully detect and attempt to evade CAPE's analysis. We are however not concerned with a sample attempting to obfuscate it's behavior from CAPE to a large degree. This would be because our work utilized Process Monitor exclusively. Our literature research did not indicate that currently available malware samples are verifying the existence of Process Monitor within the sandbox. This of course does not imply that such detection did not take place.

To conclude, in this section we discussed several possible limitations regarding our choice to utilize CAPE over Drakvuf. We examined any stealth considerations resulting from the different virtualization technologies the sandboxes offer and any potential accuracy issues that may have arisen. We also touched upon the accuracy of our results. Out of all the examined limitations, we find NT API Hooking the one that could not be adequately dealt with. This limitation could lead our samples to perform actions that deviate from what could be considered their normal operation. These deviations can take a number of forms, ranging from exiting without taking any action to attempting to attack the VM itself.

Provided that this is the case, and considering how we equipped CAPE with the same runtime environment and SDKS as the Drakvuf sandbox that was used to obtain the injection timestamp we reach a conclusion. Amongst all of the limitations that we faced, this is the most impacting one. Assuming that our sandbox was indeed detected by some of the samples, we theorize that this would be the reason for our sandbox failing to analyze several samples from our dataset.

Chapter 9

Future Work

In our future work, we identify several experiments that we can conduct to enhance our understanding of how what kind of behavior is being missed by current dynamic analysis systems. We begin by denoting that our experiments could also be conducted by having an Antivirus solution installed in our sandbox. Malware is known to take evasive action in the presence of AntiVirus programs, often rendering their use inapplicable as shown by Thamsirarak et al. [53]. We would thus also be interested in examining how this set of operations changes when malware finds itself under the presence of an AV solution.

We would also be interested in examining the malware under different configurations. These configurations include hardware, OS version, available target application versions as well as usage artifacts. The goal would be to determine whether significant differences can be observed in the execution of samples under examination. The goal of this would be to understand to what extent the behavior changes as the configuration changes.

Finally, we would be interested in examining our dataset samples under different network conditions as well as a different network topology. This would take place in tandem with performing an analysis of the generated traffic. Different network conditions can include an isolated network (examining samples with no internet connectivity), a spoofed sandbox IP as well several virtual network hosts. The spoofed sandbox IP would be utilized to determine whether the sample under examination attempts to reach different network targets based on its geographical location. Having an additional number of network hosts would serve the purpose of gaining insight as to whether the target application could be used as an infection vector, targeting adjacent network hosts.

Chapter 10

Related Work

The task of examining the behavior of a malicious or infected application is not novel. Beginning with methods of static analysis we encounter system call and byte-sequence analysis at the forefront. These methods are currently being used to extrapolate properties of malware [54] and determine potential malicious intent [55]. These methods however require a corpus of information that needs to be constantly kept updated and can suffer from false positives.

Considering kernel and system states are behavioral data, non-transient state changes potentially triggered by malware were studied in [37]. Under the same concept, K-Tracer was developed which makes use of kernel data in an attempt to identify unwanted manipulations of sensitive data [36].

One cannot of course neglect mentioning the use of API call sequence information. Two methods are primarily used, the N-Gram Approach [56] and the Graph Approach [57]. The N-gram approach is characterized by two methods, namely Op-Code n-gram [58] and System Call n-gram [59], both of which extract the so-called byte sequence n-gram [60], in a similar fashion to how other AI solutions work. Like any other method, they exhibit drawbacks like poor interpretability, loss of information, and the ”curse of dimensionality” which entails that more and more data are required as the number of dimensions under study increases. Theory suggests that these drawbacks are present because the API Arguments are not taken into consideration [61].

Graph approaches have their divisions, namely API Call Graph [57] and Op-Code [62]. Unlike the N-Gram approach, Graph approaches take into consideration the API arguments to deal with the corresponding drawbacks. These methods place focus on said arguments since API output arguments can often be utilized as inputs to other calls, thus allowing one to obtain a more concrete specification of real malware behavior [61]. The results of this process are either Contract Subgraphs [57] or Significant Graphs [63], both generated with the explicit purpose of detecting malicious behavioral patterns from malware as well as benign applications.

Chapter 11

Conclusion

Through our measurements we have uncovered that code-injection malware portray variable behavior. This behavior is primarily exhibited after infection of a target process has taken place. This hinders any current dynamic analysis solutions from adequately understanding which actions are performed by samples under analysis. This fact has been supported by our experiments, where CAPE's behavior signatures were not triggered.

We observed that there was a common set of preferred targeted processes, across the examined code injection types. This trend was followed by a preferred set of operation targets. Said targets appeared to be file system objects of the Windows system.

We also observed that a number of network addresses (IP, UDP, TCP, and Domain) were reached by the samples we analyzed. These addresses spanned multiple countries and appeared to be pointing towards services available to malware authors.

Bibliography

[1] Abukar, Yahye and Maarof, Mohd and Hassan, Fuad and Mohamed, Abshir, “Survey of keylogger technologies,” *I nternational J ournal of C omputer S cience and T elecommunications*, vol. 5, pp. 25–31, 02 2014.

[2] H. Orman, “The morris worm: a fifteen-year perspective,” *IEEE Security Privacy*, vol. 1, no. 5, pp. 35–43, 2003.

[3] AV-Test. Malware. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[4] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, “Bee master: Detecting host-based code injection attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, S. Dietrich, Ed. Cham: Springer International Publishing, 2014, pp. 235–254.

[5] T. Barabosch and E. Gerhards-Padilla, “Host-based code injection attacks: A popular technique used by malware,” *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pp. 8–17, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:24022

[6] A. Petrosyan. Malware - statistics facts. [Online]. Available: https://www.statista.com/topics/8338/malware/#topicOverview

[7] J. Starink, M. Huisman, A. Peter, and A. Continella, “Understanding and measuring inter-process code injection in windows malware,” in *Proceedings of the International Conference on Security and Privacy in Communication Networks (SecureComm 2023)*, Oct. 2023, 19th International Conference on Security and Privacy in Communication Networks, SecureComm 2023, SecureComm ; Conference date: 19-10-2023 Through 21-10-2023.

[8] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, “Dynamic malware analysis in the modern era—a state of the art survey,” *ACM Comput. Surv.*, vol. 52, no. 5, sep 2019. [Online]. Available: https://doi.org/10.1145/3329786

[9] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, “Dynamic analysis of malicious code,” *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug 2006. [Online]. Available: https://doi.org/10.1007/s11416-006-0012-2

[10] A. Vasudevan, “Maltrak: Tracking and eliminating unknown malware,” 12 2008, pp. 311–321.

[11] A. Vasudevan and R. Yerraballi, “Cobra: Fine-grained malware analysis using stealth localized-executions,” vol. 2006, 06 2006, pp. 15 pp.–.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, Z. Lintao, and P. Barham, “Vigilante: End-to-end containment of internet worm epidemics,” *ACM Trans. Comput. Syst.*, vol. 26, 12 2008.

[13] J. Rhee, R. Riley, D. Xu, and X. Jiang, “Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory,” in *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 178–197.

[14] C. Gorecki, F. Freiling, M. Kührer, and T. Holz, “Trumanbox: Improving dynamic malware analysis by emulating the internet,” vol. 6976, 10 2011, pp. 208–222.

[15] X. Wang and R. Karri, “Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters,” in *Proceedings of the 50th Annual Design Automation Conference, DAC 2013*, ser. Proceedings - Design Automation Conference, 2013, 50th Annual Design Automation Conference, DAC 2013 ; Conference date: 29-05-2013 Through 07-06-2013.

