Understanding and Measuring Inter-Process Code Injection in Windows Malware

Jerre Starink[1], Marieke Huisman[1], Andreas Peter[1,2], Andrea Continella[1]

[1] University of Twente, The Netherlands
{j.a.l.starink,m.huisman,a.continella}@utwente.nl,
[2] University of Oldenburg, Germany
andreas.peter@uol.de

Abstract. Malware aims to stay undetected for as long as possible. One common method for avoiding or delaying detection is the use of code injection, by which a malicious process injects code into another running application. Despite code injection being known as one of the main features of today's malware, it is often overlooked and no prior research performed a comprehensive study to fundamentally understand and measure code injection in Windows malware. In this paper, we conduct a systematic study of code injection techniques and propose the first taxonomy to group these methods into classes based on common traits. Then, we leverage our taxonomy to implement models of the studied techniques and collect empirical evidence for the prevalence of each specific technique in the malware scene. Finally, we perform a large-scale, longitudinal measurement of the adoption of code injection, highlighting that at least 9.1% of Windows malware between 2017 and 2021 performs code injection. Our systematization and results show that Process Hollowing is the most commonly used technique across different malware families, but, more importantly, this trend is shifting towards other, less traditional methods. We conclude with takeaways that impact how future malware research should be conducted. Without comprehensively accounting for code injection and modeling emerging techniques, future studies based on dynamic analysis are bound to limited observations.

Keywords: Malware · Code Injection · Malicious Behaviors

1 Introduction

Despite a significant effort put into research and development of defense mechanisms [6,29], new malware is continuously developed at a rapid pace, making it still one of the major threats on the Internet [8]. For malware to be successful, it is in the author's best interest to make sure that their samples stay undetected for as long as possible [50]. One of the techniques used for this purpose is *code injection*. Code injection is defined as the process in which an application copies pieces of its code into another running program. This running program is then tricked into executing the injected code, making it perform something it was not originally intended to [7,11,46]. By extension, if a malicious program copies its malicious code into a legitimate application, it is not the malware itself that exhibits the malicious behavior, but rather the application that was previously considered benign. As a consequence, scanning an executable file for suspicious code might not be sufficient, making the task of automating threat detection significantly more involved.

Code injection has been often overlooked and, currently, its characterization generally relies on heuristics. This includes testing for calls to well-known Windows APIs such as `VirtualAllocEx` and `WriteProcessMemory` [7,10], finding common byte-sequences in the sample [20,25,56], or looking for artifacts such as suspicious memory pages that were injected by other processes [3,9,21,26,59]. However, there are various ways to perform code injection, including methods that do not introduce these artifacts [19,22,41,39,40]. This renders many of the heuristics insufficient. Recent research attempted to formalize code injection [11,30], laying the foundation for a more generic understanding. However, despite the positive contributions, there is still a significant number of injection techniques that fall outside the current formalization—what we define *passive* techniques in Section 3. Thus, there is a need for a better, more fundamental understanding of what code injection entails, as well as a more systematic study of injection techniques, to comprehensively classify this type of behavior.

In this research, we conduct the first systematic study of code injection techniques. First, we collect 17 injection techniques used by malware and discussed in the literature. We implement and test them to verify that they still work on modern software and hardware. Then, we compare every technique to each other and identify reoccurring features. From this, we derive a more fundamental understanding of code injection and propose a categorization of the techniques based on their common characteristics. Note that, while we mostly focus on studying code injection in malware, injection techniques are also adopted by legitimate software for benign purposes, as we show in our empirical evaluation (Section 6). Our study provides a systematization and measurement of code injection but we do *not* aim at distinguishing malicious versus benign uses.

Using our taxonomy, we perform a large-scale measurement of the adoption of code injection by analyzing 47,128 malware samples observed between 2017 and 2021. To this end, we develop a framework to determine whether a sample adopts any of the studied injection techniques and detail *which* techniques are adopted, allowing for investigations of their usage and evolution. For such classification, we leverage and re-implement well-known behavior graphs [16,35], which express malware behavior—in our case, injection techniques—in terms of API/system calls and their interdependence.

Our empirical evaluation reveals important insights. While the number of samples adopting code injection does not significantly change over time, *new emerging techniques—often overlooked—are gaining more popularity* compared to the traditional methods that rely on API calls such as `WriteProcessMemory`. This has a direct effect on the way future research on malware analysis should move forward (Section 7). Without proper consideration of how malware might propagate via code injection, studies based on dynamic analysis may miss significant portions of malicious behaviors.

In short, we make the following contributions:

- **Taxonomy of Code Injection:** We survey 17 different code injection techniques and propose the first taxonomy that classifies injection techniques based on a set of identified common traits. On the base of our taxonomy, we also release a labeled dataset of code injection samples.
- **Measurement of the Adoption of Code Injection:** We examine a set of 47,128 malware samples and depict a comprehensive picture of the current prevalence and distribution of code injection techniques in the malware scene between 2017 and 2021.
- **Takeaways for Future Research:** We provide insights and takeaways for future research, highlighting the impact of our results, and laying out lessons to take into account when designing new malware studies.

We release our implementation, as well as our labeled dataset of code injection techniques at https://github.com/utwente-scs/code-injection-malware.

2 Background

Code injection can be defined as the act of copying and executing code in the context of another process. Injections start by selecting a *victim* process and finding existing executable memory pages within this process or allocating new ones. Then, the code—referred to as the *payload*—is copied into this memory, and the victim process is tricked into executing it. The goal of code injection is to alter the behavior of the victim process, making it do something unintended.

Injecting code into another process is an effective way to hide the true (malicious) intentions of a program. Detection mechanisms that solely focus on analyzing the sample itself might not pick up on the behavior offloaded to the victim process. Especially victim processes from a known vendor are an attractive option for an *injector* process, as these programs are often blindly trusted by anti-malware [11,46]. For these reasons, several variants of code injection have been adopted by modern malware families [23,49,60].

Unfortunately, fully abolishing code injection is not practical as several types of legitimate software use code injection in benign contexts. For example, *debuggers* rely on injecting small chunks of code into the target process to stop its execution and read its internal state [27]. Also, many operating systems feature *shim infrastructures* to make up for incompatible version updates. These are implemented by hooking into an API function and redirecting it to *shim code*, simulating the original behavior of the API before a breaking change[36]. Prohibiting code injection would mean giving up on these applications and frameworks.

Many operating systems feature mitigations that prevent illegitimate code injections from happening. For example, Windows implements the User Account Control (UAC) [5], where running processes are given an access token that can be used to perform administrative tasks. Effectively, this means that a non-privileged process cannot directly interact with (and thus cannot inject into) a privileged process easily. However, the majority of processes on Windows do not require special privileges to execute. Furthermore, malware may be running under special privileges (e.g., when it is packaged inside of an installer). Thus, regardless of these mitigations, the number of processes to inject into is still high, leaving code injection as a viable option for evading detection.

3 Code Injection Systematization

To obtain a more fundamental understanding of code injection, we survey 17 different code injection techniques commonly used in the malware scene. Then, we identify similarities and differences, and we extract common traits to group them into classes. These classes represent, to the best of our knowledge, the first taxonomy of code injection.

