A Measurement Study on Interprocess Code Propagation of Malicious Software

Thorsten Jenke(✉), Simon Liessem, Elmar Padilla, and Lilli Bruckschen

Fraunhofer FKIE, Zanderstraße 5, 53177 Bonn, Germany
{thorsten.jenke,simon.liessem,elmar.padilla,
lilli.bruckschen}@fkie.fraunhofer.de

Abstract. The propagation of code from one process to another is an important aspect of many malware families and can be achieved, for example, through code injections or the launch of new instances. An in-depth understanding of how and when malware uses interprocess code propagations would be a valuable aid in the analysis of this threat, since many dynamic malware analysis and unpacking schemes rely on finding running instances of malicious code. However, despite the prevalence of such propagations, there is little research on this topic. Therefore, in this work, we aim to extend the state-of-the-art by measuring both the behavior and the prevalence of interprocess code propagations of malicious software. We developed a method based on API-tracing for measuring code propagations in dynamic malware analysis. Subsequently, we implemented this method into a proof-of-concept implementation as a basis for further research. To gain more knowledge on the prevalence of code propagations and the code propagation techniques used, we conducted a study using our implementation on a real-world data set of 4853 malware samples from 1747 families. Our results show that more than a third (38.13%) of the executables use code propagation, which can be further classified into four different topologies and 24 different code propagation techniques. We also provide a list of the most significant representative malware samples for each of these topologies and techniques as a starting point for researchers aiming to develop countermeasures against code propagation.

Keywords: Malware · Code Injections · Code Propagation

1 Introduction

Malware remains a major threat to computer security. 2019 alone saw the release of over 131 million samples [3]. Furthermore, the number of cyber attacks involving ransomware alone almost doubled between 2020 and 2021 [25], with an estimated damage of $20 billion [30]. Developing adequate protection against these threats requires an in-depth analysis of samples. However, malware authors employ various different obfuscation techniques to thwart analysis. During the

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 571, pp. 264–282, 2024.
https://doi.org/10.1007/978-3-031-56583-0_18

infection phase, malware uses various evasion/obfuscation and deployment techniques, such as packer. A commonly found technique is the propagation of code from one process to another, henceforth referred to as code propagations. Examples include injections into seemingly benign processes to conceal or launch multiple copies to achieve redundancy.

This particular evasion technique has an especially negative impact on dynamic analysis. Most debuggers offer the functionality to follow child processes, but injections are generally harder to follow. API-hooking/observation is another common way to observe malicious functionality. Although it is possible to detect code propagation with system-wide API-hooking, the analyst must be aware that observing the API calls of a singular process may not reveal the entire malicious activity, and observing the activity of all processes will yield too much information to be useful. To garner more granular information about the malware's behavior, virtual machine introspection or full system emulation is used. These methods lead to a similar problem as the system-wide API-hooking. Monitoring the singular launched malware process may not be sufficient due to code propagations, and the information overload of observing the entire system is even more prohibitive in these cases. Therefore, it is imperative to be able to track the malware through new process spawns and code injections in order to filter out the malware's behavior from the entire system's behavior. Alternatively, single-process emulation poses the problem that it must be able to emulate multiple processes to allow the malware to perform its code injections and process creations.

Code propagations have an impact on almost all dynamic analysis techniques. Therefore, the goal of this work is to extend the state of the art by addressing the distribution of malicious code across processes executed by a sample. We believe that this will improve the understanding of the evasive movement of malware and help the community develop tools to address code propagations in dynamic analysis. To gain knowledge about code propagation, we have developed a proof-of-concept implementation consisting of a sandbox part and an analysis part. The methodology of our implementation for detecting code propagations can be incorporated into different dynamic analysis techniques. Using this implementation, we conducted a longitudinal study, using a dataset of real-world malware. We also provide a list of malware samples, representing each class of code propagations found in our study, to allow researchers to immediately test their countermeasures against malicious code propagations.

This paper provides the following contributions:

- A longitudinal study on a representative real-world malware corpus showing the prevalence of code propagation and the variety of used code propagation techniques.
- A method to measure code propagations in dynamic malware analysis.
- A list of malware samples labeled with their respective code propagations.

2 Related Work

Despite the pervasiveness of code propagations and their obstacles, there is little research on this topic.

Barabosch et al. contributed three works [5–7] on process injection based on the analysis of memory dumps and the evaluation of memory regions based on a variety of attributes. They have proposed multiple ways to detect code injections. One technique is based on applying honeypot principles to processes and capturing the injections that way; another is to look for common properties in processes that have been the targets of injections. They have also proposed a taxonomy to describe code injections.

Two works by Korczynski et al. [20,21] propose a framework to capture malware injections and code reuse attacks. They have developed a program called Minerva based on the taint analysis of PANDA [12], aiming to move away from the write-then-execute metric, and propose a different framework based on the flow of information. Other taint analysis approaches include DiskDuster by Bacs et al. [4], API-Chaser by Kawakoya et al. [18], and Panorama by Yin et al. [38] by capturing system-wide information flow.

Ispoglou et al. [14] proposed a malware called malWASH that makes a great use of process injections to hide its functionality from dynamic analysis systems.

Since process injections are also commonly used in packers to thwart analysis, there have also been unpackers focused on tackling them. There has been much research focused on creating several generic unpackers [8,11,13,15–17,19,26,35–37].