[16] D. Ray and J. Ligatti, “Defining code-injection attacks,” *SIGPLAN Not.*, vol. 47, no. 1, p. 179–190, jan 2012. [Online]. Available: https://doi.org/10.1145/2103621.2103678

[17] J. A. Morales, E. Kartaltepe, S. Xu, and R. Sandhu, “Symptoms-based detection of bot processes,” in *Computer Network Security*, I. Kotenko and V. Skormin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 229–241.

[18] R. Cieślak. (2023) Dynamic linker tricks: Using ld$_preloadtocheat, injectfeaturesandinvestigateprograms.https : //rafalcieslak.wordpress.com/2013/04/02/dynamic-linker-tricks-using-ld_preload-to-cheat-inject-features-and-investigate-programs/.$

[19] J. Andress and S. Winterfeld, “Chapter 12 - non-state actors in computer network operations,” in *Cyber Warfare (Second Edition)*, second edition ed., J. Andress and

S. Winterfeld, Eds. Boston: Syngress, 2014, pp. 207–219. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012416672100012X

[20] C. Ravi and R. Manoharan, “Article: Malware detection using windows api sequence and machine learning,” *International Journal of Computer Applications*, vol. 43, no. 17, pp. 12–16, April 2012, full text available.

[21] T. Barabosch and E. Gerhards-Padilla, “Host-based code injection attacks: A popular technique used by malware,” in *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, 2014, pp. 8–17.

[22] H. A. Noman and O. M. F. Abu-Sharkh, “Code injection attacks in wireless-based internet of things (iot): A comprehensive review and practical implementations,” *Sensors*, vol. 23, no. 13, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/13/6067

[23] C. Sandbox. (2023) Cape sandbox. https://capesandbox.com/.

[24] T. K. Lengyel. (2017, June) Syscalls plugin documentation. https://github.com/tklengyel/drakvuf/wiki/DRAKVUF-Plugin-Documentationsyscalls.

[25] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 386–395. [Online]. Available: https://doi.org/10.1145/2664243.2664252

[26] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitras, “When malware changed its mind: An empirical study of variable program behaviors in the real world,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3487–3504. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/avllazagaj

[27] Microsoft. (2023, September) Process monitor v3.96. https://learn.microsoft.com/nl-nl/sysinternals/downloads/procmon.

[28] Linux-KVM. (2023) Kvm. https://linux-kvm.org/page/Main$_{Page}$.

[29] s. p. malicialab, rscampos. (2024) Avclass. https://github.com/malicialab/avclass/tree/master.

[30] S. O’Shaughnessy and F. Breitinger, “Malware family classification via efficient huffman features,” *Forensic Science International: Digital Investigation*, vol. 37, p. 301192, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721001001

[31] C. Doman. Malware families in otx. [Online]. Available: https://gist.github.com/chrisdoman/299961ba9c590c1f6b39487594e7f2a7

[32] M. Bazaar. Malware bazaar - cosmu. [Online]. Available: https://gist.github.com/chrisdoman/299961ba9c590c1f6b39487594e7f2a7

[33] whiterabb17. Gryphon. [Online]. Available: https://github.com/whiterabb17/gryphon

[34] C. Talos. Attackers leveraging dark utilities ”c2aas” platform in malware campaigns. https://pentestlab.blog/2020/05/20/persistence-com-hijacking/.

[35] A. R. A. Grégio, V. M. Afonso, D. S. F. Filho, P. L. d. Geus, and M. Jino, “Toward a Taxonomy of Malware Behaviors,” *The Computer Journal*, vol. 58, no. 10, pp. 2758–2777, 07 2015. [Online]. Available: https://doi.org/10.1093/comjnl/bxv047

[36] A. Lanzi, M. I. Sharif, and W. Lee, “K-tracer: A system for extracting kernel malware behavior,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, 2009. [Online]. Available: https://www.ndss-symposium.org/ndss2009/k-tracer-system-extracting-kernel-malware-behavior/

[37] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, “Automated classification and analysis of internet malware,” in *Recent Advances in Intrusion Detection*, C. Kruegel, R. Lippmann, and A. Clark, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 178–197.

[38] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *2007 IEEE Symposium on Security and Privacy (SP ’07)*, 2007, pp. 231–245.

[39] Y. Fratantonio, A. Bianchi, W. K. Robertson, E. Kirda, C. Krügel, and G. Vigna, “Triggerscope: Towards detecting logic bombs in android applications,” *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 377–396, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:2858111

[40] J. Gardiner, M. Cova, and S. Nagaraja, “Command control: Understanding, denying and detecting - a review of malware c2 techniques, detection and defences,” 2015.

[41] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen, “Prudent practices for designing malware experiments: Status quo and outlook,” in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 65–79.

[42] J. Starink, “Analysis and automated detection of host-based code injection techniques in malware,” September 2021. [Online]. Available: http://essay.utwente.nl/88617/

[43] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, “Malware dynamic analysis evasion techniques: A survey,” *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3365001

[44] V. Ray. Malware sandbox evasion techniques: All you need to know. [Online]. Available: https://www.vmray.com/sandbox-evasion-techniques/

[45] ——. Invisible sandbox evasion. [Online]. Available: https://research.checkpoint.com/2022/invisible-cuckoo-cape-sandbox-evasion/

[46] D00m3dr4v3n. Modifying kvm (qemu-kvm) settings for malware analysis. [Online]. Available: https://www.doomedraven.com/2016/05/kvm.html#modifying-kvm-qemu-kvm-settings-for-malware-analysis

[47] jonte. Emulating a tmp105 temperature sensor using qemu and linux. [Online]. Available: https://gist.github.com/jonte/b4bd83a5f2e8330418b1f3322bff74f2

[48] MITRE. Query registry. [Online]. Available: https://attack.mitre.org/techniques/T1012/

[49] CAPEv2. Modifying kvm (qemu-kvm) settings for malware analysis. [Online]. Available: https://capev2.readthedocs.io/en/latest/usage/monitor.html

[50] UnProtect.it. Virtualization/sandbox evasion: Time based evasion. [Online]. Available: https://unprotect.it/technique/virtualizationsandbox-evasion-time-based-evasion/

[51] MITRE. Virtualization/sandbox evasion: Time based evasion. [Online]. Available: https://attack.mitre.org/techniques/T1497/003/

[52] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, “Does every second count? time-based evolution of malware behavior in sandboxes,” 01 2021.

[53] N. Thamsirarak, T. Seethongchuen, and P. Ratanaworabhan, “A case for malware that make antivirus irrelevant,” in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2015, pp. 1–6.

[54] D. Vidyarthi, C. R. S. Kumar, S. Rakshit, and S. Chansarkar, “Static malware analysis to identify ransomware properties,” 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:250557385

[55] K. Baker. (2023) Malware analysis. https://www.crowdstrike.com/cybersecurity-101/malware/malware-analysis/.

[56] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *Information Security Technical Report*, vol. 14, no. 1, pp. 16–29, 2009, malware. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1363412709000041

[57] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," *SSYM'03*, p. 12, 2003.

[58] I. Santos, "Using opcode sequences in single-class learning to detect unknown malware," *IET Information Security*, vol. 5, pp. 220–227(7), December 2011. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-ifs.2010.0180

[59] C. Ravi and R. Manoharan, "Article: Malware detection using windows api sequence and machine learning," *International Journal of Computer Applications*, vol. 43, no. 17, pp. 12–16, April 2012, full text available.

[60] D. K. S. Reddy, S. K. Dash, and A. K. Pujari, "New malicious code detection using variable length n-grams," in *Information Systems Security*, A. Bagchi and V. Atluri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 276–288.

