

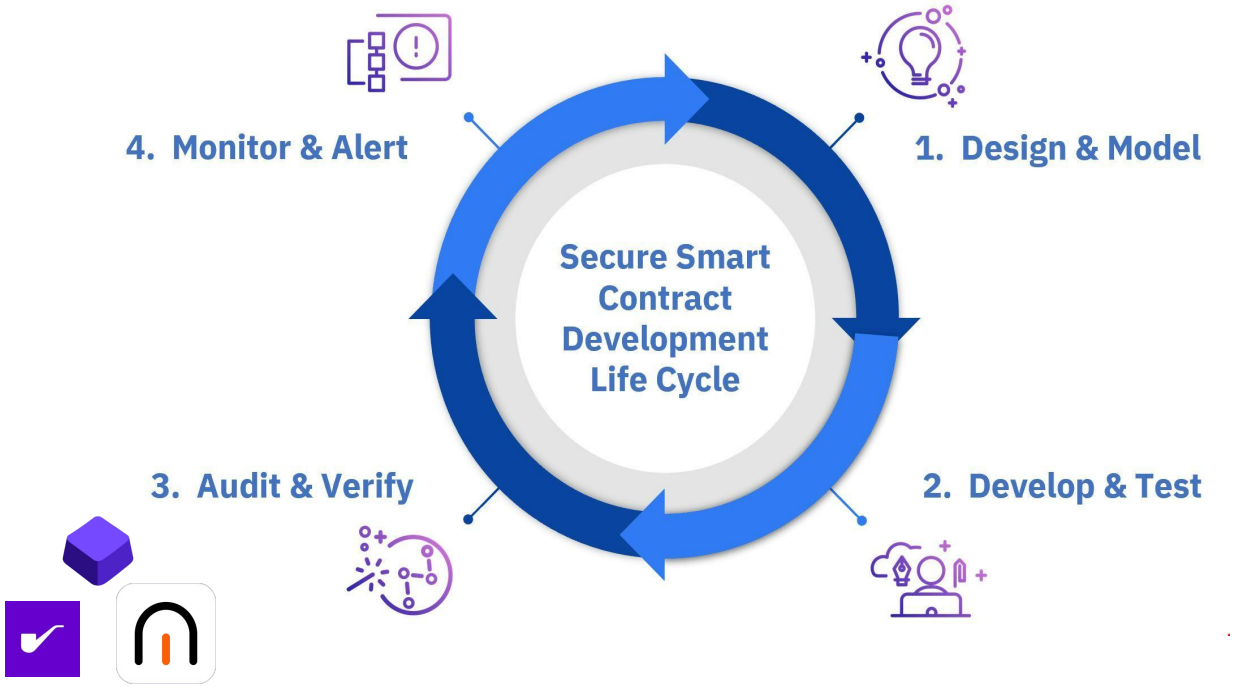
Paper

PoCo: Agentic PoC Exploit Generation for Smart Contracts

Vivi Andersson*, **Sofia Bobadilla***, Harald Hobbelhagen, Martin Monperrus
KTH Royal Institute of Technology

ACM TOSEM · Special Issue on Agentic AI in Software
ASE 2026 Journal-First Track · 15 October 2026

External Security Reviews as the Backbone of Blockchain Protocol Security



Important:

1. **Smart contracts live under and extremely adversarial environment**
2. **Security reviews are what keep billions in assets protected from hackers.**
3. **Better security reviews, better fixes, better protection**

Submitted by adriro

[M-03] Flash loan fee is incorrect in Private Pool contract

Description

According to the documentation, changeFee is meant to represent a value with 4 decimal precision (e.g., 25 = 0.0025 ETH), but the flashFee() function returns changeFee directly without scaling.

As a result, the flash loan fee is several orders of magnitude smaller than intended (e.g., 25 wei instead of 0.0025 ETH).

```
750 function flashFee(address, uint256) public view returns (uint256) {
751     return changeFee;
752 }
```

Impact

Attackers can execute flash loans at a drastically reduced cost, paying only a few wei instead of the proper fee.
This allows borrowers to use NFT flash loans essentially for free, leading to loss of protocol revenue and potential abuse of the lending mechanism.

Recommended Mitigation Steps

The flashFee function should properly scale the value of the changeFee variable, similar to how it is implemented in changeFeeQuote.