3.1 Technique Selection

To obtain a representative set of code injection techniques, we queried various sources that are well-known in the security community. These include the MITRE framework, as well as technical malware briefings provided by six well-known security companies, including The Infosec Institute, Elastic Security, MalwareBytes, F-Secure, Symantec and Kaspersky. We also included various blog posts of individual security researchers with example implementations and variations of the techniques. Since malware authors typically aim to maximize their attack surface, we select only the techniques that work on Windows 10 (as it is the most market-dominant OS at the time of conducting this research [55]), and do not have a dependency on extra (third-party) software that needs to be installed separately. With this process, we selected the following 17 techniques:

Shellcode Injection. This technique is the most fundamental form of code injection and serves as a base for many other techniques. First, memory is allocated in the victim process using a function such as `NtAllocateVirtualMemory` with the `PAGE_EXECUTE_READWRITE` protection bit set. Then, shellcode is transmitted into this allocated memory (e.g., using `NtWriteVirtualMemory`). Finally, a thread with the address of the injected shellcode as its entry point is created within the victim process (e.g., using `NtCreateThreadEx`) [21].

PE Injection. PE injection extends Shellcode Injection by including additional logic to support injecting entire Portable Executable (PE) files. This allows for easier development of larger, more complex payloads written in higher-level languages as opposed to small (handcrafted) assembly code [54].

Classic DLL Injection. Classic DLL Injection avoids `PAGE_EXECUTE_READWRITE` allocations by storing the payload in a Dynamic-Link Library (DLL) file on the disk. The injector writes the file path into some non-executable memory (i.e., `PAGE_READWRITE`), and then creates a new remote thread starting at a function such as `LoadLibrary`. This way, the victim process loads the payload DLL as if it was a normal dependency, triggering its execution [18,42].

Reflective DLL Injection and Memory Module Injection. These two techniques are variations of Classic DLL Injection that reimplement the functionality of `LoadLibrary`. This way, they avoid the call to the original function and can also keep the payload DLL in memory. A Reflective DLL Injector executes this implementation via traditional Shellcode Injection, while a Memory Module Injector performs most of the mapping on the injector's side instead. The latter also ensures that the mapped sections have the appropriate protection bits set (as opposed to only `PAGE_EXECUTE_READWRITE`), which contributes to the stealthiness of the technique [21].

APC Shell and DLL Injection. These two techniques are variations of Shellcode and Classic DLL Injection respectively that avoid the creation of new threads by abusing the Asynchronous Procedure Call (APC) queue of an existing thread. APCs are function calls that are scheduled to be invoked by a thread when the thread is e.g., waiting for an event or user input. By queuing shellcode or a `LoadLibrary` call as an APC, Windows automatically loads and triggers the execution of the payload whenever the thread is in such a state [38].

Process Hollowing and Thread Hijacking. These are one of the most commonly used methods for performing code injection and are sometimes also referred to as RunPE. The injector either creates a new suspended process or suspends an existing one respectively, and unmaps (hollows out) all its sections from memory. Then, a new PE image is manually mapped into the victim process and the main thread is redirected to the new entry point (e.g., using `NtSetContextThread`). Finally, the process is resumed afterward [37,44].

IAT Hooking. During the loading procedure of a PE file, Windows resolves the addresses of all functions that the PE depends on and puts them in its Import Address Table (IAT). The IAT Hooking technique replaces one of these addresses with one that points to the injected shellcode. This way, when the victim process calls the original function using its IAT, the payload will be triggered instead, without using thread creation or redirection APIs [28].

CTray VTable Hooking. This technique is similar to IAT Hooking but specifically targets `explorer.exe`, the default file browser on Windows. Internally, the browser defines a class `CTray` which implements the taskbar's notification tray. By replacing the address of its `WndProc` function, which is responsible for processing every message that the tray receives (e.g., paint events), the technique activates injected shellcode the moment the tray processes such a message [43].

Shim Injection. Shim infrastructures are small programs attached to legacy software, that attempt to simulate the original behavior of an API after a breaking change was introduced by a Windows update. By extension, an injector can register itself as a shim infrastructure to load and run arbitrary code within the context of software that requires these legacy features [22].

Image File Execution Options (IFEO). Image File Execution Options are settings stored in the Windows Registry that dictate how a specific application identified by its name should be started by Windows. One of the parameters it defines is the path to a debugger program that the application's memory should be replaced with when it is being loaded. IFEO Injection registers an executable file as such a debugger [48].

AppInit_Dlls and AppCertDlls Injection. Similar to IFEO, AppInit_Dlls and AppCertDlls are two Registry keys that store the paths to extra DLL files that should be loaded whenever an application starts. The key difference is that these DLLs are loaded by *any* process that is started after the Registry change was made, as opposed to specific processes [39,40].

COM Hijacking. The Common Object Model (COM) is a Windows framework that allows for software components to be used across multiple programming languages. Components are stored in the Registry as file paths to the DLLs that implement them and are loaded and instantiated on-demand. COM Hijacking replaces one of these DLL file paths with a path of its own, tricking the victim process into loading the payload DLL instead of the original component [41].

Windows Hook Injection. The Windows API exposes functions to subscribe to various global system events such as mouse clicks and key presses. More specifically, a thread can be instructed to invoke a callback defined in a specific DLL when such an event occurs. Typically, threads are chosen from the current process. However, the API allows for selecting *any* thread running on the system. Windows Hook Injection abuses this by registering a callback for one of the victim process threads, letting it load and execute a payload DLL [19].

3.2 Common Traits

From the studied techniques, we extract common traits that help us characterize the techniques more precisely, which we will introduce below.

Moment of Execution. This trait describes the moment in which the code can be injected and executed in the victim process. Some techniques can inject payloads at any time while the process is running, whereas in others it is only possible upon startup of the victim process or operating system.

Transmitter. The transmitter is the process that is responsible for copying the code into the victim process. For many techniques, this is done by the injector process itself, usually through a call to `NtWriteVirtualMemory`. However, some techniques trick the victim process into loading the code instead, e.g., by letting it read a malicious file.

Catalyst. The catalyst is the process responsible for triggering the execution of the injected code. Similar to the *Transmitter*, this is often done on the injector's side, e.g., by creating a thread within the victim process. Alternatively, the victim may also be tricked into calling the injected code itself.

File Dependency. A good amount of techniques require a copy of the injected code on the disk, usually in the form of a Dynamic Link Library (DLL). This means that such a file needs to be stored before execution can take place.

Shellcode Dependency. Some techniques require a small chunk of code to be injected directly into the victim process to execute the final payload.

Process and Threading Model. These two traits describe how malware selects and interacts with the victim process and its threads. Some techniques interact with already running processes or threads, while others spawn new ones. Alternatively, some techniques rely on the operating system itself and do not directly interact with any process or thread at all.

Memory Manipulation Model. This describes the dependency on directly allocating or manipulating the memory of the victim process. It is often accompanied by opening a process first and is present in most classic techniques.

Configuration Model. Some injection techniques depend on changing specific settings of the victim process or underlying OS. They may alter the Windows Registry, or install malicious plugins in a user application such as a web browser. Often, they also rely on the existence of a file on the disk.

3.3 Taxonomy

Using the identified traits, we define a taxonomy for code injection (Table 1) and discuss our classes below.

