NETDEV 0X1A

NEBULA SECURITY

Finding Real Bugs Using AI

Lessons from LLM-Assisted Linux Kernel Bug Hunting

Yuan Tan · University of California, Riverside

VEGA
an LLM agent for kernel bug finding

VEGA Developers

SPEAKER

Yuan Tan

Nebula Security

Xiaochen Zou
Nebula Security · previously Microsoft

Yifan Wu
Nebula Security

Juefei Pu
University of California, Riverside

Xiaochuan Yu
Nebula Security

TEAM
TRACK RECORD

- **World's first** Nginx remote code execution
- **World's first** Android 17 full-chain exploit — from **browser to kernel**, rooting a device with a **single URL click**

We don’t just find bugs — we fix them

600+

Valid kernel bugs found — each with a working **PoC**

105

Patches merged into **mainline Linux**

10+ Exploitable for **local privilege escalation**

86

Patches in the **net / netfilter** subsystem

We find them, then we fix them — every patch closes a real bug, and **more are on the way.**

Bug-Fixing Volunteers

Xin Liu

Ren Wei

Zhengchuan Liang

VOLUNTEERS FROM

University of California, Riverside
Singapore Management University
...

- Jiexun Wang
- Luxiao Xu
- Yuqi Xu
- Nan Li
- Ruide Cao
- Yucheng Lu
- Zihan Xi
- Kai Ma
- Wyatt Feng
- Ruijie Li
- Longxuan Yu
- Yilin Zhu
- Haixin Xu
- Ao Zhou
- Junjun Fang
- Xiaonan Zhao
- Yang Yang
- Douya Le
- Haoze Xie
- Zhao Zhang
- Daming Li
- Linpu Yu
- Jie Wang
- Huihui Huang

What is VEGA?

VEGA is an agentic pipeline for large-scale bug hunting. It reads source at scale, forms hypotheses about defects, and then **tries to prove them**. Here we put it to work on the Linux kernel.

- LLMs generate **plausible** bug reports at scale.
- Spotting **suspicious code** is easy; the hard part is knowing which reports are **real**.
- VEGA is built around one idea: **verification over speculation.**

Is it real?

Can the bug actually be triggered on a running kernel, end to end?

Does it matter?

Does the trigger fit a realistic threat model?

How bad is it?

What is the *minimum privilege* an attacker needs?

SECTION 02 · THE CORE PROBLEM

Plausible ≠ Real

Turning AI output into real, in-scope, prioritized issues means clearing four hurdles.

CHALLENGE 1

Unreachable

The bug can't actually be triggered; the path is dead.

CHALLENGE 2

Out of threat model

Only fires under assumptions no real attacker has.

CHALLENGE 3

Severity

Root vs. namespace vs. unprivileged: very different risk.

CHALLENGE 4

Extreme edge cases

Technically real, but only in corners that need human judgment.

The bug that can never fire

The LLM finds “buggy” code, but the end-to-end trigger chain is broken.

- During audit, the model correctly zeroes in on **locally buggy code**.
- Yet the **path to reach it** is guarded, dead, or gated by an earlier check the model never modeled.
- Static reasoning alone **cannot** close this gap; it produces confident, unreachable reports.
- "Fixing" it anyway just **bloats the code** — dead defensive checks guarding a path that can never execute.

Symptom: a stream of well-argued reports no one can reproduce. The real question is whether *a live kernel can actually reach and detonate it.*

llm-report.txt

```
# LLM says:
"Potential use-after-free in
 foo_release(), refcount may
 drop to zero while queued."

# Looks convincing. But:
? is foo_release() reachable
? from unprivileged context
? with attacker timing control
-------------------------
Only a PoC settles it.
```

Make the agent crash the kernel

- Run the agent inside **QEMU with KASAN** and kernel debug checks enabled, and let it attempt to trigger the bug for real.
- Any **kernel panic report** — a KASAN splat, a `BUG()`, or a hard panic — is our ground-truth signal that the bug genuinely fired.
- No crash, no confirmed bug.