Unlike the related work, we do not focus on a single use-case to thwart code propagation. Instead, the purpose of our research is to demonstrate that malware often utilizes code propagation, including the generation of new processes, and to investigate the characteristics of their implementation.

3 Code Propagation

This section covers the basics of code propagation. Since our study will focus on Microsoft Windows, we will use nomenclatures from the Windows ecosystem. However, the concepts presented in this paper should also be applicable to other operating systems.

3.1 Definition

Barabosch et al. [6] propose a definition for host-based code injections. However, their definition does not include other forms of code propagation besides code injections. Therefore, we propose our own definition to broaden the scope of our work.

We define code propagation as an instance of code distribution by a malware $\mathcal{M}$ that meets the following criteria:

1. The code is propagated outside of $\mathcal{M}$'s currently running process.
2. To propagate, the malware $\mathcal{M}$ either uses previously written binaries to spawn new processes or injects code into newly created or already running processes.

3. The code is distributed with the intent of immediately executing it.
4. The distribution does not require any interaction from the user or other processes.

Criterion 1 manifests the idea that the code must leave the currently running process, as our goal is to focus on the movement of malware code across process boundaries.

Criterion 2 defines the way in which malware can propagate its code. This enforces that spawning a process from a binary that was present on the machine before the malware was launched and without injecting any code is not considered a code propagation.

We have added criterion 3 to explicitly exclude persistence techniques, such as using the scheduler or autorun to launch malware after the operating system has been restarted.

Criterion 4 ensures that the code propagation is performed completely autonomously and does not use side effects of other processes.

Code propagations can also be observed in goodware; however, we focus our attention on in-depth malware analysis. Therefore, we do not consider goodware in our research.

3.2 Representation

A set of code propagations conducted by a single malware sample can be described as a tuple consisting of a graph and a weight function that describes the code propagation technique used: Let $G_{mov} = (V, E)$ be a directed graph that describes the movement of malware between processes for a given sample, where V is the set of malware instances and E is the set of code propagations between them.

There are no reflective edges $e \in E$ since code propagations exist only between two different processes.

Additionally, we define the weight function $f : E \rightarrow X$ where X is the space of all possible code propagation techniques.

Therefore, the tuple (G_{mov}, f) describes the code propagations for a given malware. G_{mov}, with $|V| > 0$, $|E| \geq 0$, and $|E| \geq |V| - 1$, is called the topology of the code propagations. The weight function will be discussed in the next segment.

3.3 Techniques

This section explains the code propagation techniques by introducing the function $f : E \rightarrow X$.

As shown in Fig. 1, the shortest way to propagate code is to spawn a process from a previously written binary or using various injection techniques. The differences between injection techniques can be summarized in the reuse or allocation of resources. For an injection, handles to a **process**, **memory regions**, and **thread** are needed. These steps are also represented in the taxonomy proposed

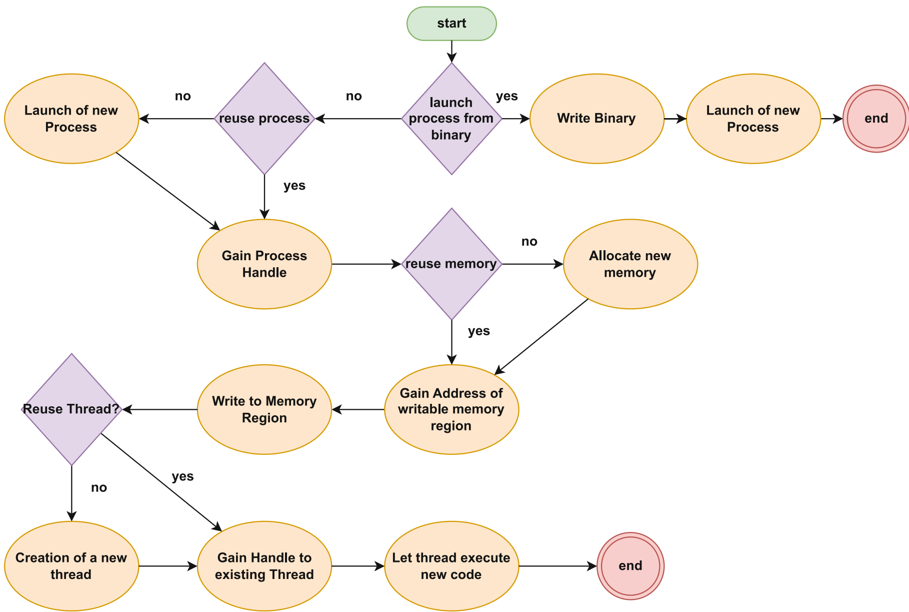

Fig. 1. Decision tree to propagate code in the system. There are two primary paths, one for starting new processes and the other for describing code injections. Code injections differ in their allocation or reuse of resources.

by Barabosch et al. [7]. For each of these resources, it is possible to allocate new ones or reuse existing ones. To ensure general applicability, our proposed method covers methods found in hundreds of different malware families. Many of them have different approaches to implementing code propagations and possibly faulty or broken implementations. There may even be incomplete propagations to subvert analysis or implementations with superfluous steps. As shown in Fig. 1, a code propagation consists of eight steps that must be taken into account. To be able to represent these steps, we have decided that the result of our weight function $f : E \rightarrow X$ should be an unsigned integer of eight bits in length. The advantage of this implementation is that it is the minimal representation possible. Each bit in X represents a trait of the code propagation, so that the eight bits describe every possible interaction between processes. With a bit value of 1 if the stated condition is true and 0 otherwise:

Bit 1: Was a new process spawned?
Bit 2: Was a handle on a process obtained?
Bit 3: Was a new thread spawned?
Bit 4: Was a handle on a thread obtained?
Bit 5: Was new memory allocated?
Bit 6: Was a memory region reused?
Bit 7: Was the data written to the memory region(s)?
Bit 8: Was the thread resumed?

