Can You Run My Code? A Close Look at Process Injection in Windows Malware

Giorgia Di Pietro
Sapienza University of Rome
Rome, Italy
g.dipietro@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
Rome, Italy
delia@diag.uniroma1.it

Leonardo Querzoni
Sapienza University of Rome
Rome, Italy
querzoni@diag.uniroma1.it

Abstract

Process injection is a core technique for malware authors to evade detection and enhance stealth. Despite its widespread use and importance in malware analysis, process injection remains under-explored in academic research, with prior work often limited to specific techniques or lacking a systematic approach.

This paper proposes a principled analysis methodology centered on fundamental operational steps inherent to all known process injection variants. By looking for the co-occurrence of a minimal set of said steps and correlating them via memory address identity, our approach overcomes the accuracy and overhead limitations of prior studies, enables reliable detection and fine-grained analysis of process injection attacks with tenable run-time costs.

We provide fresh insights into how threat actors leverage this technique by analyzing malware spotted in the wild from 2017 to 2023. An analysis of 56,340 representative samples from 2,667 malware families estimates process injection as a dominant evasion strategy, and suggests that threat actors continuously adapt their choices and implementation variants in response to evolving defense mechanisms and community knowledge. Comparative experiments then show that our method outperforms dedicated solutions and mainstream sandboxes in identifying injection activity.

To foster future research, we share with the community the implementation, dataset, and experimental logs from this study.

CCS Concepts

• **Security and privacy → Malware and its mitigation**.

Keywords

Code injection, fileless malware, malware detection

ACM Reference Format:
Giorgia Di Pietro, Daniele Cono D'Elia, and Leonardo Querzoni. 2025. Can You Run My Code? A Close Look at Process Injection in Windows Malware. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3708821.3736206

1 Introduction

Malware, the primary tool of cybercriminals, enables diverse attacks and affects numerous victims. Its rapid evolution poses significant challenges to the security community [46]. To counter these threats, researchers and vendors work to improve detection mechanisms, whereas malware authors develop better techniques to evade them.

One effective strategy to evade process monitoring-based defenses consists in running malicious code under other processes. This is the core concept of *process injection*, a general attack method for executing arbitrary code in the address space of another process, hiding malicious activities under the guise of such victim.

Process injection can give malware operations an increased level of stealthiness, improving success in defense evasion and, in some cases, enabling privilege escalation and the circumvention of allow list-based policies. Process injection is a key method for executing fileless attacks [49], which have become increasingly prevalent and nowadays represent a dominant type of cyber threat [54].

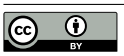
Detecting malicious behavior from injected payloads can be more challenging than with standalone samples for several reasons. Most injection techniques leave no trace on disk, inherently evading static detection methods (e.g., signature-based). In dynamic analysis, if the monitoring system fails to capture the precise moment the payload begins execution, the security solution is likely to miss the threat once its activities blend with the victim's legitimate activities [30]. Also forensics post-mortem analysis [2, 40, 41] may miss injected payloads, for example if the sample has memory wiping provisions.

A viable defense strategy would thus be to detect and prevent injection attempts. However, this task is far from straightforward. Most process injection attacks target Windows hosts, leveraging an extensive range of built-in system features designed to facilitate application interoperability and enhance functionality. Unfortunately, skilled malware authors have creatively repurposed these capabilities to inject code into the memory of other processes. This ingenuity is well-documented in white-hat blog posts [44, 70], technical reports [18, 50], and academic research [80].

In the presence of suspicious activity, anti-virus products (AVs) face an additional limitation by not being able to sustain close execution inspection for long, due to the risk of generating excessive overhead in program execution. Endpoint Detection and Response (EDR) vendors have lately explored high-level guidelines for characterizing process injection [37, 77, 79], but these are not always directly applicable and tend to cover only some techniques.

The academic community has devoted rather limited attention to analyzing the injection phenomenon, with prior efforts typically focusing on single techniques or schemes [21, 50, 51, 84]; a comprehensive analysis only emerged in 2023 [80]. Despite its prevalence and importance in the malware analysis practice, process injection remains an under-explored problem. We argue that a deeper, more systematic understanding of this phenomenon is needed.

Our approach. This paper introduces a detection and analysis methodology that targets fundamental operational traits inherent



ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/25/08
https://doi.org/10.1145/3708821.3736206

in all variants of process injection we are aware of. The methodology synergically capitalizes on two intuitions leveraged separately by prior work, and further relaxes the assumptions behind their deployment with the result of improving detection accuracy.

Starink et al. [80] survey common injection techniques and write reference implementations for them to derive behavior graphs that capture co-occurrence and inter-dependence of the API calls behind each technique. The authors use these graphs to analyze real-world malware: in an execution trace, they check for operations that manipulate *congruent memory addresses* and try to map them to known behavior graphs. We find the congruence criterion an effective proxy to discriminate related activity, removing the need for expensive taint tracking used in other works [51, 63].

As for recognizing candidate activity, we recall instead how Klein et al. [50] established that process injection tends to take place in *three operations*: allocating memory in the victim, writing the payload inside it, and triggering execution from it. They then surveyed techniques and identified the APIs of choice for each.

The core idea behind our proposal is to combine the co-occurrence of operational primitives with memory range congruence. To cope with implementation diversification from malware authors, we employ *equivalence classes* to model alternative ways of realizing each of the three core operations. This addresses a main limitation of the work of Starink et al. [80], as behavior graphs need an exact match with API identity, and also non-core operations must be identified.

But even such a method will fail if the adversary performs one of the three operations in an unanticipated way. We address this other limitation by stipulating and testing experimentally that *the presence of at least two of the three operations* is sufficient to indicate the presence of an injection behavior. This enables detection even when the injection process is not fully recognized or visible.

For attackers, they now must implement two operations in an unforeseen way, which is intrinsically harder (and analysts can update our classes as they learn one new implementation alternative).

For defenders, our approach eases the identification of variants when techniques evolve. Notably, we spotted an implementation variant now dominant in the wild for Process Hollowing, a technique that Starink et al. [80] identify as the most popular but underestimate in prevalence due to the strict patterns for the analysis.

Finally, we explicitly consider an aspect that prior work treated at orthogonal: malware evasion. By integrating our method with a state-of-the-art system for countering evasive behavior and running an ablation study, we show how samples that resort to injection techniques are often also evasive. Therefore, we rectify incomplete findings in prior work on the injection phenomenon's prevalence.

To estimate how widely the various injection techniques are used and gain insights into the choices of malware writers over time, we apply our method to a collection of Windows malware samples spotted in the wild between 2017 and 2023. We source 334,885 samples from two qualified collections: the VirusTotal Academic dataset (2017 to 2021) and the VX Underground Bazaar collection (2022 and 2023). To avoid bias from overly represented families, we apply balancing techniques and filter out near-duplicates, obtaining for the study a dataset of 56,340 representative malware samples.

Process injection remains a recurrent choice for malware writers, with on average 16.5% of analyzed samples employing at least one technique, and higher prevalence in recent years. Insights from fine-grained result analysis lead us to believe that threat actors have likely adapted their choices to the evolution of defense mechanisms and the community knowledge of emerging techniques. Finally, in a comparative experiment for detection efficacy, we note that mainstream open-source and closed-source sandboxes missed injection activity in 21.2% of the samples flagged by our approach.

We hope this research contributes to the advancement of malware analysis systems and to foster a clear understanding of an under-explored yet pivotal aspect of malware behavior within the academic community. The two contributions we propose are:

- A methodology for detecting process injection based on the (partial) co-occurrence of three operational primitives linked by memory range congruence, aimed at addressing the accuracy limitations of existing approaches;
- An experimental study that estimates the prevalence of injection attacks in Windows malware between 2017 and 2023, providing statistics and insights about the evolution of the techniques and the choices of threat actors.

We make available our code, dataset, and logs from the experiments at: https://github.com/Sap4Sec/process-injection/.

2 Background and Related Works

Process injection, or *code injection* in some literature, is a well-established attack method used by modern malware to bypass detection by end-host security solutions like AVs and EDRs.

Operation. Process injection involves loading and executing custom code within the memory space of another process, possibly a legitimate one. By executing malicious logic through a third party, malware attempts to hide its activity from monitoring solutions. This migration allows attackers to enhance a malware sample's stealth capabilities and, in some cases, escalate privileges, for example for ensuring a foothold across system reboots [81].

Process injection involves three main steps [50]: **(1)** allocating one or more memory regions in the victim, **(2)** writing the payload into the new memory region(s), and **(3)** scheduling the code for execution within the victim. Each step can take place differently depending on the specific injection technique employed, each offering distinct trade-offs in terms of stealth and reliability.

For instance, shellcode injection [64] allocates a buffer in the target victim process, writes the shellcode into it, and executes it by creating a remote thread. DLL injection from Disk writes in the victim's memory a filesystem path to a malicious library, then issues an API call to force the victim to load it. Process Hollowing [15] operates by creating a new process in suspended state, unmapping its original memory (an optional step, as our findings will show), allocating and populating memory with malicious code, and updating the process context to use such code as entry point before resuming it. In other techniques or implementation variants, attackers may hijack existing threads or use an Asynchronous Procedure Call (APC) [33] to queue the payload for execution by a legitimate thread. For brevity, we refer our readers to Appendix A for further background information on specific injection techniques. Despite their differences, all techniques manipulate the same fundamental stages, which are the common underpinnings of process injection.

For over two decades, process injection has posed significant challenges to malware analysis. Its effectiveness is recognized not

just by threat actors, who extensively use it, but also by security researchers, who have developed numerous offensive works leveraging it as a vector for implementing more sophisticated attacks [23, 30, 48]. Defensive research faces a unique challenge: the versatility of process injection has led to the development of numerous variants, each with its own implementation styles, favored by the extensive set of system APIs and exotic features (such as transacted files that become opaque to AVs) available in Windows.

Previous efforts by industry and academia have attempted to gather knowledge about process injection techniques, often reasoning on proof-of-concept implementations [38, 44, 59, 70]. The MITRE ATT&CK framework [18] models process injection as an adversary tactic, defining 12 techniques for it that include both *true injection* (i.e., the victim is a process already running in the machine) and victim process spawning (i.e., the malware sample starts and then abuses a process). Klein et al. [50] assembled a curated comprehensive catalog of true process injection techniques, categorizing them based on the three operation primitives mentioned above and detailing their requirements and limitations.

Detection. Several prior efforts have focused on detecting process injection. Barabosch et al. [21] propose a honeypot paradigm as a decoy to malware and monitor the environment for changes associated with this threat. The success of this approach depends on whether the sample chooses the decoy process as a victim, a problem that affects also network honeypots. The method handles only cases of true injection, and misses also the recursive patterns that we observed in our experiment and describe later in this paper.

Another avenue is acquiring and analyzing memory dumps [20]. Despite the potential wealth of indicators and the design opportunities for memory-based detection heuristics, this method has important shortcomings. For example, captures of single machine states may not reveal interesting patterns and are expensive to deploy frequently for online monitoring in a security solution. Furthermore, there are injection techniques that might not be detectable in a post-mortem memory dump analysis, and adversaries may exacerbate this problem by adopting memory wiping tactics.

To partially mitigate these limitations, Nasereddin et al. [61] explore the use of live memory analysis and parallel computing, using multiple agents to monitor different execution artifacts (memory protection features, PE headers, system registry contents, suspended processes) and processing traces collected with the Event Tracing for Windows (ETW) [35] technology. The work presents an experimental analysis involving six injection techniques.

