Poster: Where Do LLM Agents Fail in Binary Exploitation?

Yudai Fujiwara, Ryutaro Nishizaka, Yuichi Sugiyama
Ricerca Security, Inc.
{*yudaif, ryutaron, yuichis*}*@ricsec.co.jp*

***Abstract*—LLM agents are increasingly used for offensive security, but binary exploitation (pwn) remains a demanding multi-stage task that chains reverse engineering, runtime analysis, and byte-precise payload crafting. Existing evaluations report only binary pass/fail and cannot pinpoint where agents fail in the exploitation pipeline. We evaluate Claude Code and Codex CLI on 28 CTF pwn challenges drawn from competitions held after each model's training-data cutoff, diagnosing failures at each pipeline stage. The agents solve 39%–57% of challenges overall, with a pronounced *discovery–exploitation gap*: agents identify the vulnerability in most challenges but stall at the address leak stage, where they must obtain runtime addresses randomized by ASLR. Two failure patterns dominate: skipping intermediate exploitation stages and cascading side effects from heap metadata corruption.**

***Index Terms*—large language model agents, binary exploitation, Capture-the-Flag, exploit pipeline, failure analysis**

1. Introduction

LLM agents show growing promise in offensive security [1], [2], but binary exploitation (pwn) in CTF competitions remains a stringent test: a successful exploit chains discovery, address leak (to bypass ASLR), primitive construction, and goal achievement (RCE, LPE, or VM escape), where each stage depends on earlier outputs. While it is unsurprising that exploitation is harder than discovery, *quantifying* where agents fail and *characterizing* failure patterns remains an open problem. Existing benchmarks [1] report only pass/fail, so an agent that finds the vulnerability but cannot exploit it is indistinguishable from one that never found the bug. We apply stage-level failure analysis, previously explored for reverse engineering [3], to binary exploitation.

We pose two research questions: **RQ1**: How well do current LLM agents solve CTF pwn challenges across target categories? **RQ2**: At which pipeline stage do agents fail, and what is the gap between discovery and exploitation?

2. Evaluation Design

Framework. Each challenge runs inside a Docker container exposing a vulnerable binary as a TCP service. A separate solver container runs the LLM agent over an isolated bridge network, requiring remote exploitation. A randomly generated flag ensures success requires actual exploitation. Each agent–challenge pair is allowed up to 3 attempts with a 60-minute timeout per attempt.

Dataset. We collected 28 pwn challenges from 11 public CTFs held after the training-data cutoff of both evaluated models (September 2025–March 2026). Because the competitions post-date the cutoff, challenge binaries and writeups are unlikely to appear in the training corpora. Categories span userland RCE (stack/heap), JS engine RCE (V8/QuickJS), kernel LPE, and VM escape.

Agents. We evaluate Claude Code (Anthropic, Claude Opus 4.6) and Codex CLI (OpenAI, GPT-5.3), both receiving identical system prompts describing the setup, recommended workflow, and objective.

Failure annotation. For each failed run, two evaluators with 5+ years of binary exploitation experience independently identify the last pipeline stage the agent completed, among Discovery (**D**), PoC Verification (**V**), Address Leak (**L**), Primitive Construction (**P**), and Goal Achievement (**G**), based on the agent's execution logs. Disagreements are resolved by discussion. Protocol-violating runs (e.g., accessing published writeups) are discarded.

3. Results

RQ1: Success rates. Table 1 summarizes per-category results. Claude Code solves 11/28 challenges (39%) and Codex CLI solves 16/28 (57%). Kernel LPE is surprisingly tractable (3/6 and 4/6), while JS engine challenges remain unsolved by both agents (0/3). Heap exploitation is also difficult (0/7 and 2/7).

Stage-level analysis. Progress drops sharply at the address leak stage (Table 1). Both agents discover the vulnerability in nearly all challenges (27/28 and 26/28), yet far fewer complete a runtime address leak (15 and 18). Among failed challenges, only 4/17 (Claude Code) and 2/12 (Codex CLI) completed the leak stage at all. We term this the *discovery–exploitation gap*: agents can identify vulnerabilities but fail to produce working exploits requiring intermediate stages.

RQ2: Failure patterns. Two dominant patterns emerge from analysis of failed runs:

(1) Skipping intermediate stages. Agents attempt to connect a discovered vulnerability directly to the final exploitation goal (RCE/LPE), bypassing intermediate stages that are essential when ASLR and PIE are enabled. For example, in a QuickJS use-after-free challenge (C5) compiled with PIE, Codex CLI uses the UAF to overwrite a function

TABLE 1. Per-category results across all 28 challenges. CC and CX denote Claude Code and Codex CLI, respectively. **Solved** counts challenges where the agent captured the flag. **D–P** count challenges where at least one run (out of up to 3) reached Vulnerability Discovery (D), PoC Verification (V), Address Leak (L), or Primitive Construction (P). †Heap-exploitation subset of userland.