[61] Y. Cao, Q. Miao, J. Liu, and L. Gao, "Abstracting minimal security-relevant behaviors for malware analysis," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 4, pp. 193–204, Nov 2013. [Online]. Available: https://doi.org/10.1007/s11416-013-0186-3

[62] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *Journal in Computer Virology*, vol. 7, no. 4, pp. 247–258, Nov 2011. [Online]. Available: https://doi.org/10.1007/s11416-011-0152-x

[63] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. USA: IEEE Computer Society, 2010, p. 45–60. [Online]. Available: https://doi.org/10.1109/SP.2010.11

Chapter 12

Tables

This appendix contains all of the tables that correspond to Chapter 6. It includes sections that have been organized based on the structure of the aforementioned Chapter.

12.1 General Dataset Statistics

General Statistics	
Number of Detected Code Injection Samples	957
Number of Processed Samples	778
Number of Unique Running Sandbox Processes	44

Table 12.1: General Statistics

Metrics of Detected Code Injection Samples Before Analysis		
Injection Type	*Count*	*Percentage*
AppInit DLL Injection	66	6.89%
Classic DLL injection	1	0.1%
COM Hijack DLL Injection	548	57.26%
Process Hollowing	137	14.31%
Thread Execution Hijacking	49	5.12%
Image File Execution Options	66	6.89%
Generic Shell Injection	90	9.40%
Total: 957		

Table 12.3: Metrics of Detected Code Injection Samples Before Analysis

Successfully Processed Samples Per Code Injection Type		
Injection Type	*Count*	*Percentage*
AppInit DLL Injection	49	6.3%
Classic DLL injection	1	0.1%
COM Hijack DLL Injection	426	55.1%
Process Hollowing	110	14%
Thread Execution Hijacking	43	5.5%
Image File Execution Options	64	8.2%
Generic Shell Injection	85	10.9%
Total: 778		

Table 12.5: Counts and Percentages of Samples which resulted in a successful CAPE run, and a successfully processed Process Monitor trace.

12.2 Malware Family Distribution

Counts of Samples per Family			
singleton: 65	berbew: 106	dinwod: 155	cosmu: 56
unruy: 8	expiro: 1	simda: 3	wabot: 31
tinba: 45	gepys: 14	rebhip: 1	extcrome: 1
powp: 12	shiz: 17	uniblue: 2	ngrbot: 6
hupigon: 9	virlock: 6	virut: 10	vobfus: 11
silcon: 3	shipup: 31	gamarue: 13	allaple: 3
qqpass: 1	catalina: 4	sfone: 6	zeroaccess: 1
vtflooder: 4	zbot: 9	qbot: 1	upatre: 2
ekstak: 1	darkkomet: 4	rozena: 1	blackshades: 4
poison: 4	installcore: 7	asparnet: 1	scarsi: 1
dealply: 1	fareit: 9	scudy: 2	pajetbin: 2
sality: 7	deshacop: 1	cozyduke: 1	starter: 1
neutrinopos: 2	reconyc: 1	mailru: 1	sivis: 1
cobaltstrike: 1	jawego: 3	rack: 2	golroted: 1
hpwortrik: 2	ubibila: 1	ursu: 1	spyeye: 7
speedingupmypc: 1	wapomi: 4	parite: 2	brontok: 2
houndhack: 1	torrentlocker: 1	zegost: 1	eyestye: 6
cerber: 2	naprat: 4	rnkbend: 1	frethog: 1
msilperseus: 2	nanocore: 2	locky: 2	perflogger: 1
cutwail: 1	llac: 3	asfortal: 1	icloader: 1

hoetou: 1	disfa: 1	zboter: 1	zepfod: 1
bunitu: 1	ctblocker: 2	neurevt: 1	floder: 1
ruskill: 1	petya: 1	gator: 2	lovgate: 1
kovter: 1	cozybear: 1	regrun: 1	generictka: 1
vbkryjetor: 1	globeimposter: 1	salgorea: 1	netwiredrc: 1
palevo: 1	rbot: 1	chir: 1	regsup: 1
vawtrak: 1	mintluks: 1	neshta: 1	nemim: 1
satan: 1		fraudrop: 1	

Table 12.6: Counts of Samples per Family

AppInit DLL Injection Family Counts			
gepys: 14	shipup: 31	zbot: 3	speedingupmypc: 1
Total: 49			

Table 12.7: AppInit DLL Injection Family Counts

Classic DLL injection Family Counts
singleton: 1

Table 12.8: Classic DLL injection Family Counts

COM Hijack DLL Injection Family Counts			
singleton: 22	berbew: 106	dinwod: 155	cosmu: 1
unruy: 8	expiro: 1	wabot: 11	tinba: 2
extcrome: 1	shiz: 2	uniblue: 2	virlock: 6
virut: 3	vobfus: 10	gamarue: 1	allaple: 3
catalina: 4	sfone: 6	vtflooder: 4	zbot: 2
upatre: 2	rozena: 1	poison: 1	installcore: 7
asparnet: 1	dealply: 1	scudy: 2	pajetbin: 2
deshacop: 1	cozyduke: 1	starter: 1	mailru: 1
sivis: 1	cobaltstrike: 1	jawego: 3	wapomi: 4
parite: 2	brontok: 2	houndhack: 1	zegost: 1
cerber: 1	frethog: 1	msilperseus: 1	locky: 1
perflogger: 1	icloader: 1	disfa: 1	zepfod: 1
bunitu: 1	petya: 1	gator: 1	cozybear: 1
regrun: 1	generictka: 1	salgorea: 1	rbot: 1
vawtrak: 1		mintluks: 1	neshta: 1

Total: 405

Table 12.9: COM Hijack DLL Injection Family Counts

Process Hollowing Family Counts			
singleton: 17	tinba: 13	powp: 12	ngrbot: 6
vobfus: 1	gamarue: 7	zbot: 4	darkkomet: 2
blackshades: 4	poison: 1	fareit: 8	reconyc: 1
golroted: 1	hpwortrik: 2	ursu: 1	spyeye: 7
torrentlocker: 1	eyestye: 6	naprat: 4	rnkbend: 1
llac: 1	zboter: 1	ctblocker: 2	floder: 1
ruskill: 1	kovter: 1	vbkryjetor: 1	nemim: 1
satan: 1			
Total: 109			

Table 12.10: Process Hollowing Family Counts

Image File Execution Options Family Counts			
cosmu: 55	virut: 7	qqpass: 1	neurevt: 1

Table 12.11: Image File Execution Options Family Counts

Generic Shell Injection Family Counts			
singleton: 5	simda: 3	tinba: 30	rebhip: 1
shiz: 15	hupigon: 9	silcon: 3	zeroaccess: 1
qbot: 1	darkkomet: 2	sality: 7	cerber: 1
locky: 1	cutwail: 1	llac: 2	lovgate: 1
chir: 1		regsup: 1	
Total: 85			

Table 12.12: Generic Shell Injection Family Counts

Thread Execution Hijacking Family Counts			
singleton: 20	gamarue: 5	poison:2	scarsi: 1
fareit: 1	neutrinopos: 2	rack: 2	ubibila: 1
msilperseus: 1	nanocore: 2	asfortal: 1	hoetou: 1
globeimposter: 1	netwiredrc: 1	palevo: 1	fraudrop: 1
Total: 43			

Table 12.13: Thread Execution Hijacking Family Counts

12.3 Targeted Processes List

Non-Native Targeted Applications	
Application Name	***Executable Name***
Lightshot	`C:\ProgramFiles(x86)\Skillbrains\lightshot\5.5.0.7\Lightshot.exe`
Flux	`C:\Users\John\AppData\Local\FluxSoftware\Flux\flux.exe`
Python 3.10	`C:\Windows\pyw.exe`
Opera	`C:\ProgramFiles(x86)\Opera\opera.exe`
Chrome	`C:\ProgramFiles\Google\Chrome\Application\chrome.exe`
Firefox	`C:\ProgramFiles(x86)\MozillaFirefox\firefox.exe`
AllThreadsView	`C:\Users\John\Downloads\AllThreadsView.exe`
Count: 7	

Table 12.14: Targeted Not Native Installed Software

Native Application Categories and Description		
Application	***Category***	***Description***
WmiPrvSE.exe	Information Broker	WMI Provider Host. Provides information about the system to applications.
SearchProtocolHost.exe	Indexing	Accesses files and data sources requiring indexing to enable efficiently portraying search results.
`MicrosoftEdge_X64_120.0.2210.144_120.0.2210.91.exe`	System Browser	Microsoft Edge Executable.