Note that it is possible to generate interprocess interactions with this representation that do not fulfill our definition of code propagations.

4 Measuring Code Propagations

In this section, we describe how to determine the code propagations of a given sample. Code propagation functionality in user mode must be realized through the operating system API. So in order to record this behavior, we must record the functions used to perform the code propagation. Tackling this problem with static analysis is very time-consuming and not feasible due to the multiple possible layers of obfuscation present in modern day malware. Therefore, a dynamic analysis approach was chosen.

Using this approach, each call to a function affecting the eight states described in Subsect. 3.3 was recorded. This recording included the calling process, the parameters passed, the return value, and the values in the relevant out parameters. These recordings contain all the information needed to derive the code propagation topology and techniques used by the malware.

Every code propagation starts with the creation of the handles on a process and a thread. This is done by functions that either spawn a new process or thread or leverage existing processes. Therefore, these functions always initiate a new code propagation. At this point, the process ID (abbr. PID) is known, since the functions for opening or creating processes are directly linked to the PID, except for NtCreateUserProcess, which only returns a handle. In this case, the PID must be derived from the handle, either by using GetProcessId or by parsing the kernel structures. In the next step, the remaining interaction between the currently analyzed process and the target process is examined, e.g., if WriteProcessMemory is used, then the second to last bit is set as described in Subsect. 3.3.

This set of code propagations still includes the process spawns that do not execute code written by the malware. Therefore, the code propagations that do not belong to the family of code injections have to be filtered from the set. So, first, the function used to spawn the process is determined and its parameters are interpreted.

Most functions for spawning processes take an application name parameter and a command line parameter. The main executable used to the spawn the process is checked against a list of files manipulated by the malware and a list of all PE files on the system before the malware has been executed. However, there is a list of binaries that take a special role, like cmd or rundll. These executables are used to spawn other images, which in turn have to be further interpreted. Simply checking for a manipulated image in the command is not sufficient, since, e.g., the del command deletes the file and does not spawn it, so a block list of the shortened commands is used to filter the calls. Also, the DLL run by rundll has to be checked against the previously mentioned manipulated file and allow lists. This leaves only malicious process creations.

The results of the analysis are represented as a graph, with the processes as nodes and the system calls as edges, as described in Sect. 3.2.

4.1 Implementation

The majority of malware is released for Microsoft Windows [3]. Therefore, we focus on malware released for this operating system.

In our search for suitable frameworks to record API calls, we considered two dynamic analysis frameworks, PANDA [12] and Drakvuf [23], as well as Microsoft's hooking framework Detours [27]. PANDA is based on QEMU and provides very fine-grained information to the analyst, such as virtual machine introspection or callbacks at different stages of QEMU's execution. However, it would require a significant amount of additional work to hook arbitrary user land API-functions. Drakvuf is a sandbox based on libvmi that allows introspection into virtual machines. Unfortunately, there seem to be several hurdles when trying to hook user land API-functions [10]. Although both tools are very powerful, they require a significant investment of development time to be able to set the required user land hooks. Since the goal of this work is to explore code propagations and describe how to detect code propagations, we have chosen the most expeditious variant, which is Detours. Detours was developed by Microsoft and offers a very mature and convenient way to construct hooks by handling the installation and overall management of the hooks. However, malware is potentially able to detect the presence of Detours and alter its functionality. Also, Detours is not able to detect code propagations conducted by malware residing in kernel mode.

We have created two hooking libraries for 32-bit and 64-bit, which hook each function that shall be monitored. When a hook is called, an initial log line is created containing the parameters used to call the function. Additionally, the original function is called to preserve the functionality of the program. The return value and the out parameters are used to generate a second log line, and the return value is returned by the hook. This method ensures that every interaction with the monitored function is recorded, while also preserving the functionality of the original. Our hooking libraries are a modified version of traceapi [29]. The logs are generated with a slightly modified version of syelog [28], so that it is capable of writing logs to files.

When deciding on which functions to hook, we have focused on the high-level functions and their often undocumented low-level equivalents. Technically, it would be sufficient to target only the low-level equivalents, but we wanted to gain additional knowledge about the utilization of low-level functions. We have chosen the API functions based on the research of Plohmann et al. [33] and Mitre Att&ck [2]. The full list of hooked functions can be found in the Appendix.

The monitoring libraries are loaded into every process that loads the user32.dll library by using the AppInitDll function found in the registry. Therefore, the hooks are installed before the main image (i.e., the malware) is run.

The logs generated by the hooks are analyzed asynchronously by a Python script, which implements the analysis described above.

5 Code Propagation Study

To gain insight into how code propagations are used in malware, we conducted a longitudinal study.

5.1 Data Set

To conduct this study, we first acquired a data set with a high diversity of families and actors [34].

