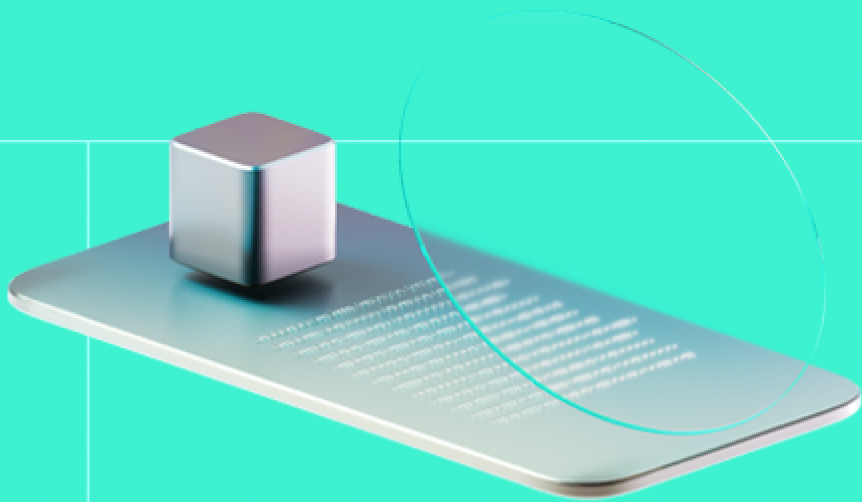
Smart Contract Code Review And Security Analysis Report



Customer: Strobe

Date: 04/07/2025

We express our gratitude to the Strobe team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Strobe Protocol's money market product supports permissionless lending and overcollateralized borrowing.

Document

Name	Smart Contract Code Review and Security Analysis Report for Strobe
Audited By	Ataberk Yavuzer, Seher Saylik
Approved By	Ivan Bondar
Website	https://strobe.finance/
Changelog	04/06/2025 - Preliminary Report
	04/07/2025 - Final Report
Platform	XRPL EVM
Language	Solidity
Tags	Lending, Borrowing, Cross-Chain, Interoperability, Incentives, Integration
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/strobe-protocol/strobe-v1-core
Commit	c3281219a141c0575692d34c8c8d1c7ce0a16b40
Retest	78f82a280b1ae9d52167259d83e5f05e452a2ef7

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

17	12	4	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	3
Medium	7
Low	2

Vulnerability	Severity
F-2025-10701 - Collateral Seizure Without Debt Repayment	Critical
F-2025-10708 - Missing msg.sender == liquidator Check Allows Front-Running and Reward+ Collateral Theft	High
F-2025-10712 - Duplicate Reserve Addition in liquidate() Leads to Invalid Rates	High
F-2025-10731 - Borrow Amount Is Always Lesser Than Advertised Amount Due To Wrong Function Parameter	High
F-2025-10709 - Dynamic LTV and Liquidation Threshold Means Borrowers' Terms Can Change Retroactively	Medium
F-2025-10715 - Zero Oracle Price Check is Missing	Medium
F-2025-10718 - Debt Calculation Logic is Broken for High-Index Reserves	Medium
F-2025-10721 - Inconsistency Between the Documentation and Code: Missing Base Rate Calculation	Medium
F-2025-10724 - Borrowing Functionality Will Be Lost When Loan-to-Value Greater Than Liquidation Threshold	Medium
F-2025-10727 - Wrong Comparison on ReserveFactor Percentage Makes The Upper Limit Inconsistent	Medium
F-2025-10750 - Lending Limit Is Not Enforced	Medium
F-2025-10679 - Missing Replay Protection for Cross-Chain Commands in AxelarPool	Low
F-2025-10734 - Hardcoded Gas Value	Low
F-2025-10713 - Unbounded, Costly Loop in Collateral Checks	Info
F-2025-10716 - Missing Zero Value Checks for liquidationThresholdPct and ltvPct	Info

Vulnerability	Severity
F-2025-10717 - Redundant applyLiquidationThreshold Parameter in Collateral Assertion Functions	Info
F-2025-10751 - Missing Zero-Address Validation	Info

Documentation quality

- Functional requirements are partially missed
- Technical description is provided. The description for index calculations is not provided
- Well-structured technical documentation covering protocol mechanics

Code quality

- The code adapts some gas-inefficient usages
- The development environment is configured

Test coverage

Code coverage of the project is **83.14%.**

- Proper Foundry configuration and development setup was provided
- Mock testing files were provided for cross-chain interaction tests.

