Nefertum: The fallout of Windows code injection in benign programs

Jordi Bals
SCS
University of Twente
Enschede, Netherlands
S2253194
j.bals@student.utwente.nl

I. Abstract

Code injection is a technique utilized by malware that injects a section of its code into other processes and tricks them into executing it. Many state-of-the-art detection systems only determine malicious behavior by looking at the malware sample. Not looking at the target process of code injection means they miss part of the malicious behavior. No research studies the effects of code injection on benign injected processes, so it is unclear how much malicious process behavior (e.g., system calls) modern solutions miss.

We propose a framework that automatically identifies behavior exhibited by injecting malware samples and their victim processes after being targeted by code injection. The framework utilizes dynamic analysis to find the system calls of the malware sample and its victim and matches the found system calls to SIGMA rules that define behavior. We then use this framework to gather the behaviors of 436 real-life samples and their victims to approximate the behavior missed in modern detection systems.
Our experiments suggest that solutions miss, on average, 56.3% of behavior when looking strictly at the amount of tracked system calls and 64% of behavior when looking at the amount of SIGMA rules found.

Keywords: Malware, Code Injection, Malicious Behaviors

II. Introduction

Nowadays, it is almost impossible to imagine a world that does not rely on the Internet for many daily tasks. People use the Internet daily for banking, gaming, social interactions, streaming, and shopping. In addition, during the past few years during the COVID-19 pandemic, many people had to work from home, which increased the use of the Internet [8].

With this increased internet use, the risk people face online also increases. One of these risks is malware. Malware, named from *MAL*icious soft*WARE*, is software designed to operate maliciously. Malware has the goal of, for example, disrupting or denying access to systems, gathering data, or gaining access to systems [10]. The high use of the Internet has resulted in a high amount of potential targets for malware. Due to the abundance of targets, malware authors develop malware at an enormous rate, about 450.000 new malware samples per day [1].

One technique that malware utilizes is code injection [20]. This technique allows malware to take a section of code and insert it into another *victim* process. The other process will then execute the code as if it were its own. The injecting of code into victims and the victim executing it makes malware much more difficult to detect by anti-viruses and other detection systems, as the malware sample does not show any malicious behavior. Instead, the victim process exhibits malicious behavior.

Many state-of-the-art detection systems attempt to detect malicious behavior and, in turn, code injection in different ways. Some examples include behavior nets [27], others use taint analysis [13], or the honeypot paradigm [4]. One thing these systems share is that they only focus on detecting that a sample uses code injection, not what behavior the victim process of the code injection shows afterward. Ignoring what behavior the victim shows means we miss part of the malicious behavior of the sample. Thus, we underestimate what these malware samples do and do not know how much we underestimate it. The victim's behavior could be a small part of the malicious behavior, but it could also be a majority. However, in the literature, no research studies the effects of code injection on benign injected processes.

We propose a framework that automatically identifies behavior exhibited by injecting malware samples and their victim processes after being targeted by code injection. Subsequently, we compare the behaviors of the victim and the injector to estimate the percentage of malicious behavior (e.g., system calls) missed if we observe only the malware sample, as many state-of-the-art detection systems do [15], [25].

We run the samples in a sandbox environment, employing system-wide monitoring to record the behavior of all processes on the system. Then, we utilize an analyzer to transform the output from the sandbox into an *Injection Timestamp* containing the moment of code injection and a handle to the victim process. Afterward, we utilize the Injection Timestamp to filter the sandbox logs for system calls of the malware sample pre-injection and the victim post-injection. These system calls are matched against SIGMA rules to determine their behavior.

Our experiments show that modern solutions miss, on average, 56.3% of behavior when they strictly look at the amount of tracked system calls. When we compare SIGMA rules, our experiments show that we miss 64% of SIGMA rules when we only look at the malware sample.

In short, the contributions of this paper are the following:

- **Framework to classify behavior:**
  We propose a framework that automatically identifies behavior exhibited by injecting malware samples and their victim processes after being targeted by code injection. The behavior is then classified utilizing SIGMA rules.
- **Approximation of missed behavior:**
  We then use this framework to gather the behaviors of 500 real-life samples and their victims to approximate the behavior missed in modern detection systems.

The rest of this paper is structured as follows. First, we cover the background knowledge required for this research in Section III. This section includes code injection techniques, an explanation of different analysis techniques, and a summary of system calls. We continue in Section IV with an explanation of the methodology used for our research. After that, in Section V, we outline the system used for our testing. In Section VI, we present our findings and discuss them in Section VII, after which limitations are covered in Section VIII. After that, we relate our results to previous research in Section IX. Finally, we conclude with a summary in Section X.

III. Background

A. Code Injection

Code injection is a technique that allows programs to insert a part of their code into other, often called *victim*, processes. The *victim* process will then be forced or tricked in some way to execute the inserted code instead of its own. There are three main reasons for malware to utilize code injections [20]:

- Hiding
- Process piggybacking
- Altering other processes

Hiding, also known as stealth, refers to the malware not wanting to be found by a human interacting with the task manager or anti-virus software checking running processes on the computer. Code injection is a good way for the malware to hide its presence. By injecting the functional part of the malware, also known as the *payload*, into a victim process, the malware ensures that the payload will get executed. After the injection, the main malware program can exit to ensure it does not get detected. Any anti-virus or human inspecting the running processes would, therefore, not detect that the malware is running, as another process is executing the malicious payload. The difficulty in detecting has multiple reasons. One reason is that some processes are trusted, and the anti-virus does not monitor them, as needing to constantly check if a process is still running as it should would have a massive processing overhead.

Process piggybacking refers to the malware potentially using other program permissions and privileges to bypass restrictions, including firewall rules and OS policies. For example, firewall rules might indicate that only specific processes can connect to the Internet. In that case, at first, the malware will try to connect to the Internet via its process, but it gets blocked by the firewall. The malware can then still try to connect to the Internet by injecting its network-specific code into one of the processes that are allowed access. This code will then be able to connect to the Internet and function as intended, as the firewall typically only checks the originating process, which is no longer the malware sample.

The last reason is *altering other processes*. Malware can have different reasons for wanting to change the behavior of another process. It could be for specific code hooking attacks or API interceptions. One example is that the malware has a file that it needs to protect, a config file. So, the malware will do anything to prevent the system from deleting that file. The malware might, for example, inject code into the API or process responsible for deleting files such that it will only work if the deletion target is not named config.txt. This way, even if an anti-virus or human finds the config file and wants to delete it, they will be unable to, given they utilize the specific API that the malware altered [20].

Some benign cases also utilize code injection, for example, debuggers and shim infrastructures. Debuggers inject pieces of code into victim processes to get internal states and pause execution at certain moments. Additionally, some operating systems use shim infrastructures to help programs that use outdated APIs. These infrastructures simulate removed functionality or functionality altered so drastically that it does not retain the original function [27].

These cases show that fully disabling code injection is not an option. One feature operating systems such as Windows have implemented is Access rights [6]. These rights help restrict what a process can do. A low-privileged process cannot gain access to protected or high-privileged processes. However, this method reduces code injection incidents only to a certain degree. Many processes are not protected or high-privilege. Additionally, running a process as an administrator gives the malware elevated privileges. So, Access Rights fails to do anything when a user gets tricked into running a malware sample administrator.

