Incident latency. Why an invalidated local artifact keeps acting once you have left it in place

Follow-up to the article on CVE-2026-31431. The residual sysctl lock case on WSL2 kernel 6.6.87.2 as empirical terrain for an operational concept: incident latency as the blind spot of operational security when an invalidated local artifact has not been uninstalled in the session that produced it…

…When operational maintenance best practices come back to remind you.

Introduction

The previous article posed the *promotion gap* as the measurable distance between the tested scope of a public recommendation and the scope where it is prescribed. It addressed artifacts received from the outside, where the issuing authority has no access to the receiving organization's target. The empirical terrain was CVE-2026-31431, known as Copy Fail, and the demonstration consisted in showing that four variants of the public recommendation had been behaviorally invalidated on WSL2 kernel 6.6.87.2 during the session of April 30, 2026. The effective mitigation had been produced elsewhere, through a surgical seccomp BPF filter on the socket(AF_ALG, …) syscall, validated by the kernel invariant /proc/<pid>/status.

This article concerns what happened afterward. More precisely, what happened during the 24 days that followed, in silence, in a corner of the system where no one was looking, and which returned brutally to attention on May 24, 2026 at the occasion of a public power outage that caused the abrupt shutdown of the host machine.

The thesis fits in one sentence. **An invalidated local artifact that has not been uninstalled keeps producing effects independent of the vulnerability it claimed to correct, and the latency between its activation and the incident it produces prevents operational attribution.**

The domain of validity is circumscribed. The argument concerns local mitigation artifacts applied in response to a public vulnerability, in environments where the cadence of application precedes the availability of an official patch. It is demonstrated on the

particular case of a kernel.modules_disabled=1 sysctl directive applied on April 30, 2026 in the Copy Fail context, and on the incident it produced on May 24, 2026 after a power outage. It generalizes to mitigations that modify the persistent state of the system (sysctl, modules, capabilities, namespaces, services) without a documented and tested removal procedure. It concerns neither the general conduct of operational cybersecurity nor the doctrine of patch management, which have their own established frameworks.

The demonstration follows four movements: the local artifact and its complete lifecycle, the empirical sequence from April 30 to May 24, the concept of incident latency, the operational criterion that derives from it.

Movement 1. The local artifact and its complete lifecycle

The previous article treated artifacts received from the outside. But an organization that operates a production environment also produces its own artifacts, particularly in emergency windows where the public recommendation is invalidated and an alternative mitigation must be designed within a few hours. These local artifacts have a different genealogy. The issuing authority is the organization itself. The tested scope theoretically coincides with the prescribed scope, since the artifact is produced for the exact target on which it applies. The promotion gap should be zero. On this point, the implicit doctrine is right more often than it is for external artifacts.

But this absence of promotion gap at application says nothing about the lifecycle of the artifact after its application. And it is here that a distinct blind spot is lodged, which deserves its own concept.

The distinction that cuts here is simple. *Applying* an artifact is not *terminating* an artifact. Application installs the artifact in the persistent state of the system. Termination cleans the artifact when it is no longer relevant, when it has been replaced by another mitigation, or when it has been behaviorally invalidated. Termination is not optional in the experimentation phase, which is precisely the phase where the majority of artifacts are invalidated. But operational doctrine generally treats application as an act and termination as an implicit consequence. It is exactly the opposite. Application produces an observable event traced in a changelog. Termination demands an explicit procedure, documented and tested. The imbalance between the two is structural.

When a local mitigation is tested and declared inoperative, two operational trajectories open. The first consists in moving to the next attempt while leaving the artifact in place, on the assumption that it does no harm since it has been invalidated. The second consists in uninstalling it cleanly before testing the next attempt. The first trajectory is dominant in practice, because it is faster and because the time pressure of the emergency window weighs more than environment hygiene. The second demands an additional discipline

that produces no immediate value. The first is exactly the one that produces incident latency.

An invalidated artifact is invalidated under a certain regime of operation. Modifying that regime can reveal effects that behavioral invalidation did not test, because no one knew they were there.