> **The lesson that makes this work.** A state-of-the-art agent already reproduces real kernel bugs from the patch alone, reliably enough that a PoC is a trustworthy signal, and a persistent **failure** to reproduce is signal too.

Empirical basis on the next slide: 100 real-world KernelCTF vulnerabilities.

qemu · KASAN

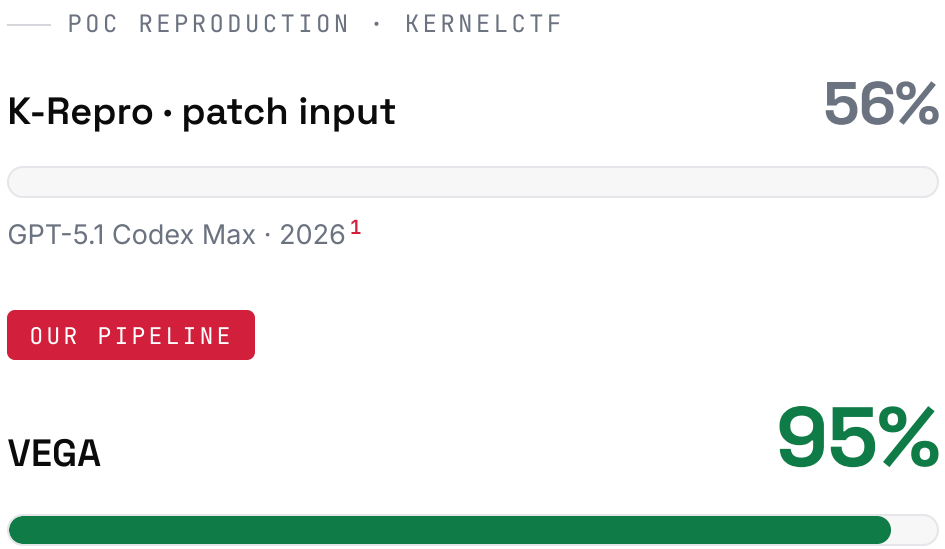
```
$ ./poc_attempt_3
[  12.44] running trigger…
==================================
BUG: KASAN: use-after-free in
     foo_release+0x1a4/0x2c0
Read of size 8 by task poc/214
Call Trace:
 foo_release
 __fput
 task_work_run
==================================
→ verified: REAL
```

VEGA already reproduces real kernel bugs

Reproducing real kernel bugs is now something strong models do well. **VEGA** reaches **95%** on the standard KernelCTF benchmark.

- Public studies set the floor: **over 50%** of 100 KernelCTF vulnerabilities reproduced from the patch alone, fully autonomous.
- Our real input is a full **bug report**, richer than a bare patch, so triggering is easier still.

> So reproduction becomes our **filter**: a bug VEGA can trigger is confirmed real; one it repeatedly can't is treated as a **false positive** and dropped.



1 Baseline: Pu, Li, Liang, Cox, Wu, Shehada, Srivastav, Qian. *Patch-to-PoC: A Systematic Study of Agentic LLM Systems for Linux Kernel N-Day Reproduction*, arXiv:2602.07287 (2026).

Observe, refine, retry until it crashes the kernel

OBSERVED BEHAVIOR

- It uses **every means available** to trigger the bug.
- It **watches system state**, sees how far the PoC got, and **feeds that back** to refine the next attempt.
- It iterates until the kernel **actually crashes**.

GUARDRAILS & LIMITS

Bounded behavior

Guardrail: **no kernel modules** allowed to trigger a bug, which would be cheating the threat model.

Coverage caveat: sanitizers flag **memory-safety violations**. A few classes — e.g. *copy-fail* bugs — don't trip KASAN; for those we watch **other signals** as the tell. In practice a small set.

PoC, or it didn’t happen

No PoC → treat as non-existent

If PoC generation is this reliable, a suspected bug that **cannot** produce a PoC is, for our purposes, **not a bug**.

Has PoC → very likely reachable

A bug that **does** produce a triggering PoC is, with high probability, **genuinely reachable**, and worth a human's time.

> Dynamic verification converts a flood of plausible reports into a **short list of proven, reproducible bugs.**

A crash is not always a vulnerability

Whether a trigger "counts" depends entirely on the attacker model.