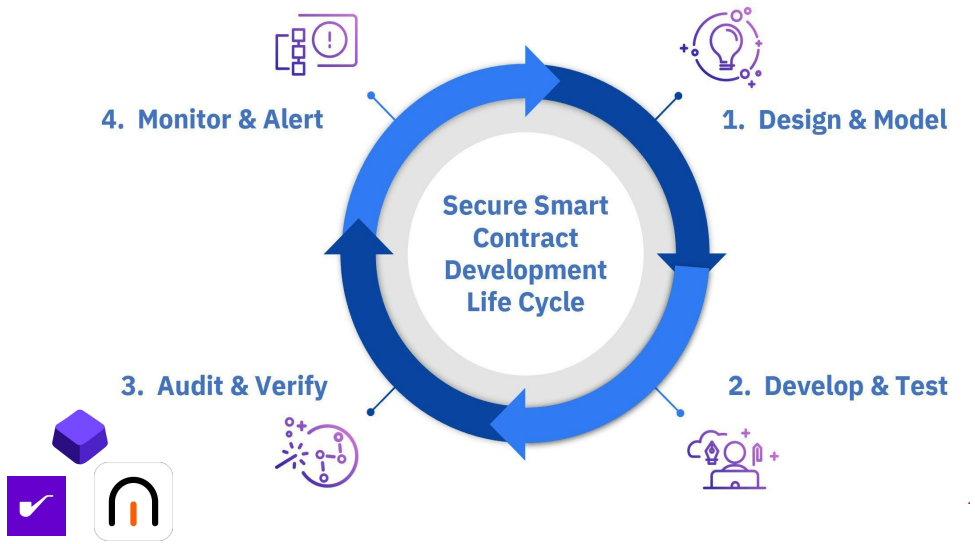
Accepted Mitigation

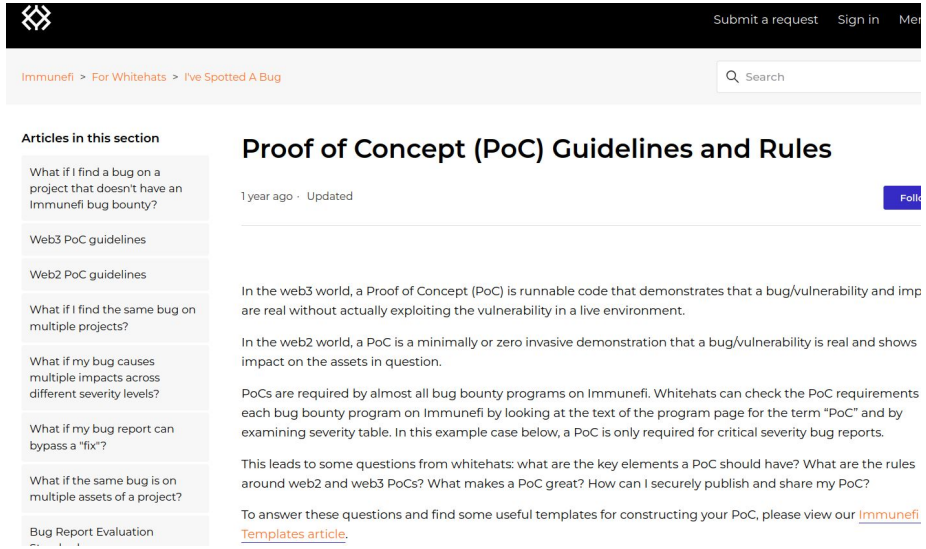
```
750 function flashFee(address, uint256) public view returns (uint256) {
751   // multiply the changeFee to get the fee per NFT (4 decimals of accuracy)
752   uint256 exponent = baseToken == address(0) ? 18 - 4 : ERC20(baseToken).decimals() - 4;
753   uint256 feePerNft = changeFee * 10 ** exponent;
754   return feePerNft;
755 }
```

Proof of Concept

```
1  function test_PrivatePool_flashLoan_IncorrectFee() public {
2      // 1) Deploy pool
3      PrivatePool privatePool = new PrivatePool(
4          address(factory),
5          address(royaltyRegistry),
6          address(stolenNftOracle)
7      );
8      uint56 changeFee = 25;
9      // 2) Initialize pool
10     privatePool.initialize(
11         address(0),           // Use native ETH
12         address(milady),      // NFT contract
13         100e18,               // Virtual base token reserves
14         10e18,                // Virtual NFT reserves
15         changeFee,            // Declared flash loan fee
16         0,                    // Fee rate
17         bytes32(0),           // Merkle root
18         false,                // Stolen NFT oracle
19         false                 // Pay royalties
20     );
21     // 3) Mint NFT to pool
22     uint256 tokenId = 0;
23     milady.mint(address(privatePool), tokenId);
24     // 4) Impersonate attacker (Alice)
25     vm.startPrank(alice);
26     // 5) Deploy borrower contract
27     FlashLoanBorrower flashLoanBorrower = new FlashLoanBorrower();
28     // 6) Fund attacker with fee
29     vm.deal(alice, changeFee);
30     // 7) Snapshot pool balance
31     uint256 balanceBefore = address(privatePool).balance;
32     // 8) Alice gets a flash loan with minimal fee; verify fee accepted and NFT returned
33     privatePool.flashLoan{value: changeFee} (flashLoanBorrower, address(milady), tokenId, "");
34     assertEq(address(privatePool).balance, balanceBefore + changeFee,"Fee not credited to pool");
35     assertEq(milady.ownerOf(tokenId), address(privatePool),"NFT not returned to pool after flash loan");
36     vm.stopPrank();
37 }
```