Active and Passive Injections. The most distinguishing feature that we observe deals with the level of interaction that is required by a code injection technique. Many techniques actively communicate with the victim process by creating or opening processes and threads and directly interacting with their memory. Since these kinds of interactions often translate to distinct sets of API calls, they can be observed by monitoring software more easily, which contributes to the stealthiness (or lack thereof) of the technique. Therefore, let us introduce the concept of *active* code injection techniques:

Definition 1 (Active Techniques). *A code injection technique is **active** if it directly interacts with the victim process or one of its threads, or actively changes the victim process' memory.*

Many existing techniques are active. For example, *Shellcode Injection* opens a handle to the victim process and uses it to directly inject executable memory into it with the help of a system call such as `NtWriteVirtualMemory` [21]. In contrast, a technique that abuses, for example, the shims infrastructure does not directly communicate with the target process, nor does it actively changes its memory. Rather, it lets the underlying OS load and execute the code instead [22]. Thus, *Shim Injection* is considered a *passive* technique.

Intrusiveness and Destructiveness. We can further subdivide active techniques by looking at the type of interaction that is required. For example, some techniques interrupt and manipulate the original execution of the victim process. Sometimes this happens in a way that parts of the application or the entire process stop working properly. Therefore, let us introduce the notion of intrusive and destructive injection techniques:

Definition 2 (Intrusiveness). *An active code injection technique is **intrusive** if it directly changes (parts of) the victim process' existing memory or threads.*

Definition 3 (Destructiveness). *A technique is **destructive** if it is intrusive and (parts of) the application stop(s) working due to the intrusive intervention.*

An example of a destructive technique is *Process Hollowing*, which creates a new victim process in a suspended state and replaces the original memory

Table 1: Taxonomy of code injection techniques and their characteristics.

			Technique	**Moment of Execution**[1]	**Transmitter**[2]	**Catalyst**[2]	**File Dependency**	**Shellcode Dependency**	**Process Model**[3]	**Threading Model**[3]	**Memory Manipulation Model**[3]	**Configuration Model**	**Functional on Windows 10**
Active	**Intrusive**	**Destructive**	Process Hollowing [44]	P	I	I		✓	N	E	N		✓
			Thread Hijacking [37]	A	I	I		✓	E	E	N		✓
			IAT Hooking [28]	A	I	V		✓	E		E		✓
			CTray Hooking [43]	A	I	V		✓	E		E		✓
			APC Shell Injection [38]	A	I	V		✓	E	E	N		✓
			APC DLL Injection [38]	A	I	V	✓		E	E	N		✓
	Non-Intrusive		Shellcode Injection [21]	A	I	I		✓	E	N	N		✓
			PE Injection [54]	A	I	I		✓	E	N	N		✓
			Reflective DLL Injection [21]	A	I	I		✓	E	N	N		✓
			Memory Module Injection [21]	A	I	I		✓	E	N	E		✓
			Classic DLL Injection [18,42]	A	I	I	✓		E	N	N		✓
Passive	**Configuration**		Shim Injection [22]	P	V	V	✓					✓	✓
			Image File Execution Options [48]	L	V	V	✓					✓	✓
			AppInit_DLLs Injection [40]	L	V	V	✓					✓	✓
			AppCertDLLs Injection [39]	L	V	V	✓					✓	✓
			COM Hijacking [41]	L	V	V	✓					✓	✓
			Windows Hook Injection [19]	A	V	I	✓						✓

[1] **A**: At any time, **P**: On process start, **L:** On library load.
[2] **I**: Injector process, **V**: Victim process.
[3] **N**: New process, thread or memory page creation, **E**: Existing process, thread or memory page manipulation.

content with new code [44]. As a result, upon resuming, the victim process does not perform its original activity anymore. This is in contrast with *Classic DLL injection*, which forces the victim to load an additional library from the disk without interrupting any threads or modifying their code [18]. Thus, *Classic DLL injection* falls under the *non-intrusive* category.

Configuration-based Injections. A more fine-grained subdivision can be made in our class of passive code injection techniques. This subdivision groups together techniques that require specific changes in the Registry, and is a direct result of the *Configuration Model* trait. An example of such a technique is *AppInit_DLLs Injection*, which registers a library file into the Registry. On the other hand, the

Windows Hook injection technique directly interfaces with system events and does not require a persistent configuration stored on the disk.

Summary and Implications. Our systematization shows that different code injection techniques take very different approaches to transmitting and executing code. As such, each technique has its own set of characteristics that a detection mechanism should take into account. Popular open-source sandboxes such as Cuckoo [3] and CAPE [2] implement detection mechanisms using API call tracing for most active techniques. They also include some more generic heuristics for detecting transmissions from one process to another by looking for API calls commonly associated with code injection (e.g., `NtWriteVirtualMemory`). However, the existence of passive techniques indicates that monitoring these common API calls might be insufficient. Most passive techniques are either not included in the signature database, or are not classified as a method of injection. Besides, since passive techniques leverage features of the underlying OS to perform their transmission and catalyst, the line between benign and injected memory pages becomes significantly more blurred—both types of pages come from the same *origin* and are allocated in the same way as normal pages. These important realizations indicate that more injections might be adopted in the malware scene than was previously thought. Thus, to uncover the landscape of code injection, we perform an empirical study on the prevalence of the different techniques.

4 Classification Models

We leverage our taxonomy to classify code injection behaviors manifested by malware. Since malware authors often obfuscate or pack their samples [14,34], static analysis is not a feasible solution to recognize code injection. Thus, we opt for an approach based on dynamic analysis, where we run a sample in a sandbox and record all APIs the system calls. Observing the entire system as opposed to a single process is crucial, as code injection is a procedure that inherently involves multiple processes. Furthermore, there exist various methods to spread malicious activity over multiple processes, including workloads that involve code injection [15,33,47]. We therefore assume a trace that includes API calls originating from *any* process running on the system.

Finding evidence of code injection in such an API call trace comes with two main challenges. First, as was discussed in Section 3.1, many techniques use standard APIs such as `NtAllocateVirtualMemory` that are also commonly used by programs for purposes other than code injection. Testing for their mere presence would make a trace such as the one presented in Table 2a indistinguishable from a trace where similar but unrelated system calls were recorded (Table 2b). The second challenge relates to injection techniques typically involving multiple API calls in sequence. Given the non-deterministic nature of concurrent systems, as well as malware intentionally reordering independent steps [13], we cannot assume a single order in which the APIs required for code injection are invoked by the sample and appear in the trace.

To overcome these two challenges, we leverage and reimplement the insights from previous work [35], and we build *behavior graphs* [16] for every code injec-

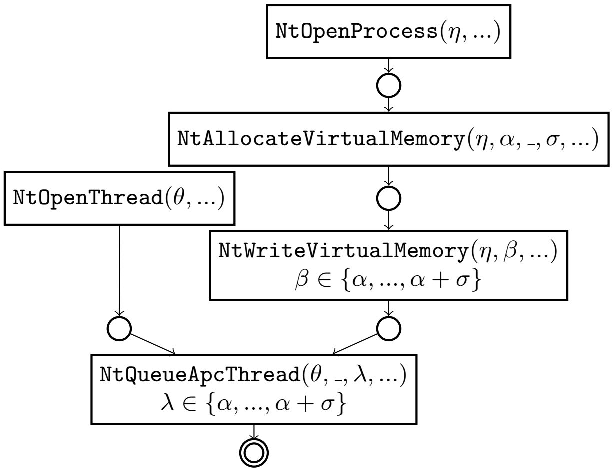
NtOpenProcess(η, ...)
NtAllocateVirtualMemory($\eta, \alpha, \_, \sigma, ...$)
NtOpenThread(θ, ...)
NtWriteVirtualMemory($\eta, \beta, ...$)
$\beta \in \{\alpha, ..., \alpha + \sigma\}$
NtQueueApcThread($\theta, \_, \lambda, ...$)
$\lambda \in \{\alpha, ..., \alpha + \sigma\}$

Fig. 1: A behavior graph implemented using a Petri net, modeling *APC Shell Injection*. For brevity, we use underscores (‘_’) and ellipses (‘...’) to discard irrelevant parameters. The accepting state is indicated by a double outline.

tion technique discussed in Section 3.1. These graphs are similar to dependency graphs that precisely specify the behavior in terms of APIs that we expect to be called when a certain code injection technique takes place. The specifications include symbolic variables and predicates that encode the relationships between the different API calls in terms of their arguments and return values. Besides, the edges in the graph describe not the exact order, but the general interdependence of each API call. This implicitly captures all possible permutations in which these API calls may appear in the trace. Finally, similar to a normal state machine, behavior graphs also contain accepting states. Once the evaluation of the graph reaches such a state, the behavior is considered *recognized*.