Starink et al. [27] recognize a taxonomy that divides existing techniques into classes based on common characteristics. These classes help determine the traits of a code injection approach and how to detect these different approaches.

The first difference is whether the malware is *active* or *passive*. A technique is active when it directly alters the memory of the victim process or directly interacts with the process or one of its threads. Conversely, when it, for example, lets the underlying operating system interact with the victim instead. The technique is considered passive. Active techniques are more traditional and most commonly used for code injection. Being used often causes most sandboxes to know about active techniques, while passive techniques are more troublesome to detect.

A commonly known active technique is *Shellcode Injection* [9]. It opens a handle to the victim and writes executable memory directly into the process, directly interacting with the victim. A well-known passive technique is *Windows Hook Injection* [12]. This technique uses a Windows API to subscribe a thread currently running to a specified event (such as a key press or mouse click). When the event triggers, the thread will load a specified function, often residing in a malicious DLL file.

The second divide is whether the passive techniques are *configuration-based* or not. Techniques are configuration-based if they require a specific change in the registry to function and need some persistent configuration stored on the system. Most passive techniques covered in this paper have this trait.

An example of a configuration-based technique is *Shim Injection* [12]. Shim infrastructures are small bits of software intended to help outdated software keep functioning after an update alters an API to such an extent that these outdated programs would no longer function. The shims help these programs by applying fixes so that the original program does not need to change its code to handle the changed APIs. Malware can use this by masquerading as a shim infrastructure and thus forcing processes to execute the payload [12]. Shim injection needs to register itself as shim infrastructure, requiring configuration files.

The third divide is if the active techniques are *intrusive*. Techniques are intrusive when they interact with a process' thread or memory and directly alter part of the thread or memory. *APC Shell* [23] is a commonly known intrusive technique. It uses the Asynchronous Procedure Call(APC) queue by first looking for a thread that is in an alterable state, then injecting a function containing its payload to the queue and running the QueueUserAPC() function to force the thread to execute the just injected payload, actively altering the thread.

The last divide made is whether the intrusive techniques are *destructive*. If the alterations made by intrusive techniques cause the application or part of the application to stop working, they are considered destructive. A clear example of this technique is *Thread Hijacking* [9]. Thread Hijacking takes a running thread, which it suspends. It unmaps the thread's code and injects its code before resuming the thread. Due to the unmapping and replacing of the code, Thread Hijacking causes the whole thread to stop working as intended.

B. System calls

Most programs, including malware, need access to hardware or memory for parts of their code. An operating system has two modes: user mode and kernel mode. Programs typically run in user mode. Programs do not have access to any system hardware in user mode. Kernel mode, in contrast, has access to all hardware on the system, can use any instruction, and can access any memory address. System calls are the only way for programs running in user mode to ask for specific tasks from the kernel, such as process creation and file management. A system call triggers a switch from user mode to kernel mode. After this trigger, the program can request the task it needs from the operating system, after which it returns the value and switches back to user mode [16]. As mentioned before, system calls are the only means of interacting with hardware or memory, making system call traces an excellent method to analyze the behavior of a program.

C. Analysis

Two main ways of analyzing malware exist, static and dynamic [2]. Static analysis tries to analyze code or binaries without needing to execute it. Dynamic analysis is the polar opposite. Dynamic analysis does not evaluate any of the code or binary. Instead, dynamic analysis executes it in a controlled environment and tries to study the behavior observed at runtime. There also exists a third way of analyzing that combines both of the previously stated analysis methods, a hybrid analysis. This method scans the binary and observes the behavior at runtime [10]. In the following, we will cover all three options and the strengths and weaknesses of each.

1) Static Analysis: As mentioned in the introduction of this chapter, static analysis tries to infer the functionality of a program from the binary or source code without actually running the program itself [2]. When applying static analysis to a program, it usually yields a model. These models can be in the form of, for example, byte sequence n-grams, control flow graphs, or operation code frequency [26].

Not having to run the malware is considered an advantage in the case of malware analysis, as the analyst does not need to execute the malware to uncover its functionality and thus does not risk infecting the machine with said malware. There also exist disadvantages belonging to static analysis. The main one is retrieving the source code from binaries. To do this, analysts need to reverse-engineer the binaries that need to be analyzed. Reverse engineering can be challenging. For example, malware authors can implement obfuscation tactics to make examining the code statically much harder. Some examples of obfuscation tactics are:

- **Metamorphism** is a tactic in which instructions are re-ordered or dead code is inserted into some samples to make different samples of the same program appear different in the analysis [17], [29]. An example of metamorphism is adding one to a variable thrice instead of adding three to the variable.
- **Opaque constant** is a tactic in which the malware author replaces, for example, a simple assignment with a series of instructions. These instructions can be in many forms, such as loops and if-statements. These instructions will be semantically equivalent to the assignment but are more complicated to analyze statically [21].
- **Polymorphism** (also known as *packing* or *Crypting*) is, in comparison to the other mentioned tactics, the most

difficult to deal with in static analysis. A malware author can implement this tactic by wrapping their compiled binary in an encryption mechanism. When the malware sample gets executed, it calls a decryption function to decrypt the code. The key used for decryption is not always included but sometimes gets sent via a command and control center, or the malware generates it via some complicated function [22]. Afterward, it loads the malware into memory and executes it. This tactic makes static analysis challenging as different samples of the program are likely encrypted with a different key or algorithm and thus do not share or share difficult-to-spot similarities [17].

2) Dynamic Analysis: As previously mentioned, dynamic analysis relies on running the program sample to analyze it instead of looking at the code. Dynamic analysis looks at the environment while running a sample instead of the code sample. Possible factors in analysis can be the program output, the system calls made at runtime, or the memory consumption. To be able to run the program samples without putting the machines at risk of infection or other undesirable outcomes, most dynamic analyzes are executed on, for example, virtual machines or simulators to be able to limit these risks [2], [26].

The main advantage of dynamic analysis over static analysis is that the previously mentioned obfuscation tactics for static analysis evasion or slowdown do not impact dynamic analysis [26]. Even if the file is polymorphic or metamorphic to ensure that every version of the program looks different, they all have the same underlying functionality and thus behave equally when dynamically analyzed. The same functionality would result in the same effects on the system. Another reason dynamic analysis could be more suited than static is it is usually less computationally demanding since it does not need to check the whole program and every possible state or input. The computational advantage makes dynamic analysis more scalable. Being more scalable is a significant advantage when analysis tools need to handle numerous samples, as is the case with anti-viruses [10].

Even though dynamic analysis has advantages over static analysis, it also has drawbacks analysts need to consider. The first one is that malware authors can, akin to anti-static analysis tactics, implement tactics to detect dynamic analysis in their program and ensure the malware does not exhibit malicious behavior. For debuggers, the program can try to see if a debugger is present on the system, and if so, choose not to exhibit malicious behavior or throw exceptions and interrupts to try and stay undetected. The program could also call API functions to check whether a debugger is in use. Examples are "CheckRemoteDebuggerPresent" and "OutputDebugString" [2]. For simulators and VMs, multiple tactics exist to see whether the program is running on a real computer or an analysis tool. For example, instructions executed in an analysis environment take longer than regular executions [2], [26]. Slower times can be measured and compared to non-virtualized execution times and thus can be used to determine whether the code is running in a VM. Another tactic is looking for artifacts introduced by the VM. Running a simulator or virtual machine will leave certain artifacts in file systems, registries, and network behavior or look for particular hardware characteristics that some VMs might not have implemented [19]. A program can attempt to look for these artifacts to reveal whether it is running on a legitimate computer or is being analyzed [26].