Our Motivation





1. How to assess the soundness of a vulnerability?

2. How to assess the soundness of its evidence (a PoC)?

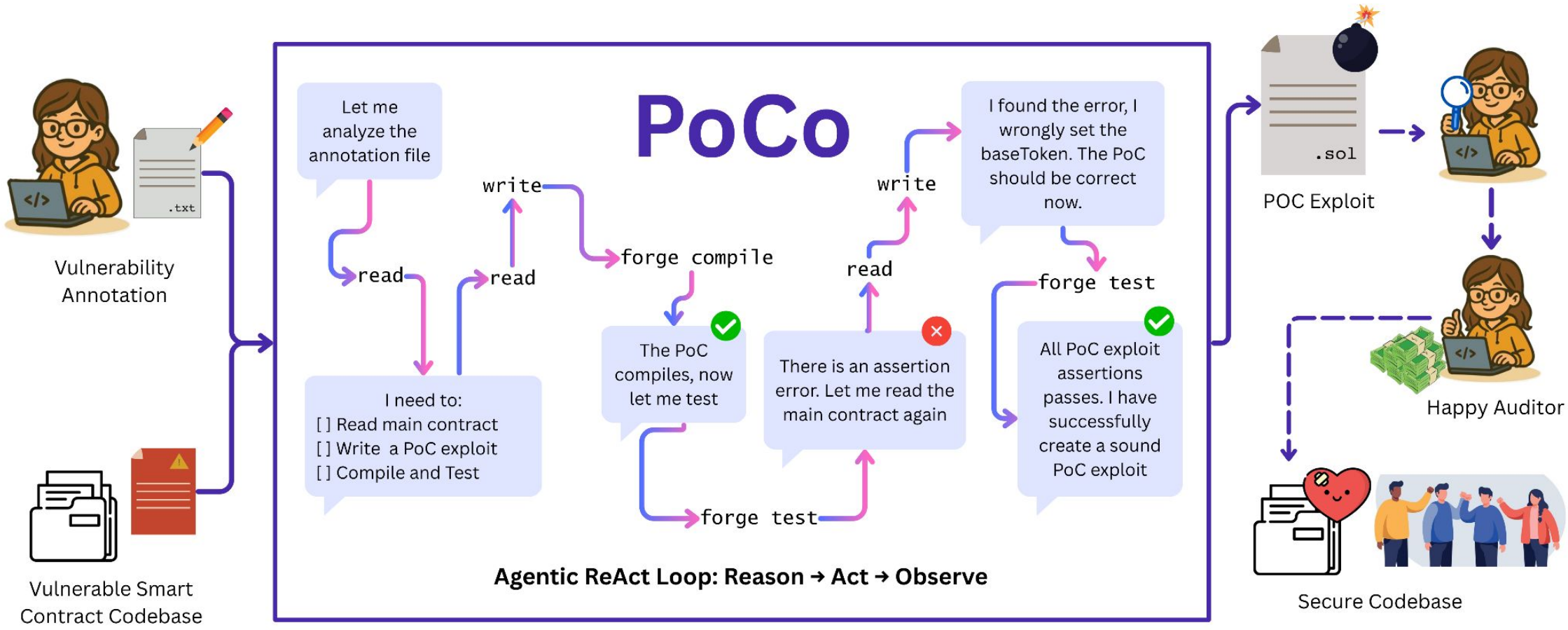
3. To what extent this task can be delegated to AI agents?

Related Work

Tool	Type	Reproducible?	Evaluation	Dataset
REX [1]	Iterative Prompting + Foundry	Not Available	Compilation + test execution (Foundry) + Manual Analysis	SmartBugs Curated + 38 real-world exploit PoCs from DeFiHacklabs
A1 [2]	Agentic (6 domain-specific tools, on-chain execution)	Not Available	Profit Oracle	VERITE benchmark (36 real-world contracts on Ethereum/BSC)-> a extraction from DeFiHackLabs
Quimera [3]	Iterative Prompt + Foundry (feedback-driven)	Not able to reproduce	Manual	DeFiHackLabs (informal, small-scale)

[1] Zeke Xiao, Yuekang Li, Qin Wang, and Shiping Chen. "Prompt to Pwn: Automated Exploit Generation for Smart Contracts." arXiv:2508.01371, 2025.
[2] Arthur Gervais and Liyi Zhou. "AI Agent Smart Contract Exploit Generation." arXiv:2507.05558, 2025.
[3] Gustavo Grieco. "Introducing Quimera: feedback-driven exploit generation for Ethereum smart contracts using LLMs." https://github.com/gustavo-grieco/quimera, 2025.