Korczynski et al. [51] present a solution based on taint analysis to expose code injection and code reuse attacks. Similarly, Arefi et al. [63] build a reverse engineering tool based on whole-system taint analysis implemented in QEMU. While taint analysis can gather rich and accurate information, it notoriously incurs considerable performance and memory overheads, which significantly complicate practical usage and downstream integration (e.g., to improve the capabilities of AVs or sandboxes) of such a system.

We previously mentioned how Starink et al. [80] model injection activities through behavior graphs, which capture APIs calls and their interdependence as nodes and transactions within Petri nets [60]. Their workflow hinges on using exact schemas that, as we show, harms accuracy when broader event abstraction is required. The study estimates the prevalence of 17 injection techniques, classifying them into *active* and *passive* methods based on their level of interaction with the victim process. Passive techniques do not interact with a target process directly, but mainly[1] have a victim program load them on startup by tampering specific registry keys. The MITRE ATT&CK framework does not categorize them as injection, but rather as Event Triggered Execution techniques [4].

Snow et al. [78] design a hardware virtualization-assisted framework for fast, accurate detection and analysis of shellcode [78]. More recently, some researchers [84, 85] have applied machine learning to detect stealthy infection vectors, including process injection.

Discussion. Despite their contributions, current solutions ultimately struggle to capture the essence of process injection, hindering a deeper understanding of the problem.

To summarize their limitations: they tend to (1) rely on costly analyses (memory dumps, taint tracking) that may still miss behaviors depending on when they are enacted and the sample's characteristics, and (2) pursue a classification-driven approach towards strict rules derived syntactically from documented injection implementations. The first choice hinders practicality and still struggles with completeness, whereas the second choice is inevitably prone to missing implementation variants and emerging techniques.

A third and more practical limitation is that much injection activity is concealed by evasion techniques, making it harder to detect in analysis systems. While addressing evasion is an orthogonal challenge, the works discussed above lack provisions for it, leading to an underestimation of process injection prevalence. More critically, researchers may overlook variations in how malware implements process injection techniques across families and even within specimens of a single family. As a reference, in an experiment we restricted to the samples from 2019 in our dataset, we noted that, under a system with no anti-evasion provisions, no less than 47.3% of samples doing an injection went undetected.

We also compared the methodology we propose in this paper with the analysis of Starink et al. [80]. By intersecting our two source datasets, we achieved 95.6% coverage of their dataset. After applying deduplication and balancing methods, we mapped the results to our final dataset, identifying 64.1% more samples exhibiting injection behaviors than they do.

These reflections and data back our argument for a different approach to better understand process injection and its implications.

3 Analysis Methodology

To approach the problem of detecting and analyzing process injection, we reviewed the sources referenced in the previous section and complemented them with technical blog posts on implementation strategies, emerging ideas for injection variants, conversations with analysts, and, to a marginal extent, our experience with malware design and reversing. In this analysis, we identify 12 process injection techniques that we find representative of what spotted until now in the wild. A detailed explanation of why these techniques form a comprehensive set is given in Appendix A.

Virtually all known injection techniques share fundamental behaviors centered around the manipulation of memory areas. This

[1]The authors consider passive also Hook Injection, a technique that provides a running victim's thread identifier to an API that causes an eventual address space manipulation.

happens in three operational steps: these regions must be obtained, populated, and then the payload must be scheduled for execution. Understanding these underlying primitives and the relationships between them can be, as our approach will show, pivotal for grasping the nature of the problem and effectively detecting injection instances, even as new techniques continue to emerge and evolve.

When we said "virtually all" above, the three-step characterization partially leaves out the techniques systematized as passive injections by Starink et al. [80]. Our stance is that these techniques, as also defined in the MITRE ATT&CK framework [18], pertain to the persistence realm of malware techniques, and can be seen as injections mostly for having an unusual vessel to initiate execution. However, by stretching the definition of memory allocation in the first step, we would still be able to recognize these techniques.

For monitoring execution, we assume the instrumentation framework is trusted and resilient to techniques designed to evade API call logging. The latter point may be relaxed if orthogonal mitigations [3] are used to strengthen the implementation against stolen code and similar attacks. The focus of the methodology and experimental study of this paper is userspace malware: rootkits or samples that abuse kernel vulnerabilities for stealth operation fall outside the scope of this study (at the time of writing, we are unaware of malware of this kind making use of process injection).

As our readers will appreciate later in the paper, a practical advantage of our methodology lies in its simple deployment requirements: it only needs to trace specific API calls and their argument values. While we developed it to gain insights in how malware uses process injection, we believe it may have potential applications to security solutions such as sandboxes and anti-virus products due to its simplicity, efficacy in detection, and limited run-time overhead.

3.1 Overview

Capitalizing on two intuitions explored only partially in prior works [50, 80], our analysis approach generalizes the three operational primitives behind injection and correlates them by looking for a high-level workflow that sees congruent memory buffers manipulated throughout a concatenation of such primitive actions.

This reasoning is the lynchpin of our analysis methodology. By further defining the minimum set of such operations that suffices to detect injection behavior, we gain a more comprehensive view of the phenomenon and improve detection.

Like most dynamic malware analysis efforts, including recent proposals in this subfield [80], our approach relies on the observation of the sequence of Windows APIs that a program executes at run time. By reasoning on the occurrence of operations from specific functional classes and on their temporal (i.e., order) and spatial (i.e., memory) relationships, we identify injection patterns from heterogeneous techniques and their implementation variants.

Data capturing is an important component of our investigation. Depending on the features of the injection technique and the choices of the malware author, each operation primitive may be instantiated using different Windows APIs, which inevitably adds complexity (and increases the risk of missing relevant operations) when tracing. To manage this variability, we build functionally equivalent classes of APIs to realize each primitive action, encompassing both high-level and Native (i.e., `Nt` system calls in Windows terminology) APIs. For instance, to allocate memory in the address space of another process, we consider `VirtualAllocEx` (and its variants) and `NtAllocateVirtualMemory`. During execution, we only need to track argument and return values of these functions, without resorting to more expensive means like memory dump analysis or taint tracking explored in other efforts.

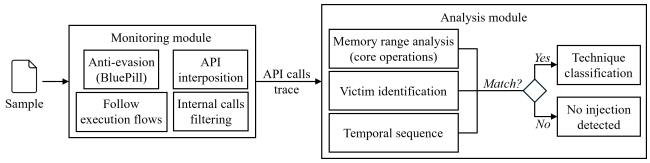


Figure 1: Architecture of the analysis framework.

Additionally, analyzing the activities of a single process may not suffice to comprehensively characterize an injection attack. Sophisticated malware may perform multi-level injections to fragment the attack, possibly to further evade detection and fingerprinting. We observe that 2.0% of injecting samples in our evaluation enact a recursive injection chain, where the sample initially infects a target process that, in turn, injects another victim. Therefore, we extend the monitoring instrumentation to other processes, specifically to the execution flows orchestrated in them by the original process undergoing analysis or, recursively, by its byproducts.

Figure 1 depicts the operation of our methodology as we implement it, and we detail the individual steps in the coming sections.

3.2 Main Challenges

Accurately detecting process injection requires addressing several critical aspects that, if mishandled, can result in poor detection outcomes. We introduce below the three main challenges we encountered when developing our method and the strategies we employed to overcome them: recognizing distinctive features, discarding false positives, and analyzing temporal relationships between operations. The remainder of the section will then expand on each challenge.

Execution monitoring-based detection involves making effective use of observed Windows API calls. While we tame implementation diversity by reasoning on classes of actions, adversaries may still come up with unanticipated ways to implement one operation, or the instrumentation of specific actions may be prone to incompleteness issues. Therefore, we identify a strategy that focuses on the *minimal* set of core operations to observe, which is sufficient to deem the presence of an injection attack. Then, we use any further available information to accurately classify the technique in use.

To address the risk of false positives, we build on insights from [80] by closely examining what memory areas within the victim process are touched by each monitored action and considering only those that operate on congruent address ranges. This lets us effectively discriminate the usage of such APIs between goodware and the injection machinery from malware, as only with the latter we observe (a minimal set of said) operations that insist on the same objects.

Finally, as malware may orchestrate the actions required for injection using multiple concurrent execution flows [80], we examine the sequence and timing of operations to validate their semantic consistency. Incidentally, this task is also key to understanding in greater depth the inner workings of a sample's strategy.

Distinctive features. A textbook process injection implementation would noticeably execute the three fundamental steps we described, possibly with the customary APIs documented for it. With real-world malware and the heterogeneity that characterizes it, the picture may become at times arguably more complex.

The first issue is implementation diversity: malware writers tend to use different, functionally equivalent APIs for one step, looking for gaps in what security products monitor or simply to hinder fingerprinting. In other cases, they adapt APIs outside their canonical usage to achieve an equivalent effect (for example, with atom tables and transacted files) for one step. Therefore, a strategy that looks for the occurrence of all three steps in the execution makes the detection method vulnerable to adversaries that successfully conceal one step by doing it in an unanticipated way.

Additionally, in some cases, an operation may be difficult to identify or two conceptually distinct steps may be realized at once. As examples, memory allocation may be implicit if targeting preallocated memory, or the allocation and writing could be carried out simultaneously, for instance leveraging the code sharing mechanism that Windows enables through memory section objects [34].

As a result, for any of the reasons above, a process injection schema may reveal fewer distinct steps in the execution than expected. However, through the analysis of various techniques (and a validation of the insight over goodware), we came to the conclusion that *the presence of at least two out of the three operational steps* is sufficient to indicate the presence of an injection behavior.

We thus devise an event analysis that identifies and correlates two such operations in order to deem if an injection took place. Any additional information present in the recorded events remains helpful to gain insights on the malware writers' choices. This analysis choice preserves efficacy: that is, it cannot introduce false negatives for the techniques we analyze. Additionally, it holds practical value, as we found that for 19.9% of samples doing injection we could witness only two of the three primitive events.

Let us discuss an exemplary scenario, depicted in Figure 2, that we distilled from a sample from the *neurevt* malware family. The sample creates a victim process in a suspended state and unmaps memory from its address space using `NtUnmapViewOfSection`.

These are typical traits of the Process Hollowing technique. However, instead of a textbook allocation of one or more memory buffers, it creates a new section with `NtCreateSection` to share parts of its memory with the victim process. As a writing operation, the malware resorts to `NtMapViewOfSection` twice: once locally to make the section point to the malicious code, and once remotely for the unmapped memory in the target process. At the end of this step, the target's memory is filled with the desired malicious code.

Finally, the malware resumes program execution by running code from the mapped view in the victim, using `SetThreadContext` and `ResumeThread` [66]. Notably, by mapping views, the malware bypasses the need to explicitly allocate memory pages, which some security tools might miss. In this case, our method treats remote section mapping as a write operation since it directly transfers code to the victim in one step. In summary, the overall workflow for this code consists of just two operations: writing and execution.

The methodology we propose is designed to capture these nuanced behaviors, enabling us to detect process injection even when

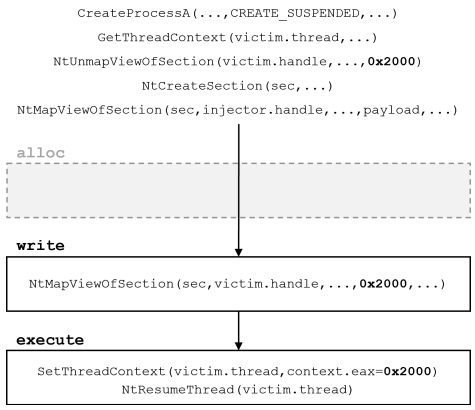


Figure 2: API call trace of a sample using Process Hollowing where only the *write* and *execute* primitives are observed. Address `0x2000` represents where the allocated memory starts and also where the injected payload's entry point resides.

traditional detection methods fall short. This benefit became particularly apparent for Process Hollowing, as through our evaluation we establish how its manifestation differs from the prior understanding in the state of the art [80]. Furthermore, as we compared our results with VirusTotal behavioral reports, even popular sandboxes struggled to recognize process injection in similar cases.