To implement behavior graphs efficiently, we use a variation of Petri nets [45] where we label the transitions with the API calls, symbolic variables and constraints. Figure 1 depicts an example of such a net that models the APC Shell Injection technique. The behavior of the transmitter is reflected in the chain of transition nodes matching on `NtOpenProcess`, `NtAllocateVirtualMemory` and `NtWriteVirtualMemory`. Note that all three transitions use a symbolic variable η for its first parameter (the process handle). This indicates that during the

Table 2: Two recorded API call traces. One sample implements APC Shell Injection and one sample uses similar but unrelated function calls.

(a) APC Shell Injection sample.

Time	**Observed API call**
t_i	`NtOpenProcess(0xA0, ...)`
t_{i+1}	`NtAllocateVirtualMemory(0xA0, ...)`
t_{i+2}	`NtWriteVirtualMemory(0xA0, ...)`

(b) Unrelated sample.

Time	**Observed API call**
t_j	`NtOpenProcess(0xD8, ...)`
t_{j+1}	`NtAllocateVirtualMemory(0x10, ...)`
t_{j+2}	`NtWriteVirtualMemory(0x28, ...)`

examination of the sample, the observed first argument for all three system calls must be equal. We also use extra constraints on the NtWriteVirtualMemory transition to restrict the value of β to the interval $\{\alpha, ..., \alpha + \sigma\}$. This indicates that β should be a memory address that falls within memory that was previously allocated in the victim process by a call to NtAllocateVirtualMemory. The use of these constraints effectively solves the problem of distinguishing between the traces shown in Tables 2a and 2b. Additionally, the transition node matching on NtQueueApcThread illustrates how the result of two independent API calls can be combined to express the catalyst of this technique, without assuming a specific order in which these APIs were invoked. Here, θ represents a thread handle obtained from a prior call to NtOpenThread, and λ is an entry point address that is constrained to be within the allocated memory range. Since our behavior graphs are implemented using Petri nets, the model does not progress until both the NtOpenThread and NtWriteVirtualMemory transitions have independently produced a result that is consistent with the constraints placed on the NtQueueApcThread call.

5 Framework Architecture

Leveraging behavior graphs, we model each injection technique presented in Section 3 and we build a framework that automatically recognizes the adoption of code injection in a given sample and specifically classifies the adopted techniques. Figure 2 depicts an overview of our framework, consisting of two components. The *Analyzer* acts as a front-end and is implemented in ∼6,000 lines of C# code. It takes samples as input and uploads them to an isolated *Examination Environment*. The Examination Environment executes each sample in a Virtual Machine (VM), and records an API call trace which is sent back to the Analyzer. The Analyzer then runs this trace through our behavior graphs, and reports back which of the code injection behaviors were recognized.

Our framework is built on top of DRAKVUF [32], which provides us with a virtualization-based, black-box binary analysis system that allows for system-wide monitoring of API calls. Given the nature of code injection techniques, this is a crucial requirement for us. Besides, DRAKVUF does not require an agent within the VM, which vastly reduces the risk of being fingerprinted by evasive samples. We use Windows 10 as the guest OS as it is the most market-dominant OS at the time of conducting this research [55]. We disable some

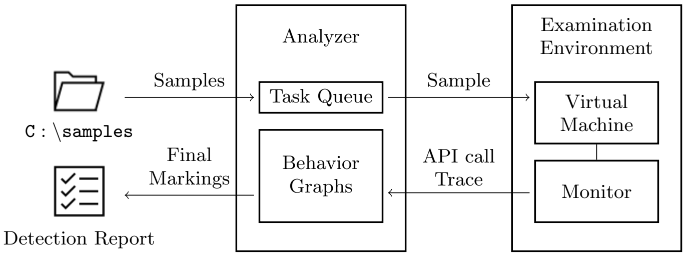

Fig. 2: Framework architecture overview.

background services that may unnecessarily prevent the samples from running or introduce artifacts in the traces, and provide it with (limited) access to the Internet. Following the community practices [51], we let malware run for a limited time, deny potentially harmful traffic (e.g., spam), and deploy our framework on a separate sub-network where no production machines are connected. After each analysis, we roll back the VM to a clean snapshot to revert side effects that malware might introduce. This also prevents potential denial of service attempts. Our setup was approved by our Ethics Committee.

6 Experimental Results

We test our automated framework to assess its capabilities to classify code injection techniques and perform a large-scale measurement of the general prevalence of code injection in malware and the distribution of the different techniques.

6.1 Datasets and Setup

To verify that our measurement framework correctly classifies the studied injection techniques, we first assembled a ground truth dataset of 63 code injection samples covering all the studied techniques. Our dataset contains both samples that we implemented ourselves, as well as handpicked open-source implementations and real-world samples. We also include 20 malicious samples that do not adopt code injection, as well as 1,147 benign applications. The benign samples include 976 executables from `C:\Windows\System32` and `C:\Windows\SysWOW64`, as well as 171 popular portable applications, e.g., VLC Media Player and WinSCP. For our measurement, we collected 47,128 random samples from VirusTotal [57] flagged by at least three AV engines (as suggested by related work [61]) and spread over 2017–2021. We then used AVClass [53] to assign samples to family labels. Table 3 describes the resulting dataset and its family distribution.

Analysis Timeout. Around 65% of malware runs completely in less than 2, and 81% does not need longer than 10 minutes [31]. Since injection is likely one of the first performed actions, we pick 6 minutes as a time limit per sample.

6.2 Framework Assessment

Table 4 shows an overview of the classification capabilities of our framework on the ground truth dataset, making a distinction between picking up on the presence of code injection and exactly classifying the techniques. In the following, we will discuss the performance of our framework in more detail.

Classification Capabilities. Our framework successfully recognizes the usage of code injection for all techniques, except for *IAT Hooking*. While this technique is destructive, we cannot recognize these injections due to behavior graphs not being able to test for faulty or *absent* behavior as a result of rerouting an API call. Furthermore, this technique only requires two calls to `NtWriteVirtualMemory` for both transmitting and preparing the catalyst respectively. While we can observe these calls, we cannot distinguish between the ones that place hooks and

Table 3: Malware family distribution in our dataset. The columns indicate the sample count and the fraction of positive samples.