Table of Contents

System Overview

Strobe is a **cross-chain decentralized lending protocol** that bridges traditional DeFi lending mechanisms with XRPL (XRP Ledger) integration through Axelar's cross-chain infrastructure. The protocol enables users to deposit, borrow, and manage collateral across different blockchain networks while maintaining unified liquidity pools and interest rate calculations.

The protocol operates on a **dual-layer architecture**:

- **Core Layer**: Manages lending pool states, interest calculations, and collateral management
- **Cross-Chain Layer**: Handles token transfers and command execution via Axelar network

Users interact with the protocol from XRPL by sending cross-chain commands that are executed on the Ethereum-based core contracts, with tokens being transferred back to XRPL upon completion.

AxelarPool — Extends Pool functionality with cross-chain capabilities through Axelar's Interchain Token Service:

- **Cross-Chain Command Processing**: Handles DEPOSIT, WITHDRAW, BORROW, REPAY, and collateral management commands
- **Token Transfer Orchestration**: Manages interchain token transfers back to source chains
- **Error Handling & Recovery**: Implements trap mechanism for failed operations
- **Chain Validation**: Ensures commands originate from accepted source chains only

Pool — The central contract managing all lending pool operations and state. It serves as a generic pool contract that handles:

- **Insights:**
  - If your collateral falls below the threshold, a single transaction can wipe out your entire position in that debt token (no partial liquidations).
  - The Pool contract tracks each user’s scaled-down deposits and debts across multiple reserves, updating per-token lending/borrowing indices on every deposit, borrow, repay, or withdrawal to accrue interest and mint treasury fees.
  - Users can toggle collateral on or off for each reserve: disabling collateral triggers an immediate undercollateralization check (reverting if debt isn’t fully covered), while enabling collateral simply marks that reserve for inclusion in future collateral-value calculations.
- **User Account Management**: Tracks deposits, debts, and collateral usage via XRPL account hashes
- **Interest Calculation**: Implements compound interest through lending/borrowing indices
- **Collateral Management**: Manages user collateral flags and health factor calculations
- **Liquidation Logic**: Enables liquidation of undercollateralized positions
- **Treasury Operations**: Accumulates protocol fees through reserve factors

PoolConfig — Abstract contract inherited by Pool.sol that manages reserve configurations and protocol parameters:

- **Reserve Management**: Adding new token reserves with risk parameters (LTV, liquidation threshold, etc.)
- **Risk Parameter Updates**: Modifying LTV ratios, liquidation thresholds, and reserve factors
- **Supply/Borrow Limits**: Setting lending and borrowing caps per reserve
- **Oracle Integration**: Connecting to price feeds via OracleConnectorHub
- **Treasury Management**: Configuring protocol treasury address

InterestRateStrategyOne — Implements a **dual-slope interest rate model** for dynamic rate calculations:

- **Utilization-Based Rates**: Calculates borrowing/lending rates based on pool utilization
- **Two-Slope Model**: Different rate curves before and after optimal utilization point
- **Parameter Validation**: Ensures rate parameters stay within safe bounds
- **Overflow Protection**: Validates maximum possible rates fit within uint104

IndexLogic — Pure mathematical library handling all interest calculations and scaling operations:

- **Compound Interest**: Calculates the latest lending/borrowing indices using a simple interest formula
- **Treasury Calculations**: Determines pending treasury amounts from reserve factors
- **Scaling Operations**: Converts between raw balances and face amounts using indices
- **Time-Based Calculations**: Handles timestamp-based interest accumulation

Constants — defines fixed protocol-wide values.

DataTypes — defines core types and data structures used across the protocol.

Privileged roles

Pool Owner: The Pool Owner has comprehensive control over protocol configuration and represents the highest level of centralized control.

Capabilities:

- **Reserve Management**: Add new token reserves with complete risk parameter configuration
- **Risk Parameter Control**: Modify LTV ratios (0-100%), liquidation thresholds (0-100%), and reserve factors (0-100%)
- **Interest Rate Strategy**: Update interest rate calculation contracts for any reserve
- **Supply & Borrow Limits**: Set lending limits (supply caps) and borrowing limits (debt caps)
- **Reserve Status**: Enable/disable reserves for lending and borrowing operations
- **Treasury Management**: Change protocol treasury address for fee collection
- **Oracle Configuration**: Modify price feed sources for token pricing

AxelarPool Contract: The AxelarPool contract has exclusive access to core pool operations, acting as the protocol's cross-chain gateway.