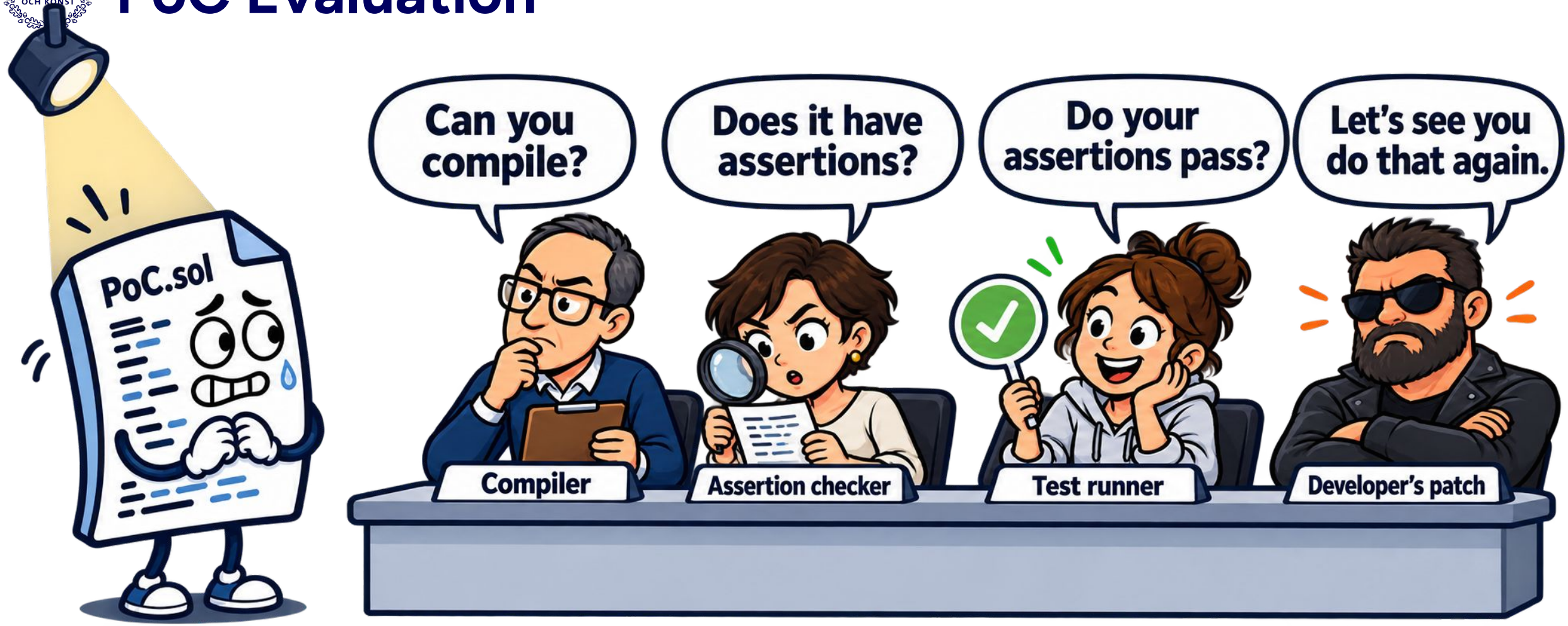
PoCo: Vulnerability Report to Exploit



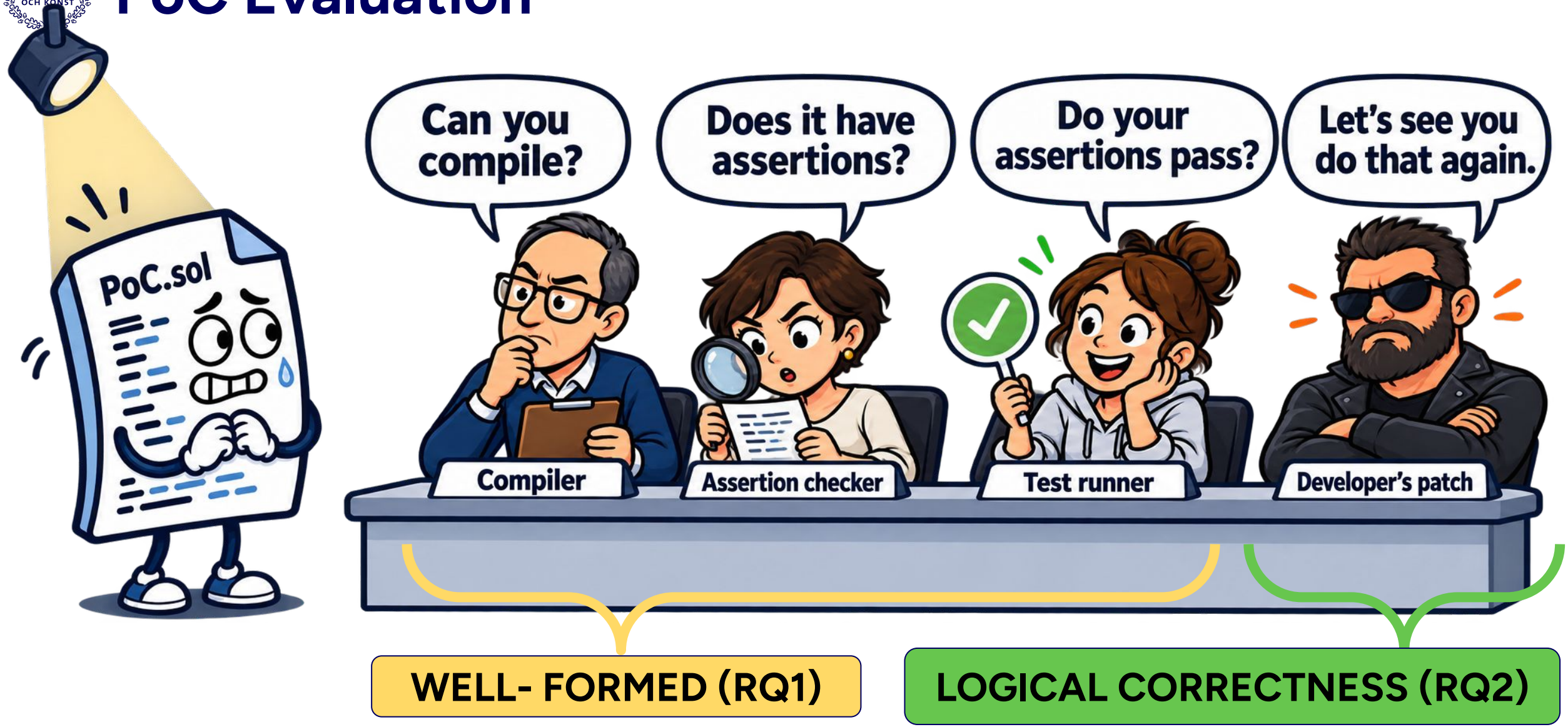
Task: generate a PoC that exploits vulnerability with passing assertions
Tools: smart contract compilation & test, read, write grep, todo

Methodology

PoC Evaluation



PoC Evaluation



PoC Evaluation

WELL- FORMED (RQ1)

Generated PoC

```
contract Exploit {
  function run() external {

IVault(target).withdraw(
      type(uint256).max);
  }
}
```

Vulnerable contract



PoC Compiles?
Contains assertions?
Assertions pass?

Any check fails

All checks pass

ILL-FORMED
PoC

WELL-FORMED
PoC

Developer patch serves as the correctness oracle, assuming a reliable patch.

PoC Evaluation

LOGICAL CORRECTNESS (RQ2)

Generated PoC

```
contract Exploit {
  function run() external {

IVault(target).withdraw(
      type(uint256).max);
  }
}
```

Patched codebase

Re-run the same PoC



FAIL

LOGICALLY CORRECT

At least one assertion fails

Does not compile

PASS

INCORRECT PoC

All assertions still pass

INCONCLUSIVE (manual review)

Developer patch serves as the correctness oracle, assuming a reliable patch.

PoC Evaluation

WELL- FORMED (RQ1)

LOGICAL CORRECTNESS (RQ2)

Generated PoC

```
contract Exploit {
  function run() external {

IVault(target).withdraw(
      type(uint256).max);
  }
}
```

Vulnerable contract



PoC Compiles?
Contains assertions?
Assertions pass?

Any check fails

ILL-FORMED
PoC

All checks pass

WELL-FORMED
PoC

Patched Codebase

Re-run the same PoC



FAIL

LOGICALLY CORRECT

At least one assertion fails

Does not compile

PASS

INCORRECT
PoC

All assertions still pass

INCONCLUSIVE
(manual review)

Developer patch serves as the correctness oracle, assuming a reliable patch.