Continued on next page

Table 12.15 – continued from previous page

Native Applications Categories and Description

Application	***Category***	***Description***
lsass.exe	System Security	Local Security Authority Subsystem Service. Enforces system security policy. Verifies users, handles password changes, creates access tokens, writes to Windows Security Log.
SearchFilterHost.exe	Indexing	Microsoft Windows Search Filter Host. Used for indexing and caching files.
dwm.exe	UI Management	Desktop Window Manager. Compositing window manager enabling hardware acceleration to render the Windows GUI.
MicrosoftEdgeUpdate.exe	System Update	Microsoft Edge Update service.
`C:\Windows\SystemApps\Microsoft.Windows.StartMenuExperienceHost\_cw5n1h2txyewy\StartMenuExperienceHost.exe`	UI Management	Managing the Start menu.
Memory Compression	Memory Management	Service that dynamically reduces the size of data before writing it to RAM.
SgrmBroker.exe	System Security	System Guard Runtime Monitor Broker Service. Responsible for monitoring and attesting to the integrity of the OS.
spoolsv.exe	Memory Management	Spooler Service. Caches into memory data ready for printing.

Continued on next page

Table 12.15 – continued from previous page

Native Applications Categories and Description		
Application	***Category***	***Description***
MoUsoCoreWorker.exe	System Update	Part of the Windows Update process.
TiWorker.exe	System Update	Executes background tasks. Associated with the Windows Update service. Installs and manages system updates.
services.exe	Process and System Management	Services Control Manager. Responsible for running, ending, and interacting with system services.
`C:\Windows\System32\taskhostw.exe`	Process and System Management	Part of Windows' task management system. Manages Windows Task Scheduler tasks such as running background processes and handling system functions.
csrss.exe	Process and System Management	Client/Server Runtime Subsystem. Provides the user mode side of the Win32 subsystem.
`C:\Windows\System32\smartscreen.exe`	System Security	Microsoft Defender SmartScreen.
`C:\Windows\System32\sihost.exe`	Process and System Management	Shell Infrastructure Host file. Executes various system processes.
SearchIndexer.exe	Indexing	Provides content indexing, property caching, and search results for files, e-mail, and other content.

Continued on next page

Table 12.15 – continued from previous page

Native Applications Categories and Description		
Application	***Category***	***Description***
`C:\Windows\System32\ctfmon.exe`	Process and System Management	Collaborative Translation Framework. Used by Microsoft Office to control the Alternative User Input Text Input Processor and the Microsoft Office Language Bar.
fontdrvhost.exe	UI Management	Responsible for managing fonts on Windows.
winlogon.exe	System Security	Responsible for handling Secure Attention Sequence, loading the user profile on logon, creates the desktops for the window station, and optionally locks the computer when a screensaver is running.
`C:\Windows\explorer.exe`	UI Management	Executable module in Windows that contains the Start menu, Taskbar, desktop and file manager.
`C:\Windows\System32\dllhost.exe`	Process and System Management	Serves as a host for DLL (Dynamic Link Library) files. Allows shared DLLs to be executed and accessed by applications running on the Windows.
setup.exe	Software Installation	Software Installer.
`C:\Windows\SystemApps\Microsoft.Windows.Search\_cw5n1h2txyewy\SearchApp.exe`	Indexing	Ensures the search bar on the taskbar works and provides accurate search results for all user queries.

Continued on next page

Table 12.15 – continued from previous page

Native Applications Categories and Description		
Application	***Category***	***Description***
`C:\Windows\System32\svchost.exe`	Process and System Manage- ment	Service Host. Shared-service process Windows uses to load DLL files. Helps host the different files and processes that Windows needs to run efficiently.
nfsclnt.exe	Process and System Management	System process that is responsible for managing the Network File System (NFS) client service. Handles client requests/responses.
`C:\Windows\System32\RuntimeBroker.exe`	System Security	Task Manager process that helps manage permissions on apps obtained from the Microsoft Store.
svchost.exe	Process and System Management	Service Host. Shared-service process Windows uses to load DLL files. Helps host the different files and processes that Windows needs to run efficiently.
TrustedInstaller.exe	System Security	Windows Module Installer service, part of Windows Resource Protection(WRP). WRP is a technology that restricts access to certain core system files, folders, and registry keys that are part of the Windows installation.

Continued on next page

Table 12.15 – continued from previous page

Native Applications Categories and Description		
Application	***Category***	***Description***
smss.exe	Process and System Management	Session Manager Subsystem. First user-mode process started by the kernel. Creates additional paging files with configuration data from the registry.
Count: 32		

Table 12.15: Native Applications Categories and Description

12.4 Target Processes per Code Injection Type

Metrics of Top 5 Injected Processes Per Code Injection Type		
AppInit DLL Injection		
Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	50	44.64
SearchApp.exe	11	9.82
chrome.exe	9	8.04
explorer.exe	6	5.36
StartMenuExperienceHost.exe	5	4.46
Other	31	27.68
COM Hijack DLL Injection		
Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	429	32.33
chrome.exe	128	9.65
SearchApp.exe	107	8.06
explorer.exe	78	5.88
firefox.exe	45	3.39
Other	540	40.69

Continued on next page

Table 12.16 – continued from previous page

Process Hollowing		
Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	119	23.56
chrome.exe	62	12.28
explorer.exe	33	6.53
SearchApp.exe	26	5.15
firefox.exe	24	4.75
Other	241	47.72

Image File Execution Options		
Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	69	23.55
chrome.exe	37	12.63
firefox.exe	20	6.83
SearchApp.exe	18	6.14
explorer.exe	17	5.8
Other	132	45.05

Generic Shell Injection		
Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	133	19.05
chrome.exe	69	9.89
SearchApp.exe	60	8.6
explorer.exe - Memory Compression	50	7.16
Services.exe	44	6.3
Other	342	49.0

Thread Execution Hijacking		
Target Process	*Counts of Processes*	*Percentage of Samples*

Continued on next page

Table 12.16 – continued from previous page

svchost.exe	52	30.23
chrome.exe	20	11.63
SearchApp.exe	15	8.72
firefox.exe/explorer.exe	8	4.65
Opera.exe	7	4.07
Other	70	40.7

Classic DLL injection

Target Process	*Counts of Processes*	*Percentage of Samples*
svchost.exe	1	100.0
Other	0	0.0

Table 12.16: Metrics of Top 5 Injected Processes Per Code Injection Type

Attack Profiles Specification

Profile	***Comprising Processes***	***Code Injection Types***
Profile 1	svchost.exe SearchAp.exe chrome.exe explorer.exe	AppInitDLL Injection COM Hijack DLL Injection Process Hollowing Image File Execution Options
Profile 2	chrome.exe SearchApp.exe explorer.exe firefox.exe	COM Hijack DLL Injection Process Hollowing Image File Execution Options
Profile 3	svchost.exe explorer.exe SearchApp.exe firefox.exe	COM Hijack DLL Injection Process Hollowing Image File Execution Options

Table 12.17: Attack Profiles and Comprising Processes

12.5 Target Process Actions per Code Injection Type

Percentages Of Sandbox Aggregated Actions With Samples Loaded

Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per App***	***Average Percentage Per Sample***
File Operations	$78,598$	40.08 %	$1,786.31$	$\approx 0.05\%$
Process Operations	$27,181$	13.86 %	617.75	$\approx 0.16\%$
Query Operations	$29,418$	15.0 %	668.59	$\approx 0.14\%$
Registry Operations	$60,694$	30.95 %	$1,379.4$	$\approx 0.07\%$
Set Information File Operations	106	0.05 %	2.4	$\approx 41\ \%$
TCP Operations	66	0.03 %	1.5	$\approx 66\ \%$
UDP Operations	48	0.02 %	1.09	$\approx 91\ \%$

Table 12.18: Percentages Of Sandbox Aggregated Actions With Samples Loaded.