The final tactic to determine whether the program is under analysis is a reverse Turing test. A reverse Turing test uses different methods to see whether a human has interacted with the system. If this is the case, then the chance the program is running on a genuine system is high, while if the test finds out that a human has not interacted with the system, the chances of being analyzed are higher. There are different ways the program can achieve this goal. It can actively look for human inputs, such as mouse or keyboard inputs, by displaying a window on the screen. The screen would need to be interacted with to disappear. It can also check for human interaction passively, for example, by checking the system for past mouse movements, keyboard inputs, process creation, or clipboards [2], [7].

3) Hybrid Analysis: The last type of analysis covered in this paper is hybrid analysis. As mentioned in the introduction, hybrid analysis combines aspects of static and dynamic analysis and thus combines their respective types' strengths [10]. With hybrid analysis, there is less worry about obfuscation tactics to circumvent static analysis. The dynamic part of the hybrid type will cover that weakness. Similarly, the static part of the analysis will help cover the potential non-malicious behavior the program might exhibit due to it detecting the dynamic analysis environment. There are some drawbacks to hybrid analysis. If a program implements tactics to circumvent both types of analysis, hybrid analysis will still struggle with the program as static or dynamic analysis would. The other drawback is that, since a hybrid analysis uses both other types, it is very resource intensive [26].

IV. Methodology

In this section, we outline our methodology for this research.

The goal of this research is to automatically identify what is executed by the injecting malware sample and the victim process after being targeted by code injection. We do this to estimate the percentage of malicious behavior missed if we observe only the malware sample.

We base our approach on dynamic analysis because, as mentioned in section III-C, malware authors often obfuscate or pack their samples before they release them onto the Internet. These tactics make it challenging to use static analysis because, as also mentioned in section III-C, static analysis focuses on analyzing binaries and has difficulties with these obfuscation tactics. In addition, even if static analysis successfully identifies the payload injected into another process, analysis of the payload would be troublesome. The payload may have a different format depending on the technique used. For example, Shellcode Injection has shell code as a payload [9], Thread Hijacking can have an entire PE file [9], and DLL Injection has a DLL file [12]. We would then need to translate the payload into system calls, as our research focuses on process behavior. This translation is difficult as each type of payload requires a different method of converting. To circumvent these difficulties, we use dynamic analysis. Dynamic analysis means we run the sample in an environment and look at the system calls the sample and the victim to determine the behavior.

We divide our methodology into three phases:

In the first phase, we dynamically run samples in a sandbox enabled with system-wide monitoring. We have to monitor the sample and its victim in the sandbox without knowing which process is the victim because only after running the sample is the victim known. The best solution is to monitor the entire system and filter the information afterward. This research focuses on what the victim does independently of the malware sample. We mainly monitor system calls that allow processes to interact with files, other processes, networking, and the registry, as these system calls describe process behavior best.

In the second phase, we leverage existing behavioral models to parse the collected traces and identify instances of code injections. After the model has detected code injection, we further analyze the produced traces to identify the victim process and an *Injection Timestamp*. We propose a definition of the Injection Timestamp:

Definition 1 (Injection Timestamp): The moment-in-time code injection has happened, combined with a process handle pointing to the victim process of this code injection.

The Injection Timestamp is the point in execution where it is clear that code injection has happened in the victim process. Any behavior from the victim after this moment is potentially influenced by the malware sample and could be malicious. We find the injection time by looking at the system call chain utilized by code injection techniques. These chains indicate when a sample completed code injection, namely after it calls the last system call in the chain. If we match the system calls of the malware sample to the chain, we can determine which call matches the final call. With this method, the process handle of the victim process is also trivial to find. This handle allows us to determine the victim process as the handle will point to it. As mentioned in section III-A, active code injection techniques interact directly with other processes, which the malware can only do using system calls. The malware needing to use system calls means that for active code injection, the handle is identifiable from the same system calls used previously for the injection time.

Definition 2 (Pre-Injection Behavior): The behavior of the malware sample before code injection has occurred.

Definition 3 (Post-Injection Bahavior): The behavior of the victim process after code injection has occurred.

When looking at the malicious behavior exhibited by the sample and the victim, we look at these two periods. As shown in figure 1, most of the malicious behavior is in the malware sample before it injects code into another process (Pre-injection behavior) and in the victim after the malware has injected code (Post-injection behavior) as it behaves according to the code injected. Before the malware injects its code, the victim process behaves as it should, so any behavior in this time frame is nonmalicious, and should thus be ignored. Similarly, the malware sample after code injection also does not exhibit much malicious behavior, as one of the reasons for code injection mentioned in section III-A is hiding.

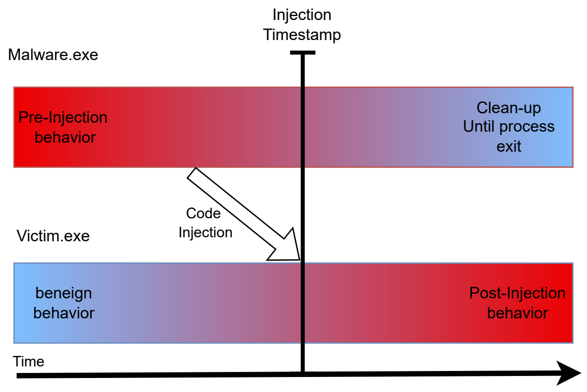

Fig. 1. Model of victim and malware behavior

In the last phase, we extract the meaningful system calls from the sandbox traces for both the victim and the malware sample. As mentioned in phase one, these traces mainly contain system calls for interacting with files, other processes, networking, and the registry. For the victim, we look at post-injection behavior and for the sample pre-injection behavior. We then use these system calls to match them to a rule-based system based on system calls for quantifying process behavior. These would ensure that most information would easily be accessible from the sandbox environment. These rules are general enough to detect pre- and post-injection behavior.

V. System Architecture

This section covers the system we developed to implement our methodology. Figure 2 shows the parts that form the system divided over the three phases explained in section IV.

Phase one contains a sample queue and the Drakvuf sandbox, a black-box malware analysis system, for running the malware samples. Drakvuf supplies the logs formed by running the sample to phase two. Phase one also consists of an agent and a server to extract additional information not covered by the sandbox and supply it to the system of phase three.

Phase two is a slightly altered version of the system built by Starink et al. [27] handling the detection of code injection in a given sample. The system utilizes the logs from phase

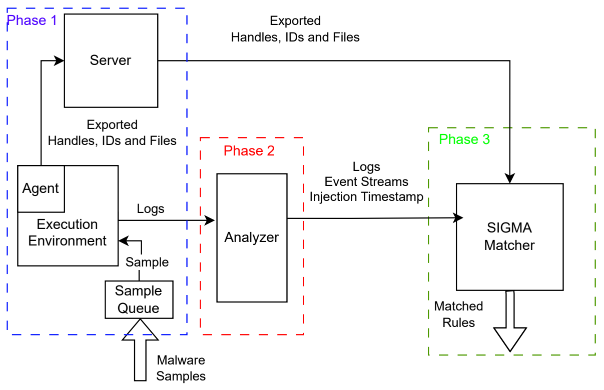

Fig. 2. System Overview

one and provides those logs, event streams, and the Injection Timestamp to phase three.