We noticed that, when modeling code injection with behavior graphs, Starink et al. [80] assume a fixed set of operations to reach the final accepting state for each technique. This makes their system less effective at handling cases with fewer exercised primitives. Their detection approach also relies on additional information, such as APIs related to opening process or thread handles associated with the victim process. While necessary for executing process injection, this information can be acquired more stealthily by malware, potentially causing the analysis to fail. As a result, their method may lead to inaccurate analyses due to false negatives.

Once an injection attempt is detected, our system differentiates between techniques using *distinctive* APIs, with operational patterns serving as secondary identifiers to resolve any ambiguities. For example, in the Process Hollowing case shown in Figure 2, the technique is classified based on the creation of a suspended victim process. When distinctive APIs are unavailable or too general, the core operations themselves allow us to distinguish between techniques. For instance, analyzing features of memory allocation and writing operations can reveal key differences. In the case of Shellcode Injection, the payload written into the victim process exhibits different characteristics compared to the otherwise very similar PE Injection, which can be identified by the (larger) size of the payload or the higher number of write operations (e.g., one per section). This approach allows us to detect subtle distinctions by closely examining the specific details of each technique.

Relevant operations. A design aspect to take into account for accurately detecting process injection is that many Windows APIs used for it are also commonly employed for legitimate purposes by programs and, in turn, by the Windows DLLs they rely on. For

example, Windows allows processes to write to each other's memory to support valid functionalities such as debugging, application interoperability (e.g., atom tables), and system monitoring. Therefore, API call traces from a program execution may include many occurrences of these APIs that turn out to be unrelated to injection attempts, and an analysis like our should discard them without affecting its ability to identify true injection attempts.

Our approach hinges on the principle that injection operations *consistently operate on congruent memory areas*, a concept previously introduced by Starink et al. [80]. During API trace analysis, we carefully examine the involved memory regions in the virtual address space of the target process, ensuring that different operations occur within the same locations. For instance, if a memory region is allocated in the address range $[\alpha, \alpha + \beta]$, malware must both write its custom code within that interval and initiate execution from there to successfully perform an injection.

Noisy background activity can present challenges as well. Consider the execution flow of a sample from the *sality* malware family, shown in Figure 3. This sample performs numerous operations that could potentially mislead the analysis by generating false positives across several memory regions. In this example, we have already filtered out operations performed on processes other than the actual victim, as including these (without discriminating them by target process identity) could offer further room for false positives. By accurately intersecting the memory ranges involved through interval lookup, our methodology effectively differentiates valid, completed injection attempts during execution. Additionally, our interval analysis is designed to cover multiple processes, enabling the detection of several injection attempts targeting different victims.

While Windows provides several functionally similar APIs for disparate tasks, we observe that the functions responsible for these operations form a finite set. Furthermore, even if attackers devise new strategies over time, these will inevitably revolve around (or can conceptually be mapped to) the three core steps that prior literature established for injection. While implementation limitations or newly identified APIs may make us miss one step, we discussed how our approach mitigates the risks of false negatives by relying on the occurrence of just two of the three steps. This premise reduces the element of surprise for analysts, ensuring that our methodology remains effective in detecting and countering evolving threats.

Temporal correlations. When analyzing execution traces, due to the frequent presence of multiple execution flows in a process, it is essential to consider the time dimension to verify whether a candidate sequence truly represents an injection. Our methodology addresses this by examining the temporal relationships between operations, ensuring they occur in a *logical and coherent order*. We impose constraints on the primitives involved in process injection, recognizing that memory allocation generally precedes writing, which is followed by code execution. Even when these steps overlap or combine, we preserve the integrity of the logical flow by mapping these complexities into corresponding sub-schemes for validation.

Moreover, some injection techniques involve additional operations that may occur out of the canonical order but are helpful for identifying the specific technique. For example, in DLL Injection from Disk, the resolution of the `LoadLibrary` function's address (which serves as the entry point for the execution unit) can happen at various stages, but necessarily before execution. To handle this variability, we modeled constraints focusing on the essential set of operations, allowing for a degree of flexibility in their sequence. This approach effectively captures all potential permutations of API calls within execution traces, accounting for the specific variations of different injection techniques. By doing so, we enhance our ability to detect suspicious behavior while maintaining accuracy in identifying process injection attempts.

3.3 Implementation

To evaluate our approach and gather insights about how threat actors resort to process injection when writing malware, we developed an automated dynamic analysis system for detecting and dissecting process injection behaviors. Detection involves identifying the presence of process injection attempts; then, an accurate classification of the technique(s) used by the authors follows.

A malware sample conducts each step by resorting to one or more Windows API calls, and these events are within the reach of standard means for API interposition in security solutions. As anticipated, we are interested in tracking APIs in use to one or more injection techniques, and in typical variants (e.g., `Ex`) of such APIs.

We implement detection routines by instrumenting the target executable at run-time. Furthermore, since malware often employs evasion techniques to recognize analysis systems, leading to incomplete or misleading investigations [56], we build on off-the-shelf anti-evasion mechanisms [29] to pursue a more thorough analysis.

The architecture of our system comprises two modules (Figure 1):

(1) a dynamic analysis module that executes samples in a controlled environment and captures execution traces;
(2) an analysis module that interprets the dynamic analysis traces, identifying and classifying injection behaviors.

Monitoring module. The approach presented in the past sections can be integrated into any dynamic analysis system that intercepts API calls. For this study, we used *BluePill* [29], an extensible framework for dynamic analysis of possibly evasive malware, implemented atop the dynamic binary instrumentation (DBI) of Pin [55].

We chose BluePill for its ability in neutralizing many known evasion techniques commonly deployed against automated analysis tools. Its ease of customizability allowed us to interpose on Windows APIs associated with different process injection techniques.

We instrumented APIs related to process and memory manipulation for injecting code from one process to another, which, as explained before, are sufficient for detection. Additionally, we included APIs that handle registry keys, window memory, hook procedures, and TxF activity to improve our ability to identify the specific technique in use. In total, we track 74 APIs (101 if we consider `A/W` version distinctions) that are of interest to process injection techniques. In Appendix B, we provide a list of the APIs associated with the three operation primitives, and refer to the implementation for brevity for the others. We added hooks to BluePill to intercept functions at both the library and system call levels, redirecting them to custom procedures that capture call argument values.

We then extended the run-time tracking to cover the hierarchy of relevant processes, from the initial one running the sample to any external processes (whether already running or newly created) into which code is injected at run-time. To this end, we configure Pin

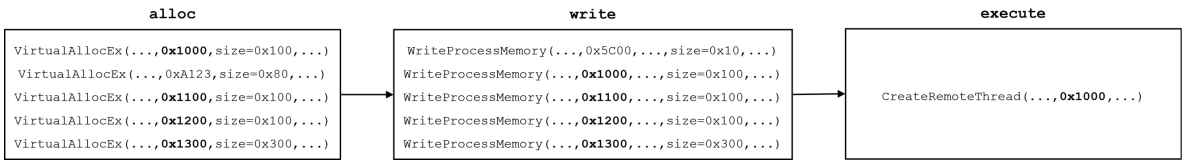


Figure 3: API call trace of a sample with misleading activities on the victim process (for brevity, we stripped operations involving other third processes). The allocation at `0xA123` and the write at `0x5c000` are not congruent by range with any other operation.

(`-follow_execv`) to extend its instrumentation to child processes and remote threads if the monitored sample creates one.

To improve the accuracy of our analysis, we included a mechanism to filter out noise from internal calls [31] within Windows components that do not provide valuable insights. Following the approach in [31], we defined a call as relevant only if it originates from the code under analysis and eventually returns to it. Accordingly, we employed a binary interval tree structure, populated with the code section ranges of legitimate Windows DLLs as they are loaded and unloaded. If the return address of an API call falls within one of these intervals, we discard the information as irrelevant.

We acknowledge a residual risk of false negatives from filtering internal calls in two cases. In one, the attacker hijacks the control-flow within a Windows DLL API to malware code: such an attack may be highly constrained in practice, as it requires knowing the DLL version in use at the victim, a vulnerability in it, and possibly bypassing the Control Flow Guard mitigation of Windows DLLs. In the other, the attacker sets as return address a gadget in DLL code that returns to malware code: this can be countered by verifying the legitimacy of the return address at API call or return time.

Albeit far from perfect in terms of transparency, DBI techniques offers a sweet spot for implementing dynamic analysis tools, providing fine-grained analysis of malware behaviors while enabling implementation opportunities for incorporating anti-evasion strategies to counter known evasions. Other recent works in malware analysis have similarly employed DBI [19, 28], even for large-scale studies of evasive malware. As mentioned before, attempting to address evasive behavior is necessary to mitigate the risks of missing injection behaviors from samples that, for example, exhibit benign behavior once they recognize an analysis environment [19, 56].

To estimate the impact of evasions, we implemented the same analysis procedures on top of Microsoft Detours [25], a library for dynamically intercepting arbitrary Win32 binary functions and adding instrumentation. While Detours offers a lightweight and reliable tracing mechanism, it uses conspicuous trampolines for API hooking and, unlike BluePill, does not come with provisions for evasions targeting virtual environments or delays from the analysis. These characteristics make this baseline implementation in Detours a helpful tool for ablation studies on the impact of evasions.

Analysis module. This component processes the data collected by the monitoring module to identify specific process injection techniques and classify malicious attempts. We group APIs in equivalence classes based on their operation types, abstracting implementation differences from APIs able to complete a given task. For example, memory may be written to by using a byte copying API, mapping a section to memory, writing to the Atom table, or asking Windows to explicitly load a module. Appendix B lists APIs in the equivalence classes for the three core operational primitives.

We then proceed with correlating memory allocation, writing, and execution operations in the call trace. Our matching mechanism employs *interval lookup* operations to check the congruence of memory ranges among operations. For correlation, we consider the identity of the victim process (using its process identifier) and the referenced objects. We further verify if all core operations (i.e., allocation, writing, execution) occur on coherent addresses, as partial matches could be false positives[2]. As for referenced objects, whenever an operation involves handle values, we treat them as an opaque entity already in the monitoring module, looking up and mapping them to the real object identifier (whereas Starink et al. [80] attempt to match handle values from logs). This choice removes potential ambiguity from handle reuse or similar effects.

Next, by comparing temporal relationships between these actions through their recorded timestamps, we accurately detect the sequence of injection activities. If a valid sequence of at least two core steps is found, we run another analysis of the full API call trace to pinpoint the specific technique(s) in use. This analysis looks for distinctive APIs associated with a given technique: for example, with Process Hollowing, the creation of a suspended process.

4 Experiments Setup

This section details the dataset construction process for gathering insights on on how threat actors use injection techniques and the setup of the analysis environment for running our experiments.

4.1 Dataset

To build our dataset, we sought samples from two sources: the VirusTotal Academic dataset [83] made available to researchers and the Vx Underground Bazaar collection [26]. The VirusTotal Academic dataset features samples observed by the vendor between 2017 and 2021 and is a popular choice in recent academic works. However, the vendor stopped updating it after 2021, as confirmed through private communication. To extend our study with more recent years, we turned to the Bazaar collection, a publicly accessible repository that is well-regarded among industry practitioners.