Proof-of-Patch



Manually curated real-world smart contract vulns
23 samples across 15 projects from 2022-2025

Each sample:

1. Vulnerability description
2. Vulnerable codebase
3. Scoped patch

Severity

8 High	15 Medium

Auditor-written PoC available

13 yes	10 no

23 vulnerabilities from 15 audited projects

#001	Size	2024	multicall bypasses deposit limits
#003	PoolTogether	2023	mint shares, steal the yield fee
#008	Centrifuge	2023	rounding gives investors excess shares
#009	Caviar	2023	royalties trapped at zero address
#015	PoolTogether	2023	hook disrupts prize distribution
#018	Caviar	2023	former owner reclaims assets via approvals
#020	DODO GSP	2023	first LP inflates share price, DoS
#032	Putty	2022	strike amount stuck in contract
#033	Caviar	2023	flash-loan fees miscalculated
#039	Axis Finance	2024	refund errors lock seller funds
#041	Axis Finance	2024	hijack a prefunded auction
#042	Cap	2025	rounding skews interest rates
#046	xETH	2023	zero-amount transfer blocks rewards
#048	Caviar	2023	royalty recipient drains the pool
#049	Cooler	2023	lender changes loan terms alone
#051	Centrifuge	2023	deposit on behalf of others, DoS
#054	Cally	2022	unchecked transfers create empty vaults
#058	Putty	2022	Ether locked in the contract
#066	Kelp	2023	rsETH mint miscalculated
#070	Phi	2024	NFT transfers while paused
#077	AI Arena	2024	reentrancy claims extra rewards
#091	Basin	2023	reported reserves can be manipulated
#098	Cally	2022	fake balances for nonexistent tokens

Total: 23

Curated Security Patches



```
--- a/2023-11-kelp/src/LRTDepositPool.sol
+++ b/2023-11-kelp/src/LRTDepositPool.sol
@@ -125,6 +125,11 @@
         nonReentrant
         onlySupportedAsset(asset)
     {
+
+        // interactions
+        uint256 rsethAmountMinted = _mintRsETH(asset, depositAmount);
+
+
         // checks
         if (depositAmount == 0) {
             revert InvalidAmount();
@@ -137,8 +142,6 @@
             revert TokenTransferFailed();
         }

-        // interactions
-        uint256 rsethAmountMinted = _mintRsETH(asset, depositAmount);

         emit AssetDeposit(asset, depositAmount, rsethAmountMinted);
     }
```

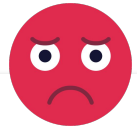
Moves function call before a transfer

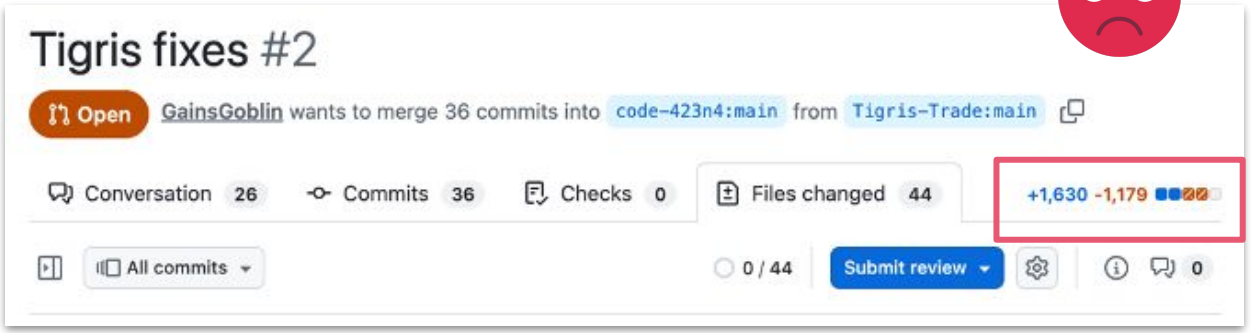


```
--- a/2022-05-cally/contracts/src/Cally.sol
+++ b/2022-05-cally/contracts/src/Cally.sol
@@ -169,6 +169,7 @@
         require(dutchAuctionReserveStrike < strikeOptions[dutchAuctionStartingStrikeIndex],
"Reserve strike too small");
         require(durationDays > 0, "durationDays too small");
         require(tokenType == TokenType.ERC721 || tokenType == TokenType.ERC20, "Invalid
token type");
+        require(token.code.length > 0, "token is not contract");

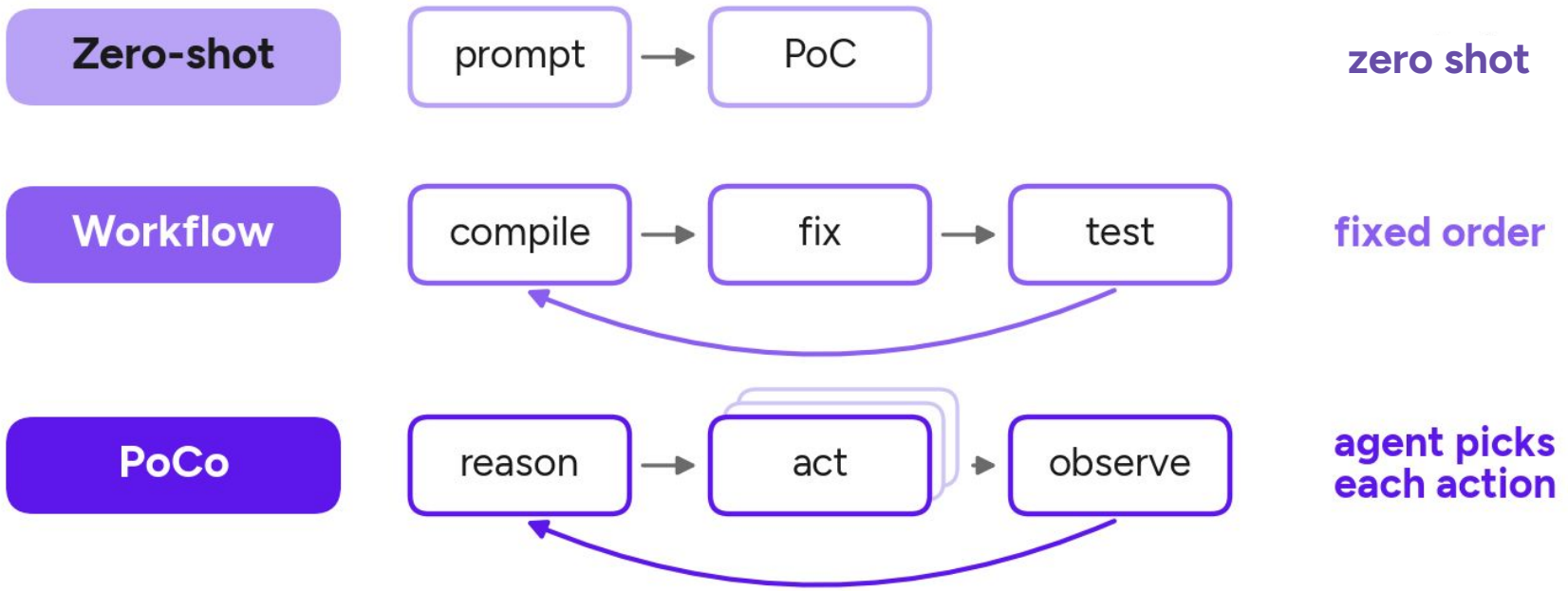
         Vault memory vault = Vault({
             tokenIdOrAmount: tokenIdOrAmount,
```