Movement 2. The empirical sequence from April 30 to May 24

Article VI documents four successive attempts at public mitigation on Copy Fail, all invalidated by the same syscall test. Attempt 2 mobilized the kernel.modules_disabled=1 sysctl lock. The lock forbids loading kernel modules at runtime, which was supposed to prevent the automatic reloading of algif_aead by request_module(). Behavioral invalidation showed that the lock did not prevent the bootstrap primitive, because the modules were already loaded in memory and the lock did not purge memory. Attempt 2 was documented as inoperative in Article VI, and the approach moved to attempt 3 then 4, until the seccomp BPF mitigation that held.

But the file /etc/sysctl.d/99-cve-2026-31431.conf containing the directive kernel.modules_disabled=1 remained in place. It had been created for attempt 2, it survived its invalidation, and it was not removed in the session that invalidated it.

For 24 days, from April 30 to May 24, the lock remained active in the sysctl sense. But its practical effect was masked. The modules iptable_nat, iptable_filter, nf_conntrack, br_netfilter and ip6_tables were already loaded in WSL2 kernel memory since the initial Docker installation in March 2026. As long as kernel memory was not flushed by a cold reboot, the modules remained available, and the services that depended on them (Docker 29.3.0, Tailscale, containerd erofs plugin, Open WebUI container) operated nominally. The Docker journal of May 6, 2026 shows a fully operational daemon, with image pull, active bridge networking, container management. No apparent symptom of the residual artifact.

On May 24, 2026 at 22:13, an outage of the public electrical network caused the abrupt shutdown of the Windows Server 2025 host machine running WSL2. Power restoration restarted Windows. The scheduled task WSL2-AI-Startup, configured on boot trigger, attempted to invoke wsl.exe -d Ubuntu-24.04 -- /home/jeromev/scripts/start-all.sh at 22:13:11, that is, 8 seconds after the Windows boot. For reasons that remain to be investigated in a separate post-mortem, the command failed silently, and WSL2 only actually started at 23:41 upon manual intervention through terminal opening.

During those 88 minutes, the public demo at twingital-ventures.com/realisations/toxtwin was unavailable. This unavailability is a separate subject to be treated elsewhere. What concerns us here is what happened when WSL2 finally booted.

At fresh boot, WSL2 kernel memory was initialized blank.

- The netfilter modules pre-loaded since March vanished with the memory.
- The kernel.modules_disabled=1 lock, persistent in /etc/sysctl.d/, was applied by systemd-sysctl.service during boot.
- From that moment on, the WSL2 kernel refused any module load, including for iptable_nat, which was necessary for Docker.
- The Docker daemon, started by systemd, attempted to create its NAT chain, received in return the error iptables v1.8.10 (legacy): can't initialize iptables table 'nat': Table does not exist, and crashed in a loop through three attempts before systemd marked it as failed.
- Tailscale, started in parallel, failed in the same way on its IPv6 rules with the message modprobe: ERROR: could not insert 'ip6_tables': Operation not permitted. The containerd erofs plugin refused to load for the same reason.

Four apparently distinct symptoms, in four independent services. A single root cause. The root cause was not the power outage. The outage merely revealed a condition that had been present for 24 days. Without the outage, the system would have continued to function indefinitely in its masked state, until another cold reboot event (Windows update, scheduled restart, manual shutdown) produced the same result.

Diagnosis took about two hours, the bulk of it spent exploring false leads: Docker 29 bug, Microsoft WSL2 kernel regression, nftables vs. iptables-legacy conflict, namespace capabilities issue. These leads were all plausible, and some were locally true (the WSL2 kernel does not allow runtime modprobe outside a privileged context). But none of them was the root cause. The root cause was a 26-byte file in /etc/sysctl.d/, created for an invalidated mitigation 24 days earlier, never cleaned up.

Movement 3. Incident latency as an operational concept

The prolonged Copy Fail case makes explicit what remained implicit in the doctrine of the promotion gate and the promotion gap. Every persistent artifact introduced into the state of the system possesses a temporal dimension of its own, distinct from its moment of application and its moment of invalidation. This dimension has no name in standard operational doctrine, and that is precisely why it is ignored. I propose to call it *incident latency*.

Operational definition. *Incident latency* designates the temporal distance between the activation of a persistent artifact and the incident it produces, insofar as that distance blurs causal attribution.

