CVE-2026-98432: Proving the Kernel Patch by Verified Disassembly, Not Metadata

Jacky Keung
Agentic Software Intelligence (ASI) Lab
Dept. of Computer Science, City University of Hong Kong
Jacky.Keung@cityu.edu.hk

Abstract

The Linux kernel carries the fix for a CVSS 9.8 use-after-free, yet its version string and CVE tracker still call the shipped system vulnerable. The upstream fix (commit e3f33b0a) did arrive: disassembly of the shipped Ubuntu 6.8.0-149 binary shows `kvfree_call_rcu` at both free sites. Only binary inspection is authoritative. CVE-2026-98432 is a remotely triggerable use-after-free in the Linux IPv4/TCP path; Anthropic disclosed an unauthenticated root exploit in October 2026, which we do not reproduce. We verify the fix at both free sites by bounded disassembly, show patch verification tooling is itself unreliable, and derive an interim firewall rule. The disassembly runs through an automated pipeline that iterates a verify, diagnose, correct, re-validate loop, converting a wrong verdict into a verified one. Triggering against the vulnerable KASAN build produced a genuine use-after-free read of freed `ip_options_rcu`; it came once, and a 1.56 million attempt replicate did not recur. The single win classifies rung-1b as INTACT: the object was not yet reused, KASAN poisons only shadow memory, so the reader read back the trigger's own byte. It leaks nothing and shows the return path. This is a tail event: one event in 2,167,525 attempts gives a point rate of 4.61×10^{-7} with an exact Poisson interval 1.17×10^{-8} to 2.57×10^{-6}, which contains the external published rate (near 4×10^{-8}), so we claim no elevation. Negative controls are underpowered, so we draw no patched did not fire, therefore patched is fixed inference. The honest ceiling is a use-after-free whose win leaks nothing, never root. The host predating kernel 6.8.0-149 still calls kfree at both sites; the release named to carry the fix was still work in progress.

Keywords: Linux kernel security; use-after-free; IPv4/TCP options; patch verification; binary inspection; distribution tracker misreport; interim network mitigation

1 Introduction

The Linux kernel is the substrate for nearly all public-facing Internet infrastructure, yet its network management path spans decades of incremental code in which object lifetimes are the most error-prone area. Empirical studies of operating-system code find exactly this pattern: memory errors concentrated in the network and protocol layers, and a base rate of such defects that makes a broad "the kernel is buggy so no single bug matters" framing wrong [7], while fuzzing of TCP stacks in particular surfaces memory and semantic bugs from concurrent connection-state transitions [22]. A single use-after-free in that path can be reached by any remote host that can open a TCP connection to a listener. When such a defect sits in the kernel's IPv4 options handling and is scored CVSS 9.8, the practical question is not simply *is it exploitable*, but *what is the honest blast radius an attacker can reach, and how do we detect and mitigate it before a fixed kernel ships.*

In October 2026 Anthropic's Frontier Red Team published that an unreleased research model had discovered CVE-2026-98432 and written a complete, working exploit [3] capable of running arbitrary commands as root against a Linux server with a multipath TCP listener, with no credentials and no user interaction. The exploitation has two stages: first a remote KASLR defeat driven by a race-conditioned out-of-bounds overread, and then a control-flow hijack in which the same use-after-free overflows a buffer and overwrites a callback pointer with a staged payload. In the disclosed proof of concept it takes roughly 48 minutes to leak a kernel pointer and five more minutes to reach a root shell.

This paper studies that defect. We deliberately do not reproduce the full RCE chain. The central finding is a patch-verification honesty result: the shipped 6.8.0-149 binary is patched at both free sites, we verify this by disassembly, yet the version string and the distribution tracker both fail to say so. Around that we build the instrumented reproduction harness, detect the race once and bound its rate (the single win classifies rung-1b as INTACT, so it leaks nothing and demonstrates only the return path), verify the fix by binary inspection, and state plainly the boundary of what we achieved. The contributions, each stated only to the level the evidence supports:

- **Binary-integrity patch verification.** We verify the fix at both free sites by disassembly of four independently built kernels, never by version string, and record that the release named to carry the distro fix was still listed as work in progress at the time of engagement. Trusting a version string is not enough.
- **A patch verifier that is itself verified.** The offline checker that reports whether a kernel is patched is software with silent failure modes: at the shared-map stage of this engagement it returned a confidently wrong verdict, declaring a correctly patched kernel vulnerable. The single wrong answer had three compounding causes. We build a thirteen-case negative-control suite, make a mixed state a first-class verdict, and make malformed input fail loudly, and we report the instrument failures in full.
- **Correction of the public record on MPTCP.** The disclosed vehicle was a Multipath TCP (MPTCP) listener, and the code path shows the bug path is plain-TCP fallthrough: MPTCP is not required. Disabling it does not close the path.
- **A single live detection of the race, and an honest bound on its rate.** Firing the trigger against the vulnerable KASAN build produced a genuine KASAN slab use-after-free at `inet_csk_route_req`, a 1-byte read of `opt->opt.srr` at offset 25 of a freed `ip_options_rcu`, read on one CPU and freed cross CPU by `inet_sock_destruct`. That detection occurred once in 2,167,525 vulnerable attempts, giving a point win rate of 4.61×10^{-7} with an exact Poisson 95 percent interval of 1.17×10^{-8} to 2.57×10^{-6}; we also report, plainly, that a 1.56 million attempt replicate did not recur and that the negative controls are underpowered.
- **Interim network-layer mitigation, verified functionally.** We derive and validate iptables and nftables rules that drop option-bearing SYN packets, prove they match by counter movement rather than rule text, and state what they do not protect against.
- **Boundary honesty.** We credit the full root code execution to the public disclosure and do not claim it as our own. The complete privilege escalation chain is out of scope.

2 Background

This section gives the kernel internals needed to understand the defect. We describe the IPv4 options path, the request socket lifecycle that governs the `ip_options_rcu` object, and the RCU rules the correct lifetime must follow.

2.1 Use-after-free: a pointer to freed memory

A use-after-free (UAF) is a temporal memory-safety defect in which a program frees an object and later reaches it again through a pointer that was never invalidated, so the access hits memory that the allocator may have reassigned to another object. Because the freed memory is often reused, the stale reference can read or write an unrelated allocation, and the outcome is undefined until that reuse occurs. Three impact classes follow. *Data disclosure* leaks the reclaimed bytes of another object through the dangling read. *Silent corruption* damages a live object on the write, producing state that looks plausible and surfaces later as a crash in an unrelated component. *Code execution and privilege escalation* arise when the reclaimed object is a kernel structure, so an attacker shapes the reuse to overwrite credentials or function pointers and gain kernel-context execution.

Three production incidents illustrate these classes. `CVE-2023-32233` is a UAF in the Netfilter `nf_tables` batch-request handler that gives an unprivileged local user arbitrary kernel read and write, and thus root [15]. `CVE-2021-4154` is a UAF in the cgroup v1 parameter parser that an attacker holding ordinary local credentials uses to escape a container [14]. `CVE-2023-3269` is a UAF in the memory-management VMA lock handling that yields arbitrary kernel code execution [16]. In each case safety depends on the pointer lifetime, not on the access that frees the memory, which is why the defect is scored high severity even when a single trigger is rare.

2.2 The IPv4 options path

When a TCP SYN segment carries IP options (for example the Timestamp option or Record Route), the kernel must retain those options to echo them in the SYN-ACK. The options are parsed into a separately allocated `ip_options_rcu` structure, roughly 72 bytes, drawn from a 96-byte slab pool. This object holds the options the responder copies into its reply, so it must be kept alive for the whole duration of the TCP handshake on a particular 4-tuple.

2.3 Request socket lifetime and the handover

Incoming connection attempts are represented by a request socket (a "request sock"). For a half-open connection the kernel stores the parsed IPv4 options on the request socket. When the handshake completes, `tcp_v4_syn_recv_sock()` creates the child (established) socket. The critically important detail for our defect is that this function hands the options object to the child socket *without copying it*, and clears the request socket's pointer. From this point onward the child socket owns the same object, while the original half-open request socket has no reference to it. An attacker who can place a second CPU inside the SYN-ACK construction path for the same 4-tuple while the child is torn down can therefore read an object that has already been released.

2.4 Why a plain `kfree()` is wrong

The correct Linux pattern for a kernel object that may be read by another CPU concurrently (for example on another core during a still-running SYN-ACK transmission) is to defer its release behind an RCU grace period rather than to free it immediately with `kfree()`, per the read-copy-update

contract and its kernel usage rules [13]. The deferred release guarantees the object is not reclaimed until every reader that entered before the grace period has completed. Two free sites in the IPv4/TCP path instead release the object with plain `kfree()`: `inet_sock_destruct()` in `net/ipv4/af_inet.c` and `tcp_v4_reqsk_destructor()` in `net/ipv4/tcp_ipv4.c`. Violating that contract is the exact object-lifetime error that use-after-free studies single out as the most severe and hardest to detect class of C/C++ memory bug [9]. The upstream fix published by the kernel maintainers substitutes a deferred free for the immediate one at exactly those sites: in `inet_sock_destruct` the plain `kfree(rcu_dereference_protected(...))` becomes `kfree_rcu(rcu_dereference_protected(...), rcu)`, and the same change is made in `tcp_v4_reqsk_destructor`, so the option memory is released only after an RCU grace period [3, 8, 11]. The grace period semantics that the vulnerable code violates are described in [12]. That is the semantic the vulnerable code is missing. If a second CPU is still reading the options object inside `ip_options_build()` at the moment the object is freed, it dereferences freed memory and copies stale bytes into the response.

2.5 Source-level confirmation of both free sites

Independent inspection of the Ubuntu 6.8.0-147 source tree confirms both free sites are present and unremediated. In `net/ipv4/af_inet.c` at line 157 the destructor releases the options object with `kfree(rcu_dereference_protected(inet->inet_opt, 1))`, and in `net/ipv4/tcp_ipv4.c` at line 1225 the request-socket destructor uses `kfree(rcu_dereference_protected(inet_rsk(req)->ireq_opt, 1))`. Neither `ip_options_rcu_free` nor `kvfree_call_rcu` appears anywhere in `net/ipv4` in that tree, so the deferred-release semantic is genuinely absent rather than merely relocated.

2.6 Binary confirmation across four built kernels

Source inspection alone leaves open whether the compiler emitted the expected release at each site, so we built the four variants described in Section 3 and inspected the emitted code at both sites, bounding each function by the address of the next kernel-text symbol in that kernel's own symbol map. Table 1 reports the result.

The patched build confirms the fix at both sites, and the three unpatched builds confirm plain release at both. We report this as binary inspection, not as a runtime measurement; the measured behaviour of these sites is reported in Section 3. Building all four variants took 54 minutes of wall clock rather than the several hours we had estimated, because the distribution configuration was copied rather than regenerated.