Adds invariant check to prevent fake asset





Baselines vs. PoCo

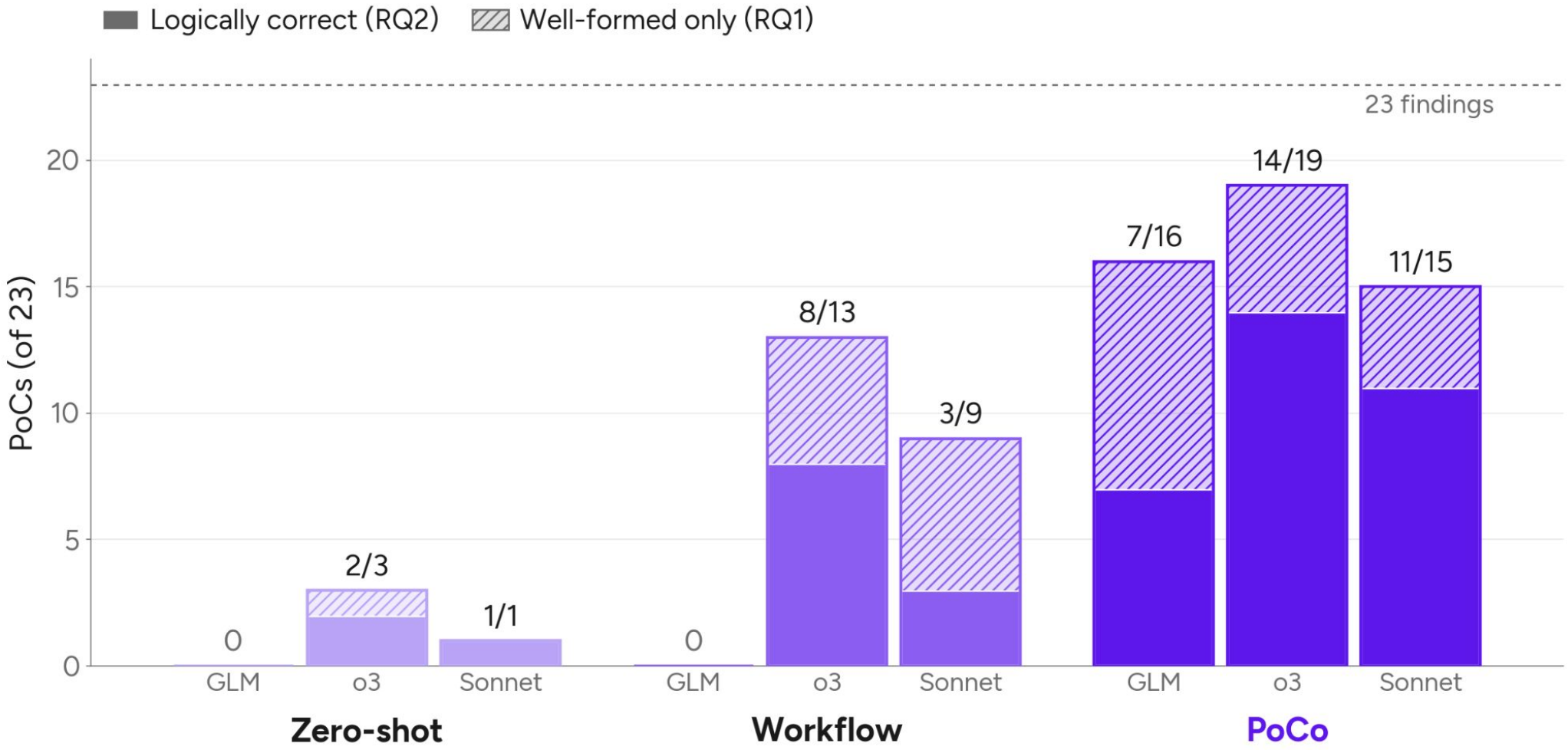


Models: Sonnet 4.5 · OpenAI o3 · GLM 4.6, each with all three approaches
Budget per PoC: **$3** or max 10 smart-contract tool calls