Capabilities:

- **User Operations**: Execute deposits, withdrawals, borrowing, and repayment on behalf of XRPL users
- **Collateral Management**: Enable/disable collateral usage for user positions
- **Liquidation Execution**: Perform liquidations of undercollateralized positions
- **Cross-Chain Coordination**: Coordinate token transfers with lending operations

Potential Risks

- Because admin updates take effect independently of cross-chain message ordering, a **malicious admin could observe pending deposits** or borrows and then lower the liquidation threshold or increase the LTV immediately before execution, causing sudden under-collateralization and forced liquidations.
- The owner's ability to arbitrarily enable or disable any reserve introduces centralization risk, potentially blocking user withdrawals, deposits, or repayments and impacting protocol functionality.
- Parameter validation allows liquidation threshold < LTV, making users unliquidatable and causing protocol to accumulate bad debt.
- Heavy reliance on Band Protocol Oracle creates a single point of failure for price data that affects the entire protocol.
- No circuit breaker exists to halt protocol operations during critical vulnerabilities or market stress.
- The functioning of the system significantly relies on specific external structures (Axelar). Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
- The project utilizes libraries or contracts without security audits, potentially introducing vulnerabilities. This compromises the security of the audited system, making it susceptible to attacks exploiting these external weaknesses. Among these contracts, **Interchain Token Service, InterchainTokenExecutable** and **InterchainToken** can be considered third-party and unaudited dependencies.
- Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions, increasing the risk of rapid exploitation and irreversible changes.
- The absence of restrictions on state variable modifications by the owner leads to arbitrary changes, affecting contract integrity and user trust, especially during critical operations like minting phases.
- Allowing the admin to set oracle addresses without constraints or verification mechanisms introduces the risk of incorrect or malicious oracle selection, affecting the accuracy of data and potentially leading to financial losses.

Findings

Vulnerability Details

F-2025-10701 - Collateral Seizure Without Debt Repayment - Critical

Description:

In `Pool.liquidate()`, the ERC-20 repayment logic only executes when `liquidator != address(0)`, a check intended to be true only for calls originating from the Axelar gateway. However, because any external user can set `liquidator = address(0)`, they can bypass the ERC-20 transfer entirely. This allows an attacker—bypassing `address(0)` as `liquidator` and their own XRPL key as `liquidationRewardRecipient`—to seize a borrower's full collateral (plus bonus) without ever actually repaying the debt.

As we can see, the `liquidate()` function can be called by setting the `liquidator` to any address including zero;

```
function liquidate(
    address       liquidator,
    bytes memory  liquidationRewardRecipient,
    bytes memory  liquidatee,
    address       debtToken,
    uint256       amount,
    address       collateralToken
) external reserveEnabled(debtToken) reserveEnabled(collateralToken) nonReent
rant {
 //...
    DataTypes.DebtRepaid memory debtRepaid =
        _repayDebtRouteInternal(liquidateeHash, debtToken, amount, liquidator)
;
 //...
}
```

And it skips the ERC-20 transfer part when the liquidator = address(0);

```
function _repayDebtInternal(...) internal {
    // ...
    if (liquidationRepaymentData.liquidator != address(0)) {
      //...
```

```
            _updateRatesAndRawTotalBorrowing(...);
             IERC20(token).safeTransferFrom(
                 liquidationRepaymentData.liquidator, address(_axelarPool), liq
uidationRepaymentData.repayAmount
             );
        } else {
        _updateRatesAndRawTotalBorrowing(...);
        }
        //...
    }
```

Impact:

- An attacker calling `liquidate(  address(0),   // …);`, forces the protocol to reduce internal reserves by `amount` and then send all collateral plus bonus to the attacker—without the protocol ever receiving any debt tokens in return.
- By repeating this attack on one or multiple undercollateralized positions, the attacker empties the pool's collateral reserves at zero cost by stealing from the protocol.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: Only allow the Axelar gateway to specify the liquidator as the **zero address**. Otherwise, if called by a user with the `liquidator = address(0)`, it should **revert**.