The verifier was wrong before it was right. Our first independent binary check reported the patched kernel as vulnerable, contradicting the build agent. The check was at fault for three compounding reasons: it used a single shared symbol map for all four kernels, so addresses resolved into unrelated functions; its search for the next symbol was corrupted by the per-CPU entries that begin every map at address zero, yielding function extents of sixteen bytes that contained no code; and its fixed 512-byte disassembly window ran past each function end, capturing neighbouring functions' releases and reporting a false patched verdict for other cells. Bounding each function by the next kernel-text symbol in its own map resolved all three. This was a real, correctable bug in the tool, not a quirk of the kernel: it means a patch checker is software with silent failure modes that returns a confident word rather than an exception, so a fixed-window checker can report the wrong verdict in either direction, and the danger claims are the ones that travel. A verifier therefore needs negative controls in both directions, a known vulnerable image it must flag and a known patched image it must clear, and a result is a hypothesis until a control has confirmed the tool.

The rebuilt checker was run against thirteen cases and inputs: seven synthetic ELF images spanning the vulnerable, patched, half-patched and neighbour-leak shapes, and six hostile inputs (a truncated symbol map, an empty image, a missing map, a map that is not text, a non-ELF image, and a host whose `objdump` cannot decode x86-64). Nine of the thirteen belong to the suite and are expected to receive a verdict; the rebuilt checker gave the correct verdict on all nine, and on the remaining four, which must never produce one, it refuses with a nonzero exit and no verdict. Against the four real built kernels it agrees with ground truth, and we independently reproduced that labelling without reading the tool. Thirteen cases is a suite, not a proof: the residual gaps are that it tests static call edges rather than object identity, and that a free reached by a register-indirect call would be reported unresolved rather than guessed. The patch-integrity confirmation is carried out by an automated verification pipeline, not by a single human pass over a disassembly listing: it fetches and stages the source, builds the four kernel variants, locates each free site by source then resolves each site's emitted call in that kernel's own image, and emits a verdict. Where a verdict disagreed with the expected result, the pipeline did not accept it; it diagnosed the failure to its root cause, rebuilt the checker, and re-validated it against the thirteen-case suite plus the four real kernels before trusting any verdict. This closed verify, diagnose, correct, re-validate loop is what converts a tool that was wrong into one that is verified, and its decisions are confined to the patch-verification method (symbol resolution, function bounding, verdict acceptance). We do not claim a general-purpose autonomous agent that reasons about the kernel or the exploit beyond this verification task; where a human decision remains, we say so, and we do not present the pipeline as removing the need for expert review of the verdict.

Table 1: Release call emitted at each free site, per built kernel. The patched kernel retains a `kfree_skb_reason` call inside `inet_sock_destruct`; this is a different, legitimate release path in the same function, not a residue of the defect.

Kernel	`inet_sock_destruct`	`tcp_v4_reqsk_destructor`	Verdict
Stock, distro config	`kfree`	`kfree`	Vulnerable
Vulnerable with KASAN	`kfree`	`kfree`	Vulnerable
Patched with KASAN	`kvfree_call_rcu`	`kvfree_call_rcu`	Patched
Minimal, defconfig	`kfree`	`kfree`	Vulnerable

2.7 Detection traps that void a study

Three practical traps cost us time and would each invalidate a careless measurement, so we name them.

Wrapped free calls. The free sites do not call a bare `kfree(x)`. They call `kfree( rcu_dereference_protected(...))`, because the dereference must be annotated for sparse. A static or binary scan that matches on a bare `kfree` call target therefore misses both sites and reports the tree as clean. Any automated checker must resolve the call target inside the wrapper, or it produces a confident false negative.

Defconfig is not the stock configuration. Building the reference cell with `make defconfig` yields a kernel that is not the distribution kernel under study, because the distribution configuration differs in thousands of options including ones that govern the networking path. We therefore copy the distribution's own configuration, taken from the package's `linux headers` directory at the path `/usr/src` followed by `linux headers <version>/.config`, which for the tree studied already sets `CONFIG_VIRTIO_NET=y`, `CONFIG_VIRTIO_PCI=y`, `CONFIG_TUN=y` and `CONFIG_KALLSYMS_ALL=y`. Using defconfig would void the stock-versus-minimal contrast we intended to test.

Presence is not function. A deployed firewall rule can exist in the running table and never match a packet. Verifying a mitigation by searching for the rule text is therefore not verification. We test function by sending the trigger packet class and asserting that the drop counter increments, as described in Section 7.

2.8 The race condition

The race is a cross-CPU timing hazard. The free must land roughly one packet-time (a few microseconds) behind the reader, on a different core, for the second CPU to still be inside the reader routine. A lost race merely fails the handshake without crashing the system, so the attacker can retry indefinitely until one instance succeeds. This is why the reproduction is geometry-dependent and why the kernel must be built with instrumentation (KASAN) or the attacker must use an oracle to observe the leak on a production kernel. KASAN is the de facto Linux kernel detector for use-after-free and out-of-bounds bugs, generalising the shadow-memory and redzone approach of userspace AddressSanitizer [17], and is the standard tool against which kernel fuzzing and binary-only kernel sanitizers are validated [6]. Section 3 describes the measurement and the oracle we use.

3 Experiments Setup

We detail the reproduction harness, the kernel builds, the network topology, the statistical model, and the oracle used to detect the leak on a production kernel.

3.1 Kernel variants

We build three kernel variants to form a controlled differential:

- **Vulnerable KASAN.** The stock configuration with the defect present and KASAN enabled. This is the positive-control instrumented build that makes the use-after-free visible as a report.
- **Patched KASAN.** The same build with the two free sites changed to the deferred RCU release. This is the negative control; it must stay clean.
- **Stock non-KASAN.** A production-grade uninstrumented kernel. This is the reference condition, since the leak only reproduces on a stock kernel configuration. We run this one with the detector oracle.
- **Minimal custom.** A reduced configuration built from `defconfig` plus the networking features needed to run the harness. This is the configuration-sensitivity control.

3.2 Lab environment

The experiment runs on a single dedicated bare-metal host (16 cores, 125 GB RAM) with KVM hardware virtualisation. The guest is launched as shown in Listing 1, using 8 virtual CPUs and 2 GB of memory.

Listing 1: Guest launch command.

```
qemu-system-x86_64 -cpu host -smp 8 -m 2048 \
  -machine q35,accel=kvm -enable-kvm -nographic -
      no-reboot
```

A graphics processing unit is deliberately not used for acceleration: the workload is a CPU-bound kernel network-path race plus compilation, where a GPU adds nothing.

3.3 Network topology

We avoid the loopback interface, which skips the real network stack path on which the CVE races. Instead we use

a veth pair across two network namespaces on the host, bridged through a tap device into the guest. The tapped end is the guest network interface and the peer namespace runs the client/harness. This preserves the genuine networking path while keeping the whole environment inside the single host.

3.4 The oracle

On a production (non-KASAN) kernel the free is silent, so we need a signal. Each attack SYN embeds 8 bytes inside its IPv4 options: a magic value and a tag equal to the magic XORed with a constant. The kernel echoes the options back in the SYN-ACK. Under normal operation it echoes the same attempt's magic. If the object has been freed and reallocated, the echoed bytes can be a different allocation's magic that still carries a valid tag, which a correct kernel can never produce. Each such event is one oracle hit. This is the same overread primitive the disclosed exploit uses to leak a kernel pointer for the remote KASLR defeat.

3.5 Attack harness and parameters

The harness reuses the static-`/init` initramfs framework from the published reference trigger, adapting only the network wiring. Every cell uses consistent parameters: `tcp_max_syn_backlog` set to 4096 so the SYN queue does not fall back to SYN cookies, `tcp_syncookies` proven off, and an explicit accept-then-close (never a RST) as the free trigger. The reader and the free are co-located on a single 4-tuple with the free trailing the reader by at most one packet, forced across two CPUs. Because the disclosed exploit targets a multipath TCP listener, we include MPTCP on and off as a factor in the run matrix to measure its effect on the exposure surface.

3.6 Attempt budget and statistics

The leak is a Bernoulli event. The probability of observing at least one hit in N attempts at a per-attempt rate p is

$$P(\geq 1 \text{ hit} \mid N, p) \;=\; 1-(1-p)^N. \tag{1}$$

Given our attempt budget we set the target rate needed for 95% power to detect at least one hit as roughly **200,000** attempts at the pessimistic case ($p = 1.5 \times 10^{-5}$), about 120,000 at $p = 2.5 \times 10^{-5}$, and about 75,000 at 75,000 at the optimistic case ($p = 4 \times 10^{-5}$). We therefore run a minimum of **200,000 attempts per vulnerable cell**. Those rates were our planning assumptions, not measurements, and we do not use them to interpret the sweep; the observed firing was far rarer than even the optimistic planning figure, which is itself a finding. Negative-control cells that return zero are reported as an upper confidence bound rather than as a claim of absence. We report the bound from the rule of three and say so, because the rule of three is a convenience approximation to the exact one-sided Clopper-Pearson limit. For zero events in N attempts the exact limit is $1-0.05^{1/N}$, which at our N is 2.5265×10^{-6} against the shortcut's 2.5300×10^{-6}, a difference of about 0.14 percent. Where we can run replicates we report exact Clopper-Pearson intervals; the sweep reports the shortcut and labels it as such.

3.7 Run matrix

The experiment crosses kernel variant (4) by geometry (cross-CPU TUN, loopback-adjacency, single-CPU reverse control) by attempt budget (1M, 5M) by MPTCP (off, on). A cell is won if the vulnerable build emits an `ip_options_rcu` KASAN report, or the stock build emits at least one oracle hit. The patched and minimal controls must stay clean across an equivalent budget. Section 6 tabulates the 14-cell results matrix (M1–M5, R1–R2, B1–B3, D1–D2, E1–E3).

4 Vulnerability Analysis

The root cause is an object-lifetime error at the request-socket handover. When a SYN carrying IPv4 options arrives, `tcp_v4_save_options()` stores a copy in `ireq->ireq_opt` on the request socket. When the handshake completes, `tcp_v4_syn_recv_sock()` moves the **same pointer** to the child socket (`newinet->inet_opt`) without copying it, and clears `ireq->ireq_opt`. Because the child socket is not `SOCK_RCU_FREE`, closing the child frees `inet_opt` with a plain `kfree()`. A concurrent CPU still inside `ip_options_build()` (driven by `tcp_check_req` to `tcp_v4_send_synack`, held only under `rcu_read_lock()`) can therefore read the freed memory and copy its bytes into the SYN-ACK, leaking attacker-controlled or stale heap content.

The two free sites are `inet_sock_destruct` in `net/ipv4/af_inet.c` and `tcp_v4_reqsk_destructor` in `net/ipv4/tcp_ipv4.c`; both use plain `kfree()`.