Phase three is a system that utilizes information provided by the server from phase one and the analyzer from phase two to match the behaviors of the victim and the sample to SIGMA rules to associate the observed system calls with known malicious behavior patterns.

In the following, we explain SIGMA rules to ensure the rest of the sections are easier to understand. Afterward, we discuss how each component works in more detail.

```
Title: CACTUSTORCH Remote Thread
    Creation
id: 2e4e488a-6164-4811-9ea1-
    f960c7359c40
description: Detects remote thread
    creation from CACTUSTORCH
logsource:
  product: windows
  category: create_remote_thread
detection:
  selection:
    SourceImage|endwith:
      - '\System32\cscript.exe'
      - ...
    TargetImage|contains: '\SysWOW64'
    StartModule: null
  condition: selection
```

Listing 1. Example SIGMA Rule for detecting remote thread creation by a CACTUSTORCH like program.

A. SIGMA Rules

Our system uses SIGMA rules to quantify behavior. SIGMA rules are a generic signature format flexible enough for most log sources [11].

Listing 1 shows a stripped-down example of a SIGMA rule. This rule detects when a program whose source image matches known instances of CACTUSTORCH, a particular strain of malware, creates a remote thread into a process whose path contains SysWOW64, the Windows folder containing resources that support 32-bit programs on 64-bit systems.

The most important parts of the rule for phase three are the Logsource and Detection fields. The Logsource only contains the category of a rule, which dictates which API calls link to it. For the example in Listing 1, the category is create_remote_thread, which links to, among others, the `CreateRemoteThread` API. The detection field contains one or more selection fields and the condition field. The condition field lets the matcher know what other fields should hold for the rule. In the example, it is the selection field. For the selection to apply, all fields should be valid, or in the case of a list, at least one field should be valid. The fields all apply to a specific characteristic. In the example, the characteristics are the target image name, the source image name, and the start module of the target, where the source image name is a list of possibilities, and the others have one option.

One of the reasons we use SIGMA rules is that Avllazagaj et al. [3] showed that SIGMA has a sufficiently large set of signatures for dynamic analysis. In addition, the cybersecurity landscape recognizes SIGMA as a reliable rule-set. SIGMA continuously updates its rule set to update older rules and add rules for new threads, making it a great option to future-proof this system.

B. Agent and server

The second part of phase one is an agent and a related receiving server. Some data required for the SIGMA rules is not included in the Drakvuf logs and thus requires supplementary monitoring. Some examples include file hashes and file signature data. In addition, some data is incomplete or difficult to find for all files or programs, including command lines and DNS queries.

The agent has two main functions. The first is to obtain the command lines of all programs running on the test environment and send them to the server. Some fields in the SIGMA rules need the command line of the source or target, making them required. Darkvuf logs contain command lines of processes started with `CreateUserProcess` but not the already active processes. Those command lines are what the agent provides. The second function is to launch the program specified in its command line and inject an agent DLL that implements additional logic to monitor the process in more detail. This injection shows up in the logs as an instance of code injection for the analyzer. To combat this, we placed a filter on the logs to remove any occurrence of our agent by checking the source program of the API calls.

Inside the sandbox, the agent and associated DLL have two versions, 32-bit and 64-bit. To inject a DLL into a sample, the agent and DLL require a matching bit version of the malware sample, requiring both a 32-bit and 64-bit since malware authors can compile their sample as either.

The server is listening to a predetermined address and port. When the agent or hooks submits the supplementary logs, the server handles the request, sometimes performs processing steps such as calculating hashes of received files or decoding base64, and saves the results to the disk for use during SIGMA matching.

The goal of the agent is to assist in processing the SIGMA rules by sending additional data from the virtual machine running the sample to the server.

In the following sections, we discuss the DLL functions in more detail.

1) Hooking: The agent DLL uses API hooking to gather the additional information. The APIs that get hooked are the following: `NtCreateThreadEx`, `NtOpenThread`, and `CreateRemoteThread` are similar in functionality. `CreateRemoteThread` opens a new thread in a specified process. `NtCreateThreadEx` is the underlying syscall for `CreateRemoteThread`, having the same function. `CreateRemoteThread` is hooked in addition to its underlying syscall because `NtCreateThreadEx` has different handles, making linking more difficult unless both are hooked. `NtOpenThread` opens a pre-existing thread object. The Injection Timestamp contains either a thread or process handle, which needs to be translated into a process ID to get to the victim of the process injection. Drakvuf logs have a partial list of handles and IDs, but not enough to guarantee the victim process is present in that list. Therefore, we are required to gather them ourselves. The hooks send pSairs of thread handles and process IDs to the server for future use.

Similarly, we hook `NtOpenProcess` and `NtCrea-teUserProcess`. `NtOpenprocess` is an API that opens a pre-existing process object, while `NtCreateUserProcess` creates a new process and thread. These APIs also send process IDs and handles to the server. Both also send the source image of the process they are opening to the server. The image is required because some fields in the SIGMA rules apply to the hash values of the image. We send the files instead of the hash values to keep the hook callback as computationally light as possible. In addition, `NtCreateUserProcess` also sends the current directory of the call and command line specified in the API call to the server. The command line is because, as mentioned previously, Drakvuf has command lines, but not all of them. The current directory is because Drakvuf does not have a log for this.

`LdrLoadDll` is the API that loads modules into the calling process's address space. The `LdrLoadDll` hook is necessary because many SIGMA rules require knowledge of loaded modules, which is missing in Drakvuf logs. The `LdrLoadDll` hook sends any DLL file loaded by a process to the server. The matcher later uses these files to verify signature data. This hook required a separate thread to send the data. To ensure `LdrLoadDll` always completes and retains its normal behavior, we do not send the collected data immediately in our hook callback but use a separate thread instead. The sending thread is critical because sending a request to the agent server may load other libraries, occasionally causing the callback to take too much time and crash as the library used to send the data loaded a DLL file, creating an infinite recursion of `LdrLoadDll` calls.

To ensure the thread has finished sending all the files, we hook `ExitProcess`. `ExitProcess` gets called after a program terminates under normal conditions. As soon as the malware sample exits, Windows starts terminating any running threads, including the thread sending DLL files to the server. Our hook ensures the sending thread finishes before allowing ExitProcess to continue.

Lastly, the DNS APIs. These API help processes request DNS records. The DNS API hooks send any query done by a process in combination with that process's ID to the server. These hooks are necessary due to the difficulty in linking DNS requests in the PCAP given by the Drakvuf sandbox to individual processes. The difficulty lies in the fact that the timestamps of the API calls do not perfectly match the PCAP times. There may be a time difference between registering the request and the API call.