We have chosen Malpedia[1] [32] as the basis for our data set. It is a strictly manually curated malware corpus, with 7161 samples across 2686 families, and strives for representative coverage of as many malware families as possible by providing a sample for every version of every malware family. For almost every sample, the data set provides an unpacked and dumped version, as well as meta-information, and YARA rules [1]. Since this data set strives to include every version of every family [32], there is no bias towards any particular version of a family. Adding more samples to our data set would not provide any further benefit, as it would only create redundancy and introduce a bias towards a specific version of a family, unless we provide an additional sample for every version of every family. Therefore, we focused our attention on evaluating a single data set.

Since we focus our study on Windows malware, we have filtered the 7161 samples in the data set by excluding every file that is not a PE file or the main representative of the family version. Therefore, we always chose the representative that has been found in the wild. 5758 samples remain after filtering and are therefore our data set.

5.2 Study Setup

We analyzed the Malpedia samples on three Oracle VirtualBox VMs [31], each with 4 GB RAM and 2 CPUs running at 3.20 GHz. To maximize the number of compatible malware with virtual machines, we have chosen Windows 7 as the operating system. The installation includes every service pack and the MS Visual C runtime (MSVCRT) library released by Microsoft for Windows 7. The hardening of the virtual machine was performed as described in [9].

The malware samples are automatically loaded into the VM and launched with a simulated double-click. The double-click ensures that the malware is launched in the most organic way possible, since its parent process is the Explorer. DLL-based malware was run using rundll.exe by calling the DLLmain. The hooks were already installed and the VM was re-booted. The analysis of a sample timed out after 120 s. As shown in [15], most malware finished unpacking after 25 s, and Küchler et al. [22] argue that the majority of samples run for less than two minutes or more than ten minutes. Since we are only interested in the

[1] Git commit d366eb0 - Jan 6, 2023.

code propagation of malware, which typically occurs at the very beginning of the execution, we argue that 120 s is sufficient. After the timeout, the logs were collected and sent back to the host for further analysis.

5.3 Results: Topologies

In this section, we discuss the results of our study by analyzing the resulting topologies. We use the graph definition from Sect. 3.2 to describe the code propagations in this section.

Of the entire set of samples, 199 samples (3.5%) failed to create the initial graph and 735 samples (13%) failed the interpretation step. In other words, the malware was never run or the analysis was broken to the point that no start of the interpretation could be found. However, it should be noted that a significant portion of the PE files in Malpedia are not intended to be executed by the user, such as modules. Therefore, it was expected that a significant portion of the samples would fail the interpretation step. These samples were omitted from the statistics. In total, 4853 samples from 1747 families remain, which were successfully analyzed.

There are 1446 DLLs and 3407 executables in the data set. 1351 samples used code propagations. 52 of these samples are DLLs; therefore, 3.6% of the analyzed DLLs use code propagations. 1299 executables (38.13%) have used code propagations. From these numbers, we can see that code propagations play a much larger role for executables than for DLLs. This may be due to the fact that DLLs have a different use case, because DLLs are often loaded by an initial dropper file, so hiding in the system may not be important for them, as this could already have been achieved by the dropper.

Apart from 45 samples (3.33%) that produced circular propagation graphs in our experiments, 1306 (96.67%) samples produced tree structures.

Table 1. 68.24% of the trees have a maximum degree of one. We coin these trees as paths. So, 31.76% of the samples spread their code to two or more processes.

Max Degree	Paths	Star St.	Bran. St.	C. Graphs	$\sum$
1	922	0	0	3	925
2	0	212	36	11	259
3	0	36	6	1	43
4	0	24	5	0	29
5	0	8	5	1	14
6	0	5	2	3	10
7	0	2	1	2	5
8	0	7	1	5	13
9	0	6	2	9	17
>= 10	0	24	2	10	36

Table 2. The longest path in the trees grouped by their topology types. The most common lengths are two and three. The most common length of the longest path for the branching structures is three.

Longest Path	Paths	Star St.	Bran. St.	$\sum$
2	713	203	0	916
3	136	75	36	247
4	44	30	7	81
5	24	9	10	43
6	1	1	2	4
7	1	1	0	2
8	0	1	3	4
9	0	1	0	1
>= 10	3	3	2	8

To further classify the tree structures, we first looked at their maximum degree. We found that 922 (68.25%) of these samples had a maximum degree of one, i.e., they can be categorized as paths (subsequently referred to as path samples), as shown in Table 1. We also found that 713 (77.33%) of all path samples carried out exactly one code propagation, as depicted in Table 2.

To classify the remaining trees, we looked at how many nodes had a degree greater than one. As shown in Fig. 2, 324 (23.98%) trees meet this condition. Due to their similarity to starlike trees [24], we named this class star structures[2].

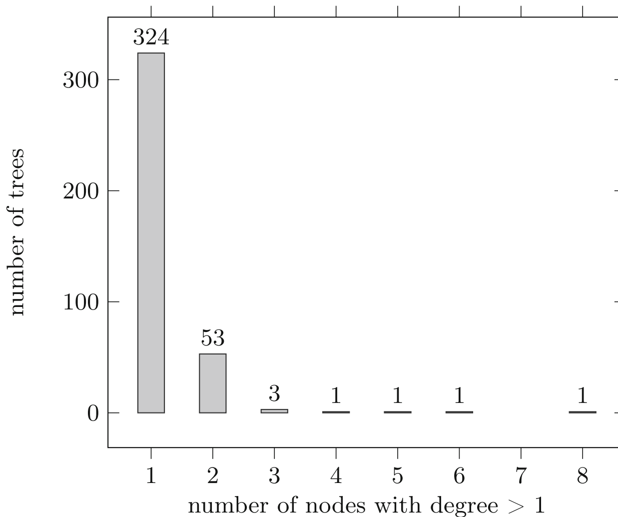

Fig. 2. This graph illustrates the number of trees with the number of nodes with a degree greater than one. There are 324 trees with exactly one node with a degree greater than one.