AppInit DLL Injection				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$2,098,075$	10.36 %	$31,789.02$	$\approx 0.21\%$
Process Operations	$845,683$	4.18 %	$12,813.38$	$\approx 0.53\%$
Query Operations	$245,477$	1.21 %	$3,719.35$	$\approx 2.01\%$
Registry Operations	$17,043,084$	84.17 %	$258,228.55$	$\approx 0.02\%$
Set Information File Operations	$4,718$	0.02 %	71.48	$\approx 94\%$

TCP Operations	$6,335$	0.03 %	95.98	$\approx 69\%$
UDP Operations	$5,305$	0.03 %	80.38	$\approx 82\%$

Table 12.19: Metrics of Aggregated Actions for AppInit DLL Injection

Classic DLL Injection				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$39,368$	9.17%	$39,368.0$	$\approx 25.4\%$
Process Operations	$17,406$	4.06%	$17,406.0$	$\approx 57.4\%$
Query Operations	$5,179$	1.21%	$5,179.0$	$\approx 19\%$
Registry Operations	$36,6817$	85.48%	$366,817$	$\approx 27.2\%$
Set Information File Operations	100	0.02%	100	$\approx 79.8\%$
TCP Operations	134	0.03%	134	$\approx 74\%$
UDP Operations	113	0.03%	113	$\approx 88\%$

Table 12.20: Metrics of Aggregated Actions for Classic DLL Injection

COM Hijack DLL Injection				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$15,847,625$	9.63%	$28,919.02$	$\approx 1.9\%$
Process Operations	$7,357,633$	4.47%	$13,426.34$	$\approx 4.09\%$

Query Operations	$2,006,632$	1.22%	$3,661.74$	$\approx 15.18\%$
Registry Operations	$139,256,706$	84.6%	$254,118.08$	$\approx 0.21 \cdot 10^{10-2}\%$
Set Information File Operations	$37,627$	0.02%	68.66	$\approx 79.8\%$
TCP Operations	$50,595$	0.03%	92.33	$\approx 59.4\%$
UDP Operations	$42,918$	0.03%	78.32	$\approx 69.9\%$

Table 12.21: Metrics of Aggregated Actions for COM Hijack DLL Injection

Process Hollowing				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$4,621,164$	11.48%	$33,731.12$	$\approx 0.41 \cdot 10^{-2}\%$
Process Operations	$1,898,555$	4.72%	$13,858.07$	$\approx 1 \cdot 10^{-2}\%$
Query Operations	$483,109$	1.2%	$3,526.34$	$\approx 4.13\%$
Registry Operations	$33,225,026$	82.53%	$242,518.44$	$\approx 0.05\%$
Set Information File Operations	$8,800$	0.02%	64.23	$\approx 21.3\%$
TCP Operations	$11,834$	0.03%	86.38	$\approx 15.8\%$
UDP Operations	$10,424$	0.03%	76.09	$\approx 18\%$

Table 12.22: Metrics of Aggregated Actions for Process Hollowing

Thread Execution Hijacking				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$1,707,810$	9.14%	$34,853.27$	$\approx 0.14\%$
Process Operations	$767,194$	4.11%	$15,657.02$	$\approx 0.32\%$
Query Operations	$231,056$	1.24%	$4,715.43$	$\approx 1.19\%$
Registry Operations	$15,966,437$	85.44%	$325,845.65$	$\approx 0.01\%$
Set Information File Operations	$4,300$	0.02%	87.76	$\approx 0.57\%$
TCP Operations	$5,790$	0.03%	118.16	$\approx 41\%$
UDP Operations	$4,859$	0.03	99.16	$\approx 49\%$

Table 12.23: Metrics of Aggregated Actions for Thread Execution Hijacking

Image File Execution Options				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	$3,102,369$	29.81%	$47,005.59$	$\approx 0.14\%$
Process Operations	$901,156$	8.66%	$13,653.88$	$\approx 0.49\%$
Query Operations	$98,261$	0.94%	$1,488.8$	$\approx 6.32\%$
Registry Operations	$6,298,689$	60.53%	$95,434.68$	$\approx 0.06\%$

Set Information File Operations	1,400	0.01%	21.21	≈ 33.63%
TCP Operations	1,883	0.02 %	28.53	≈ 23.97%
UDP Operations	2,638	0.03 %	39.97	≈ 16.81%

Table 12.24: Metrics of Aggregated Actions for Image File Execution Options

Generic Shell Injection				
Aggregated Action	***Count of Hits***	***Percentage***	***Average Actions per Sample***	***Average Percentage per Sample***
File Operations	4,544,027	20.19%	50,489.19	≈ 0.18%
Process Operations	1,439,452	6.4%	15,993.91	≈ 0.57%
Query Operations	285,339	1.27%	3,170.43	≈ 3.16%
Registry Operations	16,223,982	72.08%	180,266.47	≈ 0.05%
Set Information File Operations	3,800	0.02%	42.22	≈ 21.92%
TCP Operations	5,148	0.02%	57.2	≈ 15.93%
UDP Operations	5,494	0.02%	61.04	≈ 14.87%

Table 12.25: Metrics of Aggregated Actions for Generic Shell Injection

12.6 Sample Targets per Operation Type Code Injection Type

12.6.1 File Operations

File Operations

AppInit DLL Injection

Target	Count of Hits	Percentage
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	$393,792$	20.8%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	$276,960$	14.63%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	$214,656$	11.34%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	$196,512$	10.38%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	$102,768$	5.43%
`Other`	$708,559$	37.43%

COM Hijack DLL Injection

Target	Count of Hits	Percentage
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	$3,109,316$	20.65%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	$2,186,847$	14.52%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	$1,694,888$	11.26%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	$1,551,626$	10.3%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	$821,775$	5.46%

Continued on next page

Table 12.26 – continued from previous page

Other	5,692,796	37.81%

Process Hollowing		
Target	**Count of Hits**	**Percentage**
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	721,952	20.33%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	507,780	14.3%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	393,536	11.08%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	360,272	10.15%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	200,568	5.65%
Other	1,366,959	38.49%

Image File Execution Options		
Target	**Count of Hits**	**Percentage**
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	114,856	16.91%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	80,824	11.9%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	62,608	9.22%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	57,316	8.44%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	56,726	8.35%
Other	306,714	45.17%

Generic Shell Injection		
Target	**Count of Hits**	**Percentage**

Continued on next page

Table 12.26 – continued from previous page

`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	$311,752$	17.91%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	$219,310$	12.6%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	$169,936$	9.76%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	$155,572$	8.94%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	$111,758$	6.42%
`Other`	$772,775$	44.38%

Thread Execution Hijacking

Target	Count of Hits	Percentage
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	$352,772$	20.66%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	$248,110$	14.53%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	$192,296$	11.26%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	$176,042$	10.31%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	$92,063$	5.39%
`Other`	$646,527$	37.86%

Classic DLL Injection

Target	Count of Hits	Percentage
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Deployment.srd`	$8,204$	20.84%

Continued on next page

Table 12.26 – continued from previous page

`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd`	$5,770$	14.66%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERD6B0.tmp.csv`	$4,472$	11.36%
`C:\ProgramData\Microsoft\Windows\WER\Temp\WERAE61.tmp.csv`	$4,094$	10.4%
`C:\ProgramData\Microsoft\Windows\AppRepository\StateRepository-Machine.srd-shm`	$2,141$	5.44%
`Other`	$14,687$	37.31%

Table 12.26: File Operations: Sample Targets per Operation Type Code Injection Type.