Our initial malware dataset consisted of 334,885 PE executables. To create a more cohesive and meaningful selection, we first sorted the samples by their first submission date on VirusTotal and discarded Bazaar samples flagged as malicious by less than 15 AV vendors, following the criterion VirusTotal used for their dataset.

[2]For example, the analyzer of [80] checks for matching memory addresses only for allocation and write operations for Process Hollowing and Thread Execution Hijacking. However, the semantic validity of execution is not guaranteed unless the entry point is coherently updated in the thread's context.

We then deduplicated the samples using their *vhash*, an in-house static fingerprint provided by VirusTotal, to cluster a representative subset from the extensive original dataset. Appendix C discusses how this choices comes with almost no semantic loss for our study and removes potential bias from overly represented samples. Moreover, to avoid a similar problem with families, we ensured that no single malware family occurred for more than 5% of the sample selected for each year, using AVClass [76] to obtain plausible malware family labels. The final dataset accounts for 56,340 samples.

4.2 Analysis Environment

We deployed our analysis framework on 5 VirtualBox virtual machines running on a local server with an Intel Xeon E5-2620 v3 CPU @ 2.40 GHz with 24 cores and 64GB RAM. Each VM had 2 GB of RAM and 4 logic CPUs, running Microsoft Windows 7 for better compatibility with malware from different years.

To resemble a legitimate target, we followed best practices by introducing wear-and-tear environmental artifacts typical on real systems, such as documents, history, and installed programs [58]. Additionally, we included a simulator for common internet services to enable network traffic for malware samples without having to grant them internet access. Our analysis manager controls each VM: specifically, it spawns the VM, send the sample to the in-guest agent responsible for analysis, and collects the report. After each execution, we restore the virtual machine to a clean state for the next analysis using a live snapshot. All these provisions concretely mitigate the risks of harm from the experimentation.

During preliminary experiments, we tested 26,391 samples (46.8% of the size of the final dataset) for up to 8 minutes. We observed that the first injection attempt occurred within 1.5 minutes for 95.2% of the samples and within 2 minutes for all but 215 samples. These values are coherent with a recent study [52] that shows how most malware behaviors are observed during the first two minutes of execution, with little additional information gained from longer runs. Also, due to the purpose of process injection, we expect it to happen in the early stages of the execution, likely after any potential evasive activity ends. We opted to chose for a 4-minute execution window for conducting the full study, allowing for a larger budget to mitigate residual concerns for DBI overheads.

5 Experimental Findings

In this section, we present the results of our experimental analysis, aiming to answer the following research questions:

RQ1: What are the most common process injection techniques adopted by modern malware? Does the release of information about these techniques influence their adoption?

RQ2: Is there a correlation between malware families and the injection techniques they exploit? Have the injection techniques adopted by each family evolved over time?

RQ3: May the methodology see downstream uses in security solutions, and if so with what performance (e.g., false positives)?

5.1 Adoption of Process Injection Techniques

General Trends. We observed injection attempts in 16.5% of the samples (9,314) executed under our DBI-based monitoring system. This percentage fluctuates between 10.7% to 29.1% across different years (Table 1). As mentioned, our samples come from two collections: VirusTotal (2017 to 2021) and Bazaar (2022 to 2023). As we noted appreciable differences in the frequency of observed injection attempts between the two collections, we tested additional[3] 8,241 (balanced) samples from the Bazaar collection available for 2021, so as to have a common temporal ground for comparison. Also these 2021 Bazaar samples were generally more "active" in terms of conspicuous activities, including injection attempts (31.6%).

Table 1: Prevalence of injection in the balanced dataset.

Year	Samples	Injection
2017	5,859	1,321 (22.6%)
2018	14,650	1,800 (12.3%)
2019	9,978	1,181 (11.8%)
2020	5,321	567 (10.7%)
2021	5,042	553 (11.0%)
2022	8,067	2,348 (29.1%)
2023	7,423	1,544 (20.8%)
Total	56,340	9,314 (**16.5%**)

After careful analysis, we concluded that the difference most likely stems from the different construction methods of the two datasets. The VirusTotal Academic dataset is a subset of samples submitted to the VirusTotal portal over an unspecified number of months within the reference year (with one or two collections available per year) and that alerted at least 15 antivirus products. Many of these samples may be second-stage payloads that by nature do not need to resort to process injection, suspicious files that may not be malware, or infected installer programs. On the contrary, the Bazaar collection features on a monthly basis samples collected and reported as malicious by professionals. These different characteristics make both clearly valuable for general malware studies, and we believe they are behind the higher frequency of intercepted injection attempts in the Bazaar samples.

Another interesting aspect relates to malware evasion. Recent longitudinal studies on this topic [42, 56] suggest a subtle but steady increase in the prevalence of evasive behaviors over the years. To assess the impact of these behaviors on our analysis, we tested our approach using its baseline implementation in Microsoft Detours on the balanced collection dating back to 2019, as it is the most recent year shared between our collection and the two studies cited above. Our findings revealed that only 6.2% of the samples exhibited their injecting nature compared to 11.8% detected by the DBI-based implementation. This discrepancy highlights the significant influence of evasive techniques on malware analysis, suggesting that the actual number of samples doing process injection could be higher (i.e., several samples likely evaded also BluePill), and thereby confirming the limitations of insights from prior injection-related literature.

Our results largely differ from what Starink et al. observed in [80]. The authors measured an occurrence of injection attempts for 8.3% of the tested samples in 2017, 13.4% in 2018, 13.6% in 2019, 5.1% in 2020, and 6.1% in 2021. We explain this with multiple factors:

- The authors did not balance the VirusTotal Academic dataset (which was their only dataset), leading to over-representation of certain families in their statistics;
- Their system does not come with anti-evasion provisions, a limitation the authors acknowledge and partially mitigate by

[3] These samples do not concur with the statistics presented in the rest of the paper.

Table 2: Distribution of process injection techniques in the balanced malware dataset from 2017 to 2023.

Injection Technique	2017	2018	2019	2020	2021	2022	2023	Total
Shellcode Injection	74 (5.6%)	453 (25.2%)	200 (17.0%)	75 (13.2%)	61 (11.0%)	25 (1.1%)	5 (0.3%)	893 (9.6%)
PE Injection	61 (4.6%)	357 (19.8%)	246 (20.8%)	191 (33.7%)	47 (8.5%)	172 (7.3%)	51 (3.3%)	1,125 (12.1%)
DLL Injection from Disk	3 (0.2%)	3 (0.2%)	18 (1.5%)	0 (0.0%)	3 (0.5%)	5 (0.2%)	1 (0.1%)	33 (0.4%)
APC Injection	0 (0.0%)	1 (0.1%)	0 (0.0%)	4 (0.7%)	1 (0.2%)	0 (0.0%)	1 (0.1%)	7 (0.1%)
Process Hollowing	919 (69.6%)	880 (48.9%)	414 (35.1%)	148 (26.1%)	152 (27.5%)	2,104 (89.6%)	1,482 (96.0%)	6,099 (65.5%)
Thread Execution Hijacking	9 (0.7%)	41 (2.3%)	10 (0.9%)	3 (0.5%)	58 (10.5%)	2 (0.1%)	0 (0.0%)	123 (1.3%)
Early Bird	161 (12.2%)	0 (0.0%)	1 (0.1%)	0 (0.0%)	1 (0.2%)	0 (0.0%)	1 (0.1%)	164 (1.8%)
Hook Injection	77 (5.8%)	83 (4.6%)	137 (11.6%)	28 (5.0%)	33 (6.0%)	19 (0.8%)	4 (0.3%)	381 (4.1%)
Atom Bombing	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
Extra Window Memory Injection	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Process Doppelgänging	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
Passive Injection	35 (2.7%)	91 (5.1%)	237 (20.1%)	151 (26.6%)	224 (40.5%)	45 (1.9%)	9 (0.6%)	792 (8.5%)

deploying their system atop a virtual machine introspection-based sandbox (which remains vulnerable, e.g., to timing attacks to detect VMs or stall analyzers);
- Our methodology is more general and captures more injection patterns, as with the *no-unmap* variant of Process Hollowing, which occurs in a significant fraction of samples exhibiting injection activity.

Due to the generally higher percentages (and the more techniques) we observe, we tend to believe that our study captures better the varied landscape of process injection. While the attentive reader would argue that the two datasets are different, we applied our balancing method (Section 4.1, Appendix C) on their results, measuring for our study a 95.6% coverage of their dataset, and 64.1% more samples exhibiting injection behaviors than Starink et al. do.

Before moving to the analysis of techniques and trends, we report on the efficacy of the choice of identifying an injection based on the observation of just two related primitive events (Section 3.2). For 19.9% of the samples for which we detected an injection, we find only two of the three primitive events in the execution trace. Within individual techniques, we observe appreciably higher percentages for Process Hollowing (36.1%) and PE Injection (29.9%), which we relate to the technical evolution trends that we discuss next.

Techniques Distribution and Evolution. We measured the occurrence of process injection techniques recognized by our method over the years, with statistics provided in Table 2. Many techniques were observed throughout the entire study period, suggesting (as expected) that their first appearance may predate our analysis timeframe. Analogously, techniques detected near the end of our dataset (December 2023) are likely to remain in use in the future, indicating that malware authors will continue to leverage effective methods while phasing out those that become obsolete for efficacy.

Our data revealed two patterns: *certain techniques are markedly more prevalent* than others, and many of these techniques *tend to cluster within specific time periods.*

The most frequently exploited techniques are Process Hollowing (65.5%), PE Injection (12.1%), and Shellcode Injection (9.6%). Process Hollowing stands out due to its effectiveness in achieving stealth with relative simplicity, making it a preferred choice for malware writers. This shows also in how it overcomes in numbers all other techniques combined in 2017 and 2022, and even more so in 2023.

During our analysis, we uncovered not only longitudinal trends but also significant vertical evolution within specific techniques.

A notable example is Process Hollowing, where our approach, focusing on essential operations, exposed implementation variants that were overlooked by prior studies. In the traditional workflow, Process Hollowing unmaps memory from the target process, a step that gives the technique its very name. However, we observed a shift in attacker behavior: instead of hollowing out the original executable, attackers modify the victim's address space directly. That is, they allocate a new memory region at a system-assigned address and write the malicious code there, leaving the original memory unmodified. This *no-unmap* variant, though occasionally mentioned in security blogs in 2016 [1] and 2022 [22], shows in our tests as significantly used as early as 2017 (36.0%), and by the following year it had surpassed the classic version in prevalence.

Our interpretation is that authors now prefer to feature relocation machinery to accommodate the payload within the remote addresses assigned randomly by Windows, rather than requesting the allocation at a fixed based address once the address space has been hollowed out. While relocation itself is not technically challenging, it significantly enhances stealth. In more detail, this no-unmap variant avoids conspicuous API calls like `NtUnmapViewOfSection`, and leaves fewer detectable memory artifacts, making it harder for security solutions to identify anomalies when checking the integrity of loaded code modules. This evolution highlights why Process Hollowing remains popular among malware authors—its versatility and adaptability have given rise to new variants, complicating detection and analysis efforts for security professionals.

While Process Hollowing occurrences have increased over the years, we observed other techniques less frequently. We underline that the choice of injecting more structured payloads (as it is the case with Process Hollowing) is also reflected by the prevalence of PE injection over simpler shellcode, a shift we believe possibly influenced by the abundance of white papers detailing capable and advanced injection techniques [45, 62, 69].