[2] Our definition differs from the definition for starlike trees in that the vertex with degree greater than 1 does not need to be the root.

The highest degree of a tree in Table 1 provides insight into how widely malware of this class spreads in the system. 212 (65.43%) of the samples in this group distribute their code to two processes, and the remaining 112 (34.58%) spread it to more than two processes.

In Fig. 2, the third largest group is 60 (4.44%) trees with two or more nodes with a degree greater than one. Curiously, we found that trees with exactly two nodes with degrees greater than one are caterpillar trees, i.e., trees in which removing all pendant nodes results in a cordless path. The remaining seven trees are more complexly branched than the caterpillar trees. Therefore, we refer to this group as branching structures, as samples of this type have the most intricate structure of all trees. No tree with a branching structure has a shorter longest path than three, as can be seen in Table 2.

In total, we have identified four different topological classes: **Circular Graphs**, **Paths** with a degree of one, **Star Structures** that have only one node with a degree greater than one, **Branching Structures** that have more than one node with degree greater than one.

The vast majority of 922 samples belong to the path topology. The second largest group, with 324 samples, belongs to the star structures. The branching structures and the circular graphs are the least important topology classes with 60 and 45 samples, respectively.

5.4 Results: Propagation Techniques

In this section, we will discuss the code propagation methods by first analyzing the number of propagations and their actual implementation.

Table 3. The number of code propagations per sample. The branching structure and graphs perform the most code propagations, and paths perform the fewest code propagations.

	Paths	Star St.	Branching St.	C. Graphs
min	1	2	4	2
25%	1.0	2.0	4.0	5.0
median	1.0	3.0	6.0	11.0
75%	1.0	5.0	9.0	12.0
average	1.50	8.55	10.30	28.73
maximum	131	357	161	892

First, we provide an overview of the number of code propagations used by the samples in their respective topology types. We hypothesize that path samples use the least number of propagations since they are of the simplest type.

The results are displayed in Table 3. It can be seen that path samples mostly use only one code propagation. However, as indicated by the discrepancy between

Table 4. Occurrence of code propagation techniques in topology types. Each propagation is counted once per sample. The most important code propagation techniques are the spawn of new processes, technique 217 (injection using a newly created process into new memory allocation and executed by a reused thread), and technique 129 (launch of a new process and thread resume). The explanation for the binary codes can be found in Sect. 3.3.

Code Propagation		Paths	Star Str.	Branching Str.	C. Graphs	$\sum$
Propagation	**Binary Code**	–	–	–	–	–
1	00000001	613	270	57	40	980
217	11011001	147	59	17	9	232
129	10000001	106	57	10	4	177
225	11100001	52	13	11	3	79
86	01010110	14	24	8	31	77
209	11010001	50	17	1	4	72
85	01010101	19	10	16	6	51
249	11111001	13	7	1	0	21
241	11110001	9	3	0	1	13
81	01010001	5	3	2	0	10
233	11101001	8	1	0	0	9
137	10001001	5	2	1	0	8
117	01110101	2	2	3	1	8
97	01100001	6	0	0	0	6
118	01110110	2	1	0	3	6
113	01110001	4	0	0	1	5
214	11010110	0	0	0	4	4
221	11011101	0	2	0	1	3
229	11100101	3	0	0	0	3
145	10010001	1	0	0	0	1
161	10100001	1	0	0	0	1
237	11101101	0	1	0	0	1
245	11110101	0	0	1	0	1
253	11111101	1	0	0	0	1

the third quartile and the average, there are a small number of samples that perform significantly more propagations. Star structures use many more code propagations, about nine on average. The branching structures use about 8 and the circular graphs even use almost 29 code propagations on average. By comparing the data, it is evident that the number of code propagations for all types except paths increases greatly.

Subsequently, we focus on the different code propagations as they are used by the samples in their respective types. To represent the code propagation

Table 5. Functions, their number of usages, and the number of samples using the respective functions.

Function	Total Usage	#Samples
WriteProcessMemory	20449	461
VirtualAllocEx	9007	445
CreateProcessW	3309	808
ResumeThread	1838	511
CreateProcessA	1668	423
VirtualProtectEx	959	93
NtWriteVirtualMemory	763	140
SetThreadContext	600	260
CreateRemoteThread	556	139
OpenProcess	446	83
ShellExecuteA	168	102
ShellExecuteExW	145	93
ShellExecuteW	124	77
NtResumeThread	103	67
CreateProcessInternalW	86	73
WinExec	85	65
VirtualQueryEx	65	5
CreateProcessInternalA	31	15
QueueUserAPC	6	6
ShellExecuteExA	3	3

techniques, we interpret the eight bits as described in Subsect. 3.3 as the value of an unsigned integer and use the resulting value for easier reference. The bit representation of the code propagation technique is shown in parentheses next to the unsigned integer. Table 4 shows the total number of code propagations used in the data set. We have chosen to narrow down on fully realized code propagations and, therefore, omit failed code propagations.