Resolution: In the fixed version, the function now explicitly checks:

```
if (liquidator == address(0)){
   require(msg.sender == _axelarPoolAddress, "Only the axelar pool can use th
```

```
e zero addressor liquidator.");
} else if (msg.sender != liquidator) {
    revert Errors.InvalidLiquidator();
}
```

Only `_axelarPoolAddress` can use `address(0)` as the `liquidator`, which aligns with the original assumption in the protocol design. (Revised commit: 2fcfec0)

Evidences

Reproduce:

Steps to reproduce the issue:

- **Setup a healthy loan position:**
  - Alice deposits 100 units of token A, valued at $50 each, with a 50% LTV → $2,500 collateral value.
  - She borrows 22.5 units of token B, valued at $100 each → $2,250 debt.
- **Trigger liquidation eligibility:**
  - Drop the price of token A to $40, reducing Alice’s collateral value to $2,000.
  - Since $2,000 < $2,250, her position becomes liquidatable.
- **Begin attack from Bob’s EVM address:**
  - Without holding or approving any token B, Bob calls the `liquidate()` function.
- **Pass malicious parameters:**
  - `liquidator` is set to `address(0)` to bypass the ERC-20 debt repayment.
  - `rewardRecipient` is set to Bob’s XRPL key to receive seized collateral.
  - The `amount` parameter claims to repay 6.25 token B—without actually sending any.
- **Protocol is tricked:**
  - Internally, the contract detects `liquidator == address(0)` and skips the token transfer.
  - Still, it proceeds to seize 18.75 worth of token A from Alice (including the bonus), transferring it to Bob’s XRPL balance.
- **Verify the theft:**
  - Using pool storage, it’s confirmed Bob’s XRPL account received the collateral without having paid any debt.
  - The pool loses assets; the attacker gains them at zero cost.

Test code:

```
function testMaliciousLiquidatorStealsCollateralWithoutPayingDebt() public {
    setupWithLoan();
```

```
// Alice originally deposited 100 token A priced at $50 with 50% LTV.
// => Collateral value: 100 * 50 * 0.5 = $2,500
// => Borrowed 22.5 token B priced at $100 = $2,250 debt.
```

[See more]

Results:

```
Ran 1 test for test/HackenPool.t.sol:PoolTest
[PASS] testMaliciousLiquidatorStealsCollateralWithoutPayingDebt() (gas: 11365
44)
Logs:
  Bob stole funds: 1875000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.40ms (690.17µs
CPU time)
```

F-2025-10708 - Missing msg.sender == liquidator Check Allows Front-Running and Reward+ Collateral Theft - High

Description: The `Pool.liquidate()` function accepts a `liquidator` parameter but never checks that `msg.sender == liquidator`. This oversight allows any attacker to **spoof a legitimate liquidator's address** and trigger the liquidation using that address's ERC-20 allowance.

The attacker can **divert the seized collateral and bonus to their own XRPL account** by setting `rewardRecipient` to their XRPL key. As a result, the honest liquidator loses both tokens and reward:

```
pool.liquidate(
   /* liquidator       = */ genuineLiquidatorAddress,
   /* rewardRecipient  = */ attackerXrplKey,
   ...
);
```

The spoofed transaction will:

- pull the ERC-20 tokens from the genuine liquidator,
- apply them to repay liquidatee's debt,
- and send the seized collateral and bonus **to the attacker**.

Impact:

- Anyone can front-run a liquidation and use someone else's `approve()`d allowance without consent.
- The attacker's XRPL key receives the *entire seized collateral*, including the original amount plus liquidation bonus.
- The real liquidator loses tokens without getting any collateral or compensation in return.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: High

Recommendations

Remediation:

Enforce `msg.sender == liquidator` and at the beginning of `liquidate()`, add:

```
if (msg.sender != liquidator) {
   revert Errors.UnapprovedLiquidator();
}
```

This prevents any caller other than the specified liquidator from using that address's allowance.