Most techniques exhibit peak usage within specific, brief periods, likely driven by active malware campaigns where cybercriminals test new tactics. For example, the Early Bird technique saw significant use in 2017, accounting for 12.0% of the samples, but sharply disappeared in subsequent years. Various sources indicate that Early Bird gained interest around 2018, as shown by the availability of security blog posts [27, 65, 73]. This suggests attackers may have leveraged this technique in earlier campaigns when it was novel and harder to detect. As it became more widely known, vendors

Table 3: Top 20 families in the dataset. # indicates the number of samples, Date refers to the first appearance (in the dataset) of a sample from that family, First Injection refers to the first appearance of a sample from that family with recorded injection activity, # Injection is the number of samples employing process injection, # Techniques represents the maximum number of techniques a sample from that family used.

Family	#	Date	First Injection	# Injection	# Tech.
virlock	1,657	2017	2018	25 (1.5%)	2
upatre	1,288	2017	2017	10 (0.8%)	1
virut	1,278	2017	2017	84 (6.6%)	2
sality	1,203	2017	2017	436 (36.2%)	4
vobfus	901	2017	2017	45 (5.0%)	1
bladabindi	771	2017	2017	34 (4.4%)	2
dealply	757	2017	-	0 (0.0%)	0
reline	723	2021	2022	718 (99.3%)	1
stop	722	2021	2021	466 (64.5%)	1
ramnit	649	2017	2017	329 (50.7%)	2
mokes	641	2019	2019	152 (23.7%)	1
allaple	603	2017	-	0 (0.0%)	0
zard	600	2018	2022	75 (12.5%)	1
ursu	570	2017	2017	49 (8.6%)	2
sfone	513	2017	-	0 (0.0%)	0
gamarue	472	2017	2017	166 (35.2%)	3
zbot	471	2017	2017	133 (28.2%)	1
tofsee	470	2017	2017	30 (6.4%)	1
strab	470	2022	2022	148 (31.5%)	1
gandcrab	469	2018	2018	25 (5.3%)	2

likely developed patches, leading to its reduced usage. This dynamic may also explain the near disappearance of APC Injection, which, as an early technique exploiting asynchronous procedure calls in victim threads, was well-known and easily detected by AV tools since its identification in 2018 or earlier [47].

Passive injection techniques that manipulate system registry keys showed their peak in activity between 2019 and 2021. This cluster aligns with the findings of Starink et al. [80], although they noted a rising adoption rate. This period also coincided with the COVID-19 pandemic, during which the number of cyberattacks and threats grew exponentially, prompting attackers to explore alternative methods for executing malicious code [44, 57].

Another textbook technique, DLL injection from Disk, appears less popular among malware authors, likely because modern security systems can easily detect this approach. However, the loading of a malicious DLL can still be accomplished in other ways, as with Reflective DLL injection (we model it under PE Injection). Analogously, we observed a small percentage of samples performing Hook Injection, suggesting a shift toward strategies like fileless techniques that avoid obvious artifacts that may trigger AV alerts.

Finally, we noticed that techniques like Atom Bombing and Process Doppelgänging appeared rarely, with instances recorded only once in 2018. Similarly, we found no cases of Extra Window Memory Injection. While practitioners have published PoC implementations of these techniques [24, 43, 71], our tests confirmed they can be exploited in real-world attacks and detected using our methodology.

Frequency of Use and Target Processes. Another interesting metric is the number of injection techniques employed by each sample. The percentages for single years shown in Table 2 do not sum to 100% because some samples use multiple techniques. Specifically, 96.8% of samples employed a single technique, 3.0% used two, and the remaining 0.2% utilized up to three different techniques. When multiple techniques are used, authors often diversify choices.

Malware may then attempt to inject multiple processes before succeeding. Our observations reveal that 85.2% of samples executed a single injection sequence, 5.1% attempted two, 1.2% carried out three, and the remaining cases involved four or more, with some failure cases looping over all running processes. These patterns suggest that malware authors typically prefer to rely on a single injection technique, targeting a limited set of processes to minimize system disruption and maximize the chances of a successful attack.

The most frequently targeted processes are those with high trust levels or commonly executed by the system. For simpler techniques like Shellcode and PE injection, 83.2% of attacks target existing processes rather than creating new ones. The most common targets picked in our tests are *svchost.exe* (48.3%) and *explorer.exe* (33.2%), with other popular user applications like *notepad.exe* also targeted in smaller percentages. With these injection techniques, we observe no particular victim preference when the sample starts it.

Process Hollowing sees instead by construction the creation of a victim process in suspended state: therefore, identifying its typical targets can offer interesting insights. In 28.6% of cases, the most commonly targeted processes are native ones, including *svchost.exe* and *taskhost.exe* that run in the background (and in multiple instances with *svchost.exe*) of Windows installations.

More interestingly, in 45.3% of cases, malware spawns a copy of itself in suspended state, solely for injection. We ran this finding with a senior malware analyst and a plausible explanation is it being a flavor of defense evasion. Specifically, the code the sample injects may evade the scrutiny of the AV/EDR's code and run-time analyses (unlike if running such code itself), and writing to the memory of a child process is easier than with an unrelated victim because the sample already holds a handle to it (and opening a process with memory write privileges is a conspicuous action).

RQ1. Process injection is a relevant phenomenon for modern malware and shows quite a dynamic nature. In our analysis, 16.5% of the samples employed at least one injection technique, with Process Hollowing and PE Injection as the most common choices. Malware authors tend to adopt techniques in clustered periods, possibly influenced by the latest findings within the community.

5.2 Families and Process Injection Techniques

As a next investigation, we looked for correlations between malware families and their use of process injection to gain insights into how these behaviors may have evolved over time.

Prevalence of Techniques in Families. Our dataset contains samples from 2,667 malware families, with 576 (21.6%) containing at least one sample that adopts an injection technique. The dataset includes also 7,219 samples (12.8%) not classified under any family by AVClass, hence we leave them out from the analysis below.

Table 3 provides details on process injection across the top 20 most prevalent families. Notably, the year a family first appears in the dataset does not always coincide with when it started using injection. For example, the *virlock*, the *reline*, and the *zard* families initially did not employ injection but later adopted it as part of their strategies. Additionally, the more numerous families (column #)

Table 4: Injection techniques observed in the top families ranked by size (SC: Shellcode Injection, PE: PE Injection, PH: Process Hollowing, HK: Hook Injection, APC: APC Injection, TH: Thread Execution Hijacking, EB: Early Bird, P: Passive Injection, -: No samples available for that year).

Family	2017	2018	2019	2020	2021	2022	2023
virlock		PE	PE,HK	PE	PE	PE	PE,PH
upatre	PH	PH,P	PH			PE,PH	
virut	SC,PH,P	SC,PE,PH,HK	SC		PH		
sality	SC,PE	SC,PE,PH,P	SC,PE,PH	SC,PE	SC,PE	SC,PE,HK,P	SC,PH
vobfus	PH	PH	SC,PE,PH	PH	PH	PH	
bladabindi	PH,HK	DLL,PE,PH	PH	PH,HK		PH,HK	PH
reline	-	-	-	-		SC,PE,PH	PH
stop	-	-	-	-	PH	PE,PH	PE,PH
ramnit	PE,PH	PE,PH,APC	SC,PE,PH,HK	PE,APC	PE	PE	-
mokes	-	-	PH			SC,PH	PH
zard	-					PE,PH	PE,PH
ursu	PH,EB	SC,PH,TH	PH	PH	PH		-
gamarue	DLL,PE,PH	PE,PH	PH	SC,PE,PH	PE,PH	SC,PE,PH	SC,PH
zbot	SC,PE,PH,EB	SC,PE,PH	PE,PH	PE,PH		SC,PE,PH	PH
tofsee	PH	PH				PE,PH	PH
strab	-	-	-	-	-	PE,PH	PH
gandcrab	-	SC,PH,TH					

are not those with the highest frequency of injection activity. The *reline* family, designed to steal user account information, stands out with 99.3% of its samples using injection, followed by the *stop* family at 64.5%. However, this high usage rate does not necessarily pair with the diversity of techniques employed. For instance, while both the *ramnit* and *bladabindi* families utilize up to two different techniques, their injection activity rates vary significantly.

Usage Consistency. We then investigated whether malware families make consistent use of process injection over time.

Among families with multiple samples exhibiting injection activity, only 39 out of 482 (8.1%) consistently used injection across all samples. In most cases, the rate of injection usage varies significantly within a family, suggesting that if one sample performs an injection technique, it is not so likely that all samples in that family will. This seems to contrast with the results of Starink et al. [80].

When examining malware families that perform injection over multiple years, we found that only 20.6% of families consistently manifest injection behavior each year they appear in the dataset. We believe this indicates that malware families often change their attack strategies over time in response to evolving defenses. Table 4 details how the behaviors of the most prevalent malware families from the dataset have evolved in recent years. For instance, the *tofsee* family employed injection in its first two years in the dataset, ceased using it for a period, and later resumed this technique.

Selection Consistency. Given that samples from the same malware family often exhibit similar malicious behavioral traits, a hypothesis worth exploring was that the set of injection techniques used may correlate with specific families. To test this hypothesis, we evaluated the correlation of injection techniques employed by samples within each family, focusing on those with recorded injection activity over at least two years. We excluded single-year families to reduce noise and enhance pattern identification.

We calculated the intersection of injection techniques across samples within each family. Our analysis showed that for 138 of 251 (55.0%) families exhibiting injection behavior over multiple years, at least one technique remained consistent in all samples. However, the remaining families manifested different trends over time, either shifting between affine techniques (e.g., PE injection and Process Hollowing) or adopting completely different ones. Also this finding seems to support our hypothesis that, as security defenses evolve, malware families may adapt their injection techniques to achieve their objectives and maintain effectiveness.

Finally, Table 4 shows how families that initially used PE Injection later transitioned to Process Hollowing. This shift likely occurred because Process Hollowing, evolving over time, may have been perceived as a more robust variant of the simpler techniques, especially for injecting whole malicious executables. As pointed out already by Starink et al. [80], the choice of injection technique alone appears insufficient for identifying a malware family.

RQ2. Within a malware family that uses process injection, it is very likely to encounter both samples using injection techniques and samples that (in our executions) do not. A significant number of malware families consistently use the same injection techniques, while others adapt to current trends. This variability suggests that relying solely on the choice of specific techniques is insufficient for accurately identifying malware families.

5.3 Feasibility of Downstream Uses

The experiments discussed until now focused on the value of our methodology for gaining deeper understanding of the dynamics of process injection, as in a longitudinal study fashion. An orthogonal dimension that we evaluate next concerns its potential downstream uses: specifically, if it may be integrated in existing solutions to more effectively counter and respond to malware techniques.

The first aspect we examine is whether our methodology can incur false positives, which is of paramount importance with non-malicious software in downstream uses. Due to space limitations, we describe in Appendix D the approach and results for the study, and report here the main insight. With the notable exception of the process management strategies of modern browsers (which also anti-virus products need to recognize and allow), our tests suggest false positives are unlikely with goodware. This is because injection activities involve specific interactions with memory regions, and legitimate software is unlikely to perform such activities in the ways that malware does. Therefore, we find that false positives would not be a hurdle for downstream integration of our method.

We thus move to a case study for the feasibility of integration. We explore whether our method could improve the results of popular closed and open-source tools already with provisions for process injection, comparing the results from our main study with theirs.