12.6.2 Process Operations

Process Operations

AppInit DLL Injection

Target	Hits Count	Percentage
`C:\Windows\System32\policymanager.dll`	$3,648$	0.44%
`C:\Windows\System32\msvcp110_win.dll`	$3,648$	0.44%
`C:\Windows\System32\OnDemandConnRouteHelper.dll`	$1,152$	0.14%
`C:\Windows\System32\ntdll.dll`	627	0.07%
`C:\Windows\System32\svchost.exe`	624	0.07%
`Other`	$828,820$	98.84%

COM Hijack DLL Injection

Target	Hits Count	Percentage
`C:\Windows\System32\policymanager.dll`	$28,804$	0.42%
`C:\Windows\System32\msvcp110_win.dll`	$28,804$	0.42%
`C:\Windows\System32\OnDemandConnRouteHelper.dll`	$9,249$	0.13%
`C:\Windows\System32\svchost.exe`	$5,019$	0.07%
`C:\Windows\System32\ntdll.dll`	$4,995$	0.07%
`Other`	$6,832,739$	98.89%

Continued on next page

Table 12.27 – continued from previous page

Process Hollowing

Target	Hits Count	Percentage
C:\Windows\System32\policymanager.dll	6,688	0.36%
C:\Windows\System32\msvcp110_win.dll	6,688	0.36%
C:\Windows\System32\OnDemandConnRouteHelper.dll	2,292	0.12%
C:\Windows\System32\svchost.exe	1,199	0.07%
C:\Windows\System32\ntdll.dll	1,182	0.06%
Other	1,824,837	99.02%

Image File Execution Options

Target	Hits Count	Percentage
C:\Windows\System32\policymanager.dll	1,064	0.12%
C:\Windows\System32\msvcp110_win.dll	1,064	0.12%
C:\Windows\System32\OnDemandConnRouteHelper.dll	732	0.08%
C:\Windows\System32\ntdll.dll	229	0.03%
C:\Windows\System32\svchost.exe	226	0.03%
Other	880,050	99.62%

Generic Shell Injection

Target	Hits Count	Percentage
C:\Windows\System32\policymanager.dll	2,888	0.2%
C:\Windows\System32\msvcp110_win.dll	2,888	0.2%
C:\Windows\System32\OnDemandConnRouteHelper.dll	1,362	0.09%
C:\Windows\System32\svchost.exe	584	0.04%
C:\Windows\System32\ntdll.dll	568	0.04%
Other	1,431,162	99.42%

Thread Execution Hijacking

Target	Hits Count	Percentage
C:\Windows\System32\policymanager.dll	3,268	0.43%
C:\Windows\System32\msvcp110_win.dll	3,268	0.43%

Continued on next page

Table 12.27 – continued from previous page

C:\Windows\System32\OnDemandConnRouteHelper.dll	1,032	0.14%
C:\Windows\System32\ntdll.dll	571	0.08%
C:\Windows\System32\svchost.exe	564	0.07%
Other	747,772	98.85%

Classic DLL Injection

Target	Hits Count	Percentage
C:\Windows\System32\policymanager.dll	76	0.44%
C:\Windows\System32\msvcp110_win.dll	76	0.44%
C:\Windows\System32\OnDemandConnRouteHelper.dll	24	0.14%
C:\Windows\System32\svchost.exe	13	0.07%
C:\Windows\System32\ntdll.dll	13	0.07%
Other	17,204	98.84%

Table 12.27: Process Operations: Sample Targets per Operation Type Code Injection Type.

12.6.3 Query Operations

Query Operations

AppInit DLL Injection

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	13,968	5.59%
C:\Windows\System32\policymanager.dll	11,040	4.42%
C:\ProgramData\Microsoft\Windows\WER	8,736	3.5%
C:\ProgramData\Microsoft\Windows\WER\Temp	8,736	3.5%
C:\Windows\Temp	8,260	3.31%
Other	199,114	79.69%

COM Hijack DLL Injection

Target	Hits Count	Percentage

Continued on next page

Table 12.28 – continued from previous page

C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	110,289	5.46%
C:\Windows\System32\policymanager.dll	87,170	4.31%
C:\ProgramData\Microsoft\Windows\WER	68,978	3.41%
C:\ProgramData\Microsoft\Windows\WER\Temp	68,978	3.41%
C:\Windows\Temp	67,264	3.33%
Other	1,618,793	80.08%

Process Hollowing

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	25,608	5.3%
C:\Windows\System32\policymanager.dll	20,240	4.19%
C:\Windows\Temp	16,964	3.51%
C:\ProgramData\Microsoft\Windows\WER	16,016	3.32%
C:\ProgramData\Microsoft\Windows\WER\Temp	16,016	3.32%
Other	388,265	80.37%

Image File Execution Options

Target	Hits Count	Percentage
C:\Windows\Temp	5,604	5.7%
C:\Windows\ServiceProfiles\NetworkService\AppData\Local\Temp	4,288	4.36%
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	4,074	4.15%
C:\Windows\System32\policymanager.dll	3,220	3.28%
C:\ProgramData\Microsoft\Windows\WER	2,548	2.59%

Continued on next page

Table 12.28 – continued from previous page

Other	78,527	79.92%

Generic Shell Injection		
Target	**Hits Count**	**Percentage**
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	11,058	3.88%
C:\Windows\Temp	10,728	3.76%
C:\Windows\System32\policymanager.dll	8,740	3.06%
C:\ProgramData\Microsoft\Windows\WER	6,916	2.42%
C:\ProgramData\Microsoft\Windows\WER\Temp	6,916	2.42%
Other	240,981	84.45%

Thread Execution Hijacking		
Target	**Hits Count**	**Percentage**
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	12,513	5.42%
C:\Windows\System32\policymanager.dll	9,890	4.28%
C:\ProgramData\Microsoft\Windows\WER	7,826	3.39%
C:\ProgramData\Microsoft\Windows\WER\Temp	7,826	3.39%
C:\Windows\Temp	7,464	3.23%
Other	185,537	80.3%

Classic DLL Injection		
Target	**Hits Count**	**Percentage**
C:\ProgramData\Microsoft\Windows\AppRepository\Packages\Microsoft.Windows.ContentDeliveryManager_10.0.19041.1023_neutral_neutral_cw5n1h2txyewy\ActivationStore.dat	291	5.62%
C:\Windows\System32\policymanager.dll	230	4.44%

Continued on next page

Table 12.28 – continued from previous page

C:\ProgramData\Microsoft\Windows\WER	182	3.51%
C:\ProgramData\Microsoft\Windows\WER\Temp	182	3.51%
C:\Windows\Temp	172	3.32%
Other	4,122	79.59%

Table 12.28: Query Operations: Sample Targets per Operation Type Code Injection Type.