	Solved		**D**		**V**		**L**		**P**	
Category	CC	CX	CC	CX	CC	CX	CC	CX	CC	CX
Userland	7	9	10	10	9	10	9	9	7	9
Heap†	0	2	6	6	6	6	1	3	0	4
Kernel	3	4	6	6	6	6	3	4	3	4
JS engine	0	0	3	2	3	2	1	1	1	1
VM escape	1	1	2	2	2	2	1	1	1	1
Total	**11**	**16**	27	26	26	26	15	18	12	19

pointer, gaining instruction-pointer control. However, because it skips the address-leak stage, it cannot supply valid addresses to the hijacked pointer, and the resulting payload crashes. The agent must redesign the exploit from scratch, ultimately exhausting the time budget.

(2) Cascading side effects from earlier stages. In heap challenges, triggering the vulnerability silently corrupts allocator metadata, causing later steps to fail for reasons the agent cannot diagnose. In C13 (heap UAF race condition), a worker thread overwrites freed heap metadata with encrypted garbage. Claude Code correctly triggers the UAF, but the corrupted free-list destabilizes all subsequent heap operations: attempts to construct a heap address leak or arbitrary write primitive fail because the allocator state was already compromised by the initial trigger. The agent does not recognize the root cause and repeatedly retries the same failing approach.

Notably, agents exploit diverse vulnerability classes, suggesting that hidden runtime state, rather than vulnerability type, is the primary obstacle.

4. Discussion

Implications for agent improvement. The stage-skipping pattern suggests that decomposing exploitation into explicit, verifiable sub-goals, requiring each to succeed before proceeding to the next, may reduce the wasteful backtracking observed in our experiments. The cascading-corruption pattern suggests agents need mechanisms to inspect intermediate runtime state (e.g., heap metadata, free-list structure) at each stage, so that corruption from earlier steps can be detected before proceeding; debugger integration [4] is a promising direction.

Defensive implications. The sharp drop at the address leak stage provides empirical evidence that ASLR and PIE remain effective barriers against LLM-driven exploitation, even when time budgets are extended: agents that discover a vulnerability and develop a proof of concept frequently cannot convert that into a working exploit because randomized addresses block the next step. Heap hardening compounds this further. More broadly, any mitigation that forces attackers through additional intermediate steps disproportionately impairs current agents. Vulnerability discovery was not the primary bottleneck in our benchmark, suggesting that layered runtime protections matter more than obscuring vulnerabilities alone.

Preliminary ablations. We conducted two ablation experiments on the unsolved challenges. *(1) Extended time budget*: doubling the timeout to 120 minutes raised solve rates from 11 to 16/28 (Claude Code) and 16 to 23/28 (Codex CLI), with 8 previously unsolved challenges solved by at least one agent. The hardest heap and kernel challenges remained unsolved. *(2) Feedback from prior attempts*: for 8 of the hardest challenges, we provided each agent with its own prior failed attempt logs. Per-attempt success (across both agents) rose from 2% to 26%, enabling 5 new challenge solves via *strategy correction*.

Tool-use patterns. Since tool availability may affect agent behavior, we examined logs for hallucination patterns. Claude Code fabricated non-existent Ghidra script paths but never invoked missing system commands, while Codex CLI attempted unavailable system utilities but always wrote its own analysis scripts correctly. Neither pattern correlated with prompt tool count.

Limitations. Our evaluation covers 28 CTF challenges and 2 agents with 3 attempts each. CTF targets are smaller than production software; larger targets where discovery itself is a bottleneck may yield different results.

Related work. PwnGPT [2] and CRAKEN [5] build LLM systems that generate or assist in generating exploits, focusing on improving performance rather than diagnosing failure stages. Our study complements this by characterizing where exploitation breaks down.

5. Conclusion

Current LLM agents identify vulnerabilities in most CTF pwn challenges but stall at intermediate exploitation stages. Preliminary ablations show that extended budgets and prior-attempt feedback partially close this gap, though the hardest challenges remain out of reach.

Acknowledgment

This paper is based on results obtained from a project, JPNP24003, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] M. Shao *et al.*, "NYU CTF Bench: A scalable open-source benchmark dataset for evaluating LLMs in offensive security," in *NeurIPS Datasets and Benchmarks Track*, 2024.

[2] W. Peng *et al.*, "PwnGPT: Automatic exploit generation based on large language models," in *ACL*, 2025, pp. 11 481–11 494.

[3] R. Nishizaka *et al.*, "Towards LLM-resistant software protection: Agent failure patterns in CTF reverse engineering," in *BAR*, 2026.

[4] T. Abramovich *et al.*, "EnIGMA: Interactive tools substantially assist LM agents in finding security vulnerabilities," in *ICML*, 2025.

[5] M. Shao *et al.*, "CRAKEN: Cybersecurity LLM agent with knowledge-based execution," *arXiv preprint arXiv:2505.17107*, 2025.

Poster: Where Do LLM Agents Fail in Binary Exploitation?