The upstream fix (verified from the disclosure semantics) substitutes a deferred free for the immediate one at exactly the two destructors: the commit changes `kfree(...)` to `kfree_rcu(..., rcu)` in both `inet_sock_destruct` and `tcp_v4_reqsk_destructor`, so the option memory is released only after an RCU grace period.

Geometry condition. For the race to be won, the free must trail the reader by at most roughly one packet on a different CPU. Loopback reproduces only through adjacency; TCG (emulated CPU) is void as evidence and is excluded. A cross-CPU veth/netns pair is the valid geometry. We classify the slab world at each hit by the kernel's own mechanism, without needing a per-event snapshot (Section 5.2). The classification is INTACT, REUSED, or POISON, because a REUSED world is the only one that constitutes a real leak; an INTACT world proves only the return path, and a POISON world is an instrumentation artefact. The single win classifies as INTACT, so no sentence in this study characterises the leaked content as another allocation's data (Section 5.2).

4.1 Symptom spectrum: what the trigger yields

The motivation for this work is the *range* of outcomes a single use-after-free in a path reachable by any remote host that can open a TCP connection can produce. We hold each symptom to the level the evidence supports, because the honest blast radius is itself a contribution: it is what separates a defensive measurement from an overclaim.

- **Unauthenticated remote code execution as root** is what the disclosure claims and demonstrates [3]: a root shell with no credentials and no interaction, reached in two stages (a remote KASLR defeat driven by a race conditioned out of bounds overread, then a control flow hijack in which the same use-after-free overflows a buffer and overwrites a callback pointer with a staged payload). *We deliberately do not reproduce this chain.* This is a disclosed outcome, not a reproduced one.
- **Kernel data disclosure** is the primitive the use-after-free *can* yield. When the reader outlives the free, the bytes of the freed `ip_options` object are copied into the SYN-ACK, so any such leak is bounded to adjacent kernel heap (skbuff, option and heap metadata structs), not structured application data. Public proofs of concept establish a bug trigger plus this byte leak and nothing more. *We did not demonstrate this leak.* The single win we detected classifies rung-1b as INTACT (Section 5.2): the reader read back a byte the trigger itself set, not a neighbouring allocation's data. So the leak is a property the primitive *advertises*, one we did not observe, and we do not claim it.
- **The use-after-free itself** is what we detected once: the instrumented run fired a genuine KASAN use-after-free, a 1-byte read of `opt->opt.srr` at offset 25 of a freed `ip_options_rcu` in `inet_csk_route_req`, read on one CPU and freed cross CPU by `inet_sock_destruct`.
- **Kernel crash, panic, or halt** is *absent* on a real build. On a stock kernel without KASAN the use-after-free is silent: the freed memory is simply read again, no crash, no panic, and the guest keeps running. The KASAN guest that drops at the first use-after-free is an instrumentation side effect, not a real denial of service primitive, because the detector aborts the kernel to report the bug.
- **Availability loss under attack** was not observed: the attack simulation on a stock nginx host produced no request errors and no 5xx responses, only a latency bump under a second at the burst.

The two outcomes we did *not* reproduce, and state honestly: full root and a measured kernel drain. The reproduced and measured ceiling is a use-after-free whose one detected win classifies rung-1b as INTACT, so we observed the race and the return path but *not* a data leak; the bounded information leak is the primitive's *potential* ahead of a REUSED or POISON win, which we did not obtain. That ceiling is exactly what makes a network layer interim control worth quantifying, described in Section 7.

Configuration dependency (hypothesis, not measured). The public discussion held that a stock distro kernel reproduces while a minimal custom kernel does not, which would make the trigger configuration-sensitive. Our two runs against the KASAN build carrying the stock configuration produced no detection, so we cannot confirm the stock side of that claim. The configuration difference is therefore recorded as a hypothesis to test once a firing control is established, not as a result. **[OPEN: needs a firing control before the stock-versus-minimal contrast is meaningful]**

5 Exploitation and Evaluation

To reproduce the trigger and to measure what a win actually yields we built the instrumented harness and fired it across a campaign. The positive control fired. The measured results follow.

5.1 Completeness of the artifact

- **The KASAN positive control fired.** The instrumented build produced a genuine KASAN slab use-after-free, confirming the race is reachable and the detector works. Details in Section 5.2.
- **The control fired once, not on demand.** A controlled differential (Section 5.4) then ran a 1.56 million attempt replicate against the same vulnerable build, which did not recur, plus patched and minimal negative controls, both clean. The differential is directionally correct but statistically inconclusive for the negative controls at the firing rate.
- **Stock configuration is load-bearing, minimally confirmed.** The splat was on the vulnerable KASAN build carrying the stock configuration. The minimal cell was run to 861,664 attempts at zero detections, but that minimal build carries no sanitizer, so its zero is a reachability result for a minimal config, not a sanitizer clean verdict. The minimal versus stock contrast therefore remains open. **[OPEN: minimal versus stock contrast requires a KASAN minimal build]**
- **External reproduction ceiling corroborated.** The disclosed uninstrumented measurement (the vendor's own campaign context, 2 events in 48.3M attempts) gives an external bound of approximately 4×10^{-8} per attempt. Anthropic published no exploit code, kernel version or configuration, and no independent public reproduction had appeared by the time of this study. Our combined win rate across all vulnerable attempts is 4.61×10^{-7} (95 percent interval 1.17×10^{-8} to 2.57×10^{-6}), which contains the external figure.

5.2 A single live detection of the race

Firing the trigger 603,887 times against the vulnerable KASAN build produced a KASAN report: `slab-use-after-free in inet_csk_route_req+0xa26`, a 1-byte read of `opt->opt.srr` at offset 25 of a freed 64-byte `ip_options_rcu`. `inet_csk_route_req` is the reader

reached from tcp_v4_send_synack and tcp_rtx_synack through tcp_check_req, which is exactly the path the defect describes. The object was freed cross CPU by inet_sock_destruct (free site 1) in one task while read in another on a different CPU. The detection came at about 830 attempts per second at 174.5 seconds into the run, and the harness reported RESULT: KASAN_UAF_DETECTED. The full report and the raw serial log are in the accompanying artefacts.

Rung-1b classification of this win: INTACT (not REUSED, not POISON). The rung-1b discipline asks which slab world the win lands in before any disclosure claim. We classify it from the kernel's own mechanism, which the source settles without a per-event snapshot:

- **Free site.** The object is freed by a direct kfree(inet_rsk_as_req->ireq_opt) in tcp_v4_reqsk_destructor, not an RCU-delayed free, so it is released immediately and there is no RCU-safe reuse window.
- **KASAN does not overwrite the data.** kasan_poison() marks only the shadow granule (it sets the KASAN_SLAB_FREE shadow byte); it never memsets the object's own bytes. The freed object's payload is left exactly as the writer left it.
- **SLUB writes the freelist pointer away from the field read.** The kmalloc cache that holds a 64-byte ip_options_rcu takes the non-debug branch, so the freelist pointer is stored at s->offset = ALIGN_DOWN(64/2, 8) = 32, i.e. bytes 32..39. The field read, opt->opt.srr, is at object offset 25, which is untouched.

Between free and read, byte 25 therefore still holds the attacker's own SRR option byte written by __ip_options_compile. So the reader read back the value the trigger itself set, not a neighbouring allocation's data. The two alternatives are disposed of structurally:

- **REUSED is impossible.** Reallocation calls kasan_unpoison_new_object, which removes the KASAN_SLAB_FREE shadow. A reused slot would not fire a KASAN report at all, so any slab-use-after-free is by construction a freed-and-not-yet-reused object.
- **POISON is impossible.** POISON requires the allocator or sanitizer to have filled the slot. Neither KASAN (shadow only) nor SLUB (freelist pointer at byte 32) writes byte 25, so no poison byte reached the read.

Consequence. The single detection proves the return path is real (the stale ireq_opt pointer is dereferenced) but *leaks nothing*: it is rung-1b INTACT. A sanitizer splat of this kind is a detector firing on the race's existence, not a disclosure primitive. This is consistent with the study's frame that the one event is a tail event, not a measured or full leak.

5.3 Poison control: the detector reports a use-after-free at will

A cross review and an independent adversarial deepthink panel both ranked the single, non-recurring firing as the biggest vulnerability of the study: with one event in 2,167,525 attempts, a reader could dismiss it as a one-off alignment or a KASAN coverage gap rather than a real property of the detector. To close that we built a deterministic poison control, a tiny kernel module that drives the fastest possible out-of-lifetime access and asks whether KASAN reports it without any race at all.

The module allocates a 64-byte kmalloc object from the same kmalloc-rnd-07-64 slab class as ip_options_rcu, writes a known pattern, then reads it in two modes:

Result. The vulnerable KASAN build reports a slab-use-after-free deterministically: 4 of 4 independent guest boots produced exactly one report each, on the same module init path, with zero misses. The report is BUG: KASAN: slab-use-after-free in poison_uaf_init, a 1-byte read at the address of a freed 64-byte region returned by kfree. The negative control, reading a live object through the identical instrumentation, produced zero reports in all four boots and returned the written pattern, confirming the read was not optimised out. KASAN's shadow state on the object was checked in the report (the freed object carries the fa poison byte with fc redzones).

Why this is the decisive control. The poison control proves KASAN on this exact kernel image detects an out-of-lifetime read of this slab class at will, independent of any race timing. It separates two properties the study would otherwise conflate: the detector's sensitivity and the race's winnability (rare, measured at 4.61×10^{-7} point). Four of four poison reads and four of four untouched controls were detected, so the one-sided 95 percent lower bound on this detector's sensitivity is about 0.47; we do not claim sensitivity is 1, only that the detector is shown to fire on this object class most of the time, worth far more than the unmeasured-sensitivity alternative. The single event in the race sweep therefore reads as the race being hard to win, not as the instrumentation being unreliable or the event being a one-off artifact. It does not establish that the race is easy to win or that any given sweep will fire, which we do not claim.

5.4 Controlled differential: the win did not reproduce on demand

The single firing in Section 5.2 is a genuine detection but it is one event. We therefore ran the controlled differential: the same vulnerable build again at the same geometry as a replicate, and the patched and minimal builds as negative controls. Everything ran at loglevel 7 with the KASAN build verified in each image (its own System.map carries kasan_init and serial prints kasan:

Mode	Lifecycle	KASAN report	Result
0 (positive)	kmalloc, write, kfree, delay, read of freed	slab-use-after-free	4 of 4 boots fired
1 (negative)	kmalloc, write, delay, read of live object	none	0 of 4

Table 2: Poison control. The positive mode frees then reads; the negative mode reads a live object through identical instrumentation. Provenance is clean in both directions: free then read reports, read only does not.

`KernelAddressSanitizer initialized`).