2) Propagation: In addition to the functions mentioned previously, the DLL propagates itself into the other programs that the current program interacts with and reports the timestamps to the server when it does. It does this by injecting this DLL into any process interacted with via the `NtOpenProcess` or `NtCreateUserProcess` APIs. This propagation ensures that all possible victims of active code injection are present in the available data. Similar to the injection done by the agent, this propagation also shows up in the logs as code injection to the analyzer. Simply dropping all mentions of the process is impossible since the injection shows up as behavior of the hooked process, which the analyzer still needs to examine. To combat this, We send a message with the current time to the server for every instance of propagation. The analyzer checks if the server saved any propagation messages and, if any, removes the `CreateRemoteThread` with a timestamp close to the saved time. This slight edit to the analyzer prevents it from recognizing the propagation as code injection.

C. Analyzer

As previously stated, the part of the system that handles the detecting code injection in these samples is the system described in a paper written by Starink et al. [27]. Starink et al.'s system utilizes the Drakvuf Sandbox [28] to collect information about the run of a sample in the form of different logs. The system then uses these logs and various behavior nets, a graph-based model for detecting specific types of software behavior, to determine whether a sample has performed code injection. If it did, the system forwards the information to the system described in Section V-D.

The reason for using this system is closely related to the sandbox used. Drakvuf, as a sandbox, monitors the entire system instead of only the malware sample. This feature is essential for our project since it focuses on the victims of the injection instead of the malware programs. Additionally, the behavior net signatures used by this approach have been tested extensively in their paper and are sound enough to produce the Injection Timestamp.

D. SIGMA matching

Phase three is the core of the system, matching SIGMA rules. Because Drakvuf does not include a native way to match system calls to SIGMA rules, we made a custom matcher for this purpose. It takes information from the analyzer and the agent to determine if the behavior matches any SIGMA rules.

First, the matcher receives the Injection Timestamp from the analyzer if it finds any code injection. This Injection Timestamp contains the malware process ID and a handle for the victim process. The handle can be translated into a process ID utilizing the information saved by the server. When the matcher has the process IDs of both programs, it starts collecting all API calls made by the processes. Then, with the help of Drakvuf logs, it begins to keep track of all files, programs, and registry keys they have interacted with.

Afterward, the matcher evaluates the SIGMA rules, checking if any rules hold for the given data. The matcher checks the category of a rule and uses that to determine what objects are relevant (for example, files for file_access or images for image_loaded, etc.) and checks every possible target for that rule. For every target, it checks if every selection property is true or false. After which, it checks if the given condition holds for the results of the selection properties and, if so, saves them to report later. At the end, the matcher gives any matched rules in addition to a list of all APIs with their targets and the number of times they have occurred. Lastly, the matcher repeats the matching process for the malware sample to allow for a comparison of the malicious behavior in the victim and the malicious behavior from the sample, showing how significant the missed behavior is.

VI. Evaluation

For the evaluation of our framework, we run two separate experiments. The first one is a small-scale test to see if our system can detect any behaviors. The second runs a large set of real-world malware samples that implement code injection and looks at the behaviors found in the malware sample compared to the behaviors detected in victims.

In the following sections, we cover the datasets for both experiments, the parameters used, and the results of these experiments.

A. Datasets

For the first experiment, we use a dataset of 30 samples where we know what code injection techniques are used by all samples. The sample set combines real-world and custom samples crafted by Starink et al. [27] and sufficiently covers the active code injection techniques. We included real-world samples to ensure our system is not biased to our custom samples.

For the second experiment, we use the VirusTotal academic Data Set. In particular, a subset of 436 samples from 2017. We specifically use the 2017 sample set as results from Starink et al. [27] show that this contains the most active code injection samples. The subset includes only the active code injection samples from this set. This subset ensures we waste no time on samples that do not implement code injection or implement passive techniques that our system cannot handle, further discussed in section VIII.

B. Experimental Setup

For the parameters of both experiments, we set the maximum execution time of samples to 10 minutes, which is four minutes longer than Starink et al. [27] advised. The reason for setting the execution time longer is that the agent needs time to collect all command lines, and our research has a greater emphasis on the behavior after code injection than the research of Starink et al. We limit the execution time to 10 minutes as Küchler et al. [14] showed that 81% of malware does not need longer than 10 minutes to comeplete.

The SIGMA rules used for this experiment are a subsection of version "Release r2024-07-17". To stick as closely to the general behavior of the application, we picked the Windows subsection of the rules. Due to time constraints, we did not implement all Windows subcategories and of the implemented categories, some have had skipped fields. Some were ignored due to difficulty in implementation, while others only applied to one or two rules and would take a disproportionate amount of time to implement for their usefulness.

For this research, there are some ethical concerns that we need to consider because we utilize dynamic analysis and run malware samples in a sandbox. First, to ensure no physical machines are compromised, we run the malware samples in an isolated execution environment. Second, we provide the sandbox, and subsequently the samples, with internet access to ensure as many samples exhibit their behavior. We protect the university network against malware spread or denial of service attacks by dropping all internet traffic that has a destination within the network. In addition, the testing environment has a firewall with strict rules. These rules block frequently used ports for TCP and UDP-based protocols. Last, we roll back the VM that runs the samples to a fresh snapshot after every cycle. This rollback ensures the elimination of any remnants of the malware, further protecting against any denial of service attempts. We verified these countermeasures in addition to the Ethics Committee of the University of Twente approving the countermeasures [24].

C. Results

In the following section, we present the results of our small-scale test proving our system can detect process behaviors. After which, we present our findings of the large-scale experiment showing what part of malicious behavior is in victim processes.

TABLE I
NUMBER OF OCCURRENCES OF EACH SIGMA RULE IN CAPABILITY TEST

	Number of Occurrences
Creation of a system .dll file in unusual location	18
Unusual target for remote thread creation	16
Access request with "Process_all_access" mask	13
Creation of a system executable in other folder	1

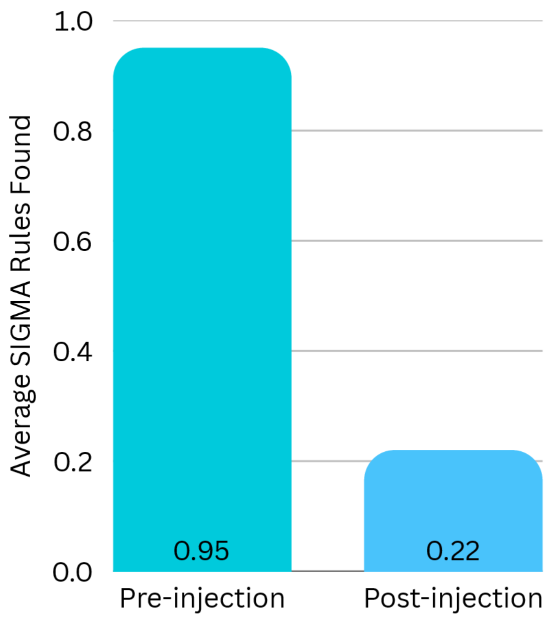

Fig. 3. Average number of SIGMA rules found pre-injection and post-injection in capability test

1) Capability Test: Figure 3 shows the average amount of SIGMA rules found pre-injection(left) and the rules found Post-injection(right). In total, we found 39 rules pre-injection and nine rules post-injection, meaning, on average, 18.8% of the malicious behavior is in the victim. Table I shows the frequency of rules in either the victim or the injector. What is notable about the results is that 'uncommon targets for remote thread creation' occurred 16 times, showing that the SIGMA matcher correctly concluded that some custom samples target the notepad application. Another rule that stands out is the 'Creation of a system .dll file in an unusual location.'. This rule occurs 18 times. A possible explanation is that several injectors utilize mimicking system DLL files in their code injection.