CASE · OOB WRITE IN GET_OPTIONS()

lib/cmdline.c · KASAN

```
# boot cmdline: st=0-2147483647
# range count wraps INT_MAX → INT_MIN
# ints + i now points before the array
---------------------------------
BUG: unable to handle page fault
     for address: ffffffd…
RIP: get_option+0x1a/0x90
Call Trace:  get_options ← st_setup
Kernel panic - not syncing
→ verified: real OOB write
```

REAL — BUT OUT OF SCOPE

- A genuine **out-of-bounds write**, crashing under KASAN — present since **v2.6.20**.
- It only fires if you **modify the boot command line** — not a runtime, unprivileged trigger.
- Whoever can set your boot args **already owns the machine** — so this crash **doesn't count**.

This is where verification stops. A PoC proves a bug is **real and reachable**; it can't tell you whether it's **in scope**. Scope is a policy we set **in the prompt** — verification doesn't settle it.

Grade severity by privilege and primitive

We rate each bug on two axes: how much privilege it takes to trigger, and what capability it grants.

WHO CAN TRIGGER IT · PRIVILEGE FLOOR

- Get a **root PoC** first, then **step privilege down**, level by level.
- The **lowest floor** that still triggers becomes the reported severity.

L0	root / CAP_SYS_ADMIN	start	BASELINE
L1	user namespace	↓ step down	ELEVATED
L2	unprivileged user	↓ step down	CRITICAL

WHAT IT GRANTS · EXPLOITATION PRIMITIVE

- The same crash can grant very different power, from a **constrained write** to **arbitrary write or control-flow hijack**.
- Static reasoning spots the **memory corruption**; a debugger-driven check **confirms the primitive** before it counts.

IF	Invalid free	WRITE
OUW	OOB / UAF write	WRITE
CFH	Control-flow hijack	CONTROL
AAW	Arbitrary write	CONTROL

When verification isn't enough

Even with dynamic checks and a threat model, some reports only fire in extreme corners.

CASE · A TIME-RANGE PACKET MATCH

cross-day contiguous match

```
// cross-day range backdates a day
stamp -= SECONDS_PER_DAY;

// if the clock sits near 1970:
stamp < 0  →  monthday wraps > 31

val = 1U << monthday;
// out-of-range shift — undefined behavior
```

REAL CRASH — PRACTICALLY UNREACHABLE

- It genuinely **crashes the kernel** — an out-of-range **1U << monthday** shift, real and reproducible.
- But it only fires if the **system clock sits near 1970** — a corner that **virtually never happens** in the real world.

Reality: cases like this need **case-by-case human judgment.** Verification narrows the field; it doesn't remove the expert.

Let the LLM write the patch

Verification changes the **fixing** problem too: the same PoC that **proves** a bug can **prove a fix.**

> The LLM proposes the patch; the **crash-to-clean** signal decides whether it holds — the same verification loop, now closing the fix.

A DRAFT PASSES ONLY WHEN…

- the PoC **stops triggering** on the patched kernel,
- and the kernel still **boots and passes** its checks.
- That crash-to-clean transition is a natural **reward** for the agent.

But the drafted patch isn't always a clean one. It proves a fix is possible and gives a good **starting point** — but it's usually not clean enough to merge as-is. A human still writes the final patch.

RECAP · WHAT WE LEARNED

Verification is what makes AI bug-finding real

It proves what's **real and reachable**. What **counts** and what **matters** are still ours to define.

- If it can't produce a **PoC**, treat it as non-existent.
- Pin down the **threat model** explicitly, in the prompt.
- Report by the **minimum privilege** a trigger needs.
- Keep a **human** for the edge cases.

VISION · WHERE THIS GOES

Toward a bug-free Linux

As models get better, two systems tackle bugs from both ends:

NEW CODE

Sashiko

Guards code as it's written — so new commits stop adding bugs.

EXISTING CODE

VEGA

Finds and fixes the bugs already living in today's kernel.

New code stays clean. Old code gets cleaned. Until, one day, no bugs are left.