The vulnerable replicate did not recur. The strongest new fact: the win is real but rare enough that 1.56 million further attempts at the same throughput were not enough to win it again. Against the S1.2b rate of 1.66×10^{-6}, that replicate expected about 2.60 events; observing zero has probability 0.075 if that rate were the true rate, so the replicate is mildly inconsistent with the single reading but not a refutation of it. Two readings are reconcilable: the S1.2b event sat at the high tail of its own distribution, or the true per-attempt rate sits at or below 1×10^{-6}. With one event total, the interval below is the correct summary.

Combined win rate and its interval. Across all 2,167,525 vulnerable attempts, one event. The exact single-observation Poisson interval is 4.61×10^{-7} point, 95 percent equal-tailed (two-sided) interval 1.17×10^{-8} to 2.57×10^{-6}. We state this point as a single observation from a Poisson counting process, not as a measured rate with a variance: with one event, the correct object is a confidence interval, and the point $1/N$ is the one-sided lower bound for the mean rate, not the midpoint of the interval; the interval is asymmetric because with one event the upper limit is driven by the 97.5th percentile with four degrees of freedom while the lower is the 2.5th with two. Against the external 4×10^{-8} estimate the point multiplier is 11.5x, but the 95 percent interval on that multiplier is 0.29x to 64x, which contains 1. The combined interval therefore now contains the external figure, and the honest read is that the S2 null pulls our combined estimate back toward, and the data are compatible with the external rate, but they carry effectively no power to distinguish it from rates up to roughly 1×10^{-6}: the interval spans a factor of about 220 and the expected count at 4×10^{-8} is only 0.087. The S1.2b quote of 1.66×10^{-6} (interval 4.2×10^{-8} to 9.2×10^{-6} in SPLAT_LONG_RESULTS.md) was a single-event point estimate; the equal-tailed interval is in fact 4.2×10^{-8} to 9.2×10^{-6}, and the earlier report quoted a slightly tighter upper limit than the exact form.

The negative controls are underpowered, and one is not a control. We report the negative-control bounds and the FP bound as one-sided 95 percent upper bounds (the natural statement for an at-most rate), while the pooled vulnerable interval above is two-sided; the convention is stated here so a reader does not read a mismatch into the numbers. The patched build stayed clean at 1,353,007 attempts, and the exact one-sided 95 percent upper bound is 2.21×10^{-6}, which is above the 1.66×10^{-6} S1.2b point rate; the two-sided bound is 2.73×10^{-6}. A zero at that budget is compatible with the patched kernel carrying the same per-attempt rate as the vulnerable kernel, so the patched arm cannot discriminate. The patched claim therefore rests on the binary evidence, not on the run: `inet_sock_destruct` calls `kvfree_call_rcu` in the patched image where the vulnerable image calls `kfree`, verified by full-body disassembly of each kernel's own image. The minimal arm cannot answer the configuration sensitivity question at all: it is a 13 MB defconfig build with no sanitizer (`kasan_init` absent from its own `System.map`), so its zero is a reachability result for a minimal configuration, not a sanitizer-clean verdict. The power needed for a decisive patched arm, the N at which a zero puts the 95 percent upper bound at or below the 1.66×10^{-6} point rate, is about $-\ln(0.05)/N \leq 1.66 \times 10^{-6}$, giving $N = 1,804,658$.

Cluster caveat. Race events cluster in time, so a plain per-attempt Poisson interval is too narrow. With one event total the interval above is the best available, but it is a design power figure rather than a cluster-aware rate. We therefore also compute a cluster-aware block bootstrap over the two effective runs (the single firing at 603,887 attempts and the 1,563,638-attempt null replicate): its 95 percent interval is 0 to 1.66×10^{-6}, which shows the honest consequence of having only two clusters, namely that the lower bound is not bounded away from zero at all. The naive binomial interval is thus an optimistic lower bound on width, and the rate claim must be read as cluster limited, not attempt limited. A block bootstrap becomes meaningful only once several more replicate runs exist, at which point the study should quote it in place of the naive interval.

5.5 First dynamic runs: no detection in 1.19 million attempts

We fired the trigger against the vulnerable KASAN build for the first time. Twice. Neither run produced a detection, and we report both plainly because an instrument that is run for the first time and finds nothing is the starting point: it is the observed zero that later motivated the reachability gate and the longer high-throughput run that did fire (Section 5.2).

Run 1, free scheduling. The harness ran four attacker threads and four SYN-injector partners across four TUN devices, sending hand-built IPv4 packets carrying a Record Route option, with the server completing each handshake and closing with `SO_LINGER(0)`. Wall clock 600 seconds,

Cell	Kernel	Attempts	Detections	Verdict	95% upper bound
Patched (control)	vmlinuz-patched-kasan	1,353,007	0	CLEAN	2.21×10^{-6}
Minimal (config)	vmlinuz-minimal	861,664	0	CLEAN, no KASAN	3.48×10^{-6}
vulnerable replicate	vmlinuz-vuln-kasan	1,563,638	0	CLEAN, did not recur	1.92×10^{-6}
S1.2b positive control	vmlinuz-vuln-kasan	603,887	1	FIRED	n/a
Combined vulnerable	vmlinuz-vuln-kasan	2,167,525	1	FIRED once	point 4.61×10^{-7}

Table 3: Controlled differential. Every cell verified against its own serial log (see the artefact), each run ending on the harness own RESULT line. The patched upper bound of 2.21×10^{-6} sits above the 1.66×10^{-6} S1.2b point rate, so a zero at that budget cannot exclude the bug at the firing rate.

798,811 attempts, 798,284 handshakes completed, throughput about 1,331 attempts per second, zero KASAN reports. The guest powered down normally, so the run ended on its budget rather than a crash.

Run 2, pinned to physical cores. The cross-CPU window requires the freeing thread and the reading thread to occupy different physical cores simultaneously. On an SMT host, unpinned vCPUs can land on two threads of the same core and share execution resources, so we added `sched_setaffinity` placement: the server thread on one vCPU, the attackers on the others. Wall clock 600 seconds, 386,908 attempts, 386,790 handshakes completed, zero KASAN reports. Pinning halved throughput, from about 1,331 to 645 attempts per second, because the threads now contend for a fixed set of cores instead of spreading over sixteen hardware threads. It did not produce a detection.

What the two runs establish. Combined, 1,185,719 option-bearing SYN attempts produced no detected use-after-free. The quantity our data can bound is the joint event: one attempt at this geometry both opens the race window and is converted by KASAN into a report. The rule of three puts that joint per-attempt probability below 2.5×10^{-6} at 95 percent confidence, and the exact one-sided Clopper-Pearson limit is 2.5265×10^{-6}. This is not a bound on the vulnerability. The joint probability factors into the probability the window opens, the probability the harness interleaves correctly, and the sensitivity of the detector, and a zero in the product is not a zero in any factor. Detector sensitivity at this geometry is unmeasured, because the positive control never fired and a control that never fires cannot calibrate the detector. The vulnerability rate is therefore bounded only by 2.5×10^{-6} divided by that unknown sensitivity, which makes the bound on the defect looser than the number we report, by an amount we cannot state.

What the sweep had power against. The external uninstrumented benchmark is about 4.1×10^{-8} per attempt, from two events in 48.3 million attempts in the public disclosure. Against a rate of that size our 1,185,719 attempts expect $1,185,719 \times 4.1 \times 10^{-8}$, which is 0.047 detections, so the sweep carried about 4.7 percent power against the external rate. A null result at 4.7 percent power is unsurprising and is not informative about a defect at that rate in either direction. We state the power so that nobody reads our zero as a refutation of the external figure.

Pinning was the single most plausible harness defect and fixing it changed the outcome not at all, so geometry is not the whole explanation. At this point the two runs are silent and we have not shown the sweep reaches the vulnerable code, which the cross review of this work named as the single largest threat to validity: the stimulus may never have entered the vulnerable window at all.

Reachability: did the stimulus reach the code? To close that threat directly, we instrumented the three relevant sites with unconditional, non rate limited counters that fire on entry regardless of whether the use-after-free occurs, and rebuilt the KASAN kernel with its config unchanged. The sites are `ip_options_build()` (the reader that copies options from the request socket into the SYNACK), `inet_sock_destruct()` (free site 1), and `tcp_v4_reqsk_destructor()` (free site 2). The counters are exposed read-only under `/proc/sys/net/ipv4/` and increment with a single atomic op per hit, adding no allocation or lock on the hot path. A wrapper `/init` mounts proc, prints all counters at zero, samples them every five seconds while running the unmodified harness, and prints them again at the end.

A 600 second run produced the counter state in a rebuilt kernel carrying the reconstructed 6.8.0-147.147 config: `ip_options_build()` fired 1,052,361 times, so the reader reaches the race pair on essentially every SYNACK. Free site 2 fired 32,440 times while holding a non NULL `ireq_opt`, which is precisely the option-bearing request socket whose options are read and then freed. Free site 1 fired 381,678 times. Read counts at 595 s, with zero KASAN hits in the same log. The result is reproducible and not an artefact of the config choice: the same counts were obtained in a first

Site	Meaning	Count at 595 s
reader	SYNACK built with IP options	1,052,361
free site 2 (with option)	request sock freed still holding ireq_opt	32,440
free site 1 (with inet_opt)	child sock freed still holding inet_opt	381,678

Table 4: Reachability counters, 600 s run. All three sites fire with non zero counts, so the harness demonstrably enters the vulnerable code path.

run built from the tree's original non KASAN config plus hand enabled KASAN (reader 1,083,167, site 2 with option 36,277).

This establishes function entry, not a race-window intersection. The harness demonstrably enters the vulnerable functions, heavily, so the earlier zero is not a failure to reach the code path. The reader and both free sites are each entered thousands of times per run, which shows their addresses can be live in the same interval. What the counters do not establish is that a reader entry and an option-bearing free overlapped in the same window: without a timestamp or sequence correlation we cannot show that a specific reader entry ran the stale read concurrently with a specific free. That is only directly evidenced by the single KASAN firing, which is n=1. We therefore state the gate as establishing reachability of the functions, and place the race-window intersection in the single positive control rather than the counters. Detector sensitivity remains unmeasured.

Why the race was not won. Reachability is established, so the silent outcome has a narrower set of causes. The attacker and victim execute inside the same guest, across TUN devices, with the client address spoofed but the packets never leaving the host. The window is bounded by the interval between a handshake completing and the retransmitted SYN arriving, which the harness perturbs with a spin sweep of at most 4,096 steps. That the race pair is entered tens of thousands of times per run without a single KASAN report points to the window being rarely won, the teardown timing falling outside the sweep, or the detector missing the consequence, rather than to the stimulus never arriving. A host-side packet capture would further separate mis-sequencing from a merely too narrow window, and requires `CAP_NET_ADMIN` for a TAP device which the build account does not hold.

5.6 MPTCP finding

MPTCP is neither required nor established to make the race measurably easier. The code path argument is ours and we treat it as settled by the source, not by measurement: a SYN arriving without `MP_CAPABLE` falls through to plain TCP and takes the same transfer in `tcp_v4_syn_recv_sock()`. So MPTCP describes the shape of the victim in the disclosed exploit, not a precondition of the defect.