Some samples from the dataset failed to run. This failure was due to the payload injected into the victim process having either no or non-malicious behavior, such as creating a MessageBox. Neither of these would show up as activity in the API logs for the victim, resulting in an error seeing no behavior.

2) Large scale Test: Table II shows the distribution of the sample set. Of the 436 samples we ran, 166 have stopped showing code injection, and 57 showed code injection but failed in the SIGMA matching. The other samples implemented code injection and worked as expected in the SIGMA matcher. An important note is that the techniques do not add up to the expected 213 as some samples implement multiple

TABLE II
DISTRIBUTION OF TECHNIQUES IN THE EXPERIMENT DATASET

	Pre-Injection
No Injection	166
Failed to match	57
Process Hollowing	138
Generic Shell Injection	44
Thread Hijacking	32
Classic DLL Injection	1

TABLE III
TOTAL SYSTEM CALLS PER INJECTION TECHNIQUE

	Pre-Injection	Post-Injection
Total Average	2.167.483 (43.7%)	2.785.266 (56.3%)
Process Hollowing	1.103.654 (41.3%)	1.570.341 (58.7%)
Generic Shell Injection	860.242 (47.0%)	971.522 (53.0%)
Thread Hijacking	190.674 (44.1%)	241.221 (55.9%)
Classic DLL Injection	12.913 (85.5%)	2.182 (14.5%)

injection techniques.

Table III shows the total amount of monitored syscalls and the percentage of system calls that appear in the injector (pre-injection) and the victim (post-injection) per injection technique. When we look at only the number of tracked system calls made by the injector and the victim processes, we see that, on average, the injector makes 43% of system calls, and the victim processes make 57% of calls. If we divide the system call counts over the different techniques, we see that the percentages have some slight differences, except for Classic DLL Injection sample with 85.5%, the amount of system calls made by the injector stays between 40 and 50 percent.

This percentage is an upper bound of missed behavior as not all system calls are attributable to influenced behavior. Some victim processes might continue their execution if the technique is not destructive. The split per technique shows Process Hollowing has 58.7 percent of system calls originating from the victim. Because Process Hollowing is a destructive technique and thus destroys most of the original functionality of the victim process, its percentage will be relatively close to reality.

We translate these system calls to SIGMA rules to give more context to the missed behavior. Figure 4 shows the average amount of SIGMA rules found in the injector(left) and the victim(right). We found 40 rules in the injectors and 71 rules in the victims, meaning we found 64% of SIGMA rules in the victims, mimicking the results we found earlier.

Table IV shows the frequency of rules in the victim and the injector combined. The two most found rules by a significant margin are processes accessing desktop.ini and processes requesting access to the LSASS process, which is responsible for enforcing security policies. The first could hint at a popular target for injecting other types of malware, as it is in all folders. While the latter could hint at malware samples wanting to alter security policies.

Lastly, Table V shows the three most frequent rules per malware technique and how often they were found for that technique. As is shown in the table, we found no rules in the Classic DLL Injection sample. Thread Hijacking samples only had two different rules found where mimicking a system process occurred once. Generic Shell Injection and Process Hollowing samples had the most rules found, which may be due to being the most prevalent in the sample set.

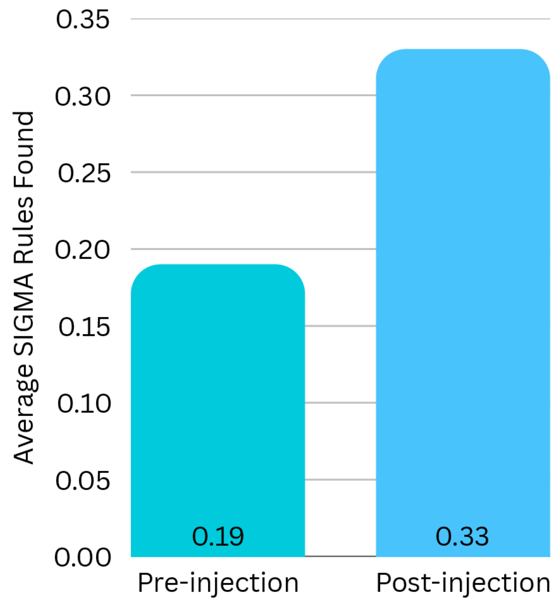

Fig. 4. Average number of SIGMA rules found pre-injection and post-injection in the large scale test

TABLE IV
NUMBER OF OCCURRENCES OF EACH SIGMA RULE IN LARGE SCALE TEST

	Number of Occurrences
unusual processes accessing desktop.ini	66
suspicious access requests to LSASS process	29
uncommon processes creating remote threads	7
executable with system process name in other folders	6
creation of PowerShell module by non-PowerShell process	4
files being created in the Windows startup directory	3
file with a suspicious extension created in the startup folder	2
suspicious processes accessing windows credential manager	2
possible malicious content triggered by application shims	1
API Master keys access by an uncommon application	1

Although translating the system calls to SIGMA rules allowed more understanding of the behaviors of the injectors and the code injection victims, they cannot fully capture all behaviors exhibited by the processes. One example of this is DNS spraying. Looking into the DNS requests of the victims of some samples, we saw many requests to seemingly random URLs. When viewed as separate requests, these requests do not trigger any rules, but when seen as a whole, it becomes clear that it is unnatural behavior. This example confirms the SIGMA rules found are only a lower bound of the total behavior while still being behavior from a victim, and many detection solutions would thus miss it.

TABLE V
MOST FREQUENT SIGMA RULES FOUND PER CATEGORY AND HOW OFTEN THEY OCCURRED

	Top 3 most common rules found
Process Hollowing	1. Access desktop.ini (44) 2. Uncommon remote thread (4) 3. System Executable other folder (4)
Generic Shell Injection	1. Acess LSASS (29) 2. Acess desktop.ini (14) 3.Uncommon remote thread (3)
Thread Hijacking	1. Access desktop.ini (10) 2. System Executable other folder (1) 3. N/A
Classic DLL Injection	1. N/A 2. N/A 3. N/A

VII. DISCUSSION

In this section, we will discuss our results and provide additional perspectives.

Table III shows the system calls recorded pre and post-injection. Most of these entries have similar ratios for the percentage of system calls counted for the injector and the victim, except for Classic DLL Injection. Combining this with Table V showing Classic DLL Injection had no rules found and the knowledge that the analyzer only detected Classis DLL Injection once in the entire sample set gives ground to consider this one sample is an outlier and ignore it when pulling conclusion from the results.

When we ignore the outlier, the results show that 53 to 58.7 percent of observed system calls are post-injection. As mentioned in section VI-C2, this percentage is an upper bound of what could be considered malicious behavior. Non-destructive techniques such as Generic Shell Injection do not remove the regular functionality, and because of that, proceed to make system calls that belong to benign behavior post-injection. In addition, even when considering destructive techniques such as Process Hollowing and Thread Hijacking, not all system calls recorded post-injection are malicious. We grouped the Drakvuf logs per process ID, so it is possible that while one thread is targeted by code injection, another continues to run as normal before finishing or crashing later. Finally, regarding system call counts, a raw count of system calls can indicate potential malicious behavior, but on its own, it is inconclusive. There is no set amount of system calls per malicious behavior. A process can require a single system call to create a remote thread if all the parameters are ready. The process may take two system calls if a process handle is needed, and in some cases, it could require even more.