The definition is restrictive on two points. First, it presupposes an artifact that modifies the state of the system, not a transient operation. A shell command that runs and terminates does not produce incident latency. A sysctl directive, a loaded kernel module, an iptables rule, a configuration file in place, an activated systemd service, do. Second, it presupposes a blurring of attribution. If the artifact produces its incident immediately, in the session that installed it, the cause is obvious. If the incident triggers a month later at the occasion of a decorrelated event, the natural attribution falls on the event, not on the artifact.

The distinction that cuts here separates two causal regimes that operational practice systematically conflates. **The event cause is what is seen.** The power outage, the scheduled restart, the kernel update, the load spike, any observable event that immediately precedes the incident. It is this cause that the usual post-mortem investigation reconstitutes, because it is traced in logs and has a clear temporal signature. **The state cause is what acts.** It is the condition of possibility of the incident, present in the system before the event, and which renders it possible. The event cause is necessary but not sufficient to produce the incident. The state cause is necessary and co-produces sufficiency.

In the May 24 case, the event cause is the power outage. The state cause is the file 99-cve-2026-31431.conf. Without the outage, the incident does not occur. But without the file, the outage would have produced only a transient interruption followed by a nominal restoration of services. The state cause is what transformed a banal event into a structural incident.

Three properties characterize incident latency.

1. **Silence during the masking phase.** While the effect of the artifact is neutralized by another element of the system state (in the present case, the modules in memory), the artifact produces no observable symptom. No log signals its presence. No functional test detects it. The organization can conduct audits, configuration reviews, penetration tests, and the artifact will pass unnoticed because it is operationally transparent as long as the masking condition holds.
2. **Triggering by a decorrelated event.** The incident occurs at the occasion of an event that has no apparent relation to the residual artifact. A power outage has nothing to do with a crypto mitigation. A kernel update has nothing to do with a residual iptables rule. A scheduled restart has nothing to do with an undead service. The apparent decorrelation between the triggering event and the residual artifact is precisely what produces the attribution blurring.

3. **Erroneous attribution by default.** The natural investigation focuses on the triggering event. The power outage becomes the subject of the post-mortem. The remediations envisaged concern resilience to outages, improvement of boot sequences, hardening of services. All of these remediations may be justified on other grounds, but none of them treats the state cause. The residual artifact stays in place. The incident can therefore reproduce, at the occasion of another decorrelated event, with another surface signature that will orient toward another erroneous investigation.

These three properties explain why incident latency is not a rare or anecdotal problem. It is the predictable consequence of the structural asymmetry between application and termination of artifact posed in Movement 1. As long as that asymmetry persists, incident latency persists. And as long as it is not named, it cannot be investigated.

Movement 4. The operational criterion. Explicit reversibility as the condition of experimentation

The concept of incident latency calls for an operational criterion, which articulates without contradiction with the promotion gate (Article V) and the promotion gap (Article VI). But it does not add to them as a supplementary requirement that one could have omitted. It is the condition of their holding over time.

The promotion gate presupposes that an artifact is promoted from experimentation to normative dependency after a deliberate and traced crossing. This doctrine implicitly presupposes that a non-promoted artifact remains in experimentation, or exits it by removal. But in observed practice, a non-promoted artifact can remain in place in the operational system without being either promoted or removed, in a hybrid status that has no name in the doctrine. It is this hybrid status that produces incident latency. The promotion gate as a verificational device becomes a one-way gate if the organization has no mirror procedure of removal.

The promotion gap designates the distance between tested scope and prescribed scope for external artifacts. A parallel reasoning applies to internal artifacts: the distance between the scope tested at the moment of application and the effective scope at the moment when the artifact acts. This distance is also non-zero, because the state of the system when the artifact acts can be different from the state in which it was tested. In the May 24 case, the kernel memory full of modules was the state at the moment of testing (April 30). The empty kernel memory was the state at the moment when the artifact acted (May 24). The artifact acted in a state in which it had not been tested, because no one had thought to test what it did in that state.