	Total		**2017**		**2018**		**2019**		**2020**		**2021**	
Family	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**
virlock	5,783	0.5%	111	1.8%	131	0.8%	301	9.3%	5,057	0.0%	183	0.0%
dinwod	3,180	0.1%	2,763	0.0%	71	2.8%	71	0.0%	72	0.0%	203	0.0%
sivis	1,066	0.0%	12	0.0%	86	0.0%	71	0.0%	19	0.0%	878	0.0%
berbew	862	99.0%	144	100.0%	12	100.0%	283	100.0%	105	96.2%	318	98.4%
upatre	862	0.2%	190	1.1%	209	0.0%	187	0.0%	189	0.0%	87	0.0%
virut	861	1.4%	200	5.0%	138	0.0%	486	0.2%	33	3.0%	4	0.0%
delf	843	5.8%	31	3.2%	52	3.9%	189	13.2%	136	15.4%	435	0.0%
kolabc	837	0.0%	2	0.0%	12	0.0%	8	0.0%	0	0.0%	815	0.0%
vobfus	816	1.2%	156	0.6%	225	1.3%	41	9.8%	14	0.0%	380	0.5%
wapomi	738	0.4%	318	0.9%	63	0.0%	17	0.0%	339	0.0%	1	0.0%
wabot	596	0.0%	377	0.0%	50	0.0%	117	0.0%	43	0.0%	9	0.0%
vindor	594	0.0%	32	0.0%	36	0.0%	37	0.0%	0	0.0%	489	0.0%
allaple	567	0.2%	193	0.5%	88	0.0%	276	0.0%	9	0.0%	1	0.0%
gator	530	0.0%	63	0.0%	2	0.0%	103	0.0%	34	0.0%	328	0.0%
hematite	470	0.0%	15	0.0%	197	0.0%	230	0.0%	26	0.0%	2	0.0%
vtflooder	462	0.4%	137	1.5%	32	0.0%	58	0.0%	23	0.0%	212	0.0%
shipup	428	88.8%	58	94.8%	251	90.0%	59	86.4%	55	81.8%	5	60.0%
gepys	418	89.2%	27	88.9%	277	89.2%	67	82.1%	40	100.0%	7	100.0%
Other	27,215	9.4%	5395	11.1%	6259	9.5%	6424	12.2%	3498	8.3%	5,642	4.9%
Total	47,128	9.1%	10,224	8.3%	8,191	13.4%	9,025	13.6%	9,692	5.1%	9,999	6.1%

inject other types of memory. Note that, this does not mean that our framework is blind to destructive techniques. For example, in *Process Hollowing*, the catalyst always calls `NtSetContextThread` and `NtResumeThread`, whose arguments can be traced back to previously observed transmitter API calls, and thus can be reliably tested for. However, our framework sometimes confuses it with *Thread Hijacking*, as many hollowing implementations are nearly identical to it, and only include an extra call to `NtUnmapViewOfSection` to "hollow" out the victim process before the payload is transmitted. Again, while behavior graphs can encode this call for Process Hollowing, they cannot encode its absence for Thread Hijacking, causing the latter to be sometimes incorrectly identified as well. Therefore, if both techniques were detected in a sample, we assume that only Process Hollowing was implemented instead.

For three techniques (*PE Injection*, *Reflective DLL Injection*, and *Memory Module Injection*), our framework can recognize the presence of an injection, but not exactly identify the specific technique. The limited granularity of the API call trace causes some techniques to have a near-identical pattern of API calls for their transmitters and catalysts. In this case, the three methods become indistinguishable from *Shellcode Injection*, and can therefore only be classified as such. This is a reasonable compromise, as, since the only difference between these techniques is the format of the actual injected memory, they can be seen as a special case of injected shellcode. Thus, while this classification does not completely reflect the exact exhibited technique, it is not an incorrect classification either. All these techniques belong to the sample class in our taxonomy. We, therefore, refer to this group of injections as *Generic Shell Injection*.

Table 4: Overview of recognized techniques. *Match*: An injection was recognized. *Exact*: The correct technique was identified. Asterisk (*): Technique may be confused with another.

Technique	**Match**	**Exact**
Process Hollowing	✓	✓
Thread Hijacking	✓	✓*
IAT Hooking		
CTray Hooking	✓	✓
APC Shell Injection	✓	✓
APC DLL Injection	✓	✓
Shellcode Injection	✓	✓
PE Injection	✓	
Reflective DLL Injection	✓	
Memory Module Injection	✓	
Classic DLL Injection	✓	✓
Shim Injection	✓	✓
Image File Execution Options	✓	✓
AppInit_DLLs Injection	✓	✓
AppCertDLLs Injection	✓	✓
COM Hijacking	✓	✓
Windows Hook Injection	✓	✓

Performance Metrics. All samples that do not implement code injection were correctly marked negative by our framework. The 1,147 benign Windows applications were also marked negative, except for one System32 program. This program (`osk.exe`) implements an on-screen keyboard and simulates key presses when the user clicks the virtual key buttons. We found that it indeed uses Windows Hook Injection to send the simulated key presses to other processes. This confirms that code injection is also used for legitimate purposes, emphasizing that the use of code injection is insufficient for classifying a sample as malicious.

Our framework has a true positive rate of 87.50% and an F1-score of 93.0% on the samples that implement code injection. The false negatives are mainly caused by some samples not activating themselves during the analyses. In particular, implementations of *Windows Hook Injection* are susceptible since their catalyst sometimes requires user input (e.g., key presses) to run the payload. Note that, this is a limitation of any dynamic analysis-based examination environment.

6.3 Prevalence Measurement

Now that we assessed that the results of our framework are reliable, we can measure the adoption of code injection in the malware scene. Table 3 summarizes the observed prevalence of code injection within our dataset of 47,128 samples. We identified a total of 4,278 samples (9.1%) that perform at least one type of code injection. Note that, as we discuss in Section 7, this number represents a lower bound. To further test whether the classification made by our framework is consistent, we picked 20 positive samples covering all the detected techniques and 20 negative samples, and we manually verified our results. To the best of our reversing effort, all the classifications made by our framework were correct. Naturally, this does not exclude the presence of undetected false negatives (Section 8). Overall, the fraction of samples observed to adopt code injection varies from 5.1% to 13.6% per year. While this fluctuation does not seem to follow any particular motif, the distribution of the implemented techniques over time reveals interesting patterns.

Distribution of Techniques. Table 5 and Figure 3 show the distribution of the different adopted techniques, and Table 6 shows the generally observed preference of techniques in each of the years 2017–2021. Note that, the percentages do not add up to 100% as some samples implement multiple code injection techniques. Specifically, 94.65% of the positive samples in our dataset manifested one injection technique, 5.26% manifested two techniques, and four samples exhibited three techniques.

We can see that *Process Hollowing* and *Generic Shell Injection* are among the more popular choices of malware authors. Since these are traditional methods, and the majority of malware authors tend to copy code from others [17], this is an expected result. However, the popularity of these techniques is decreasing, while others are rising. If we aggregate all techniques by their class, as shown in Table 7, we can see that many of these rising techniques are *Configuration-Based* injections. Most notably, in 2018, the *AppInit_DLLs Injection* technique almost overcame all active techniques combined on its own, and in 2020, the aggregation of all Configuration-Based techniques convincingly surpassed them.

Table 5: Observed general prevalence and distribution of code injection techniques in the sample sets from 2017 to 2021.