As mentioned in section VI-C2, to give context to these system calls, we mapped them to SIGMA rules. We found 111 SIGMA rules in 213 samples, split 40 in injectors and 71 in victims, keeping the pattern of observing more behavior in post-injection(64%) than pre-injection(36%). The found rules are broken down per rule in table IV, and table V shows the most repeated rules and how often they occurred for each technique. The numbers in table V do not necessarily add to the numbers in table IV as some samples implement multiple code injection techniques and rules are counted for both.

A couple of notable features stand out from both tables. The first one is that the second most occurring rule, an access request to the LSASS, is found 29 times in total, and that same rule has been found 29 times for Generic Shell Injection samples, meaning that this is the only technique in the sample set that accesses the LSASS. Further research would be necessary to confirm whether this is a coincidence of the sample set or if this rule is exclusive to this technique. The second notable feature is that the most commonly found rule, accessing desktop.ini, is spread proportionally across all the techniques. It follows that accessing desktop.ini is not only prevalent in malware using code injection but also widespread across many techniques. The last notable feature is we discovered only two different rules in Thread Hijacking samples. One is desktop.ini, which appeared 10 times, and the other is creating a file with a system executable name outside of usual system folders, which only occurred once. Similar to Generic Shell Injection samples, further research is needed to check if this is specific Thread Hijack only utilized these rules, if it was a coincidence for the sample set, or if our SIGMA rules do not cover the behavior these samples inject.

Our system has not found any rules in a sizable portion of the samples, which can have multiple reasons. The first reason is, as mentioned in section VI-B, our system only utilizes a subset of the SIGMA rules. The behaviors of these samples could still be malicious and detectable, simply not with the current set of rules. The second reason is similar to the first one. It is possible that the behaviors of these samples have not yet been recorded into SIGMA rules. SIGMA rules are updated often, and every update adds more rules. The system may find more behaviors in the sample set with a later version of the SIGMA rules. The last reason is, as mentioned in section VI-C2, the system matches SIGMA rules per system call, so behaviors like DNS spraying are not detectable with this system. Future work would need to add a different detection mechanism to identify such behavior.

A possible note to the agent and the hooked API calls is only hooking the ExitProcess API is insufficient. It might be lacking because when a process does not terminate normally, the program does not call ExitProcess. But, if a program does not exit normally, it has likely crashed. If the malware crashes, we do not know whether the sample has successfully performed code injection or stopped before. Thus, crashing is an undefined behavior, making the matching process faulty regardless.

Lastly, as mentioned in section VI-A, our results are gathered from the VirusTotal 2017 dataset. The system has been tested on all active techniques in the capability test and works with each. Even though the sample set only had four types of techniques, they are the most prevalent techniques of this time and, thus, the most important. A more in-depth sample set could provide more insights into all code injection techniques, but due to time constraints, it is not feasible for this thesis. Our framework is also general enough to be utilized on newer and older samples. Provided they implement active code injection, our system can analyze it. Additionally, the system can effortlessly process multiple samples simultaneously. The amount of simultaneous samples depends on the hardware available.

VIII. Limitations

The current implementation has some limitations future references should consider. The first limitation is that the matcher only recognizes active code injection techniques. Covering active techniques means that six out of the seventeen injection techniques are not covered. As mentioned in section III-A, active techniques interact with the victim process directly, making figuring out the victim process trivial by looking at API calls. Passive techniques would need more analysis to determine the injection method and, more importantly, which processes or processes have been affected. In addition, future work should expand the agent to collect this additional information for those processes.

The second limitation is that the analyzer uses a user-mode agent. Having the agent present could imply that some malware samples might detect that they are being observed and will not show any or reduced malicious behavior. The original idea was to have one or more Drakvuf modules have the same functionality as the agent, but that proved too difficult for quick prototyping. Future researchers could improve the system by making the agent a legitimate Drakvuf module. Should the agent be a Drakvuf module, it would ensure that malware samples would not detect the agent and, thereby, not exhibit malicious behavior as modules use hypervisor-level monitoring.

The last limitation that the analyzer faces is that we manually hook the syscalls, and it is feasible that we missed some. All categories have syscalls associated with them, but it could be that some more obscure system calls that have similar outcomes are used in some malware samples. Another possible problem with hooking syscalls is that some samples might implement indirect system calls. When a sample circumvents the standard API call, it will not trigger our hooks. Not triggering our hooks means the agent cannot collect the required information to complete the matching process. Making the agent a Drakvuf module would remedy this limitation, as hypervisor monitoring would contain these indirect system calls.

IX. Related work

Starink et al. [27] have made a taxonomy defining 17 code injection techniques. This taxonomy divides these techniques in the following way. First, they determined whether the technique is *active* or *passive*, meaning it directly interacts with the victim process if it is active or passive if it does not. After this split, they split passive techniques once more based on the technique needing to store a specific and persistent configuration on the disk, *Configuration-based*, or not. The classification of active techniques splits into two more criteria, the first being *Intrusive*, meaning that the technique alters part of the victims' memory or active threads. The second criterion is if the technique is *Destructive*, meaning that the technique causes part of or the entire victim application to stop working. In addition to the taxonomy, Starink et al. have made a classifier based on the taxonomy and behavior graphs to attempt to detect all 17 techniques.

Korczynski et al. [13] implement a code injection detection mechanism based on taint analysis named Tartaturs. They propose to flag the sample's memory as *tainted*. After every instruction, Tartarus checks whether the memory for that instruction is tainted, if it initiates code reuse, or if it belongs to a code reuse buffer. If any condition holds, the instruction gets added to the malware execution trace. To detect code injection, Tartaturs checks the trace for places where the control flow graph of the trace switches from one process to another. Afterward, Tartarus analyses the trace to create a code-injection flow graph. Tartaturus could, when tested, detect all code injections in the samples. A significant drawback of taint analysis when considering malware analysis is the speed. The amount of checks and steps required for every instruction slows down analysis drastically, making it often unfit. The slow speed is the reason our system does not utilize taint analysis.

Sandboxes are dynamic analysis systems. Some examples of sandboxes are Cuckoo [25] and Joe sandbox [15]. These systems often employ VMs to analyze samples without putting genuine systems at risk [18]. These systems are practical when many samples need to be tested, as sandbox testing is usually automatable. In addition, sandboxes typically offer a variety of reports after testing a sample. Network activity, behavior analysis, and static analysis are some examples. These factors make it so that sandboxes are used widely in malware detection and studies rated. The problem with using sandboxes for code injection is that they do not always detect them. As Korczynski et al. compared their system to two common sandboxes, Codisasm [5] and Cuckoo, they concluded that they missed 40 and 60 percent of code injection, respectively. In addition, Mills et al. [18] mention that sandboxes often have trouble detecting malware when these techniques check for monitoring, as is covered in section III-C. Combining the low chance of detecting code injection with not following the victim when code injection is detected kindled the idea of this thesis.