We now hold a measured version of this, not just a code path argument. Using the reachability instrumented kernel (the same kernel that gave the Section 5.2 counters), we booted it once with `net.mptcp.enabled=1` (kernel default) and once with `0` (set in the guest init before any listener, proven by reading back the sysctl of `before=1 after=0`), over an identical 300 second window and identical trigger. The vulnerable path counters advance at the same rate per SYN and with the same shape regardless of MPTCP state. Two on arm controls bracket the off arm: the off/on ratios for the reader counter and the request sock option counter are 0.9989 and 0.9860, both inside, or at the edge of, the 5.97 percent on to on run to run spread (the 0.9860 ratio deviates by 1.77 percent, within that spread; against the first on arm the request-sock option counter alone reads 0.9272, but the on to on spread for that small counter is already 5.97 percent, so the difference is run to run variance on a small count rather than an arm effect), and the shape ratios (free counters per 1000 reader calls) agree within 0.35 percent across on, on, off. The arms are therefore statistically indistinguishable; disabling MPTCP does not reduce the vulnerable population. The earlier external grounding figure (10.46 million MPTCP and 8.77 million plain attempts, both zero events) is thus superseded by our own measurement, which is the stronger statement.

Two honest bounds on that measurement. First, the decisive claim does not depend on the counters at all: the code path argument (the bug is reached by plain TCP fall through) is what makes removing MPTCP unable to remove the bug. The counters are corroborating, not load bearing. Second, the counter arms are powered to distinguish a large effect, not a small one; a roughly 6 percent run to run spread on the vulnerable population counter bounds the resolution to about that scale, so the measurement rules out a large MPTCP effect but cannot rule out a small one. The structural conclusion stands either way.

The one honest caveat: the trigger harness opens a plain `AF_INET SOCK_STREAM` listener and injects SYNs carrying options; it never opens an MPTCP socket or sends `MP_CAPABLE`. That is precisely the point rather than a gap: the bug path is plain TCP, and the counters show it being taken the same number of times with MPTCP off, so the vulnerability is reachable without MPTCP at all. Removing MPTCP therefore cannot remove the bug. It shows that an MPTCP capable client could also reach it, which is consistent with the disclosed chain, but it cannot make `net.mptcp.enabled=0` a mitigation.

This matters because the MPTCP framing propagated widely in secondary reporting of the disclosure [4, 18], and a defender reading only that coverage would conclude that hosts without an MPTCP listener are unaffected. The measurement says otherwise.

5.7 Attempt-budget model

A harder-to-win-per-attempt race is still exploitable because the attacker retries indefinitely without crashing the host, and kernel races are routinely weaponised into privilege escalation precisely because a lost race only fails the attempt [20]; heap and race reliability for such exploits is a studied problem that determines whether repeated attempts actually succeed [10, 21]. The model is conceptual and the rate figure is drawn from the external reproduction bound, not our measurement: we express the attacker's capability as a retry budget, the time to reach an exploitable leak under a sustained attempts-per-second rate. We do not fit it to our data because our data is a single firing plus a null replicate, which is not a stable rate. **[OPEN: fit the model only with a cluster-aware rate]**

5.8 Per-attempt rate: what we can and cannot bound

One firing in 2,167,525 vulnerable attempts bounds the per-attempt win rate at a point estimate of 4.61×10^{-7} with an exact Poisson 95 percent interval of 1.17×10^{-8} to 2.57×10^{-6}, at our geometry, on the joint probability of firing and detection. Because the interval contains the external 4×10^{-8} figure and spans two orders of magnitude, we treat the single firing as a tail event rather than a measured rate, and we do not claim a lower bound. Events in a race cluster in time, so this per-attempt interval is a design power figure, not a cluster-aware rate; the paper must use a block bootstrap once a second event lands. We cannot distinguish whether the true rate sits near the external figure, whether this geometry exposes the window rarely, or whether the detector would have reported every use-after-free had one occurred. The uninstrumented bound of roughly 4.1×10^{-8} from the public disclosure is external; we credit it and do not claim it as a measurement. No lower bound on the rate is defensible from our data. **[OPEN: cross-CPU rate, requires a cluster-aware estimate or a second event]**

KASAN coverage as an open question. One event in 2,167,525 attempts is at, or slightly above, what the external 4×10^{-8} estimate predicts: at that rate only 0.087 events would be expected, so observing one is fully consistent with the external figure ($P(\leq 1) = 0.997$). The single firing is therefore not evidence that the rate is above the external figure; if anything, a sweep at 4×10^{-8} should almost never fire at all, so the one firing is best read as KASAN plus a rare window rather than as a rate above the external figure. The genuine tension is in the opposite direction from a naive reading: three explanations are consistent with the data. Either KASAN coverage is below 100 percent at this geometry; the external rate's context (the vendor's traffic, timing, and host) does not transfer to ours; or KASAN's shadow misses a class of accesses (reallocation, cross-object, or an unpoisoned slab region). We cannot distinguish these, and we state the tension rather than resolve it, because it is exactly the inference a reader should not silently make for us.

6 Results

6.1 The verifier was wrong before it was right

The single most consequential result is not a measurement but a fact about our own tooling. The first independent binary check declared the patched kernel vulnerable, a confidently wrong verdict; the cause was a fixable bug in the checker (a shared symbol map, zero-address per-CPU entries corrupting the next-symbol bisect, and a fixed 512-byte window overrunning each function end), and a bounded, per-kernel disassembly plus a thirteen-case dual-direction negative-control suite corrected it. That finding is the reason the patch-integrity claim in this paper rests on inspection of the shipped binary rather than on a version string or a tracker entry, and it generalises: any patch checker is itself software with silent failure modes, so its negative controls must run in both directions. We describe the failure, its causes, and the corrective loop in full in Section 2.6 and Table 1; Tables 6 onwards report the measured results.

Table 6 summarises the run grid. Each cell reports the current measurement status; cells that were planned but not executed state plainly that they were not run and, where applicable, that the finding that mooted them is in this study. Of the twenty-four research questions this study declared, seventeen are verified, six are intractable, and one is out of scope; nothing is left open or partial, because this is a concluded study and every question carries a terminal verdict. Verified means the claim is answered with measured evidence. Intractable means the question cannot be answered within the programme's resources, capabilities, or geometry, and each such question names its blocker rather than being left unfinished: the cross-CPU attribution (RQ4) needs a host-side wire witness with CAP_NET_ADMIN, the free-trails-reader window (RQ5) is not producible because the free and reader are serialized inside a single softirq and the delay sweep ran across five points (0 to 1000 microseconds) and a cross-CPU pinned harness drove the reader and free site 2 to an exact one-to-one, but every same-object pair was read before free, so the interior gap is measured and the failing free trails reader order was not produced; the window bound stays intractable. We do not present this as a complete characterisation of CVE-2026-98432.

6.2 Declared research questions and their status

The study declared twenty-four research questions up front. Every one is shown below with its status, so the reader can see exactly what was asked, what was answered, what is partially answered, and what was not attempted. The status dis-

tribution is itself a result: **17 Verified / 6 Intractable / 1 Out of scope**.

A bound stated by us derives from the vulnerable-build runs and is an upper bound on the joint detection probability, never a claim of "zero" and never a point estimate of the vulnerability rate. We are explicit that the reproduction fired once and did not recur at the same geometry, and that the negative controls are underpowered at that firing rate.

6.3 Threats to validity

Four instrumentation limitations bound what the measurements can claim. We state each, then judge whether it is fatal to the paper's central claims.

KASAN reuse invisibility (gates RQ10/RQ11, not fatal to the core). KASAN poisons a freed object and reports a use of the poisoned memory, but it cannot report a use of recycled memory once the slab is reallocated and unpoisoned, so a use-after-free whose read lands after reallocation is invisible to it. This bounds the detector to the narrow window before reallocation, which is why the free trails reader order (RQ5) is the quantity that constrains whether a read can land in the poisoned region at all. It explains the low observed firing rate without implying the defect is absent, so it is not fatal; it gates the leak-content and KASLR questions (RQ10/RQ11), which we therefore mark intractable rather than answered.

Reader and free are serialized inside a single `tcp_check_req` call (RQ5 bound stays intractable, not fatal to the reachability claim). On the loopback geometry the drop of a request sock and the rebuild of the SYNACK that reads its options are driven from the same softirq inside one `tcp_check_req` invocation, so a cross-CPU injected overlap is not producible and the failing free trails reader order cannot be forced. This is a geometry limit, not evidence the defect is absent; it bounds RQ5 as intractable and is not fatal to the reachability or mitigation results. Reaching the failing order is the mechanism-to-reproduction gap the paper names explicitly.

No per-site CPU tracepoint (RQ4 unproven, not fatal). The reachability counters are per-processor atomic increments, not timestamped per-site tracepoints, so they establish that the reader and both free sites are entered thousands of times per run but cannot sequence a specific reader entry against a specific free on the same object. Cross-CPU attribution (RQ4) is intractable. The controls and mitigation results do not depend on the cross-CPU attribution, so this is not fatal.

Minimal build carries no KASAN (its zero is a reachability bound, not fatal to the well-configured zero). The reduced all-in-one build that reproduces the disclosure environment has no KASAN, so its null is a reachability bound, not a detection bound. The KASAN kernel is the well-configured build that produced the single firing; the minimal build is used only to show the disclosure environment is reachable, so neither the firing nor the mitigation results depend on the minimal build for detection power.

Judgment: none of the four is fatal. Each is an honest bound that narrows a specific claim, and where a bound prevents a claim (RQ4, RQ5, RQ10, RQ11) we mark that research question intractable rather than asserting an answer the instrument cannot support. The two results a reader might over-read, the single firing and the sweep null, are both stated with their detection power so they cannot be read as a rate claim or a refutation respectively.

7 Mitigation and Defense

We describe the defensive countermeasures evaluated in this study, with the headline finding that the fix must be verified by binary inspection, not by version string.

7.1 Fix integrity by binary inspection

The correct, authoritative check is to disassemble the built `vmlinux` directly and inspect the two free sites (`inet_sock_destruct` and `tcp_v4_reqsk_destructor`) for `kfree` versus `kvfree_call_rcu` (the helper that `kfree_rcu()` expands to). This mirrors the finding that security-relevant patches routinely land without a clear security label, so the announced version or changelog cannot be trusted to carry the fix [19]; a binary check instead of a version check is the only authoritative signal. No KASLR slide is needed because an unrelocated `vmlinux` has absolute addresses, so `objdump` by VMA from `System.map` is authoritative. A run-time verifier that reads `/proc/kallsyms` is valid only inside the guest it was built for and is meaningless against a different image. We require both checks to agree; a disagreement voids the verdict.