Technique	**2017**		**2018**		**2019**		**2020**		**2021**		**Total**	
Process Hollow	230	27.2%	260	23.6%	276	22.5%	92	18.5%	92	15.2%	950	22.2%
Thread Hijack	87	10.3%	123	11.2%	78	6.4%	12	2.4%	52	8.6%	352	8.2%
CTray Hook	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
APC Shell	2	0.2%	13	1.2%	1	0.1%	2	0.4%	3	0.5%	21	0.5%
APC DLL	0	0.0%	0	0.0%	1	0.1%	1	0.2%	0	0.0%	2	0.1%
Generic Shell	174	20.6%	138	12.6%	218	17.7%	83	16.7%	37	6.1%	650	15.2%
Classic DLL	2	0.2%	4	0.4%	70	5.7%	3	0.6%	0	0.0%	79	1.9%
Shim	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
IFEO	86	10.2%	25	2.3%	61	5.0%	75	15.1%	80	13.2%	327	7.6%
AppInit_DLLs	86	10.2%	519	47.2%	170	13.8%	137	27.5%	19	3.1%	931	21.8%
AppCertDLLs	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
COM	240	28.4%	69	6.3%	406	33.0%	111	22.3%	341	56.4%	1167	27.3%
Windows Hook	0	0.0%	4	0.4%	21	1.7%	7	1.4%	0	0.0%	32	0.8%
Total	846	8.3%	1100	13.4%	1229	13.6%	498	5.1%	605	6.1%	4278	9.1%

Since samples within a family often employ very similar behaviors [10], and families differ in size, some techniques might be overrepresented in Figure 3. Thus, Figure 4 presents a different view of the data, where all samples within the same family are considered as one instead. If one sample within a family performs

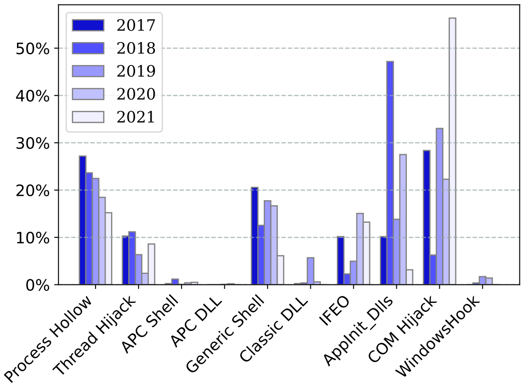

Fig. 3: Distribution of code injection techniques in malware, 2017 – 2021.

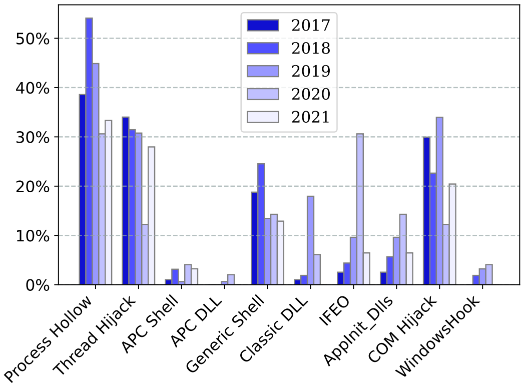

Fig. 4: Distribution of code injection techniques exhibited by malware in our dataset, normalized by malware family.

a given type of code injection, then this family is considered to implement this technique. We see *Process Hollowing* dominating the market, closely followed by *Thread Hijacking*. We also see that the adoption rates of passive techniques such as *AppInit_DLLs Injection* are reduced, but remain significant and are increasing over the years.

7 Discussion and Takeaways

Our study shed light on the understanding and adoption of code injection and provided important insights.

Trend Shift in the Adopted Techniques. In Section 3 we presented the existence of code injection techniques that can take a very different high-level approach from traditional methods. Examining our measurements, we conclude this is not merely theoretical. In fact, Figure 3 and Table 7 suggest that *the fraction of malware using passive techniques is growing towards the majority.*

Table 6: General preference of technique adopted by malware families (*AppInit*: AppInit_DLLs Injection, *COM*: COM Hijacking, *Hollow*: Process Hollowing, *IFEO*: Image File Execution Options, *Shell*: Generic Shellcode Injection, *Thread*: Thread Hijacking, and *WHook*: Windows Hook Injection).

Family	**2017**	**2018**	**2019**	**2020**	**2021**	**Total**
virlock	COM	Shell	Shell			Shell
dinwod	COM	COM				COM
berbew	COM	COM	COM	COM	COM	COM
upatre	COM					COM
virut	IFEO		Shell	WinHook		IFEO
delf	Thread	Thread	Hollow	Hollow		Hollow
vobfus	Hollow	Hollow	Hollow		Hollow	Hollow
wapomi	COM					COM
allaple	COM					COM
vtflooder	COM					COM
shipup	AppInit	AppInit	AppInit	AppInit	AppInit	AppInit
gepys	AppInit	AppInit	AppInit	AppInit	AppInit	AppInit
Other	Hollow	Hollow	Hollow	Shell	Hollow	Hollow
Total	COM	AppInit	COM	AppInit	COM	COM

Table 7: Distribution of classes of code injection techniques exhibited by malware in the sample sets from 2017 to 2021.

Class	**2017**		**2018**		**2019**		**2020**		**2021**		**Total**	
Active	495	58.5%	538	48.9%	644	52.4%	193	38.8%	184	30.4%	2054	48.0%
Intrusive	319	37.7%	396	36.0%	356	29.0%	107	21.5%	147	24.3%	1325	31.0%
Destructive	317	37.5%	383	34.8%	354	28.8%	104	20.9%	144	23.8%	1302	30.4%
Non-Intr.	176	20.8%	142	12.9%	288	23.4%	86	17.3%	37	6.1%	729	17.0%
Passive	412	48.7%	617	56.1%	658	53.5%	330	66.3%	440	72.7%	2457	57.4%
Config.	412	48.7%	613	55.7%	637	51.8%	323	64.9%	440	72.7%	2425	56.7%

This is an indication that malware developers have started to shift strategies for detection avoidance, and adopted techniques that do not depend on calls to `NtWriteVirtualMemory`. Another explanation could be that these new techniques also serve purposes other than running in the context of another process. For example, *AppInit_Dll Injection* tricks Windows into loading a DLL in new processes by changing a setting in the Registry. Since this setting is stored on the disk, this can be used for persistence.

Adoption of Unconventional Techniques. Additionally, the results in Figure 4 show that this trend shift is not only happening for large malware families but also started to appear in the bulk of smaller families as well. While most families still use *Process Hollowing*, we see an increase in the adoption rate of all passive techniques over the years. This further shows that *this shift towards less traditional methods is happening. Process Hollowing* is one of the oldest and most known techniques, which analysis systems can effectively track, and our results suggest that, for this reason, attackers now tend to disregard its adoption.

Code Injection among Families. From Table 3 we can see the positive rate of code injection techniques within a single family is typically either very low

($< 2,0\%$) or very high ($> 80,0\%$). This indicates that *if a sample implements a code injection technique, then it is very likely that its entire family implements some form of code injection*, and can thus be seen as a feature that a family can be characterized by. However, we also see that malware families switch between techniques over time (Table 6), and as such, the specifically adopted techniques are not enough to identify a family alone.