Barabosch et al. [4] implement a detection system for host-based code injection. This system, named the bee master, utilizes the honeypot paradigm. The bee master has one main process, *the queen*, that creates multiple sub-processes known as *workers*. Because the queen starts the other processes, it knows what their code and memory pages should be. When the queen detects changes in a worker, they shut them down and analyze the memory dump. This approach has some flaws. For one, the author based the bee master on the assumption that the malware samples target processes via the process name or inject its code into every available process, which not all malware samples do. Additionally, techniques such as process hollowing, which start new threads and alters those, are impossible to detect for this implementation.

Avllazagaj et al. [3] has researched malware execution across multiple operating system versions. The goal was to find an invariant that appeared for a given malware sample in most OS versions instead of one specific scenario. Their dataset contained 7.6 million execution traces of benign, malicious, and potentially unwanted programs (*PUP*) across mostly five operating system versions. They used SIGMA rules to model the behaviour per trace and later compared how different each trace of a sample was. This research concluded that malicious samples have a higher variability than benign samples. Where Avllazagaj et al. utilized SIGMA rules to model the behavior of the samples and determine when a sample is or is not malicious, we utilize SIGMA rules to model the behavior of the sample and their victim to estimate the amount of missed behavior.

X. Conclusion

We proposed a framework that automatically identifies behavior exhibited by injecting malware samples and their victim processes after being targeted by code injection. We used this framework to collect empirical evidence on the behavior of malware samples implementing code injection from 2017 and their victims. Our empirical results show that an average of 56.3% of observed system calls originated from the victim process of code injection, and 64% of detected SIGMA rules were detected in the victims. Finally, our study provided important insights and takeaways for future research on malware behavior analysis.

XI. Acknowledgments

We are very grateful to our supervisors Andrea Continella, Jerre Starink and to Antionia Affinito who have provided us with valuable feedback and proofreading.

References

[1] visited on 2025-04-2. [Online]. Available: https://www.av-test.org/en/statistics/malware/.

[2] 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, (visited on 2025-04-2), ISSN: 0360-0300. DOI: 10.1145/3365001. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3365001.

[3] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitraș, "When malware changed its mind: An empirical study of variable program behaviors in the real world," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3487–3504.

[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, ISBN: 978-3-319-08509-8.

[5] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 745–756, ISBN: 9781450338325. DOI: 10.1145/2810103.2813627. [Online]. Available: https://doi.org/10.1145/2810103.2813627.

[6] K. Bridge, *Process security and access rights - win32 apps*, visited on 2025-04-2, Jul. 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/procthread/process-security-and-access-rights.

[7] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*, ser. ROOTS, Vienna, Austria: Association for Computing Machinery, 2017, ISBN: 9781450353212. DOI: 10.1145/3150376.3150378. [Online]. Available: https://doi-org.ezproxy2.utwente.nl/10.1145/3150376.3150378.

[8] S. Dahiya, L. N. Rokanas, S. Singh, M. Yang, and J. M. Peha, "Lessons from internet use and performance during covid-19," *Journal of Information Policy*, vol. 11, pp. 202–221, 2021.

[9] J. Desimone, *Hunting in memory - elastic security labs*, visited on 2025-04-2, Jun. 2022. [Online]. Available: https://www.elastic.co/security-labs/hunting-memory.

[10] N. Dutta, N. Jadav, S. Tanwar, H. K. D. Sarma, and E. Pricop, "Introduction to malware analysis," in *Cyber Security: Issues and Current Trends*. Singapore: Springer Singapore, 2022, pp. 129–141, ISBN: 978-981-16-6597-4. DOI: 10.1007/978-981-16-6597-4_7. [Online]. Available: https://doi.org/10.1007/978-981-16-6597-4_7.

[11] *GitHub - SigmaHQ/sigma: Main Sigma Rule Repository — github.com*, https://github.com/SigmaHQ/sigma, [Accessed 04-04-2025], 2025.

[12] A. Hosseini, *Ten process injection techniques: A technical survey of common and trending process injection techniques*, visited on 2025-04-2, Oct. 2023. [Online]. Available: https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process.

[13] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1691–1708, ISBN: 9781450349468. DOI: 10.1145/3133956.3134099. [Online]. Available: https://doi.org/10.1145/3133956.3134099.

[14] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? time-based evolution of malware behavior in sandboxes," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021. DOI: 10.14722/ndss.2021.24475.

[15] J. S. LLC, *Automated malware analysis*, visited on 2025-04-2. [Online]. Available: https://www.joesandbox.com/#windows.

[16] S. Mandal, *Introduction of system call*, visited on 2025-04-2, Sep. 2023. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-system-call/.

[17] 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, (visited on 2025-04-2), ISSN: 0360-0300. DOI: 10.1145/3329786. [Online]. Available: https://doi.org/10.1145/3329786.

[18] A. Mills and P. Legg, "Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques," *Journal of Cybersecurity and Privacy*, vol. 1, no. 1, pp. 19–39, 2021, ISSN: 2624-800X. DOI: 10.3390/jcp1010003. [Online]. Available: https://www.mdpi.com/2624-800X/1/1/3.

[19] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 1009–1024.

[20] A. Mohanta and A. Saldanha, "Code injection,process hollowing, and api hooking," in *Malware analysis and detection engineering a comprehensive approach to detect and analyze modern malware*. Apress, 2020, pp. 267–267. DOI: 10.1007/978-1-4842-6193-4.

[21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC*

2007), 2007, pp. 421–430. DOI: 10.1109/ACSAC.2007.21.

[22] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim, "File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–45, 2022.

[23] *Process injection: Asynchronous procedure call, subtechnique t1055.004 - enterprise | mitre attck®*$_2$021, visited on 2025-04-2, Jan. 2021. [Online]. Available: https://attack.mitre.org/versions/v10/techniques/T1055/004/.

[24] D. Reidsma, J. van der Ham, and A. Continella, "Operationalizing cybersecurity research ethics review: From principles and guidelines to practice," in *2nd International Workshop on Ethics in Computer Security, EthiCS 2023*, Internet Society, 2023.

[25] C. Sandbox, *Cuckoo Sandbox*, visited on 2025-04-2. [Online]. Available: https://github.com/cuckoosandbox/cuckoo.

[26] R. Sihwail, K. Omar, and K. Ariffin, "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 8, no. 4-2, pp. 1662–1671, 2018, visited on 2025-04-2. DOI: 10.18517/ijaseit.8.4-2.6827. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85055342494&doi=10.18517%2fijaseit.8.4-2.6827&partnerID=40&md5=e5c8707776645064725f7f6507364062.

[27] J. Starink, M. Huisman, A. Peter, and A. Continella, "Understanding and measuring inter-process code injection in windows malware," in *19th International Conference on Security and Privacy in Communication Networks, SecureComm 2023*, 2023.

[28] Tklengyel, *Tklengyel/drakvuf: Drakvuf black-box binary analysis*, visited on 2025-04-2. [Online]. Available: https://github.com/tklengyel/drakvuf.

[29] M. Tofighi Shirazi, "Analysis of obfuscation transformations on binary code," (visited on 2025-04-2), Theses, Université Grenoble Alpes, Dec. 2019. [Online]. Available: https://theses.hal.science/tel-02907117.