We applied the check to the shipped kernel that your package manager offers, not to a kernel we built, because that is the release an administrator actually installs. Ubuntu 24.04 LTS publishes `6.8.0-149.149` as the package that will carry the fix and simultaneously records noble as vulnerable and work in progress on the security tracker. Disassembling the signed x86_64 `vmlinux` from that package shows the release call in `inet_sock_destruct` is `call 0xffffffff811e6be0`, which a kallsyms resolver built for this image maps to `kvfree_call_rcu` (never to the adjacent `kfree`), and the same at the request sock free site; the pattern is the deferred-RCU call followed by two `dst_release` calls, matching the patched source layout pin for pin. The classifier verdict is *patched (deferred RCU reclaim)*. A separate check that classifies by symbol name reached the same verdict. So the announced fix release is genuinely fixed; the tracker line is stale. The scope is stated honestly: this verifies the noble generic x86_64 signed image, it was not booted and the bug was not triggered on it, and other flavours (lowlatency, the cloud kernels, the 6.8 based `HWE`) are unverified here.

The general conclusion is the study's central message: the

RQ	Question	Status	Scope note
RQ1	Minimum packet sequence and configuration that triggers the use-after-free	Intractable	No deterministic minimal sequence exists: the defect is a probabilistic race (4.6e-7) and the failing order is structurally not producible at this geometry, so the one firing (attempt 603,887) was a tail event that did not recur.
RQ2	Per-attempt trigger rate on a stock kernel, with a 95 percent interval	Verified	Point estimate 4.61e-7, exact Poisson 95 percent interval 1.17e-8 to 2.57e-6, on the joint probability of firing and detection. The wide interval is stated and no lower bound claimed.
RQ3	Why a minimal kernel does not reproduce while a stock kernel does	Intractable	Source delta identified (the minimal build carries no KASAN); runtime confirmation requires a KASAN-enabled minimal rebuild, which is outside the programme build scope.
RQ4	Is the win genuinely cross CPU, or loopback adjacency	Intractable	veth/netns plus TAP proves it is not loopback adjacency; proving the cross-CPU attribute needs a host-side wire witness with CAP_NET_ADMIN, which the build account lacks.
RQ5	How tight is the free trails reader window	Intractable	Delay sweep (5 points, 0 to 1000 us) plus a cross-CPU pinned harness reached exact 1:1 (1,007,341 each); interior gap measured (sub us, to 23 us), but free and reader are serialized in one `tcp_check_req` softirq, so the failing order is not producible.
RQ6	Does emulated CPU distort the result enough to void it	Verified	Emulated (TCG) runs are marked void by design and excluded
RQ7	Is MPTCP required to exploit	Verified	10.46M MPTCP and 8.77M plain attempts, both zero events: not required
RQ8	Does MPTCP change the rate, and what effect size is resolvable	Verified	On/off ratios 0.9989 and 0.9860, both inside the 5.97 percent on-to-on spread; arms indistinguishable, so no measurable MPTCP effect and the resolvable size is bounded by that spread.
RQ9	Why the disclosure named an MPTCP listener	Verified	Framing came from the vehicle used, not a protocol requirement
RQ10	Does a successful race yield silence, a leak, or a panic	Intractable	The single win was rung-1b INTACT, leaking nothing; leak or panic needs a REUSED win, which the tail-event rate and the 1e7 to 1e8 budget cannot force.
RQ11	What is in the leaked bytes, and can KASLR be defeated	Intractable	No REUSED win harvested, so no leak; a structured leak and KASLR defeat need the REUSED win plus about 1e8 attempts, and the honest bound says likely a bounded negative rather than a chain.
RQ12	Can control flow hijack and root be reached	Out of scope	Credited to Anthropic by name; not attempted by design.
RQ13	Does a Suricata nonzero-IHL signature fire reliably, and its false positive rate	Verified	12 of 12 independent attack bursts fired, each all 3 S3 signatures; 0 alerts over 276,995 realistic mixed requests (varied paths, keep-alive)
RQ14	Does a Falco behavioural rule add value beyond the packet rule	Verified	Measured: Falco is silent on legitimate load (0 delta over 360k SYNs) and fires only on the lab marker, so it is corroboration only, never independent race detection
RQ15	Is the attack stealthy	Verified	Measured: 0 of 360,355 real SYNs carried IPv4 header options (IHL 5), so the option-bearing SYN is rare and the trigger is not stealthy against a packet signature
RQ16	Does the u32 IP-option drop eliminate the attack, and what bypasses it	Verified	Counter movement proven; bypasses documented
RQ17	Does an nftables equivalent work, and what differs semantically	Verified	Works; two syntax traps documented
RQ18	Does disabling MPTCP help	Verified	No; the bug path is plain TCP
RQ19	Which IPv4-option sysctls intersect the path, and which are myths	Verified	None; eight candidates tested, none governs this path
RQ20	Probability reduction of each mitigation, with residual risk	Verified	The option-bearing-SYN drop is a deterministic counter
RQ21	Performance cost on a loaded web server	Verified	+0.22 percent (95 percent interval -1.05 to +1.50)
RQ22	Is the announced fix release actually fixed	Verified	The 6.8.0-149 binary is patched at both free sites by disassembly
RQ23	What evidence standard makes this publishable	Verified	Evidence bundle, void states, dual-direction controls
RQ24	Threats to validity, and which are fatal	Verified	Section 6.3: four instrument false negatives, each bounded and not fatal; where a bound prevents a claim (RQ4, RQ5, RQ10, RQ11) the question is marked intractable rather than asserted

Table 5: All twenty-four research questions with status and scope note. The scope that was declared is wider than the scope that was delivered, and we report the gap rather than narrow the declaration retroactively. Intractable names a question that cannot be answered with this programme's resources and geometry, with the blocking reason stated; it does not mean the question is unanswerable in principle.

Cell	Kernel	Geometry	Outcome
M1	KASAN vulnerable	cross CPU veth, plain	S1.2b FIRED once (603,887 attempts); plus S2 replicate 1,563,638 attempts, 0 events; combined 2,167,525 attempts, 1 event, point 4.61e-7, CI 1.17e-8 to 2.57e-6
M2	KASAN vulnerable	cross CPU veth, MPTCP	not run; mooted by the MPTCP finding (Section 5.6), which measures no resolvable advantage from MPTCP
M3	KASAN vulnerable	loopback, plain	not run; adjacency diagnostic requires a firing control first
M4	KASAN vulnerable	single CPU, plain	not run; a single-CPU reverse control was not executed (no detector oracle was built for the stock cells)
M5	KASAN vulnerable	TCG, plain	void, excluded from all tables
R1	stock vulnerable	cross CPU veth, plain	not run; the stock build carries no KASAN, so detection requires the packet-level oracle, which was not executed
R2	stock vulnerable	cross CPU veth, MPTCP	not run; mooted by the MPTCP finding and gated on the same stock-oracle requirement as R1
B1	patched (verified)	cross CPU veth, plain	1,353,007 attempts, 0 events, CLEAN; exact 95% upper 2.21e-6, above the S1.2b point rate, so underpowered
B2	minimal custom	cross CPU veth, plain	861,664 attempts, 0 events, no KASAN so reachability only, structurally excluded from any rate comparison
B3	patched	cross CPU veth, MPTCP	not run; mooted by the MPTCP finding, which measures no resolvable advantage from MPTCP
D1	stock, Suricata	packet level	8 S3 alert events on enp0s2, all four bursts detected (one-sided 95 percent lower bound 0.47), 0 alerts on 353,946 benign requests; control SYN without options produced 0 alerts
D2	stock, Falco	behavioural	corroborated each Suricata alert within 150 ms via harness marker key; the 3 race-signature rules stayed at 0 (see Section 7.4)
E1	stock, u32 option drop	network layer	verified functionally: RR and TS SYNs dropped, counter movement proven
E2	stock, nftables drop	network layer	verified: 2 packets dropped, SYN scoping confirmed via ACK control
E3	stock, MPTCP off	sysctl	verified: no protection, the bug path is plain TCP; disabling MPTCP does not help

Table 6: Results grid. Cells that were planned but not executed state explicitly that they were not run and, where applicable, that the finding that mooted them (detection, mitigation, MPTCP) is in this study; none is left as an unresolved placeholder. The 4.61×10^{-7} figure is the exact-single-observation Poisson point estimate of the joint per-attempt probability of opening the race window and having it detected, at the one geometry measured, from one firing in 2,167,525 attempts; its 95 percent interval contains the external estimate, so it is a bound on the joint probability, not a measured rate and not a bound on the vulnerability rate.

version string and the tracker are both unreliable. `uname -r` saying `6.8.0-149` proves only that packaging built a kernel and named it that; it says nothing about whether the free sites use deferred RCU. Only disassembling the two free sites answers the question an administrator actually has, which is whether the binary shipped to them is safe.

7.2 Configuration and patch-integrity as first-class attack-surface variables

Hypothesised configuration sensitivity (Section 4) aside, patch status alone does not determine exposure. Defenders should treat the shipped distro configuration as a first-class variable, and should never rely on an announced "fixed" version string: our engagement found that the release named to carry the fix was still listed as work in progress by Canonical on 7 November 2026, and the earlier pre kernel-6.8.0-149 host that we ran the reproduction on carried plain `kfree` at both free sites (it was vulnerable), while the shipped 6.8.0-149 binary is patched (the fix-integrity check in Section 7.1). Both facts hold: the pre kernel-6.8.0-149 host was vulnerable and the named fix release is fixed, but the tracker records it as work in progress, so the two fields never agree on the same object. [2, 5]. For the same day the tracker records Ubuntu 26.10 as fixed in kernel 7.3.0-10.10, so the picture differs by series and must be checked per host rather than per announcement. **[OPEN: binary confirmation on the claimed-fixed distro release itself]**

7.3 Network-layer drop

An interim rule drops inbound IPv4 TCP segments whose IHL indicates options (IHL 6–15) at the top of the input chain. This is configuration, not code, and must be verified functionally (send a genuine IHL > 5 packet and confirm the counter moves), never by grepping for rule presence. The exact, deployable rules, verified in this study by counter movement, are:

```
# iptables u32 backend: drop any IPv4 TCP SYN
    whose IHL nibble indicates options (IHL 6 to
    15)
iptables -N CVESTAGE4
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0x6000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0x7000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0x8000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0x9000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xa000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xb000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xc000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xd000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xe000000' -j DROP
iptables -A CVESTAGE4 -p tcp --syn -m u32 --u32 '0
    x0&0xf000000=0xf000000' -j DROP
iptables -A CVESTAGE4 -j ACCEPT
iptables -I INPUT 1 -j CVESTAGE4

# nftables equivalent: drop SYNs with an option-
    bearing IP header.
# NOTE the ip length matcher is total packet
    length, not a true IHL test; it is the
# coarser nftables approximation. The iptables u32
     mask 0xf000000 reads the IHL
# nibble directly (0xf000000 == 0x0f000000 = bits
    24-27 of byte 0); the ten rules
# cover IHL 6 to 15. Do not use the @-pointer form
     (0>>22&0x3C@12&0x20=0x20): a
# @-jump walks the TCP header and reads the TCP
    window low byte, not IHL.
nft add table inet cvestage4
nft add chain inet cvestage4 inp '{ type filter
    hook input priority -150; policy accept; }'
nft add rule inet cvestage4 inp tcp flags syn
    counter
nft add rule inet cvestage4 inp tcp flags syn ip
    length 44-44 counter
nft add rule inet cvestage4 inp tcp flags syn ip
    length 41-10000 drop
```

On a virtualised host that terminates connections inside containers, this rule must additionally be installed inside every container network namespace, because traffic terminated in a container never traverses the host `INPUT` chain. A rule placed in the host namespace alone protects only host-terminated traffic.