Insufficiency of Common Heuristics. A trend shift has important implications from a defender's perspective. The main insight is that *standard heuristics, e.g., looking for artifacts such as suspicious memory pages or API calls to* `NtWriteVirtualMemory` *(both commonly used in state-of-the-art [3,7,9,21,26,59]), are insufficient* for reliably detecting an entire class of code injections. This means that endpoint protection solutions need more sophisticated mechanisms to recognize and track emerging techniques (i.e., passive techniques).

Need for Combination of Behavioral Models. As discussed in Section 6, behavior graphs based on API/system calls cannot recognize *IAT Hooking* injections. Since this technique does not depend on APIs to activate the injected payload, signatures that look for evidence in an API trace cannot find any. This suggests that *sandbox developers and future researchers should combine multiple approaches to find evidence of the use of code injection*, or use strategies that can track behavior on a lower level, e.g., the approach suggested in [30].

Implications for Future Studies. Our results directly affect future research on malware analysis. Studies based on *dynamic analysis are bound to miss significant portions of malicious behaviors if they do not comprehensively account for the variety of code injection techniques, including passive techniques.* Furthermore, while code injection is significantly more present in malware, injection is also used by legitimate applications for benign purposes (as seen in the `osk.exe` example). Thus, although observing code injection is a strong indicator of suspicious behaviors, *it cannot be solely adopted as a criterion for malware detection.*

Adoption Rate. Finally, it is important to mention that the results of our measurement are likely a *lower bound* on the actual adoption of code injection, which further stresses the importance of our insights. While we took some countermeasures to mitigate evasion (e.g., using a stealthy, agent-less instrumentation environment), addressing evasive malware is an undecidable problem, and it is thus possible that evasive samples did not manifest their behavior (i.e., code injection) during our analyses. As a consequence, the number of samples implementing code injection is likely higher than what was observed in our study, further stressing that *future research should carefully take code injection into account to avoid biases* when performing malware behavior analysis.

8 Limitations and Future Work

Our study does not come without limitations. First, while the theoretical model of behavior graphs is fairly generalized, our implementation might be too specific to recognize mutations in the techniques. This is especially the case with Configuration-based techniques, as they access specific keys in the Registry, and thus require matching on API calls with specific paths. While we can match those paths, this does not encapsulate the core characteristic of abusing the settings of the OS. If another technique uses a different key, a new model is required.

A similar limitation can be found in the use of exact system calls in behavior graphs. This has the downside that different but semantically equivalent sets of API calls require multiple transition nodes to be added. This could be improved by adding a preprocessing phase that lifts API calls in the trace into *event classes* (e.g., similar to [35]), and then using these classes of events in our transition nodes instead. Alternatively, behavior graphs could be extended to allow for matching on multiple different types of APIs within a single transition node, such that these equivalence classes could be directly built into the graphs themselves. Both options would allow the graph to match a higher abstraction of events that the sandbox observes while avoiding additional structural complexity.

Evasion. Although our attempts to mitigate evasion (e.g., using a stealthy, agent-less instrumentation environment), and despite our goal is not to implement a detection system, malware could evade our framework and, thus, limit our measurement. First, behavior graphs are still a form of signature-based classification, making new unknown techniques invisible to our framework. This might also mean that the *IAT hooking* technique could be widely adopted but remained unnoticed in our observations (Section 6.2). Besides, malware that strictly depends on a remote C2 server might not exhibit its behavior if this server is offline. Furthermore, samples that require some form of interaction (e.g., Trojan Horses embedded in a GUI-based application) would not execute properly in our examination environment. Finally, samples might not execute code injection within our analysis timeout (although the majority of samples run within our limit [31]) or might still recognize our analysis environment [24,13,32]. Nonetheless, the potential presence of evasive samples could imply that code injection is more frequent than what we observed, further stressing our insights.

9 Related Work

Code Injection Classification. The idea of identifying common characteristics and placing code injection techniques in classes is a relatively new concept. Barabosch et al. introduced two forms of injections and execution models that are similar to our *process-* and *thread model* [11]. However, their focus is on threads only, leaving out a whole class of techniques. As discussed in Section 3, there exist techniques that do not create or manipulate threads directly.

Behavior Modeling. Similar to ours, most automated systems for behavior analysis rely on dynamic analysis [3,12,32]. While these systems have been very thorough with their examination, they stop at providing basic interpretations. Martignoni et al. refer to this as the *semantic gap* [35], and also address this with behavior graphs. However, their approach relies on taint-analysis within a *single* process. Thus, behaviors distributed over multiple processes cannot be modeled, a crucial element in the context of code injection.

Various methods exist to detect stealthy malware by the means of anomaly detection [52,58]. One limitation of adopting such a strategy is that it requires a database of normal behavior profiles for every potential victim process. In the case of code injection, this task becomes infeasible, as many victim processes (such as `explorer.exe`) are either closed source or too complex to model in a single automaton or graph. Besides, anomalies are at most a weak indicator for code injection, as there are other ways to let benign processes behave abnormally.

Code Injection Identification. Barabosch et al. proposed a method for detecting code injection leveraging the honeypot paradigm [10], by imitating attractive victim processes and monitoring for anomalies. However, this heavily relies on malware selecting these decoy processes as victims. Furthermore, it also faces the problem of not being able to monitor child processes, making popular techniques such as *Process Hollowing* undetectable. As an alternative approach, they also proposed to dump the system's memory and search for suspicious memory pages [9]. However, this assumes that benign pages can be distinguished from injected ones, which can be difficult for passive techniques. Furthermore, only some states of a machine are captured, requiring the victim process to be alive upon taking snapshots if we want to find any evidence. Finally, Korczynski et al. presented an approach based on system-wide taint-analysis to detect the presence of code injection and identify the responsible instructions [30]. Unfortunately, all these approaches are not applicable to our measurement framework, as they do not distinguish and classify different injection techniques, which is essential to perform an in-depth study like ours. Proprietary sandboxes, such as ANY.RUN [1] and Joe Sandbox [4], provide indicators of the occurrence of code injection. However, they do not recognize the specific techniques and do not provide information about their analysis approach, as they are fully closed-source.

10 Conclusion

We conducted a systematic and empirical study on code injection techniques and proposed a taxonomy to group such techniques into classes. Leveraging our taxonomy, we implemented a framework to classify the adoption of code injection in malware samples. We used our framework to collect empirical evidence on the prevalence of code injection, as well as the distribution of the adopted techniques, in the malware scene from 2017 to 2021. Our empirical results show that at least 9.1% of all examined samples perform code injection Besides, we showed a shift in trend: More traditional techniques are getting less used, while new, previously overlooked techniques are becoming more prevalent. Finally, our study provided important insights and takeaways for future research in the field of malware behavior analysis.

Acknowledgements

We are grateful to the reviewers as well as to Maya Daneva who have provided us with valuable insights and feedback. We also thank VirusTotal for granting us access to their academic malware sample datasets. This work has been partially supported by the Mercedes project, Grant No. 639.023.710, funded by The Netherlands Organisation for Scientific Research (NWO).

