Patching at AI Pace

What Verified Security Submissions Should Look Like

Rajat Gupta

netdevconf 0x1A | Rome 2026

whoami

Rajat Gupta

Offensive Security @ Qualcomm

@z3ta_rjt

Reported multiple local privilege escalation exploits in Windows and Linux kernel

DEF CON CTF Finals — 2020, 2021, 2022

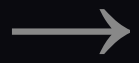
Recent passion: AI for Vulnerability Research

The Inbox Today

2024

2-3

reports / week



2026

5-10

reports / day

Mostly raw LLM output. No reproducer. No analysis.

The only differentiator is evidence quality.

4 Verification Gates

1. Trigger	Does the bug actually exist?	Yes
2. RCA Evidence	Do we know WHY it happens?	Partial
3. Patch Verification	Does the patch fix it?	Yes
4. Regression	Does the patch break anything else?	Yes

Each gate passed → higher priority. More evidence → faster review.

Gate 1 — Trigger + Impact

No trigger = lowest priority.

Sanitizer fires	Bug is real	Queue it
Controlled trigger	Reachable unprivileged	Real surface
Primitive demo	Attacker controls corruption	Exploitable
Full exploit	Privilege escalation	Drop everything

Automatable: Build + QEMU + run + check output

FAILS Gate 1

```
From: John Smith <jsmith@example.com>
Subject: [PATCH] net: fix use-after-free in foo()

I identified a potential use-after-free vulnerability in foo().
Under certain conditions, ptr may reference freed memory,
leading to kernel memory corruption and potential privilege
escalation.

The fix adds a reference count increment to prevent early free.

Signed-off-by: John Smith <jsmith@example.com>
---
 net/ipv4/foo.c | 1 +

+       refcount_inc(&ptr->refcnt);
        result = ptr->callback(skb);
```

"potential" — never triggered it. No reproducer. No crash log. No KASAN output.
What are the "certain conditions"? Wastes 30 min of reviewer time.

PASSES Gate 1 — Escalating Impact

```
LEVEL 1 — Bug is real (sanitizer confirms):
  $ ./trigger
  BUG: KASAN: slab-use-after-free in rb_next+0x14 (kmalloc-512)
  Call Trace: tcp_collapse_ofo_queue+0x1a4

LEVEL 2 — Reachable without privileges:
  $ unshare -Urn ./trigger
  [same KASAN splat from unprivileged user namespace]

LEVEL 3 — Attacker controls the corruption:
  $ ./trigger_controlled
  [*] Freed object reclaimed via slab spray (kmalloc-512)
  [*] Controlled RIP: 0x4141414141414141

LEVEL 4 — Full privilege escalation:
  $ ./exploit
  # id
  uid=0(root) gid=0(root) groups=0(root)
```

Each level tells you how fast to move. Level 1 is the minimum bar.

Gate 2 — Root Cause Evidence

Prose is not evidence.

Memory safety	KASAN/KMSAN backtrace at exact site
Data races	KCSAN report: both racing accesses
Arithmetic	kprobe showing exact values
Logic/semantic	Differential test: patched vs unpatched

Show me something I can run. Not something I must trust.

Partially automatable

FAILS Gate 2

```
From: John Smith <jsmith@example.com>
Subject: [PATCH] net: fix NULL deref in foo()

Root Cause:
  foo() dereferences ptr without checking if it's NULL.
  Under certain conditions ptr can be NULL, leading to
  a kernel oops.

Fix: add NULL check.

+       if (!ptr)
+               return -EINVAL;
        result = ptr->callback(skb);
```

This just restates the crash in English. "ptr is NULL" is the EFFECT, not the CAUSE.
What are the "certain conditions"? Who sets ptr to NULL? When? Why?

Trigger tells you WHERE it crashes. RCA must tell you WHY.

PASSES Gate 2 — Same bug, actual root cause

```
Root Cause: bar() sets sk->handler = NULL during teardown,
but the timer callback in foo() still holds a reference.

Trace (bpftrace):
  kprobe:bar   { printf("bar: setting handler=NULL, sk_state=%d\n", ...); }
  kprobe:foo   { printf("foo: reading handler=%p\n", ...); }

  [  12.001] bar: setting handler=NULL, sk_state=7 (TCP_CLOSE)
  [  12.003] foo: reading handler=0x0000000000000000 -> crash

Ordering: bar() nulls at line 342 BEFORE timer fires foo() at line 891.
The timer should have been cancelled in bar() before nulling the handler.

Fix: del_timer_sync(&sk->timer) in bar() BEFORE setting handler = NULL.
NOT: NULL check in foo() (hides the lifetime bug, timer still fires).
```

WHO nulled it (bar), WHEN (TCP_CLOSE teardown), WHY it's wrong (timer not cancelled),
WHERE to fix (bar, not foo). Reviewer verifies with bpftrace in 2 minutes.

Gate 3 — Patch Verification

"I think this fixes it" is not evidence.

Before patch

Trigger crashes

After patch

Trigger passes

Two kernel builds. One trigger. That's it.

Fully automatable — CI applies patch, builds, runs trigger, compares output

Gate 3 — Patch Verification

```
BEFORE PATCH (baseline — confirm bug exists):
  $ ./trigger
  BUG: KASAN: slab-out-of-bounds in nf_conntrack_find+0x62
  Bug is real.

AFTER PATCH (apply patch, same trigger):
  $ ./trigger
  [clean exit, no KASAN/UBSAN/lockdep splat after 1000 iterations]
  Bug is gone.

Still crashes after patch? -> low priority. Patch doesn't fix the bug.
Clean after patch? -> PASS.
```

Two kernel builds. One trigger. Fully automatable as CI.

Gate 4 — Regression

If you have QEMU for a trigger, you have QEMU for selftests.

FAILS

```
"I tested this on my development
machine running Ubuntu 24.04.
Everything works fine."
```

"Normal usage" = opened Firefox

PASSES

```
$ make kselftest TARGETS=net
  1..62
  ok 1 - tcp_mmap
  ok 2 - tcp_fastopen
  ...
  ok 62 - reuseport_bpf
  # pass:62 fail:0 skip:0
```

Reproducible. Anyone can re-run.

Fully automatable

The System

1	Build + run trigger in QEMU	No crash → low priority	Yes
2	Compare RCA evidence to trigger	Mismatch → flag for review	Partial
3	Apply patch, re-run trigger	Still crashes → low priority	Yes
4	Run subsystem selftests	Regressions → low priority	Yes

Fail any gate → low priority.

Pass all gates → arrives with evidence attached.

Honest Limitations

Eliminates

- Bug doesn't exist
- Hallucinated RCA
- Untested patches
- Zero-evidence reports

~80% of slop

Doesn't solve

- Symptom vs root cause
- Fix completeness
- Design decisions

Still needs human judgment.
But humans start from evidence,
not from scratch.

The Hard Truth About Gate 2

Both patches pass Gates 1, 3, and 4:

NULL check in foo()

```
BEFORE: crash  (bug is real)
AFTER:  no crash
Selftests: pass
```

Hides the bug. Timer still fires on dead object. Will crash elsewhere later.

del_timer_sync in bar()

```
BEFORE: crash  (bug is real)
AFTER:  no crash
Selftests: pass
```

Fixes the cause. Timer cancelled before teardown.

Gates 1, 3, and 4 cannot distinguish these two patches.

Gate 2 is where human judgment enters — but with evidence, not guesswork.

Thank You

Rajat Gupta

@z3ta_rjt

rjtgupta09@gmail.com