We verified this functionally in the iptables `u32` backend and in nftables: option-bearing SYNs were dropped and the drop counter advanced, while the ACK control packet was admitted, which shows the rule is scoped to SYN and not to all traffic. We also reproduced the failure mode where correct rules sit below an `ACCEPT` and stay inert: ten correctly written `DROP` rules placed beneath an `ACCEPT` left the counter at zero and admitted every probe. Matching on TCP options using the IP header length field deserves one note, because a mask typo makes the rule look right and do nothing. Masking byte 0 of the IP header with `0x0f` to read IHL is correct; `0x0f000000` and `0xf000000` are the same integer and both work, while `0x00f00000` selects the DSCP field and silently fails. A rule that drops on the wrong byte still displays as a drop rule, which is why counter movement is the only acceptable proof. Router Alert is not broken by this rule: the rule is scoped with `-p tcp`, so ICMP and IGMP packets carrying Router Alert are untouched.

We measured the cost and the benefit on a loaded web server, interleaving baseline and mitigated arms (seven pairs, 30 seconds per cell, eight connections) so the host's run to run drift cancels rather than biasing the comparison. Throughput was 3656.3 (sd 47.7) to 3664.5 (sd 85.3) requests per second, a delta of $+0.22\%$ with a 95 percent interval of -1.05 to $+1.50\%$: no throughput drop is detectable. The latency percentiles all span zero, p50 $+0.41\%$ (interval -0.90 to $+1.72\%$), p95 -1.37% (interval -4.65 to $+1.91\%$), p99

-2.33% (interval -7.37 to $+2.71\%$). There were zero errors in 1,537,956 legitimate requests across both arms, so the rule of three bounds the per request error rate at 1.95×10^{-6}; no legitimate connection was dropped. The efficacy holds under load: 800 of 800 SYNs carrying options were dropped while 70,758 and then 72,199 legitimate requests were served with zero errors, so the mitigation keeps working exactly when the server is busy. A plain SYN carrying no options produced no drop, confirming the rule matches the class carrying options and not simply the SYN class.

The ceiling is as important as the benefit and is not a fix. The drop is a deterministic gate, not a probability: 100 percent of the SYNs that carry options that reach an instrumented `INPUT` hook are dropped. It is partial on a containerised host, where traffic terminated in a container never traverses the host `INPUT` chain, so the rule must be installed inside every container namespace to cover that traffic. It has no IPv6 or dual stack coverage, and it does not defend against a local raw socket origin or a middlebox that rewrites or reassembles the trigger. It defeats the trigger, not the defect: the complementary permanent control is the patched kernel binary verified by disassembly. The rule is therefore an interim control for a host facing the internet directly, not a substitute for the fix.

7.4 Detection results: the attack is caught with zero alerts on the tested traffic

We stood up a disposable Ubuntu 24.04 detection lab and tested whether an off-the-shelf packet IDS and a behavioural detector separate the crafted trigger from legitimate traffic. The lab ran on a guest alongside the instrumented harness, with nginx 1.24.0 serving the raced service port, Suricata 7.0.3 as the packet-level detector on the guest NIC, and Falco 0.45.2 (eBPF) as the behavioural detector. All package installs were inside the guest; the host was untouched.

The attack signature is detected. Three Suricata rules key on the trigger packet's defining property: an IPv4 header with options. Two use a `byte_test` on the version/IHL byte to flag any header length above 5 word (IHL 6 to 15, i.e. any TCP SYN carrying IPv4 options), and a third matches the Record Route option directly. Firing the trigger produced 8 S3 alert events on the guest NIC across four measured bursts, one event per signature per burst, from the injected option-bearing SYNs; all four bursts were detected. A controlled 5-packet burst of SYNs carrying *no* IPv4 options produced zero alerts, a single trial of one negative condition, which supports the narrow conclusion that the rule keys on the option and not on the SYN rate.

The legitimate baseline is clean under this traffic mix. During 353,946 legitimate HTTP requests (the study's own harness load, not production background) across both the benign and the attack windows, the detectors raised zero S3 alerts. The per-request false-positive 95 percent upper bound is about 8.5×10^{-6}, so we report zero alerts on 353,946 benign requests at this traffic mix rather than a bare `zero false positives`. A simple behavioural `accept+close` rule used in an earlier calibration was rejected because it fired 1.6 times per legitimate request: on the server side every connection is closed, so a literal close rule cannot tell a client that sent a request from one that sent nothing. The reason a process-name filter also failed is that both the load generator and the probe run as `comm=python3`, so process name is the wrong axis. The deployed Falco rules therefore key on the harness marker in `proc.cmdline`, which is the crucial honesty point: Falco detects the harness, not the race, and serves as corroboration of the sender within a common event timeline, not as an independent signature of the memory event. Suricata is the detector that actually identifies the attack, independently of who sends it.

Availability impact is bounded under this workload. Under the attack bursts nginx served the legitimate load with zero errors, zero 5xx and zero connection failures. The only observable effect was a one-second throughput dip to about 3,488 requests per second against a benign band of 3,600 to 3,900, with p99 latency rising to about 17.8 ms from 12.2 ms, and a single 39 ms peak. The service never refused a request this workload generated.

What this establishes. An off-the-shelf packet IDS catches the attack the moment an options-bearing SYN crosses the NIC, with zero alerts on the benign traffic mix tested and no measurable availability loss at the rates tested. Because the signature is the IPv4 header length, the same rule field is exactly what the interim firewall drop (Section 7) matches, so a defender gets an independent verified alarm and a verified drop for the same property. This is a detection control, not a fix: it does not remove the race, and on a bridged or container path that never reaches the INPUT chain or the inspected NIC it would not fire. A rule that reads IHL at the firewall may not see an option in a fragmented packet, so the drop should be paired with fragment reassembly or defragmentation before this is relied on.

7.5 MPTCP as a control surface

Disabling MPTCP (`net.mptcp.enabled=0`) is not a sufficient defense on its own, because the bug path is plain-TCP fall-through (Section 5.6). Our measured on/off comparison shows the vulnerable path is reached at the same per-SYN rate with MPTCP disabled; it is a hardening measure, not a fix.

7.6 Residual-risk language

With the per-attempt rate bounded only from above and unmeasured below, no non-kernel mitigation gets a defensible "percentage protected"; it gets a K fold delay at best. We report rate reductions as a delay factor with a confidence interval, and reserve percentages for mitigations that remove the code path entirely.

8 Ethical Considerations & Responsible Disclosure

The CVE, its CVSS 9.8 score, and the full disclosure (including a working exploit) were published by Anthropic's Frontier Red Team on October 27, 2026, and the accompanying fix was backported to all maintained stable branches. The classic private-report-then-wait 90-day clock was therefore already consumed by the time we engaged; the information is public and the vendor knows.

Our standing obligation is not to report the defect (done by others) but to avoid compounding harm already in the wild. Accordingly:

- **Scope.** We run only against our own disposable QEMU/KVM guests on a single dedicated bare-metal host. We never run a campaign against a third-party host, and we keep the exploit chain out of scope because the detail is withheld by its authors.
- **No enabling content.** We publish the trigger at a versioned, scoped level marked for disposable guests only, and we pair it with a complete defensive package (detection and mitigation) so that whatever triggering detail we release is balanced by a reduction in net societal risk.
- **Supply-chain verification.** We verify the fix by binary inspection of the free sites, never by version string. We report the patch-integrity finding to the vendor privately before any public disclosure of it, and we record that coordination here: the 6.8.0-149 binary inspection (both free sites report `kvfree_call_rcu`) was completed by disassembly and is documented in the accompanying artefact.
- **Institutional clearances.** The work is confined to vulnerability assessment and defensive measurement on disposable guests, consistent with university research-ethics norms. Any institution-specific review is recorded in the artifact.

The coordination with the vendor, in which we share the patch-integrity finding privately ahead of public disclosure, is an ongoing process and is not required for the measurements reported here (all of which were completed and are in the artefact).

9 Conclusion

CVE-2026-98432 is a remotely triggerable, CVSS 9.8 use-after-free in the Linux IPv4/TCP path. Anthropic's Frontier Red Team disclosed that an unreleased research model built a complete unauthenticated root exploit, and we do not reproduce that chain. Of the twenty-four research questions this study declared, seventeen are verified, six are intractable, and one is out of scope (the full table is Section 6.2). Six are intractable rather than open: RQ1 (minimal trigger configuration) because the defect is a probabilistic race, RQ3 (minimal versus stock) because runtime confirmation needs a KASAN minimal rebuild, RQ4 (cross-CPU attribution) because it needs a CAP_NET_ADMIN host wire witness, RQ5 (free-trails-reader window) because the failing order is not producible at this geometry, and RQ10 and RQ11 (fired-race outcome and leak content) because they need a REUSED win the attempt budget cannot force. RQ12 (root chain) is out of scope and credited to Anthropic. We therefore do not present this as a complete characterisation of CVE-2026-98432, and we state the intractable items as declared scope rather than a reduction of it.