For closed solutions, we relied on the VirusTotal Academic API to access behavioral reports from sandboxes and other malware scanners available on the platform. At the time of the writing, VirusTotal did not provide behavioral analysis for 2.8% of the samples, which we therefore excluded from the comparison. We note that 2,172 samples detected by our approach (24.0%) were not tagged for injection activity by commercial tools. Notably, 29.4% of these samples were flagged by our method by witnessing only two core operations (i.e., not all three) in the execution traces. We find this a strong indicator of the potential practical value of our methodology.

For open-source solutions, we evaluated reports from the Config And Payload Extraction (CAPE) sandbox [74], which derives from Cuckoo [75] and uses the *malfind* [41] Volatility plugin to detect

injected code and several anti-evasion provisions. While VirusTotal sometimes identifies specific techniques based on the MITRE ATT&CK framework, CAPE primarily focuses on detailed memory-related insights during execution. Despite malfind aids in detecting suspicious memory regions, it can still miss injection instances. After excluding 59 samples apparently too large for CAPE to handle, 1,694 samples (18.3%) flagged by our approach were not recognized as injection cases by CAPE, and for 13.7% of these samples our system witnessed only two core actions in the execution trace.

These results expose gaps in publicly available sandboxes and tools that are well-regarded among practitioners, with an average of 21.2% samples doing injection activities that result into false negatives. We believe integrating our analysis techniques into these systems would be straightforward, as many already hook a significant fraction of the APIs required for our monitoring. Moving forward, both existing tools and future research efforts may want to consider deconstructing the injection process into smaller, more granular components, as proposed in our methodology. This approach could help defense architects reduce false negatives, improve detection of malicious activities, and contribute to the development of more robust malware analysis systems for the security community.

To conclude the investigation, we also estimate the run-time overhead our approach adds atop a monitoring solution, which would be particularly relevant for integrating our method in security products for online detection (e.g., AVs and EDRs). We reproduce the experimental setup of recent literature on API call tracing in security applications [31], opting for a worst-case analysis scenario on a collection of call-intensive benchmarks. We use the Wine conformance tests and the workloads used in [31] for 12 popular DLLs, as they exercise in their code a large deal of APIs relevant to both goodware and malware, including operations used in process injection machinery. For measuring the overhead specific to our analysis, we compared the running time of our system against a baseline DBI tool configured with the same hooks but without analysis code for the calls. The baseline tool embodies the operation of a security solution that witnesses the invocation of the APIs of interest (these are relevant for general malware detection tasks and must be tracked anyway) without processing them. This comparison ensured that any measured overhead was solely attributable to the added instrumentation logic for detecting injection. The outcomes of the tests show an average overhead of 1.35% (Appendix E), which speaks in favor of the practicality of our methodology and may also be reduced through an optimized implementation.

RQ3. Our methodology may enhance process injection detection in existing malware analysis solutions, which missed 21.2% of the samples identified in our tests. The community should prioritize developing new, principled approaches to better understand instances of process injection in real-world malware.

6 Discussion and Concluding Remarks

In this paper, we explored the elusive nature of process injection in Windows malware, drawing key insights into its adoption over seven years (2017-2023). In spite of the variety of process injection techniques and implementation alternatives within techniques, our approach was able to analyze them by focusing on the minimal set of core operations sufficient to detect an injection attack.

From a longitudinal perspective, our findings indicate that process injection remains an impactful and favored attack vector for adversaries to bypass defenses. We observed a trend towards increasingly evolved, stealth-enhancing methods. This evolution underscores the need for future detection solutions to capture the fundamental properties of process injection in order to be able to quickly adapt to new developments. Moreover, our results suggest that threat actor strategies may have shifted in reaction to what the white-hat community discovered about the attack vectors in use.

The continued popularity of process injection likely stems from its proven effectiveness, despite being a well-known problem in the security community for over two decades. As the technique exploits legitimate operations made available for benign programs, security vendors have to exercise caution when blocking these operations due to the risks of interfering with and possibly breaking normal system functionality. However, in our goodware experiments, we have seen how our approach is not keen to false positives, as it deems such operations relevant for injection only under specific operating conditions (by spatial and temporal congruence).

The statistics from our study can serve only as a lower bound to estimate the prevalence of process injection in the wild. A common challenge in malware studies is the lack of an ideal analysis environment that is fully transparent and able to cope with other environmental dependencies of malware, such as trigger-based behavior (e.g., user interaction) or targeting (e.g., specific programs or files expected at the victim's). Our comparative findings between the BluePill and Detours-based implementations show that incorporating off-the-shelf anti-evasion measures can expose significantly more injection activity. We remark that one advantage of our analysis method is its simplicity, which makes it amenable to straightforward integration in other monitoring technologies.

Our findings revealed more activity compared to prior studies. By moving away from strict patterns and implementation-specific rules, we were able to capture more implementation variants of each technique. In the case of Process Hollowing, the most frequent technique in our observations, we underline that nearly half of the occurrences relate to the *no-unmap* variant that prior work missed. This improvement overcomes a key limitation of the methodology behind the study of Starink et al. [80]. Also, our work reveals insights on the evolution of techniques and the apparent choices of threat actors that were not available in any prior studies.

Finally, our methodology may be beneficial for improving malware detection and analysis systems. Our evaluation shows that, despite relying on resource-intensive techniques to detect and dump injected payloads, popular sandboxes missed about one fifth of the samples for which our lightweight approach succeeded in detection.

Hoping to stimulate further research and new studies, we release the source code, dataset, and experimental materials from this paper.

Acknowledgments

We thank our anonymous reviewers for their feedback, VirusTotal for providing access to their Academic malware dataset, and Nicola Bottura for helping with the experiments. This work has been partially supported by projects SERICS (PE00000014) and Rome Technopole (ECS00000024) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Monnappa K A. 2016. Detecting Deceptive Process Hollowing Techniques Using Hollowfind Volatility Plugin. https://cysinfo.com/detecting-deceptive-hollowing-techniques/ (Accessed: January 4, 2025).

[2] Monnappa K A. 2024. HollowFind. https://github.com/monnappa22/HollowFind (Accessed: December 12, 2024).

[3] Cristian Assaiante, Simone Nicchi, Daniele Cono D'Elia, and Leonardo Querzoni. 2024. Evading Userland API Hooking, Again: Novel Attacks and a Principled Defense Method. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 150–173. https://doi.org/10.1007/978-3-031-64171-8_8

[4] MITRE ATT&CK. 2020. Event Triggered Execution. https://attack.mitre.org/techniques/T1546/ (Accessed: December 12, 2024).

[5] MITRE ATT&CK. 2020. Event Triggered Execution: AppCert DLLs. https://attack.mitre.org/techniques/T1546/009/ (Accessed: January 10, 2025).

[6] MITRE ATT&CK. 2020. Event Triggered Execution: AppInit DLLs. https://attack.mitre.org/techniques/T1546/010/ (Accessed: January 10, 2025).

[7] MITRE ATT&CK. 2020. Event Triggered Execution: Application Shimming. https://attack.mitre.org/techniques/T1546/011/ (Accessed: January 10, 2025).

[8] MITRE ATT&CK. 2020. Event Triggered Execution: Component Object Model Hijacking. https://attack.mitre.org/techniques/T1546/015/ (Accessed: January 10, 2025).

[9] MITRE ATT&CK. 2020. Event Triggered Execution: Image File Execution Options Injection. https://attack.mitre.org/techniques/T1546/012/ (Accessed: January 10, 2025).

[10] MITRE ATT&CK. 2020. Process Injection: Asynchronous Procedure Call. https://attack.mitre.org/techniques/T1055/004/ (Accessed: January 10, 2025).

[11] MITRE ATT&CK. 2020. Process Injection: Dynamic-link Library Injection. https://attack.mitre.org/techniques/T1055/001/ (Accessed: January 10, 2025).

[12] MITRE ATT&CK. 2020. Process Injection: Extra Window Memory Injection. https://attack.mitre.org/techniques/T1055/011/ (Accessed: January 10, 2025).

[13] MITRE ATT&CK. 2020. Process Injection: Portable Executable Injection. https://attack.mitre.org/techniques/T1055/002/ (Accessed: January 10, 2025).

[14] MITRE ATT&CK. 2020. Process Injection: Process Doppelgänging. https://attack.mitre.org/techniques/T1055/013/ (Accessed: January 10, 2025).

[15] MITRE ATT&CK. 2020. Process Injection: Process Hollowing. https://attack.mitre.org/techniques/T1055/012/ (Accessed: January 10, 2025).

[16] MITRE ATT&CK. 2020. Process Injection: Thread Execution Hijacking. https://attack.mitre.org/techniques/T1055/003/ (Accessed: January 10, 2025).

[17] MITRE ATT&CK. 2021. ListPlanting. https://attack.mitre.org/techniques/T1055/015/ (Accessed: January 10, 2025).

[18] MITRE ATT&CK. 2024. Process Injection. https://attack.mitre.org/techniques/T1055/ (Accessed: December 12, 2024).

[19] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *17th Annual Network and Distributed System Security Symposium (NDSS 2010)*.

[20] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. 2017. Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 209–229. https://doi.org/10.1007/978-3-319-60876-1_10

[21] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. 2014. Bee Master: Detecting Host-Based Code Injection Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 235–254. https://doi.org/10.1007/978-3-319-08509-8_13

[22] Frank Block. 2022. Some Experiments With Process Hollowing. https://insinuator.net/2022/09/some-experiments-with-process-hollowing/ (Accessed: January 4, 2025).

[23] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. 1962–1970. doi:10.1145/3297280.3297472

[24] BreakingMalwareResearch. 2016. atom-bombing. https://github.com/BreakingMalwareResearch/atom-bombing (Accessed: January 15, 2025).

[25] Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium*.

[26] Vx Underground Bazaar Collection. 2024. https://vx-underground.org/Samples/Bazaar%20Collection (Accessed: December 13, 2024).

[27] cyberbit. 2018. New 'Early Bird' Code Injection Technique Discovered. https://www.cyberbit.com/endpoint-security/new-early-bird-code-injection-technique-discovered/ (Accessed: January 4, 2025).

[28] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. 15–27. doi:10.1145/3321705.3329819

[29] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. 2020. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2750–2765. doi:10.1109/TIFS.2020.2976559

[30] Daniele Cono D'Elia, Lorenzo Invidia, and Leonardo Querzoni. 2021. Rope: Covert Multi-process Malware Execution with Return-Oriented Programming. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS 2021)*. 197–217. doi:10.1007/978-3-030-88418-5_10

[31] Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro. 2021. Designing Robust API Monitoring Solutions. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2021), 392–406. doi:10.1109/TDSC.2021.3133729

[32] Microsoft Windows Documentation. 2021. About Transactional NTFS. https://learn.microsoft.com/en-us/windows/win32/fileio/about-transactional-ntfs (Accessed: January 10, 2025).

[33] Microsoft Windows Documentation. 2021. Asynchronous Procedure Calls. https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls?redirectedfrom=MSDN (Accessed: December 13, 2024).

[34] Microsoft Windows Documentation. 2021. Section Objects and Views. https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views (Accessed: December 13, 2024).

[35] Microsoft Windows Documentation. 2024. Event Tracing for Windows (ETW). https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw- (Accessed: December 14, 2024).

[36] VirusTotal Documentation. 2024. Files. https://docs.virustotal.com/reference/files (Accessed: December 23, 2024).

[37] Elastic. 2024. detection-rules. https://github.com/elastic/detection-rules (Accessed: December 12, 2024).

[38] Csaba Fitzl. 2024. injection. https://github.com/theevilbit/injection (Accessed: December 12, 2024).

[39] Chocolatey: The Package Manager for Windows. 2024. https://chocolatey.org/ (Accessed: December 31, 2024).