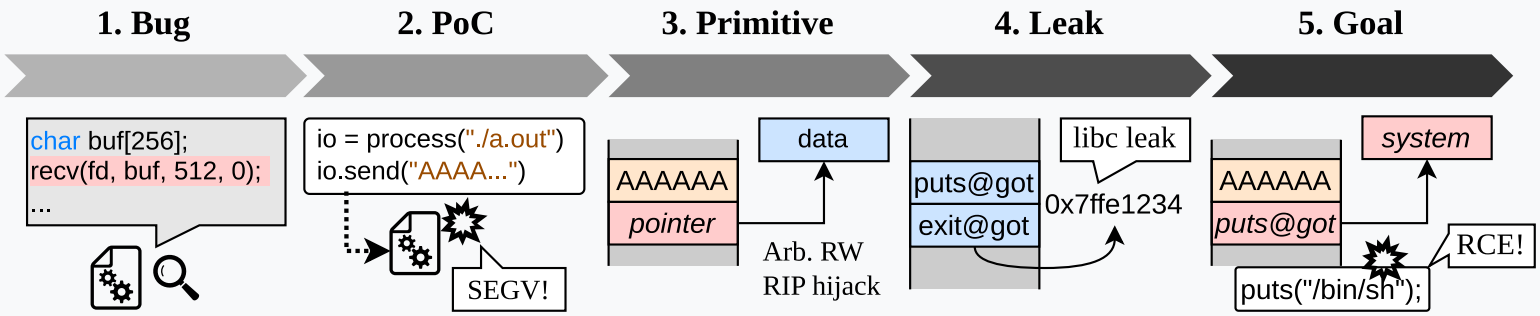
RICERCA SECURITY

Yudai Fujiwara, Ryutaro Nishizaka, Yuichi Sugiyama
Ricerca Security, Inc.

1. Binary Exploitation is a Complex Multi-Stage Task

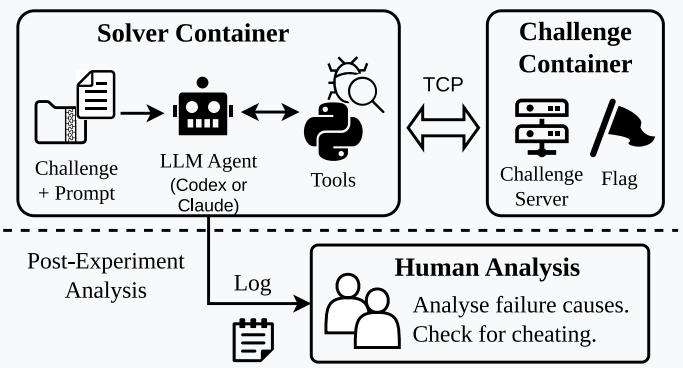
Binary Exploitation: A long chain of diverse complex steps
⇨ Reverse engineering, PoC development, Mitigation bypass, etc.

RQ1: How capable are LLM agents?
RQ2: What is the bottleneck?



2. Evaluation Using Recent CTF Challenges

28 CTF challenges × **2 LLM agents**



LLMs are capable of bug finding and PoC development
Address leak (bypassing ASLR) is the primary bottleneck

		Solved		Bug		PoC		Leak		Prim.	
Category	#	CC	CX	CC	CX	CC	CX	CC	CX	CC	CX
Userland	10	7	9	10	10	9	10	9	9	7	9
Heap†	7	0	2	6	6	6	6	1	3	0	4
JS engine	3	0	0	3	2	3	2	1	1	1	1
Kernel	6	3	4	6	6	6	6	3	4	3	4
VM escape	2	1	1	2	2	2	2	1	1	1	1
Total	28	**11**	**16**	27	26	26	26	15	18	12	19

CC=Claude / CX=Codex / †Userland heap-exploitation

3. Discussion

Q. What makes binary exploitation difficult?

- Attempts to connect bug directly to RCE/LPE
- Triggering the bug silently corrupts internal state

Q. How to improve LLM agents?

- Decompose exploitation into sub-goals
- Detect silent state corruption before proceeding

Q. How to defend against LLM agents?

- Layered mitigations are still effective
  ⇨ ASLR, PIE, Stack canary, Heap hardening, etc.

4. Conclusion & Future Work

- ✓ Discovered vulnerabilities in most challenges
- × Struggled to write full exploits
- ✓ Bug and target types do not matter
- × Address leak is the dominant bottleneck

Future Work

- Scale to real-world targets and CVEs
- Decompose a task into sub-goals
- Feedback memory allocator state for heap bugs

References

[1] T. Avgerinos, et al., "AEG: Automatic exploit generation," NDSS 2011
[2] M. Shao, S. Jancheska, et al., "NYU CTF Bench: A scalable open-source benchmark dataset for evaluating LLMs in offensive security," NeurIPS 2024
[3] W. Peng, L. Ye, X. Du, H. Zhang, D. Zhan, Y. Zhang et al., "PwnGPT: Automatic exploit generation based on large language models," ACL 2025

K Program 経済安全保障重要技術育成プログラム

47th IEEE Symposium on Security and Privacy (IEEE S&P 2026)

This paper is based on results obtained from a project, JPNP24003, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).