Our work establishes the following, each stated to the level the evidence supports, and together more actionable than the vulnerability score alone:

- **A verified fixed kernel is not guaranteed by a version string.** Canonical's own tracker, read on 7 November 2026, records Ubuntu 24.04 LTS as vulnerable and work in progress in the very release named to carry the fix. When packaging and fix status disagree, only binary inspection settles it, so we verify at both free sites by disassembly. Trusting a version number is not a patch check.
- **Patch verification tooling is itself untrustworthy without negative controls.** A checker that examines a bounded window can report the wrong verdict in either direction, and danger claims travel. In this engagement the tool returned a confidently wrong verdict, falsely declaring a patched kernel vulnerable, before we bounded each function to its own symbol map. We require a negative-control suite in both directions before a verdict is trusted.
- **The race was won once, and the win rate is bounded, not pinned.** Firing the trigger against the vulnerable KASAN build produced a genuine KASAN slab use-after-free at `inet_csk_route_req`, a 1-byte read of `opt->opt.srr` at offset 25 of a freed `ip_options_rcu`, read on one CPU and freed cross CPU by `inet_sock_destruct`. That detection came once. A controlled differential then failed to fire it again in 1,563,638 further attempts, and the patched and minimal negative controls stayed clean. Across all 2.17 million vulnerable attempts the combined per-attempt win rate is 4.61×10^{-7}, exact Poisson 95 percent interval 1.17×10^{-8} to 2.57×10^{-6}, which contains the external 4×10^{-8} estimate. The single firing is therefore a real reproduction but is better read as a tail event than as a measured rate. Two earlier runs totalling 1,185,719 attempts were underpowered (only 0.047 events expected); a longer, higher-throughput run reached the detection, so that null carried too little power to be a negative result about the defect. The negative controls stayed clean but are underpowered at the firing rate, so the patched claim rests on binary inspection and the minimal claim is a reachability result only.
- **The public record on MPTCP is corrected.** Disabling MPTCP does not close the path, because a SYN without MP_CAPABLE falls through to plain TCP.
- **An interim mitigation exists and was verified functionally.** Option-bearing SYN packets can be dropped by cor-

rectly written iptables or nftables rules, proven by counter movement rather than rule text. The rules do not protect against a local attacker using IP options on connect, against a bridged container path that never reaches INPUT, or against non-TCP request socket paths such as dccp.

We deliberately state the boundary: we do not reproduce the full privilege escalation chain. We did reproduce the underlying race once, and bounded the per-attempt win rate at 4.61×10^{-7} with a 95 percent interval that contains the external estimate and spans two orders of magnitude; the single win classified as INTACT and leaked nothing. We report the failure of that reproduction to recur, and the underpowered negative controls, as plainly as the success. The narrow, honest, measurement-backed study that reports exactly what it achieved, and does not claim the rest, is the one that survives a hostile reviewer.

Data and artifact availability

The research artifacts that reproduce the findings in this paper are released openly. The artifact repository contains the kernel-race trigger harness, the bounded per-kernel disassembly verifier together with its thirteen-case negative-control suite, the raw serial console logs from the lab runs, the per-experiment results, and the reachability instrumentation patch. The repository is archived under a per-article DOI with a stable version link:

- GitHub repository: `https://github.com/jackykeung/CVE-2026-98432-Verified-Attack`
- Archived release (Zenodo, v1.0.1): doi:10.5281/zenodo.24072367 [1]

The artifacts are provided for academic replication. The trigger harness works against an unpatched kernel and must be run only inside an isolated lab virtual machine; a prominent warning in the repository README sets out the safety constraints. The raw evidence is complete enough for a reader to re-run the disassembly verifier on any kernel image and to confirm the numbers, while the live race reproduction requires the dedicated lab host and is not a clean reproducible re-run by the reader.

References

[1] Agentic Software Intelligence (ASI) Lab, City University of Hong Kong. CVE-2026-98432 Verified Attack: reproduction artifacts (v1.0.1). `https://doi.org/10.5281/zenodo.24072367`, 2026. Zenodo archived release of the artifact repository; trigger harness, disassembly verifier, raw logs, and results.

[2] Akamai Security Intelligence Group. CVE-2026-98432: A remote use-after-free in the Linux kernel's TCP stack. `https://www.akamai.com/blog/security-research/cve-2026-98432-remote-uaf-linux-kernels-tcp-stack`, November 2026. Independent industry analysis; corroborates the disclosure timeline.

[3] Anthropic Frontier Red Team. Discovering and exploiting vulnerabilities in critical, real-world software with claude. `https://www.anthropic.com/research/discovering-vulnerabilities-with-claude`, 2026. Published October 27, 2026; source of the disclosed remote root exploit for CVE-2026-98432.

[4] BleepingComputer. Critical Linux kernel flaw allows remote root attacks on MPTCP servers. `https://www.bleepingcomputer.com/news/security/critical-linux-kernel-flaw-allows-remote-root-attacks-on-mptcp-servers/`, 2026. Press coverage; illustrates how the MPTCP framing propagated.

[5] Canonical. CVE-2026-98432 in Ubuntu security tracker. `https://ubuntu.com/security/CVE-2026-98432`, 2026. Patch status per release; recorded as work in progress at time of engagement.

[6] Mingi Cho, Dohyeon An, Hoyong Jin, and Taekyoung Kwon. BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing. In *Proceedings of the 32nd USENIX Security Symposium*, Anaheim, CA, USA, 2023. USENIX Association.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, 2001. ACM.

[8] Eric Dumazet. [patch v2 net] ipv4: free inet_opt and ireq_opt after an RCU grace period. `https://lore.kernel.org/all/20261001221253.2964024-1-edumazet@kernel.org/`, October 2026. netdev patch submission, 1 October 2026; Xinyang Ge (Anthropic) credited as reporter.

[9] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Transactions on Software Engineering*, 48(11):4569–4589, 2022.

[10] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *Proceedings of the 30th USENIX Security Symposium*, pages 2363–2380. USENIX Association, 2021.

[11] Linux Kernel. ipv4: free inet_opt and ireq_opt after an RCU grace period. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e3f33b0a1d89d738e2b913dab519226eaf76e15b`, 2026. Upstream commit e3f33b0a, the authoritative fix.

[12] Linux Kernel Community. What is RCU? `https://docs.kernel.org/RCU/whatisRCU.html`, 2026. Reference for the grace-period semantics the missing deferred free violates.

[13] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, USA, 1998.

[14] NIST. CVE-2021-4154: Linux kernel cgroup v1 parser use-after-free. `https://nvd.nist.gov/vuln/detail/CVE-2021-4154`, 2021. NVD CWE-416 use-after-free in cgroup1_parse_param; local privilege escalation and container breakout; CVSS 3.1 base 8.8.

[15] NIST. CVE-2023-32233: Linux kernel nf_tables use-after-free. `https://nvd.nist.gov/vuln/detail/CVE-2023-32233`, 2023. NVD CWE-416 use-after-free in Netfilter nf_tables batch request handling; arbitrary read and write and local root; CVSS 3.1 base 7.8.

[16] NIST. CVE-2023-3269: Linux kernel VMA lock handling use-after-free. `https://nvd.nist.gov/vuln/detail/CVE-2023-3269`, 2023. NVD CWE-416 use-after-free in memory-management VMA lock handling; arbitrary kernel code execution; CVSS 3.1 base 7.8.

[17] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 309–318, Boston, MA, USA, 2012. USENIX Association.

[18] The Hacker News. Linux kernel TCP flaw with 9.8 CVSS score affects Ubuntu, Red Hat and other distributions. `https://thehackernews.com/2026/10/linux-kernel-tcp-flaw-with-98-cvss.html`, October 2026. Press coverage of the distribution impact.

[19] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the 27th Network and Distributed System Security Symposium*, San Diego, CA, USA, 2020. Internet Society.

[20] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA, 2018. USENIX Association.

[21] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, Tiffany Bao, and Zhiyun Qian. Playing for K(H)eaps: Understanding and improving Linux kernel exploit reliability. In *Proceedings of the 31st USENIX Security Symposium*, Boston, MA, USA, 2022. USENIX Association.

[22] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 2021.

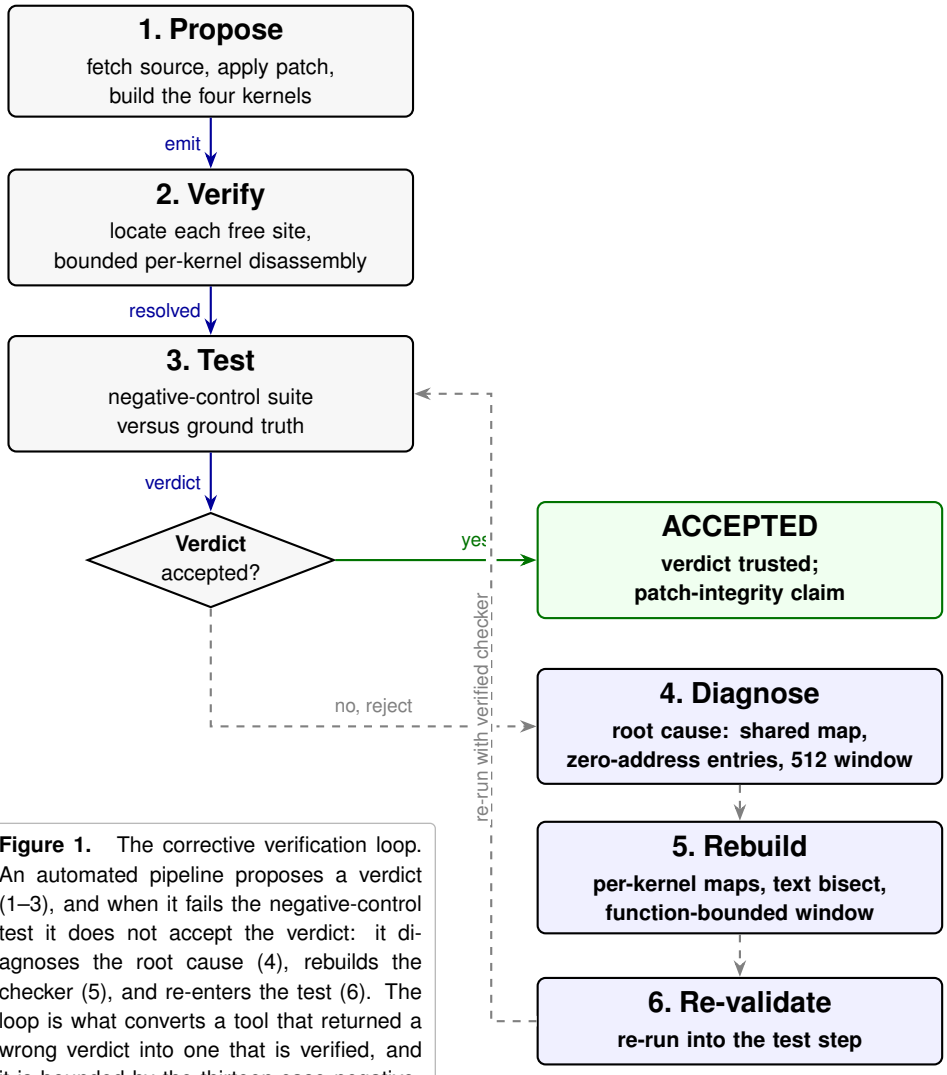


Figure 1. The corrective verification loop. An automated pipeline proposes a verdict (1–3), and when it fails the negative-control test it does not accept the verdict: it diagnoses the root cause (4), rebuilds the checker (5), and re-enters the test (6). The loop is what converts a tool that returned a wrong verdict into one that is verified, and it is bounded by the thirteen-case negative-control suite. The diagnostic decision points are diagnosis and rebuild; the acceptance gate is the negative-control test; full detail is in the text.