References

1. ANY.RUN - Interactive Malware Hunting Service. https://any.run/ (2022)
2. CAPE Sandbox. https://capesandbox.com/ (2022)
3. Cuckoo Sandbox. https://cuckoosandbox.org/ (2022)
4. Joe Sandbox - Deep Malware Analysis. https://www.joesandbox.com/ (2022)
5. User Account Control. https://docs.microsoft.com/en-us/windows/security/identity-protection/user-account-control/user-account-control-overview (2022)
6. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS) (2020)
7. Alrawi, O., Ike, M., Pruett, M., Kasturi, R.P., Barua, S., Hirani, T., Hill, B., Saltaformaggio, B.: Forecasting malware capabilities from cyber attack memory images. In: Proceedings of the USENIX Security Symposium (2021)
8. AVTest: Malware Statistics & Trends Report. https://www.av-test.org/en/statistics/malware/ (2021)
9. Barabosch, T., Bergmann, N., Dombeck, A., Padilla, E.: Quincy: Detecting host-based code injection attacks in memory dumps. In: Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2017)
10. Barabosch, T., Eschweiler, S., Gerhards-Padilla, E.: Bee Master: Detecting Host-Based Code Injection Attacks. In: Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2014)
11. Barabosch, T., Gerhards-Padilla, E.: Host-based code injection attacks: A popular technique used by malware. In: Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE) (2014)
12. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology (2006)
13. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: An overview of malware detection and evasion techniques. In: Proceedings of the International Symposium on Leveraging Applications of Formal Methods (2018)
14. Borello, J.M., Mé, L.: Code Obfuscation Techniques for Metamorphic Viruses. Journal in Computer Virology (2008)
15. Botacin, M., de Geus, P.L., Grégio, A.: "VANILLA" Malware: Vanishing Antiviruses by Interleaving Layers and Layers of Attacks. Journal of Computer Virology and Hacking Techniques **15**(4), 233–247 (Dec 2019)
16. Christodorescu, M., Jha, S., Kruegel, C.: Mining Specifications of Malicious Behavior. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. Association for Computing Machinery (2007)
17. de la Cuadra, F.: The geneology of malware. Network Security (2007)
18. Dejan Lukan: Using CreateRemoteThread for DLL Injection on Windows. https://resources.infosecinstitute.com/topic/using-createremotethread-for-dll-injection-on-windows/ (2013)
19. Dejan Lukan: Using SetWindowsHookEx for DLL Injection on Windows. https://resources.infosecinstitute.com/topic/using-setwindowshookex-for-dll-injection-on-windows/ (2013)
20. Du, M., Hu, W., Hewlett, W.: AutoCombo: Automatic Malware Signature Generation Through Combination Rule Mining. In: Proceedings of the ACM International Conference on Information & Knowledge Management (2021)

21. Elastic Security: Hunting In Memory. https://www.elastic.co/blog/hunting-memory (2019)
22. F-Secure: Hunting for Application Shim Databases. https://blog.f-secure.com/hunting-for-application-shim-databases/ (2018)
23. Falliere, N., Murchu, L.O., Chien, E.: W32.Stuxnet dossier. White paper, Symantec Corp., Security Response (2011)
24. Galloro, N., Polino, M., Carminati, M., Continella, A., Zanero, S.: A systematical and longitudinal study of evasive behaviors in windows malware. Computers & Security (November 2021)
25. Griffin, K., Schneider, S., Hu, X., Chiueh, T.c.: Automatic generation of string signatures for malware detection. In: Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID) (2009)
26. Hasherezade: PE-Sieve. https://github.com/hasherezade/pe-sieve (2018)
27. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual (2022), Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z
28. iRed.team: Import Adress Table (IAT) Hooking. https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking (2020)
29. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: UNVEIL: A Large-Scale, automated approach to detecting ransomware. In: Proceedings of the USENIX Security Symposium (2016)
30. Korczynski, D., Yin, H.: Capturing Malware Propagations with Code Injections and Code-Reuse Attacks. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
31. 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 ISOC Network and Distributed System Security Symposium (NDSS) (2021)
32. 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 Annual Computer Security Applications Conference (ACSAC) (2014)
33. Ma, W., Duan, P., Liu, S., Gu, G., Liu, J.C.: Shadow Attacks: Automatically Evading System-Call-Behavior based Malware Detection. Journal in Computer Virology (2012)
34. Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., Balzarotti, D.: Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS) (2020)
35. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID) (2008)
36. Microsoft: Understanding Shims. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-7/dd837644(v=ws.10) (2012)
37. MITRE ATT&CK: Process Injection: Thread Execution Hijacking. https://attack.mitre.org/techniques/T1055/011/ (0220)
38. MITRE ATT&CK: Process Injection: Asynchronous Procedure Call. https://attack.mitre.org/techniques/T1055/004/ (2020)
39. MITRE ATT&CK: Event Triggered Execution: AppCert DLLs. https://attack.mitre.org/techniques/T1546/009/ (2020)
40. MITRE ATT&CK: Event Triggered Execution: AppInit DLLs. https://attack.mitre.org/techniques/T1546/010/ (2020)

41. MITRE ATT&CK: Event Triggered Execution: Component Object Model Hijacking. https://attack.mitre.org/techniques/T1546/015/ (2020)
42. MITRE ATT&CK: Process Injection: Dynamic-link Library Injection. https://attack.mitre.org/techniques/T1055/001/ (2020)
43. MITRE ATT&CK: Process Injection: Extra Window Memory Injection. https://attack.mitre.org/techniques/T1055/011/ (2020)
44. MITRE ATT&CK: Process Injection: Process Hollowing. https://attack.mitre.org/techniques/T1055/012/ (2020)
45. Murata, T.: Petri nets: Properties, analysis and applications. In: Proceedings of the IEEE (1989)
46. Olaimat, M.N., Aizaini Maarof, M., Al-rimy, B.A.S.: Ransomware Anti-Analysis and Evasion Techniques: A Survey and Research Directions. In: Proceedings of the International Cyber Resilience Conference (CRC) (2021)
47. Pavithran, J., Patnaik, M., Rebeiro, C.: D-TIME: Distributed threadless independent malware execution for runtime obfuscation. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT 19) (2019)
48. Pieter Arntz: An Introduction to Image File Execution Options . https://blog.malwarebytes.com/101/2015/12/an-introduction-to-image-file-execution-options/ (2015)
49. Polska, C.: More human than human - Flame's code injection techniques (2014)
50. Quarta, D., Salvioni, F., Continella, A., Zanero, S.: Toward Systematically Exploring Antivirus Engines. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) (2018)
51. Rossow, C., Dietrich, C.J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., Van Steen, M.: Prudent practices for designing malware experiments: Status quo and outlook. In: Proceedings of the IEEE Symposium on Security & Privacy (S&P) (2012)
52. Schneider, F.B.: Enforceable Security Policies. ACM Trans. Inf. Syst. Secur. **3**(1) (feb 2000)
53. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: Avclass: A tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID). Cham (2016)
54. Sevagas: PE Injection Explained. https://blog.sevagas.com/PE-injection-explained (2014)
55. statcounter: Desktop Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/desktop/worldwide/ (2021)
56. Victor M. Alvarez: YARA. http://virustotal.github.io/yara/ (2021)
57. VirusTotal: VirusTotal Malware Academic Dataset. https://www.virustotal.com/ (2021)
58. Wang, Q., Hassan, W.U., Li, D., Jee, K., Yu, X., Zou, K., Rhee, J., Chen, Z., Cheng, W., Gunter, C.A., et al.: You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS) (2020)
59. White, A., Schatz, B., Foo, E.: Integrity verification of user space code. In: Proceedings of the Thirteenth Annual DFRWS Conference (2013)
60. Wyke, J.: The ZeroAccess Botnet Mining and Fraud for Massive Financial Gain. Sophos Technical Paper (2012)
61. Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G.: Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines. In: Proceedings of the USENIX Security Symposium (2020)