12.6.4 Registry Operations

Registry Operations

AppInit DLL Injection

Target	Hits Count	Percentage
HKLM	1,177,429	19.92%
HKU	77,832	1.32%
HKLM\System\CurrentControlSet\Control\SessionManager\Environment	70,211	1.19%
HKU\.DEFAULT	69,315	1.17%
HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages	59,328	1.0%
Other	4,455,675	75.39%

COM Hijack DLL Injection

Target	Hits Count	Percentage
HKLM	9,417,237	19.93%
HKU	621,781	1.32%
HKLM\System\CurrentControlSet\Control\SessionManager\Environment	571,761	1.21%
HKU\.DEFAULT	554,111	1.17%
HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages	468,444	0.99%
Other	35,608,249	75.37%

Process Hollowing

Target	Hits Count	Percentage

Continued on next page

Table 12.29 – continued from previous page

HKLM	2,277,481	20.17%
HKU	148,631	1.32%
HKLM\System\CurrentControlSet\Control\SessionManager\Environment	144,200	1.28%
HKU\.DEFAULT	130,316	1.15%
HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages	108,768	0.96%
Other	8,483,152	75.12%

Image File Execution Options

Target	Hits Count	Percentage
HKLM	579,159	23.33%
HKLM\System\CurrentControlSet\Control\SessionManager\Environment	47,635	1.92%
HKU	32,444	1.31%
HKU\.DEFAULT	22,392	0.9%
HKLM\SYSTEM\Setup	20,799	0.84%
Other	1,779,677	71.7%

Generic Shell Injection

Target	Hits Count	Percentage
HKLM	1,221,590	21.02%
HKLM\System\CurrentControlSet\Control\SessionManager\Environment	91,196	1.57%
HKU	75,181	1.29%
HKU\.DEFAULT	67,245	1.16%
HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages	46,968	0.81%
Other	4,308,981	74.15%

Thread Execution Hijacking

Target	Hits Count	Percentage
HKLM	1,062,942	19.88%
HKU	70,736	1.32%

Continued on next page

Table 12.29 – continued from previous page

`HKLM\System\CurrentControlSet\Control\SessionManager\Environment`	63,448	1.19%
`HKU\.DEFAULT`	63,112	1.18%
`HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages`	53,148	0.99%
`Other`	4,032,890	75.43%

Classic DLL Injection		
Target	**Hits Count**	**Percentage**
`HKLM`	24,482	19.93%
`HKU`	1,615	1.31%
`HKLM\System\CurrentControlSet\Control\SessionManager\Environment`	1,462	1.19%
`HKU\.DEFAULT`	1,439	1.17%
`HKCR\LocalSettings\Software\Microsoft\Windows\CurrentVersion\AppModel\PackageRepository\Packages`	1,236	1.01%
`Other`	92,619	75.39%

Table 12.29: Registry Operations: Sample Targets per Operation Type Code Injection Type.

12.6.5 Set Information File Operations

Set Information File Operations		
AppInit DLL Injection		
Target	**Hits Count**	**Percentage**
`C:\ProgramData\Microsoft\Windows\WER\Temp`	912	19.0%
`C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506`	288	6.0%
`C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749`	288	6.0%

Continued on next page

Table 12.30 – continued from previous page

C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	288	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	240	5.0%
Other	2,784	58.0%

COM Hijack DLL Injection

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\WER\Temp	7,201	19.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	2,274	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	2,274	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	2,274	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	1,895	5.0%
Other	21,982	58.0%

Process Hollowing

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\WER\Temp	1,672	19.0%

Continued on next page

Table 12.30 – continued from previous page

C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	528	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	528	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	528	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	440	5.0%
Other	5,104	58.0%

Image File Execution Options

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\WER\Temp	266	19.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	84	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	84	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	84	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	70	5.0%
Other	812	58.0%

Continued on next page

Table 12.30 – continued from previous page

Generic Shell Injection		
Target	**Hits Count**	**Percentage**
C:\ProgramData\Microsoft\Windows\WER\Temp	722	19.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	228	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	228	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	228	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	190	5.0%
Other	2,204	58.0%

Thread Execution Hijacking		
Target	**Hits Count**	**Percentage**
C:\ProgramData\Microsoft\Windows\WER\Temp	817	19.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	258	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	258	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	258	6.0%

Continued on next page

Table 12.30 – continued from previous page

C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	215	5.0%
Other	2,494	58.0%

Classic DLL Injection

Target	Hits Count	Percentage
C:\ProgramData\Microsoft\Windows\WER\Temp	19	19.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\77EC63BDA74BD0D0E0426DC8F8008506	6	6.0%
C:\Users\John\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\FB0D848F74F70BB2EAA93746D24D9749	6	6.0%
C:\Windows\System32\config\systemprofile\AppData\LocalLow\Microsoft\CryptnetUrlCache\MetaData\7423F88C7F265F0DEFC08EA88C3BDE45_AA1E8580D4EBC816148CE81268683776	6	6.0%
C:\Windows\SystemApps\MicrosoftWindows.Client.CBS_cw5n1h2txyewy\microsoft.system.package.metadata\Autogen\JSByteCodeCache_64	5	5.0%
Other	58	58.0%

Table 12.30: Set Information File Operations: Sample Targets per Operation Type Code Injection Type.

12.6.6 TCP Operations

TCP Operations

AppInit DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	768	11.93%

Continued on next page

Table 12.31 – continued from previous page

DESKTOP-VQUTB06:51102->51.132.193.105:https	768	11.93%
DESKTOP-VQUTB06:51109->20.50.201.195:https	768	11.93%
DESKTOP-VQUTB06:51097->20.190.181.1:https	720	11.18%
DESKTOP-VQUTB06:51079->20.190.181.1:https	672	10.44%
Other	2,743	42.6%

COM Hijack DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	6,064	11.91%
DESKTOP-VQUTB06:51102->51.132.193.105:https	6,064	11.91%
DESKTOP-VQUTB06:51109->20.50.201.195:https	6,064	11.91%
DESKTOP-VQUTB06:51097->20.190.181.1:https	5,685	11.17%
DESKTOP-VQUTB06:51079->20.190.181.1:https	5,306	10.42%
Other	21,722	42.67%

Process Hollowing

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	1,408	11.9%
DESKTOP-VQUTB06:51102->51.132.193.105:https	1,408	11.9%
DESKTOP-VQUTB06:51109->20.50.201.195:https	1,408	11.9%
DESKTOP-VQUTB06:51097->20.190.181.1:https	1,320	11.15%
DESKTOP-VQUTB06:51079->20.190.181.1:https	1,232	10.41%
Other	5,058	42.74%

Image File Execution Options

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	224	11.9%

Continued on next page

Table 12.31 – continued from previous page

DESKTOP-VQUTB06:51102->51.132.193.105:https	224	11.9%
DESKTOP-VQUTB06:51109->20.50.201.195:https	224	11.9%
DESKTOP-VQUTB06:51097->20.190.181.1:https	210	11.15%
DESKTOP-VQUTB06:51079->20.190.181.1:https	196	10.41%
Other	805	42.75%

Generic Shell Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	608	11.81%
DESKTOP-VQUTB06:51102->51.132.193.105:https	608	11.81%
DESKTOP-VQUTB06:51109->20.50.201.195:https	608	11.81%
DESKTOP-VQUTB06:51097->20.190.181.1:https	570	11.07%
DESKTOP-VQUTB06:51079->20.190.181.1:https	532	10.33%
Other	2,222	43.16%

Thread Execution Hijacking

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	688	11.88%
DESKTOP-VQUTB06:51102->51.132.193.105:https	688	11.88%
DESKTOP-VQUTB06:51109->20.50.201.195:https	688	11.88%
DESKTOP-VQUTB06:51097->20.190.181.1:https	645	11.14%
DESKTOP-VQUTB06:51079->20.190.181.1:https	602	10.4%
Other	2,479	42.82%

Classic DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:51100->51.132.193.105:https	16	11.94%

Continued on next page

Table 12.31 – continued from previous page

DESKTOP-VQUTB06:51102->51.132.193.105:https	16	11.94%
DESKTOP-VQUTB06:51109->20.50.201.195:https	16	11.94%
DESKTOP-VQUTB06:51097->20.190.181.1:https	15	11.19%
DESKTOP-VQUTB06:51079->20.190.181.1:https	14	10.45%
Other	57	42.54%

Table 12.31: TCP Operations: Sample Targets per Operation Type Code Injection Type.