In our data set, 24 code propagation techniques are used. The vast majority of samples use the technique of simply spawning a malicious process 1 (00000001). This technique is used in 980 samples, with the majority being in path and star structures at 631 and 270 respectively. The second most widely used propagation technique, 217 (11011001), is an injection into a newly created process with new memory allocation and a thread reuse. Technique 129 (10000001) launches a new process and resumes the main thread. The fourth most common technique is 225 (11100001), which is the spawn of a new process and the reuse of threads and memory.

These findings emphasize the notion that malware uses many ways to distribute its code throughout the system. Consequently, a dynamic analysis

machine must be able to keep track of a variety of different code propagation techniques. Failure to do so will result in the inability to properly detect all code propagations.

The next segment focuses on the actual implementation of the code propagations. For this purpose, we look at the usage of functions. The numbers in Table 5 represent the total usage of functions in code propagations in the data set and the number of samples that use the respective functions. We only count the number of calls made directly by the malware. For example, a call to CreateProcessInternalA made by CreateProcessA is not counted. However, a call to CreateProcessInternalA made directly by the malware is counted.

The four most used functions are WriteProcessMemory, VirtualAllocEx, CreateProcessW, and ResumeThread. The function used by most individual samples is CreateProcessW. Taking into account Fig. 1, there are two bottlenecks in the graph. There is either the path of starting a new process or the path of injections. In the injection route, the step of writing data to memory (WriteProcessMemory) is the bottleneck, while spawning a process (CreateProcess-Family) is the bottleneck on the other side. Therefore, these functions are essential for code propagations. Thus, they are the topmost used functions in Table 5.

6 Recommended Samples

As a result of our experiments, we want to provide a list of recommended samples that researchers can use to evaluate their countermeasures against code propagations. In this way, researchers do not have to rely solely on theoretical analysis to assess their approaches. First, we provide samples of malware families that belong to the **Path Topology**:

- **Laziok**[3] performs code propagation techniques **85** and **1**.
- **Bfbot**[4] uses code propagation techniques **217** and **86**.
- **Emotet**[5] performs the code propagation technique **225**.
- **Zloader**[6] uses the code propagation technique **249**.
- **Shylock**[7] covers the code propagation technique **209**.
- **Kins**[8] uses the code propagation technique **229**.
- **Ramnit**[9] uses the code propagation technique **253**.
- **Dridex**[10] uses the code propagation technique **97**.
- **Solarbot**[11] uses the code propagation technique **113**.

---

[3] 6adecfaec434b41ecce9911f00b48e4e8ae6e3e8b9081d59e1b46480e9f7dbfc.
[4] f19ce795b4b2421a82ff71a3f3a271032578c80cadd0cc44b1714848b5bb81c0.
[5] f9ef36da6a3786dd672e049aa4028d12d0cd33a4f4771ec70309c89f8f482930.
[6] bd882e2eefd0145ff169d868c1815df272f84a5ad1e501cfa5c3336839774171.
[7] a7a29da4c53d424e1997ff8f2702aea6b76e9f5b60d704f306c353e01cea4d76.
[8] 520ae48364d7e5fe6bdb0a59c9cd1370dee5b26e648677fa84f1f601f727d280.
[9] 89b138eaaade5a1ec36e2d1422ae38059f138e81b722301e713b65a74de521c7.
[10] f24354e54e4b59f6c327b1f7e144092647e726505acde5595a8386e7c2c6fa8a.
[11] 40fa0ae6c2f73af93c304b3e12d22ee38100ac0e18798f2e96b1db37abbca8e8.

- **gameover_dga**[12] uses the code propagation techniques **214** and **86**.
- **Lokipws**[13] uses the code propagation technique **145**.
- **ngioweb**[14] uses the code propagation technique **161**.

For the **Star Structure Topology**, we recommend the following samples:

- **Gozi**[15] uses the code propagation technique **118**.
- **Gootkit**[16] covers the code propagation technique **241**.
- **Cutwail**[17] uses the code propagation technique **221**.
- **Lethic**[18] uses the code propagation technique **233**.
- **Xsplus**[19] uses the code propagation technique **137**.
- **Shifu**[20] uses the code propagation technique **237**.

For the **Branching Structure Topology**, we suggest the following samples:

- **Qakbot**[21] uses the code propagation techniques **225**, **86**, and **1**.
- **Ramnit**[22] covers the code propagation technique **117**.
- **Opghoul**[23] covers code propagation techniques **129** and **81**.

At last, for the **Circular Graph Structure Topology**, we recommend:

- **Gameover DGA**[24] uses code propagation techniques **86** and **217**.
- **Snifula**[25] uses the code propagation technique **245**.

7 Discussion and Conclusion

In this paper, we set out to study the movement of malicious code through the system. We presented a definition for code propagations and a systematized representation of code propagation topologies and techniques. Also, we proposed a methodology for the detection and measurement of code propagations of malicious software.