Resolution:

The issue is fixed by implementing the "`msg.sender == liquidator`" check in the `liquidate()` function:

```
function liquidate(…) external reserveEnabled(debtToken) reserveEnabled(colla
teralToken) nonReentrant {
     if (liquidator == address(0)){
         require(msg.sender == _axelarPoolAddress, "Only the axelar pool can
use the zero addressor liquidator.");
     }
     else if (msg.sender != liquidator) {
         revert Errors.InvalidLiquidator();
}
```

(Revised commit: d5aa85f6)

Evidences

PoC

Reproduce:

Steps to reproduce the issue:

- **Prepare Borrower (Alice):**
  - Deposit 100 tokens of `tokenA` priced at $50 each (total collateral value = $2,500).
  - Borrow 22.5 tokens of `tokenB` priced at $100 each (total debt = $2,250).
  - Collateral-to-debt ratio = safe (2,500 > 2,250).
- **Force Under-Collateralization:**
  - Simulate a price drop of `tokenA` to $40.

- New collateral value = 100 × 40 × 0.5 (LTV) = $2,000.
- Now, Alice becomes eligible for liquidation since $2,000 < $2,250.

- **Honest Liquidator Prepares (Bob):**
  - Mint 6.25 tokens of `tokenB` to Bob's EVM address.
  - Approve the pool to pull 6.25 `tokenB` from Bob (ERC-20 `approve()`).
- **Attacker Front-Runs Liquidation:**
  - Attacker sends a liquidation transaction with:
  - `liquidator` set to **Bob's EVM address** (to exploit Bob's ERC-20 allowance),
  - `rewardRecipient` set to **attacker's own XRPL address** (to steal the reward).
  - Call proceeds without validating `msg.sender == liquidator`.
- **Liquidation Outcome:**
  - The pool pulls 6.25 `tokenB` from Bob's balance using his allowance.
  - The borrower's position is closed.
  - The **entire seized collateral (including the liquidation bonus)** is transferred to the attacker's XRPL key, not to Bob.
- **Resulting Exploit:**
  - Bob loses his tokens but receives **no reward**.
  - The attacker gains **18.75 tokens of** `tokenA` (which includes the 20% liquidation bonus).
  - The protocol unintentionally rewards a malicious actor and punishes the honest partici
    See more

Results:

```
Ran 1 test for test/HackenPool.t.sol:PoolTest
[PASS] testLiquidateAllowsSpoofingLiquidator() (gas: 1167259)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.44ms (938.17µs
CPU time)
Ran 1 test suite in 257.71ms (2.44ms CPU time): 1 tests passed, 0 failed, 0 s
kipped (1 total tests)
```

F-2025-10712 - Duplicate Reserve Addition in liquidate() Leads to Invalid Rates - High

Description:

In `Pool.sol` contract's `liquidate()` function, it updates `totalReserveAmounts[debtToken]` by adding the repayment amount once, then immediately calls the internal rate-update routine—passing that same repayment amount as a positive "delta" to be added *again* when computing `reserveBalanceAfter`. As a result, the pool's on-chain reserves are artificially inflated by **twice** the true inbound liquidity. This inflated reserve number is then fed into the interest-rate strategy, yielding abnormally low borrowing and lending rates right after any liquidation.

First addition (in `liquidate()`):

```
totalReserveAmounts[debtToken] += amount;
```

Second addition (inside rate update):

```
// In _updateRatesAndRawTotalBorrowing:
uint256 reserveBalanceBefore = totalReserveAmounts[token];
reserveBalanceAfter = reserveBalanceBefore + absDeltaReserveBalance; // absDe
ltaReserveBalance == amount
```

Because `absDeltaReserveBalance` is the same `amount` already added, the pool's reserves end up inflated by **2×** the real incoming tokens.

As a result, the utilization ratio (`totalDebt / totalReserves`) is artificially low, causing both lending and borrowing rates to drop more than they should immediately after liquidation.

Impact:

- **Incorrect Borrow Rates:** Borrowers see unnaturally low interest shortly after a liquidation event, since the strategy thinks there's extra unused liquidity.
- **Incorrect Lending Returns:** Depositors' rates fall too much because the pool appears over-capitalized.
- **Market Distortion:** If multiple liquidations occur before reserves are used elsewhere, rates remain skewed over several blocks. Attackers or large liquidators could exploit this window to game borrow or deposit conditions.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: Move the `totalReserveAmounts[debtToken] += amount;` line to **after** the internal repayment call, so that the rate update only sees a single increment. For example:

```
function liquidate(...) external reserveEnabled(debtToken) reserveEnabled(collateralToken) nonReentrant {
    // ... rest of the code
    DataTypes.DebtRepaid memory debtRepaid =
        _repayDebtRouteInternal(liquidateeHash, debtToken, amount, liquidator
);

    totalReserveAmounts[debtToken] += amount;
    // ... rest of the code
}
```