[40] The Volatility Foundation. 2024. https://volatilityfoundation.org/ (Accessed: December 12, 2024).

[41] Volatility Foundation. 2024. malfind. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#malfind (Accessed: December 12, 2024).

[42] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. 2022. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & security* 113 (2022), 102550. https://www.sciencedirect.com/science/article/pii/S0167404821003746

[43] Hasherezade. 2017. process_doppelganging. https://github.com/hasherezade/process_doppelganging (Accessed: January 15, 2025).

[44] Ashkan Hosseini. 2017. Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques. https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process (Accessed: December 12, 2024).

[45] Marcus Hutchins. 2013. Portable Executable Injection For Beginners. https://www.malwaretech.com/2013/11/portable-executable-injection-for.html (Accessed: January 4, 2025).

[46] AV-TEST: The Independent IT-Security Institute. 2024. Malware Statistics. https://www.av-test.org/en/statistics/malware/ (Accessed: December 12, 2024).

[47] Mossé Cyber Security Institute. 2022. Malware Injection Techniques: APC Injection. https://library.mosse-institute.com/articles/2022/05/malware-injection-techniques-apc-injection/malware-injection-techniques-apc-injection.html (Accessed: January 4, 2025).

[48] Kyriakos K. Ispoglou and Mathias Payer. 2016. malWASH: Washing Malware to Evade Dynamic Analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT '16)*.

[49] Ilker Kara. 2023. Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges. *Expert Systems with Applications* 214 (2023), 119133. doi:10.1016/j.eswa.2022.119133

[50] Amit Klein and Itzik Kotler. 2019. Windows Process Injection in 2019. *Black Hat USA* (2019).

[51] David Korczynski and Heng Yin. 2017. Capturing Malware Propagations with Code Injections and Code-Reuse Attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 1691–1708. doi:10.1145/3133956.3134099

[52] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *28th Network and Distributed Systems Security Symposium (NDSS 2021)*. doi:10.14722/ndss.2021.24475

[53] Tal Liberman and Eugene Kogan. 2017. Lost in Transaction: Process Doppelgänging. *Black Hat EU* (2017).

[54] Side Liu, Guojun Peng, Haitao Zeng, and Jianming Fu. 2024. A survey on the evolution of fileless attacks and detection techniques. *Computers & Security* 137 (2024), 103653. doi:10.1016/j.cose.2023.103653

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI '05). 190–200. doi:10.1145/1065010.1065034
[56] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. 2021. Longitudinal Study of the Prevalence of Malware Evasive Techniques. arXiv:2112.11289 [cs.CR] https://arxiv.org/abs/2112.11289
[57] Medium. 2019. Utilizing Image File Execution Options (IFEO) For Stealthy Persistence. https://securityblueteam.medium.com/utilizing-image-file-execution-options-ifeo-for-stealthy-persistence-331bc972554e (Accessed: January 4, 2025).
[58] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*. 1009–1024. doi:10.1109/SP.2017.42
[59] Modexp. 2019. Windows Process Injection: WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPlanting, Treepoline. https://modexp.wordpress.com/2019/04/25/seven-window-injection-methods/ (Accessed: December 12, 2024).
[60] Tadao Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580. doi:10.1109/5.24143
[61] Mohammed Nasereddin and Raad Al-Qassas. 2024. A new approach for detecting process injection attacks using memory analysis. *International Journal of Information Security* 23 (2024), 2099–2121. doi:10.1007/s10207-024-00836-w
[62] Emeric Nasi. 2014. PE Injection Explained. https://blog.sevagas.com/PE-injection-explained (Accessed: January 4, 2025).
[63] Meisam Navaki Arefi, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Faloutsos, Xuetao Wei, Daniela Seabra Oliveira, and Jedidiah R. Crandall. 2018. FAROS: Illuminating In-memory Injection Attacks via Provenance-Based Whole-System Dynamic Information Flow Tracking. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 231–242. doi:10.1109/DSN.2018.00034
[64] Red Team Notes. 2019. CreateRemoteThread Shellcode Injection. https://www.ired.team/offensive-security/code-injection-process-injection/process-injection (Accessed: January 10, 2025).
[65] Red Team Notes. 2019. Early Bird APC Queue Code Injection. https://www.ired.team/offensive-security/code-injection-process-injection/early-bird-apc-queue-code-injection (Accessed: January 4, 2025).
[66] Red Team Notes. 2020. NtCreateSection + NtMapViewOfSection Code Injection. https://www.ired.team/offensive-security/code-injection-process-injection/ntcreatesection-+-ntmapviewofsection-code-injection (Accessed: December 13, 2024).
[67] Red Team Notes. 2020. Reflective DLL Injection. https://www.ired.team/offensive-security/code-injection-process-injection/reflective-dll-injection (Accessed: January 10, 2025).
[68] Red Team Notes. 2020. SetWindowHookEx Code Injection. https://www.ired.team/offensive-security/code-injection-process-injection/setwindowhookex-code-injection (Accessed: January 10, 2025).
[69] Red Team Notes. 2021. PE Injection: Executing PEs inside Remote Processes. https://www.ired.team/offensive-security/code-injection-process-injection/pe-injection-executing-pes-inside-remote-processes (Accessed: January 4, 2025).
[70] Red Team Notes. 2024. Code & Process Injection. https://www.ired.team/offensive-security/code-injection-process-injection (Accessed: December 12, 2024).
[71] Unprotect Project. 2019. Extra Window Memory Injection. https://unprotect.it/technique/extra-window-memory-injection/ (Accessed: January 15, 2025).
[72] FortiGuard Labs Threat Research. 2016. AtomBombing - A Brand New Code Injection Technique for Windows. https://www.fortinet.com/blog/threat-research/atombombing-brand-new-code-injection-technique-for-windows (Accessed: January 10, 2025).
[73] Rinse and REpeat analysis. 2019. Early Bird Injection - APC Abuse. https://rinseandrepeatanalysis.blogspot.com/2019/04/early-bird-injection-apc-abuse.html?m=1 (Accessed: January 4, 2025).
[74] CAPE Sandbox. 2024. https://capev2.readthedocs.io/en/latest/ (Accessed: December 31, 2024).
[75] Cuckoo Sandbox. 2024. https://cuckoosandbox.org/ (Accessed: December 31, 2024).
[76] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A Tool for Massive Malware Labeling. In *Research in Attacks, Intrusions, and Defenses (RAID)*. Springer International Publishing, Cham, 230–253.
[77] SigmaHQ. 2024. sigma. https://github.com/SigmaHQ/sigma (Accessed: December 12, 2024).
[78] Kevin Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. 2011. SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *20th USENIX Security Symposium (USENIX Security '11)*. https://www.usenix.org/conference/usenix-security-11/shellos-enabling-fast-detection-and-forensic-analysis-code-injection
[79] Splunk. 2024. security_content. https://github.com/splunk/security_content (Accessed: December 12, 2024).
[80] Jerre Starink, Marieke Huisman, Andreas Peter, and Andrea Continella. 2023. Understanding and Measuring Inter-process Code Injection in Windows Malware. In *Security and Privacy in Communication Networks (SecureComm)*. 490–514.
[81] Microsoft Defender Security Research Team. 2017. Uncovering Cross-Process Injection with Windows Defender ATP. https://www.microsoft.com/en-us/security/blog/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/?source=mmpc (Accessed: December 12, 2024).
[82] Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. 2023. A Deep Dive into the VirusTotal File Feed. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 155–176. https://doi.org/10.1007/978-3-031-35504-2_8
[83] VirusTotal. 2024. https://www.virustotal.com/ (Accessed: December 13, 2024).
[84] Juan Wang, Chenjun Ma, Ziang Li, Huanyu Yuan, and Jie Wang. 2022. ProcGuard: Process Injection Behaviours Detection Using Fine-grained Analysis of API Call Chain with Deep Learning. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 778–785. doi:10.1109/TrustCom56396.2022.00109
[85] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A. Gunter, and Haifeng Chen. 2020. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *27th Annual Network and Distributed System Security Symposium (NDSS 2020)*. doi:10.14722/ndss.2020.24167

A Representative Process Injection Techniques

For our research, we reviewed a variety of process injection techniques and implementations discussed in the literature referenced in Section 2. Our goal was to define a comprehensive set of techniques observed in real-world scenarios, providing a solid foundation for a thorough and effective study of the process injection phenomenon.

Although Starink et al. [80] systematized 17 process injection techniques, some share similar underlying patterns but differ in their specific implementations, such as passive injection techniques that rely on system registry keys. To reduce redundancy and better align with the MITRE ATT&CK framework, which classifies these techniques under Event Triggered Execution [4], we grouped them into a broader category while retaining the ability to distinguish them at a more granular level. In other cases, we refined their framework to more clearly differentiate between specific techniques, such as distinguishing PE Injection from simpler Shellcode Injection. We also included techniques not covered in their review, such as Early Bird, along with some more exotic methods like Atom Bombing and Process Dopplegänging.

We remark that while newer techniques recognized by MITRE ATT&CK and the security community—such as Reflective DLL Injection [67] and ListPlanting [17]—introduce additional complexities and conjectures beyond basic memory range correlation, they can still be classified within the framework of the three core operations that underpin our methodology. These advanced techniques, along with other emerging methods used by adversaries, generally fall under broader categories like Shellcode and PE injection, which serve as the base case for many of these techniques.

As a result, our selection offers a comprehensive framework for understanding the core concepts of process injection, thereby enhancing the potential for effective detection. In total, we identified 12 techniques representative of different injections designs:

Shellcode Injection [64] involves allocating memory within the virtual address space of the target process, writing shellcode into this memory, and then executing the malicious code in the context of the victim process via a new thread.

PE Injection [13] involves allocating memory within the virtual address space of the target process, writing a whole Portable Executable (PE) into this memory, and then executing the malicious code in the context of the victim process via a new thread.

DLL Injection from Disk [11] involves allocating memory within the virtual address space of the target process, writing the path to a Dynamic-Link Library (DLL) into this memory, and then loading the DLL by creating a new thread within the target process that calls the `LoadLibrary` API. The semantics of this API prescribe that the requested DLL must be stored on, and thus read from, disk.

APC Injection [10] involves allocating memory within the virtual address space of the target process, writing the payload into this memory, and then executing the malicious code by attaching it to the Asynchronous Procedure Call (APC) queue of a victim thread. The target thread must be in an alertable state.

Process Hollowing [15] involves creating a new victim process in a suspended state, (as our findings reveal, optionally) unmapping the original executable code from memory, and replacing it with malicious code. The process is then resumed, with execution starting from the entry point of the injected code.

Thread Execution Hijacking [16] involves suspending an existing thread in the target process, allocating memory within its virtual address space, and writing the malicious code to this memory. The thread is then resumed and made to execute the injected code.

Early Bird [65] involves creating a new victim process in a suspended state, allocating memory within its address space, and writing the malicious code to this memory. The code is then attached to the APC queue of a victim thread, and the thread is resumed, with execution starting once the thread is in an alertable state.

Hook Injection [68] involves inserting a hook into the victim process for processing specific Windows-defined events, causing the registered callback to trigger the execution of malicious code. Such code is hosted in a DLL that the sample initially loads in itself, and the registered hook will eventually make the victim do so, too.

Atom Bombing [72] involves exploiting global atom tables to write a malicious payload in the virtual address space of the target process. The injected code is then executed by attaching it to the APC queue of a victim thread.

Extra Window Memory Injection [12] involves exploiting the Extra Window Memory (EWM) allocated during window class registration to execute malicious code in the context of the victim.