Results

In short

PoCo improves logical correctness across all models over non-agentic baselines



Logically Correct PoC: Price Oracle Manipulation (Basin DEX)

- **Vulnerability:** shift() changes Basin's asset balances without updating its price feed
- **Exploit:** An attacker can manipulate Basin's price data and steal funds from protocols that trust it

PoC

```
// Verify reserves were updated by shift()
assertTrue(reservesAfterShift[0] > initialReserves[0], "Reserve 0 should have increased");

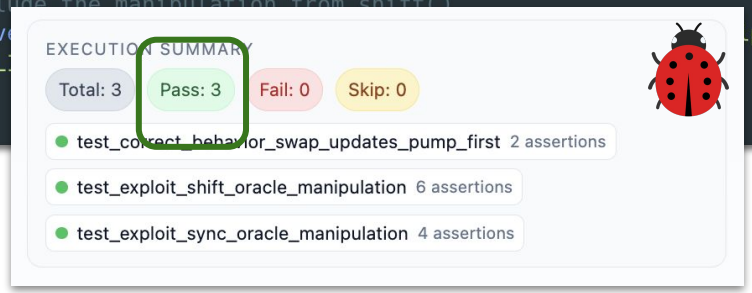
// CRITICAL OBSERVATION: Pump has NOT been updated yet
assertEq(trackingPump.updateCount(), initialUpdateCount, "Pump should not be updated by
shift()");

// These reserves include the manipulation from shift()
assertTrue( pumpReceiv                                                    ipulated
reserve 0 - VULNERABIL
);
```

Balances changed

No oracle update

The price feed receives manipulated balances



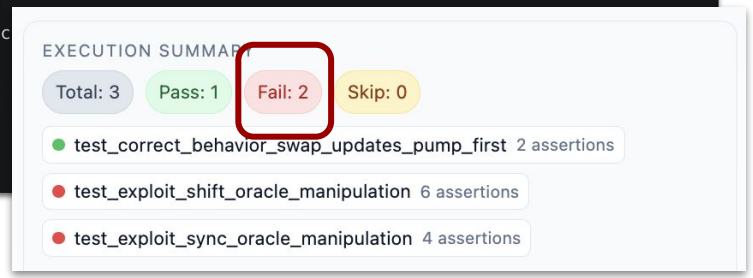
claude sonnet 4.5

Patch

```
--- a/2023-07-basin/src/Well.sol
+++ b/2023-07-basin/src/Well.sol
@@ -355,7 +355,7 @@
         address recipient
     ) external nonReentrant returns (uint256 amountOut) {
         IERC20[] memory _tokens = tokens();
-        uint256[] memory reserves = new uint256[](_tokens.length);
+        uint256[] memory reserves = _updatePumps(_tokens.length);

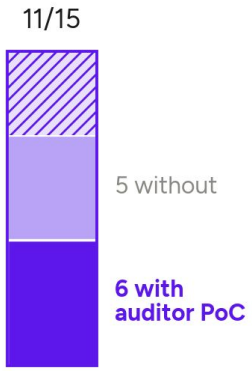
         // Use the balances of the pool instead of the stored reserves.
         // If there is a change in token balances relative to the
currently
@@ -589,7 +589,7 @@
      */
     func
-
+
```

Update the price feed before reading new balances.



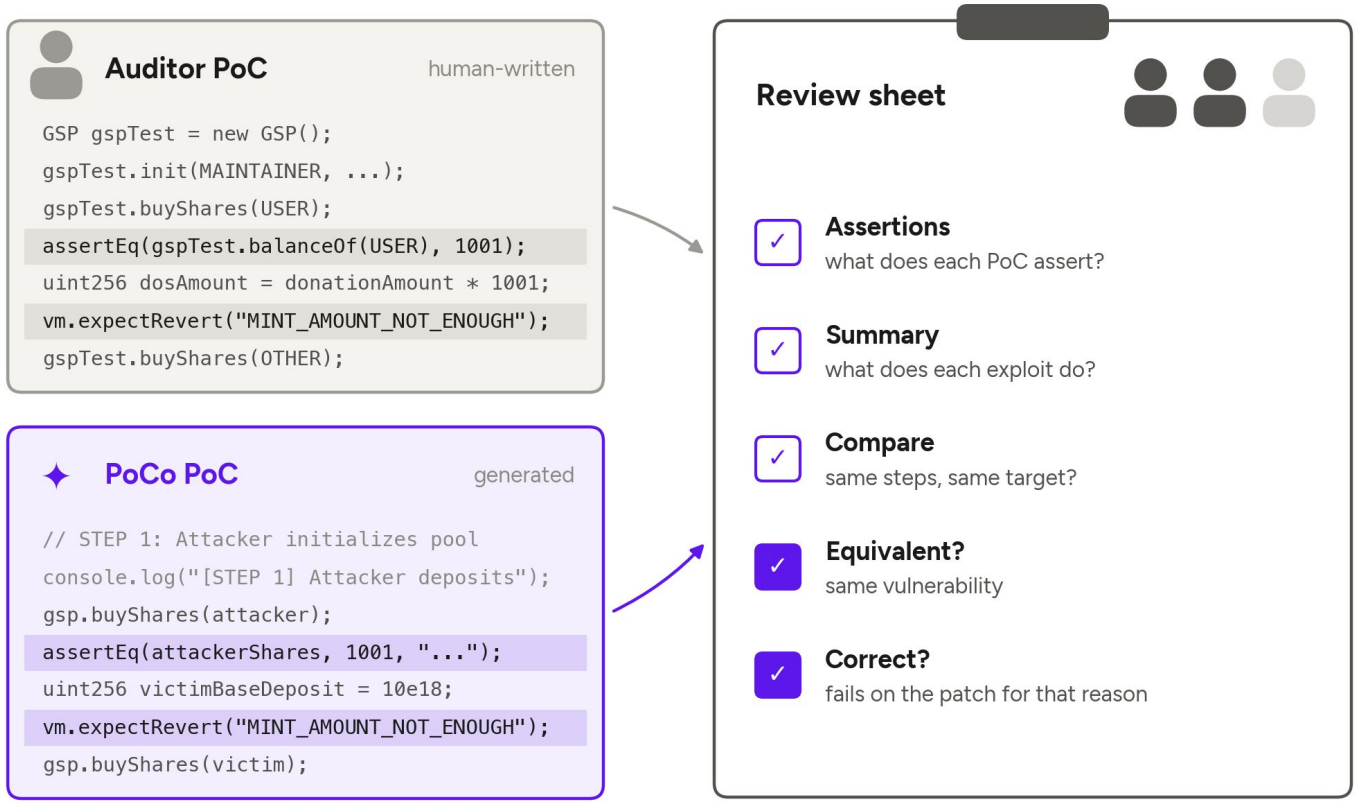
Do PoCs fail for the right reason?