Resolution: The Strobe team fixed the issue by moving the total reserve amount addition line to after the internal repayment call:

```
function liquidate(…) external reserveEnabled(debtToken) reserveEnabled(collateralToken) nonReentrant {
    // …
    DataTypes.DebtRepaid memory debtRepaid = _repayDebtRouteInternal(
        liquidateeHash,
        debtToken,
        amount,
        liquidator
    );
    totalReserveAmounts[debtToken] += amount;
    // …
}
```

(Revised commit: 5c0e4bd)

Evidences

PoC

Reproduce:

Steps to reproduce the issue:

- **Deploy and Initialize Pool with an Active Loan**
  - Set up Alice's position so she deposits token A as collateral and borrows token B.
- **Record Pre-Liquidation Reserve and Borrowing Data**
  - Read and store the pool's `totalReserveAmounts[tokenB]` and the reserve's `rawTotalBorrowing` for token B. These will serve as the baseline "before liquidation."
- **Force Alice into an Undercollateralized State**
  - Adjust the mock oracle so that token A's price falls sharply. This makes Alice's collateral insufficient to back her existing token B debt, qualifying her position for liquidation.
- **Prepare Bob as the Liquidator**
  - Switch context to Bob's address, mint exactly the same amount of token B that Alice owes, and approve the pool to pull that exact amount. This ensures Bob can repay Alice's debt in full.
- **Execute Liquidation**
  - Have Bob call `liquidate(...)` with his own address, his XRPL key as the reward recipient, Alice's XRPL key as the borrower, token B as the debt token, the exact borrowed amount, and token A as the collateral token. The pool will internally add Bob's `amount` to `totalReserveAmounts[tokenB]`, then call the rate-update helper with that same `amount` again, effectively counting it twice.
- **Compute Expected Post-Liquidation Values**
- **Expected Total Reserve for token B:** should be *initialReserve + borrowedAmount* (only a single addition).
- **Expected Raw Total Borrowing:** should be *initialRawTotalBorrowing - the raw units corresponding to the borrowedAmount*.
- **Expected Scaled-Up Debt:** calculate from the updated raw total bor
  See more

Results:

```
Failing tests:
Encountered 1 failing test in test/HackenPool.t.sol:PoolTest
[FAIL: assertion failed: 81858582296121139153737 != 8191015625000000000000]
```

```
testReserveAmountAddedTwice() (gas: 1208185)

Encountered a total of 1 failing tests, 0 tests succeeded
```

F-2025-10731 - Borrow Amount Is Always Lesser Than Advertised Amount Due To Wrong Function Parameter - High

Description:

During the security audit, it was identified a critical parameter bug in the core borrowing function that significantly reduces user borrowing capacity below advertised levels. This bug prevents users from accessing their full LTV (Loan-to-Value) borrowing rights, creating a competitive disadvantage and false advertising.

The `borrow()` function incorrectly uses **liquidation threshold** logic instead of **LTV logic**.

Pool.sol:

```
function borrow(DataTypes.XrplAccountHash borrowerHash, bytes memory borrower
, address token, uint256 amount)
    external
    reserveEnabled(token)
    onlyAxelarPool
{
    . . .

    // Confirm collateralization after all calculations
    _assertNotUnderCollateralized(borrowerHash, true); // @audit-issue - it s
hould be false

    . . .
}
```

With typical DeFi parameters (75% LTV, 85% liquidation threshold):

Maximum Borrowable Amount

- **Advertised (LTV-based)**: $1000 collateral × 75% = $750
- **Actual (Bug)**: $1000 collateral × 75% × 85% = $637.50
- **Capacity Loss**: $750 - $637.50 = $112.50 (15% reduction)

Borrowing should use an LTV limit, not a liquidation threshold, to allow maximum advertised borrowing capacity. Setting the `_assertNotUnderCollateralized(borrowerHash, true)` to `_assertNotUnderCollateralized(borrowerHash, false)` helps users to reach the advertised amounts for the `borrow()` operation.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Mitigated

Classification

Impact Rate: 3/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Medium

Severity: High

Recommendations

Remediation: Change the boolean parameter from `true` to `false`:

```
function borrow(...) external {
   // ... existing logic ...

   // Confirm collateralization after all calculations
   _assertNotUnderCollateralized(borrowerHash, false);

   totalReserveAmounts[token] -= amount;
   emit Borrowing(borrower, token, amount, amount);
}
```

Resolution: **Mitigated:** The finding was stated as a **design choice** by the client with the following statement.