The criterion that follows is formulated thus: **every persistent artifact must have a documented and tested removal procedure before its application, not after.** This formulation is demanding, deliberately so. It acknowledges that the emergency phase does not lend itself to the simultaneous production of a clean removal procedure. It therefore demands that the removal procedure be thought through before the emergency phase, as an integral part of the local mitigation toolkit. An organization that has no pre-drafted procedure for removing a sysctl directive should not apply a sysctl directive. An organization that has no procedure for detaching a seccomp BPF filter should not attach a seccomp BPF filter.

The operational corollary is even more constraining: **every invalidated local mitigation must be cleaned in the operational session that invalidated it.** Not deferred to a later cleanup session. Not registered in a technical debt backlog. Cleaned immediately, in the same session, by the same person, with the same mental context. The deferral is exactly what produces incident latency. The mental context that allows one to understand why an artifact can be removed without risk is ephemeral. Once the session is over, that context dissolves, and removal becomes an autonomous operation that requires reconstituting history, which costs more than immediate cleanup would have cost.

The predictable objection to this criterion is that it shifts the burden onto operators who intervene in the emergency window, adding to the pressure of the moment the demand for supplementary hygiene. The objection is just, and that is the point. The burden has never been elsewhere. It was simply deferred onto the future operator, generally unknown, who would face the incident produced by the residual artifact without context to understand it. The doctrine of incident latency does not shift the burden, it makes visible the burden that was already there and that current practice allowed to accumulate in silence.

A second predictable objection is that the criterion, applied strictly, leads to operational paralysis in emergency windows where time is too short to produce clean removal procedures before each attempt. The objection is partially just. The criterion must be applied with discernment regarding the nature of the artifact. A transient shell command does not demand a removal procedure. An iptables rule added on the fly demands at minimum a session note. A persistent sysctl directive demands a pre-drafted removal procedure. A loaded kernel module demands an unloading test. An activated systemd service demands a tested deactivation procedure. The granularity of the criterion must be proportioned to the persistence and blast radius of the artifact. But it cannot be zero for artifacts of high blast radius.

Experimentation distinguishes itself doctrinally from production by its explicit reversibility, not by its intention. The intention to test without long commitment is no protection if the test leaves persistent traces that no one cleans. Explicit reversibility, that is to say the documented and tested capacity to return to the prior state, is the property

that distinguishes an experimentation environment from a production environment that ignores it is in production. Without that property, experimentation is a disguised production deployment, and all the engagements of reliability, security and governance that should apply to a production environment apply in reality, without the organization being aware of it.

Conclusion

An artifact applied in an emergency window, declared inoperative after behavioral testing, and left in place by default of a removal procedure, produces a risk whose temporal signature is different from the risk the mitigation was meant to correct. The original vulnerability, in the Copy Fail case, was never exploited on the environment considered, because the effective seccomp BPF mitigation held. But the residual artifact of the invalidated attempt produced, 24 days later, a complete interruption of services independent of crypto, at the occasion of a banal event. The incident was not caused by Copy Fail. It was caused by the trace that Copy Fail left in the system when the organization stopped paying attention to Copy Fail.

This asymmetry between the original vulnerability and the incident produced by its residual mitigation is the most destabilizing property of incident latency. It implies that the operational security of an environment cannot be evaluated as the sum of its active mitigations. It must be evaluated as the sum of its active mitigations plus the sum of the persistent traces left by past mitigations, in their interactions with the current state of the system. This second sum is, in most organizations, unknown. It is neither inventoried nor tested. It accumulates silently in /etc/sysctl.d/, in /etc/modules-load.d/, in systemd services, in SELinux hooks, in capabilities, in namespaces, in all the places where the state of the system is modified without a complete documented lifecycle.

The doctrine of the promotion gate and the promotion gap concerned the promotion of artifacts into normative dependency. That of incident latency concerns what happens when an artifact has not been promoted but has not been removed either. It is, operationally, the most frequent situation in heterogeneous production environments. It is also the least addressed by standard security frameworks. Its handling is not optional for regulated domains, where the erroneous attribution of an incident to its visible event cause can lead to remediations that do not treat the actual risk, and to an accumulation of residual debt that prepares the next incident with a different signature.

Experimentation distinguishes itself from production only by its explicit reversibility. As long as an artifact has no tested removal procedure, it is in production. The conscious knowledge of this equivalence is the condition of an environment hygiene that holds over time. Supposing it acquired by default is what produces incident latency.