Consequently, we have used our proof-of-concept implementation to conduct a study on a diverse real-world data set called Malpedia [32]. Our study shows that code propagations are widely used in malware. More than a third of the

[12] 072cdcf66b81772724648da4c0ca2429a39504599e07ccfca2ba8af73ec24adc.
[13] 97a614c078ca4302c31a8af24cf19317d76507c5fee17b4df10149157127b19b.
[14] df70581c5a712e2eda579221145347041 66f93dc2158c302c58d61a487330546.
[15] be65dc1c2d2cb1ddbb7b08780e608eb0d9cabc706491f5bd7657326018c0c518.
[16] e7fa2707166283e1f0e7422546ee387aae01b5ee5c255a62909da0a3b6cb19c0.
[17] 92c0cc5879215255478b3325bee34353090e08337aa61a92506f0498f7907500.
[18] 92bb2efeea875eb5e8779f13cc50d1a831b3c538eb73e15384f8748266be8ff1.
[19] bff06d770eec594c363a217effbe2ea4e8a618b7ef95da1100e5aef9c847403f.
[20] b2c6c7e9d8bb6f75865324788cf311a5a951e2d4e69137937ecfb0879ebae1ce.
[21] d7489e3f876cb41d61b08bb1f91ed9a9f862761416954649c4ee2c26b5c3c199.
[22] 80823b2e354ed28badde4e8a7525113be5fc61b4a48f64a5f33da9491d2d2aa9.
[23] d22f9035ac8c69bb391bd478b01305c00bef0cb7b1b0b2ea716ad31a3fcc07cb.
[24] 3ff49706e78067613aa1dcf0174968963b17f15e9a6bc54396a9f233d382d0e6.
[25] 104428ccf005b36edfb62d110203a43bdbb417052b31eb4646395309645c9944.

executable samples in our data set use code propagations. We derived typical malware code propagation behavior from a representative data set by investigating the movement of code through the system as well as determining the type of propagation technique used. We have identified four types of code propagation topologies and a set of 24 code propagation techniques. Our study focuses on Windows malware. An analogous study focusing on Android malware may be beneficial due to the increasing popularity of mobile malware.

Lastly, we have provided a list of samples for each code propagation technique and topology. In subsequent work, this list can be used to generate mitigation strategies addressing the various code propagation topologies and techniques.

8 Appendix

8.1 Hooked Functions

The functions that focus on process(-handle) creation on Microsoft Windows are:

- OpenProcess & NtOpenProcess & CreateProcessW & CreateProcessA & WinExec
- CreateProcessInternalA & CreateProcessInternalW & NtCreateUserProcess
- ShellExecuteExW & ShellExecuteExA & ShellExecuteA & ShellExecuteW

The hooked functions that are used to create threads and thread handles on Microsoft Windows are:

- OpenThread & SetThreadContext & NtSetContextThread & NtOpenThread
- ResumeThread & NtResumeThread & CreateRemoteThread & QueueUserAPC

The functions that allocate new memory regions on Microsoft Windows are:

- VirtualAllocEx & VirtualAllocExNuma
- WriteProcessMemory & NtWriteVirtualMemory & VirtualQueryEx

However, we decided not to hook NtAllocateVirtualMemory because a hook on this function makes the virtual machine unresponsive.

8.2 Example for Log Analysis

```
479024d: 2072: +CreateProcessW(<NULL>,C:\Windows\SysWOW64\upnpcont.exe,0,0,0,0,0,<NULL>,18ff04,18ff50)
47b824f: 2072: -CreateProcessW(,,,,,,,,,) -> 1 (proc:2168/f0, thrd:2172/ec
47b828e: 2072: +OpenProcess(1f0fff,0,pid=2168)
47b8339: 2072: -OpenProcess(,,) -> f8
47b838c: 2072: +VirtualAllocEx(f8,0,9a4a,1000,40)
47b83c7: 2072: -VirtualAllocEx(,,,,) -> 100000
47b841a: 2072: +WriteProcessMemory(f8,@100000..109a49,27bea0,0)
47b8515: 2072: -WriteProcessMemory(,,,,) -> 1
47b8573: 2072: +CreateRemoteThread(f8,0,0,100000,0,0,18ff6c)
47b87d5: 2072: -CreateRemoteThread(,,,,,,) -> f4 (2176
488e0fa: 2072: +ExitProcess(0)
```

Fig. 3. This log file snippet is taken from the original malware process, launched by the Explorer. The sample belongs to the Laziok family. It is heavily truncated for readability and shows an injection into a new process.

Figure 3 shows a snippet of a log file for a sample of the Laziok family[26]. In the first two lines, the malware launches a process from the upnpcont.exe binary with CreateProcessW. Afterwards, the malware opens the process with OpenProcess to gain a handle, even though CreateProcessW also returns a handle to the new process. The handle f8 is used to allocate new memory in the new process to which the code is written. This memory is executed by a new thread that is spawned at the address where the data were previously written. The process then terminates. At the end of the sandboxing, this new process is still running.

References

1. Alvarez, V.M.: YARA: the pattern matching swiss knife for malware researchers (and everyone else). http://virustotal.github.io/yara/. Accessed 16 Aug 2023
2. ATT&CK, M.: Mitre att&ck (2021). https://attack.mitre.org
3. AVTest: security report 2019/2020. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2019-2020.pdf. Accessed 16 Aug 2023
4. Bacs, A., Vermeulen, R., Slowinska, A., Bos, H.: System-level support for intrusion recovery. In: Flegel, U., Markatos, E., Robertson, W. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. Lecture Notes in Computer Science, vol. 7591, pp. 144–163. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-37300-8_9
5. Barabosch, T., Bergmann, N., Dombeck, A., Padilla, E.: Quincy: Detecting host-based code injection attacks in memory dumps. In: Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Bonn, Germany (2017)
6. Barabosch, T., Eschweiler, S., Gerhards-Padilla, E.: Bee master: detecting host-based code injection attacks. In: Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), London, UK (2014)

[26] 6adecfaec434b41ecce9911f00b48e4e8ae6e3e8b9081d59e1b46480e9f7dbfc.