- **Patch failure alone could be incidental breakage.**
- We check if PoCo exploits the same vulnerability as auditor-written PoCs.



Subset:
PoCo with
Sonnet 4.5

(6 samples)



PoCo exploits the same vulnerability as the auditor

- All 6 examined cases exploit the same vulnerability as the auditor-written PoC
- Failing exploit:
  - (1) the patch invalidated the **exploit assertion** or
  - (2) blocked the **exploit operation**

ID	Project	#Tests	Verdict	PoC Test Outcome (✗ / ✓ on patch)	Exploit Assertion/Reverted Call	
020	2023-12-dodo-gsp	1	Correct	✗ test_buyShares_lowLiquidity	gsp.buyShares(attacker)	(2)
033	2023-04-caviar	5	Correct	✗ test_FlashLoanFeeIncorrectlyScaled	assertEq(expectedFlashFee/actualFlashFee,10**14)	
			Correct	✗ test_ChangeFeeVsFlashFeeDiscrepancy	assertEq(flashFeeAmount, 25)	
			Correct	✗ test_EconomicImpactOfVulnerability	assertEq(actuallyPays, 2500)	
			Correct	✗ test_ExploitFlashLoanWithMinimalFee	assertEq(feeReceived, 25)	
			Correct	✗ test_FlashLoanFeeIncorrectlyScaledWithERC20	assertEq(actualFee, 25)	
041	2024-03-axis-finance	2	N/A	✓ test_RootCause_LotIdInitializedToZero	N/A	
			Correct	✗ test_StealPrefundedTokensByOverwritingLot...	assertEq(seller0After, ATTACKER)	(1)
046	2023-05-xeth	3	Correct	✗ test_getReward_RevertsOnZeroBalance	vm.expectRevert("RevertOnZeroToken: zero amount")	
			Correct	✗ test_getReward_BlocksLegitimateRewards	vm.expectRevert("RevertOnZeroToken: zero amount")	
			Correct	✗ test_getReward_PermanentDoS	vm.expectRevert("RevertOnZeroToken: zero amount")	
070	2024-08-ph	2	Correct	✗ test_PauseDoesNotPreventTransfers	nft.safeTransferFrom(attacker, victim, tokenId, 1, "");	
			Correct	✗ test_PauseDoesNotPreventBatchTransfers	nft.safeBatchTransferFrom(.., .., tokenIds, amounts, "")	
091	2023-07-basin	3	N/A	✓ test_correct_behavior_swap_updates_pump_first	N/A	
			Correct	✗ test_exploit_sync_oracle_manipulation	assertEq(pump.updateCount(), initialUpdateCount + 1)	
			Correct	✗ test_exploit_shift_oracle_manipulation	assertEq(pump.updateCount(), initialUpdateCount + 1)	

Patch-based evaluation provides strong evidence of genuine exploitation.

Discussion

How does the patch based oracle compare?

Patch-based evaluation needs no vulnerability-specific success condition.

Oracle	**Evidence of exploitation**	**Limitation**
Task-specific (PoCGen)	Expected symptom occurs	May succeed for wrong reason
Profitability (A1)	Attacker gains value	Misses non-profitable exploits
Assertion-based (REX)	Generated Foundry test passes	Requires manual correctness validation
Patch-based (PoCo)	**Vulnerable ✓ / Patched ✕**	**Requires trusted scoped patch**

EVMbench uses PoCs to evaluate patch correctness (reverse direction). See also Bobadilla, Jin & Monperrus, TSE 2025.

Simsek et al., “PoCGen: Generating Proof-of-Concept Exploits for Vulnerabilities in Npm Packages,” FSE 2026, doi:10.1145/3808178.
Gervais & Zhou, “AI Agent Smart Contract Exploit Generation,” FC 2026.
Xiao et al., “Prompt to Pwn: Automated Exploit Generation for Smart Contracts,” ACISP 2026, pp. 445–473. Wang et al.,
“EVMbench: Evaluating AI Agents on Smart Contract Security,” ICML 2026 (PMLR 306).

Trusted well-scoped patches provide a strong oracle for exploit correctness

Trajectories & PoCs (!)

Proof of patch dataset

Trusted well-scoped patches provide a strong oracle for exploit correctness

Contributions:

- PoCo, an agentic blueprint for smart contract PoC generation
- Proof-of-Patch, a manually curated smart contract vulnerability benchmark
- Patch-based evaluation methodology for logical correctness

PoCo: Agentic Exploit Generation for Smart Contracts

Vivi Andersson, Sofia Bobadilla*, Harald Hobbelhagen, Martin Monperrus, 2026.*

ACM TOSEM Special issue on Agentic AI in Software doi/10.1145/3816704