12.6.7 UDP Operations

UDP Operations

AppInit DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:56722->thinkPad:domain	96	1.77%
DESKTOP-VQUTB06:58558->ff02::1:3:llmnr	96	1.77%
DESKTOP-VQUTB06:58558->224.0.0.252:llmnr	96	1.77%
DESKTOP-VQUTB06:53962->thinkPad:domain	96	1.77%
DESKTOP-VQUTB06:56820->thinkPad:domain	96	1.77%
Other	4,944	91.15%

COM Hijack DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:56722->thinkPad:domain	758	1.75%
DESKTOP-VQUTB06:58558->ff02::1:3:llmnr	758	1.75%
DESKTOP-VQUTB06:58558->224.0.0.252:llmnr	758	1.75%
DESKTOP-VQUTB06:53962->thinkPad:domain	758	1.75%
DESKTOP-VQUTB06:56820->thinkPad:domain	758	1.75%
Other	39,445	91.23%

Process Hollowing

Target	Hits Count	Percentage
DESKTOP-VQUTB06:56722->thinkPad:domain	176	1.69%
DESKTOP-VQUTB06:58558->ff02::1:3:llmnr	176	1.69%

Continued on next page

Table 12.32 – continued from previous page

DESKTOP-VQUTB06:58558->224.0.0.252:llmnr	176	1.69%
DESKTOP-VQUTB06:53962->thinkPad:domain	176	1.69%
DESKTOP-VQUTB06:56820->thinkPad:domain	176	1.69%
Other	9,544	91.56%

Image File Execution Options

Target	Hits Count	Percentage
DESKTOP-VQUTB06:60261->192.168.125.1:domain	88	3.34%
DESKTOP-VQUTB06:64909->192.168.125.1:domain	88	3.34%
DESKTOP-VQUTB06:49668->192.168.125.1:domain	88	3.34%
DESKTOP-VQUTB06:62593->192.168.125.1:domain	88	3.34%
DESKTOP-VQUTB06:51179->ff02::1:3:llmnr	88	3.34%
Other	2,198	83.32%

Generic Shell Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:60261->192.168.125.1:domain	100	1.82%
DESKTOP-VQUTB06:64909->192.168.125.1:domain	100	1.82%
DESKTOP-VQUTB06:49668->192.168.125.1:domain	100	1.82%
DESKTOP-VQUTB06:62593->192.168.125.1:domain	100	1.82%
DESKTOP-VQUTB06:51179->ff02::1:3:llmnr	100	1.82%
Other	4,994	90.9%

Thread Execution Hijacking

Target	Hits Count	Percentage
DESKTOP-VQUTB06:56722->thinkPad:domain	86	1.77%
DESKTOP-VQUTB06:58558->ff02::1:3:llmnr	86	1.77%
DESKTOP-VQUTB06:58558->224.0.0.252:llmnr	86	1.77%

Continued on next page

Table 12.32 – continued from previous page

DESKTOP-VQUTB06:53962->thinkPad:domain	86	1.77%
DESKTOP-VQUTB06:56820->thinkPad:domain	86	1.77%
Other	4,429	91.15%

Classic DLL Injection

Target	Hits Count	Percentage
DESKTOP-VQUTB06:56722->thinkPad:domain	2	1.77%
DESKTOP-VQUTB06:58558->ff02::1:3:llmnr	2	1.77%
DESKTOP-VQUTB06:58558->224.0.0.252:llmnr	2	1.77%
DESKTOP-VQUTB06:53962->thinkPad:domain	2	1.77%
DESKTOP-VQUTB06:56820->thinkPad:domain	2	1.77%
Other	103	91.15%

Table 12.32: UDP Operations: Sample Targets per Operation Type Code Injection Type.

12.7 IP And Domain Addresses Metrics

General Dataset Metrics

Metric	*Count*
Unique IP Addresses Count	98
Unique Domain Addresses Count	71
Unique countries Count	10
Unique Cities Count	39
Observed IP Addresses (Incl. Duplicates)	91,251
Observed Domain Addresses Count (Incl. Duplicates)	4,426
Observed IP and Domain Address Count (Incl. Duplicates)	95,677
Samples Reaching an IP Address	778
Samples Reaching a Domain Address	196
Samples Reaching an IP and Domain Address	196
Percentage of Samples Connecting to an Address	100

Table 12.33: Counts of various general IP and Domain metrics.

List of Unique Countries	
Country	***Count of IPs***
United States	60
Canada	8
The Netherlands	7
Russia	6
Singapore	5
China	5
France	2
British Virgin Islands	2
Germany	2
Ireland	1

Table 12.34: List of unique countries that IP addresses matched to.

List of Unique Cities	
City	***Count of IPs***
Secaucus	8
Toronto	6
Seattle	6
Mountain View	6
Moscow	6
Singapore	5
Hangzhou	5
Ashburn	5
Menifee	4
New York	4
Council Bluffs	3
Amsterdam	3
Glenside	3
Dallas	3
Dublin	3
Montreal	2
Paris	2
Road Town	2
Chicago	2

Continued on next page

Table 12.35 – continued from previous page

Haarlem	1
Clifton	1
Mahwah	1
Brooklyn	1
Franklin Square	1
Los Angeles	1
Phoenix	1
Cumming	1
San Francisco	1
Flushing	1
Fremont	1
Edison	1
Minneapolis	1
Frankfurt am Main	1
Kansas City	1
North Charleston	1
Lansing	1
Capelle aan den IJssel	1
Cologne	1
Amsterdam	1

Table 12.35: List of unique cities that IP addresses matched to.

12.7.1 Metrics per Code Injection Type

Address Metrics per Code Injection Type

AppInit DLL Injection

City	***Metric***
Observed IP Addresses (Incl. Duplicates)	$12,396$
Observed Domain Addresses	$1,152$
Observed Addresses Sum	$13,548$
Observed Addresses Percentage	$14,16\%$

Continued on next page

Table 12.36 – continued from previous page

Observed IP Addresses per Sample	$1,377.3$
Observed Domain Addresses per Sample	128
COM Hijack DLL Injection	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	$22,863$
Observed Domain Addresses	725
Observed Addresses Sum	$23,588$
Observed Addresses Percentage	24.65%
Observed IP Addresses per Sample	$47,04$
Observed Domain Addresses per Sample	$1,70$
Process Hollowing	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	$14,285$
Observed Domain Addresses	896
Observed Addresses Sum	$15,181$
Observed Addresses Percentage	15.86%
Observed IP Addresses per Sample	129.86
Observed Domain Addresses per Sample	8.14
Image File Execution Options	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	$22,585$

Continued on next page

Table 12.36 – continued from previous page

Observed Domain Addresses	593
Observed Addresses Sum	$23,178$
Observed Addresses Percentage	24.22%
Observed IP Addresses per Sample	352.89
Observed Domain Addresses per Sample	9.26
Generic Shell Injection	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	$18,296$
Observed Domain Addresses	528
Observed Addresses Sum	$18,824$
Observed Addresses Percentage	19.12%
Observed IP Addresses per Sample	215.24
Observed Domain Addresses per Sample	6.21
Thread Execution Hijacking	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	794
Observed Domain Addresses	514
Observed Addresses Sum	$1,308$
Observed Addresses Percentage	1.36%
Observed IP Addresses per Sample	18.46
Observed Domain Addresses per Sample	11.95

Continued on next page

Table 12.36 – continued from previous page

Classic DLL Injection	
City	***Metric***
Observed IP Addresses (Incl. Duplicates)	32
Observed Domain Addresses	18
Observed Addresses Sum	50
Observed Addresses Percentage	0.05%
Observed IP Addresses per Sample	32
Observed Domain Addresses per Sample	18

Table 12.36: Counts of Domain and IP Addresses Per Injection Type.