Process Doppelgänging [14, 53] involves abusing Windows Transactional NTFS (TxF) [32] to modify the contents of an executable pre-loading without making visible changes to the file on disk.

Passive Injection involves manipulating specific registry keys to force other processes to load malicious libraries. This technique encompasses several sub-techniques, including AppCertDlls [5], AppInit_Dlls [6], Application Shimming [7], Image File Execution Options [9], and Component Object Model (COM) Hijacking [8].

B Monitored APIs for Basic Three Operations

We report in Table 5, Table 6, and Table 7 the list of APIs that we monitor in our implementation and use to identify the three basic operation steps for an injection attack. For brevity, we refer to the implementation for details of the additional APIs that we use for fine-grained characterization of the specific technique in use.

Table 5: Relevant APIs for memory allocation.

API Name	Description

`VirtualAllocEx`
`VirtualAllocExNuma`
`NtAllocateVirtualMemory`
	Allocates a region of memory within the virtual address space of the specified process

Table 6: Relevant APIs for memory writing.

API Name	Description
`WriteProcessMemory` `NtWriteVirtualMemory`	Writes data to an area of memory in the specified process
`NtMapViewOfSection`	Maps a view of a section into the virtual address space of the specified process

`LoadLibraryA/W`
`LoadLibraryExA/W`
`LdrLoadDll`
	Loads the specified module into the address space of the *calling* process (needed in Hook Injection)

`GlobalAddAtomA/W` `NtAddAtom`	Adds a character string to the global atom table

Table 7: Relevant APIs for initiating code execution.

API Name	Description

`CreateRemoteThread`
`CreateRemoteThreadEx`
`NtCreateThreadEx`
`RtlCreateUserThread`
	Creates a thread that runs in the virtual address space of the specified process

`QueueUserAPC`
`NtQueueApcThread`
`NtQueueApcThreadEx`
	Adds a user-mode Asynchronous Procedure Call object to the APC queue of the specified thread

`SetThreadContext`
`NtSetContextThread`
combined with
`ResumeThread`
`NtResumeThread`
`NtAlertResumeThread`
	Sets the context for the specified thread
	
	Resumes the execution of the specified thread

`SetWindowsHookExA/W`	Installs an application-defined hook procedure for certain types of events
`SetWindowLongA/W` `SetWindowLongPtrA/W`	Sets a value at the specified offset in the extra window memory

C Representativity of Malware Dataset

In constructing our malware dataset, our primary goal was to select a minimal yet representative set of samples from each of the seven yearly collections, drawn from a large initial pool. To achieve this, we used VirusTotal's vhash, a hash computed from each sample's content, as the key feature for clustering. The vhash enables for the rapid identification and grouping of similar files or malware variants, even when they have been modified or obfuscated [36].

We opted for clustering as it allowed us to analyze a single, representative sample from each cluster, reducing redundancy and distribution bias. Although the vhash algorithm remains proprietary and undocumented, a recent authoritative study [82] has validated its efficacy as a good feature for clustering malware samples.

Table 8: Vhash-based clustering details for malware samples performing a process injection attack.

Year	Distinct vhash	Random vhash	100% Injecting	>80% Injecting	100% Same Injection Technique
2017	6,804	100	85	86	100
2018	16,611	100	69	81	98
2019	11,475	100	67	83	98
2020	16,368	91	75	82	90
2021	5,625	100	80	90	94
Total	56,883	491	376 (**76.6%**)	422 (**86.0%**)	480 (**97.8%**)

Table 9: Composition of the dataset filtered by *vhash*.

Year	Total	w/o vhash	w/ vhash	Distinct vhash
2017	26,093	575	25,518	6,804
2018	64,355	3,520	60,835	16,611
2019	59,098	315	58,783	11,475
2020	52,512	97	52,415	16,368
2021	25,176	42	25,134	5,625
2022	59,452	0	59,452	8,555
2023	48,199	0	48,199	8,484
Total	334,885	4,549	330,336	73,922

To evaluate the suitability of vhash as a clustering feature for malware samples performing process injection attacks (i.e., whether it would lead us to miss behaviors), we first retrieved the vhash values for all 334,885 samples in our initial dataset. Among these, 4,549 samples had no vhash value and were excluded from further analysis. Our investigation revealed that many samples within each yearly collection shared identical vhash values. To address this, we selected a single representative sample for each distinct vhash value, as outlined in Table 9. This process resulted in a final dataset of 73,922 samples, not accounting for any family balancing.

We further examined injection behaviors within the same yearly vhash clusters. To facilitate this, we compared our dataset with that used by Starink et al. [80], whose study is the most recent and comprehensive examination of process injection. Their dataset significantly overlaps with our VirusTotal samples due to the same source. However, since they did not attempt to remove near-duplicates and mitigate distribution bias (e.g., families, Section 4.1), we retrieved the vhash values of their published samples and mapped them to our collection, allowing us to reconstruct 95.6% of their dataset.

For a balanced comparison, we randomly selected 990 vhash values from the datasets intersection, ensuring equal representation of injecting (positive) and non-injecting (negative) samples. This resulted in 14,629 corresponding samples for evaluation.

Our analysis revealed consistent injection behaviors within the same vhash clusters across both categories. Table 8 provides a detailed breakdown of vhash-based clustering for samples performing process injection according to our methodology. Among the 491 vhash values considered, 76.6% of the clusters exclusively exhibited process injection behavior across all their samples. When we relaxed this requirement, allowing for the presence of injection behavior in at least a subset of samples within a vhash cluster, over 86.0% of clusters showed injection behavior in at least 80.0% of their samples. Overall, our analysis demonstrated a high level of consistency in the injection techniques employed within each vhash cluster, with 97.8% of clusters exhibiting uniformity in injection technique choice. This indicates that the vhash feature provides high precision for clustering the initial dataset according to injection behaviors, which in turn eased our evaluation on the complete dataset of 334,885 from the VirusTotal and Bazaar collections.

For the clusters classified as non-injecting (i.e., not performing a process injection) by our methodology, for all vhashes but two (0.4%) the non-injecting behavior was confirmed across samples.

D Risks of False Positives in Detection

We conducted two experiments to estimate the likelihood for our methodology to incur false positives. This aspect is especially pertinent as, in our approach, we relax the need for strict pattern matching present in prior work, especially Starink et al. [80], as it caused many false negatives when analyzing real-world malware.

While the experiments on the malware dataset let us appreciate the accuracy of our method when analyzing malware, for studying the risk of false positives we test it using goodware binaries.

In a first experiment, we follow the methodology used by Starink et al. [80], collecting 425 executables from the Windows OS build (`C:\Windows\System32`) and 97 popular applications shipped in portable form by Chocolatey, a package manager for Windows [39]. In accordance with recent longitudinal studies on malware behaviors [42, 56], we ensured the integrity of the dataset by filtering out any executables flagged as malicious by at least one antivirus vendor on VirusTotal. This step removed 55 benign programs exhibiting highly privileged behaviors typically abused by malware: these programs are debugging, monitoring, and security utilities for administrators. We were thus left with 467 binaries to analyze.

None of the executables triggered injection alerts, suggesting a low risk of false positives in everyday use cases with common software deployments. As for the debugging, monitoring, and security utilities filtered out by the VirusTotal scan, we observe that power users and system administrators are expected to collect them from trusted sources; also, to the best of our knowledge, AV and EDR vendors have explicit provisions to recognize popular tools.

However, we find that a residual risk exists with more complex applications that may legitimately allocate executable memory or load objects across processes as part of their normal operation.

We thus conducted a second experiment on software with higher degree of sophistication, testing the 20 programs listed in Table 10. Some of these programs are intrinsically harder to analyze: for instance, some may manipulate memory in multiple processes for the sake of displaying contents in a more secure way, using sandboxing mechanisms and isolation through multi-processing.

In this second experiment, we experienced 3 false positives, all related to a specific flavor of secure multi-process architecture. In Chrome and Firefox ESR, components such as the rendering engine and plugins are arranged into distinct processes. These browsers

Table 10: Programs for the second goodware experiment.

Category	Program
Browsers (3)	Chrome, Firefox ESR, Internet Explorer
Editors (3)	Notepad++, ONLYOFFICE, Sublime Text
Media (4)	Audacity, IrfanView, VLC, Windows Media Player
Microsoft Office (4)	Excel, OneNote, PowerPoint, Word
PDF readers (3)	Adobe Reader, PDF-XChange Editor, SumatraPDF
Utilities (3)	Skype, Zotero, 7-Zip

Table 11: Wine test workload and overhead percentages.

		Baseline tool	Injection detection	
Test	**Workload**	Time (s)	Time (s)	**Overhead**
kernel32	atom	8.147	8.379	2.85%
advapi32	registry	8.48	8.698	2.57%
user32	edit	14.427	14.645	1.51%
shell32	shellpath	20.415	20.673	1.26%
kernelbase	sync	4.383	4.437	1.23%
oleaut32	safearray	5.498	5.563	1.18%
ole32	storage32	33.994	34.391	1.17%
iphlapi	default	5.249	5.308	1.12%
ws2_32	protocol	6.804	6.872	1.00%
gdi32	metafile	22.751	22.968	0.95%
crypt32	encode	10.255	10.345	0.88%
wininet	internet	20.817	20.923	0.51%

create a suspended copy of themselves, allocate executable memory in the target process, write to it, and then resume execution. This method closely matches the operation with Process Hollowing, with the difference that the browsers resume execution without modifying the process context for where to resume execution (i.e., the process resumes from its original entry point). The same dynamics occurred with Skype, as it uses the Electron framework that, in turn, relies on the Chromium engine that backs also Chrome.

The operation of our method is technically correct in flagging these three programs, as they map new code just like malware does, and what is currently missing is a validation step or an allow-list policy for checking the legitimacy of the resumption point. We find that this dynamic resembles the challenges AV and EDR vendors face when dealing with suspicious programs: upon executing conspicuous actions, they start monitoring execution more closely, and block the program only when sufficient further evidence is gathered. In the current embodiment of our method, which primarily targets injection analysis in malware, we check for range congruence when `ResumeThread` is invoked. As a result, we discard the logged execution operational primitive as the resumption point does not fall within the memory allocated and written, but we still deem an injection due to the other two observed primitives. This setback could be rectified by factoring in contextual knowledge.

An anecdotal finding from this study is that Microsoft Office programs, specifically Word in our experiments, can use Hook injection to show the license activation popup to users. Our method spots a hook injection with `wwlib.dll` for filtering messages originating from GUI activity (`idhook=-1`). We underline that this is not a false positive, but a textbook execution of a legitimate Windows mechanism, and only the inspection of the DLL contents and behavior can reveal whether the DLL is malicious (in this case, an integrity check on contents and source may be sufficient).

Summarizing, from a semantics perspective, injection activities involve specific interactions with memory regions, and most legitimate software is unlikely to perform such activities, especially in the way that malware does. Nevertheless, not all code loading activity is malicious, and programs can employ legitimate features incidentally abused by malware. Incorporating behavioral and contextual awareness like AVs and EDRs do seems promising to improve the accuracy of the detection logic; it may even come as a commodity if the method were to be deployed in such a security solution.

E Detailed Figures for Overhead Experiments

In Section 5.3, we discussed the overhead introduced by our monitoring system compared to a baseline DBI monitoring system. Table 11 provides the workload details (chosen as in [31]) for each test along with the corresponding execution time values.

As shown, the highest slowdowns occur in the `kernel32` and `advapi32` tests, as these DLLs make heavier use of key instrumented APIs (especially for allocation and writing) related to process injection, using by design low-level primitives even for basic tasks (which instead is unlikely with client user programs).