7. Barabosch, T., Gerhards-Padilla, E.: Host-based code injection attacks: a popular technique used by malware. In: 2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE), pp. 8–17. IEEE (2014)
8. Bohne, L., Holz, T.: Pandora's Bochs: automated malware unpacking. Master's thesis, RWTH Aachen University (2008)
9. ByteAtlas: Knowledge fragment: Hardening win7 x64 on virtualbox for malware analysis. http://byte-atlas.blogspot.com/2017/02/hardening-vbox-win7x64.html. Accessed 16 Aug 2023
10. D'Elia, D.C., Nicchi, S., Mariani, M., Marini, M., Palmaro, F.: Designing robust API monitoring solutions. IEEE Trans. Dependable Secure Comput. **01**, 1–1 (2021)
11. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference On Computer and Communications Security, pp. 51–62. ACM (2008)
12. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with panda. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, pp. 1–11 (2015)
13. Isawa, R., Morii, M., Inoue, D.: Comparing malware samples for unpacking: a feasibility study. In: 2016 11th Asia Joint Conference on Information Security (AsiaJCIS), pp. 155–160. IEEE (2016)
14. Ispoglou, K.K., Payer, M.: malWASH: washing malware to evade dynamic analysis. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (2016). https://www.usenix.org/conference/woot16/workshop-program/presentation/ispoglou
15. Jenke, T., Plohmann, D., Padilla, E.: RoAMer: the robust automated malware unpacker. In: 14th International Conference on Malicious and Unwanted Software (MALWARE), Nantucket, MA, USA, 2019, pp. 67–74 (2019)
16. Jeong, G., Choo, E., Lee, J., Bat-Erdene, M., Lee, H.: Generic unpacking using entropy analysis. In: 2010 5th International Conference on Malicious and Unwanted Software, pp. 98–105. IEEE (2010)
17. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode, pp. 46–53. ACM (2007)
18. Kawakoya, Y., Shioji, E., Iwamura, M., Miyoshi, J.: API chaser: taint-assisted sandbox for evasive malware analysis. J. Inf. Proc. **27**, 297–314 (2019)
19. Korczynski, D.: RePEconstruct: reconstructing binaries with self-modifying code and import address table destruction. In: 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), pp. 1–8. IEEE (2016)
20. Korczynski, D.: Precise system-wide concatic malware unpacking. arXiv preprint: arXiv:1908.09204 (2019)
21. Korczynski, D., Yin, H.: Capturing malware propagations with code injections and code-reuse attacks. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1691–1708 (2017)
22. Küchler, A., Mantovani, A., Han, Y., Bilge, L., Balzarotti, D.: Does every second count? time-based evolution of malware behavior in sandboxes. In: Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society (2021)
23. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 386–395 (2014)

24. Lepović, M., Gutman, I.: No starlike trees are cospectral. Discret. Math. **242**(1–3), 291–295 (2002)
25. Magazine, S.: Ransomware attacks nearly doubled in 2021 (2022)
26. Martignoni, L., Christodorescu, M., Jha, S.: OmniUnpack: fast, generic, and safe unpacking of malware. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, pp. 431–441. IEEE (2007)
27. Microsoft: Microsoft detours. https://github.com/microsoft/Detours. Accessed 16 Aug 2023
28. Microsoft: Samples: Syelog. https://documentation.help/Detours/Sam_Syelog.htm. Accessed 16 Aug 2023
29. Microsoft: Samples: Traceapi. https://documentation.help/Detours/Sam_Traceapi.htm. Accessed 16 Aug 2023
30. Mohammad, A.H.: Ransomware evolution, growth and recommendation for detection. Mod. Appl. Sci. **14**(3), 68 (2020)
31. Oracle: Oracle virtualbox. https://www.virtualbox.org/. Accessed 16 Aug 2023
32. Plohmann, D., Clauss, M., Enders, S., Padilla, E.: Malpedia: a collaborative effort to inventorize the malware landscape. In: Proceedings of the Botconf (2017)
33. Plohmann, D., Enders, S., Padilla, E.: ApiScout: robust windows API usage recovery for malware characterization and similarity analysis. J Cybercrime Digit. Invest. **4**, 1–6 (2018)
34. Rossow, C., et al.: Prudent practices for designing malware experiments: status quo and outlook. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), San Francisco, CA (2012)
35. Royal, P., Halpin, M., Dagon, D.: PolyUnpack: automating the hidden-code extraction of unpack-executing malware. In: ACSAC, pp 289–300 (2006)
36. Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., Lee, W.: Eureka: a framework for enabling static malware analysis. In: Jajodia, S., Lopez, J. (eds.) Computer Security - ESORICS 2008. Lecture Notes in Computer Science, vol. 5283, pp. 481–500. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-88313-5_31
37. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: RAMBO: run-time packer analysis with multiple branch observation. In: Caballero, J., Zurutuza, U., Rodriguez, R. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. Lecture Notes in Computer Science(), vol. 9721, pp. 186–206. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_10
38. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 116–127 (2007)