> So, this is by design
> It essentially acts as a risk factor that inflates the borrow value relative to the underlying debt value.
> e.g. using XRP:
> - XRP price = $5
> - I deposit 50 XRP → underlying value = 50 × $5 = $250
> - The LTV ratio is 90%, so my borrowing power = $250 × 0.9 = $225
>
> However, if I choose to borrow XRP, there's an additional constraint:
> - The Liquidation Threshold (LT) for XRP is 70%, meaning borrowing XRP directly requires inflating the borrow value to borrowed amount / 70%.
> To stay within the $225 limit:
> - Maximum borrowable XRP = $225 × 0.7 / $5 = 31.5 XRP
> Although I'm borrowing 31.5 XRP, its actual market value is

only $157.5. but for liquidation purposes, it's counted as $225. At that moment my collateral ratio is 1.And price going down wont trigger liquidation as collateral value and debt value equalized. However, it still being liquidated as borrow interest always goes faster than the deposit interest, causing the borrowing value > collateral value, collateral ratio < 1

Evidences

PoC

Reproduce:

Test code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from '../lib/forge-std/src/Test.sol';
import {Pool} from '../src/core/Pool.sol';
import {AxelarPool} from '../src/axelar/AxelarPool.sol';
import {InterestRateStrategyOne} from '../src/core/irs/InterestRateStrategyOn
e.sol';
import {MockERC20} from './mocks/MockERC20.sol';
import {MockStdReference} from './mocks/MockStdReference.sol';
import {BandProtocolConnector} from '../src/oracles/BandProtocolConnector.sol
';
import {OracleConnectorHub} from '../src/oracles/OracleConnectorHub.sol';
import {DataTypes} from '../src/core/libraries/DataTypes.sol';
import {Math} from '../src/math/Math.sol';

contract BorrowCollateralCheckBugPoC is Test {
    AxelarPool axelarPool;
    Pool pool;
    MockERC20 token;

    // Test users
    DataTypes.XrplAccountHash borrowerHash = DataTypes.bytesToXrplAccountHash
(abi.encode("borrower"));
    DataTypes.XrplAccountHash lenderHash = DataTypes.bytesToXrplAccountHash(a
bi.encode("lender"));

    // Protocol configuration
    uint8 constant LTV = 75;                    // 75% LTV (users should borr
ow up to 75%)
    uint8 constant LIQUIDATION_THRESHOLD = 85;  // 85% liquidation threshold
```

```
    function setUp() public {
        // Setup protocol components
        token = new MockERC20("Token", "TKN", 18);

        DataTypes.InterestRateStrateyOneParams memory params = DataTypes.Inte
restRateStrateyOneParams({
            slope0: 15, slope1: 60, baseRate: 5, optimalRate: 75
        });
        InterestRateStrategyOne strategy = new InterestRateStrategyOne(params
);

        MockStdReference mockRef = new MockStdReference();
        mockRef.setReferenceData("TKN", "USD", 1e18, block.timestamp, block.t
imestamp);

        BandProtocolConnector connector = new BandProtocolConnector(mockRef,
"TKN", 40 minutes);
        OracleConnectorHub oracleHub = new OracleCo
```

[See more]

Results:

```
Logs:
  Setup: $10000 collateral, 75% LTV, 85% liquidation threshold
  Expected borrowing capacity: $7500
  Actual borrowing capacity: $6375
  User loss: $1125 (15% reduction)

  1. Testing bug-limited amount ($6,375):
     [SUCCESS] Can borrow $6375
  2. Testing advertised amount ($7,500):
     [EXPECTED] Cannot borrow advertised amount
```

F-2025-10709 - Dynamic LTV and Liquidation Threshold Means Borrowers' Terms Can Change Retroactively - Medium

Description:

When a user borrows, the contract uses the *current* global `ltvPct` and `liquidationThresholdPct` for all collateralization and withdrawal checks. Because these parameters are owner-controlled and not "locked in" per loan, the owner can change them at any time—instantly altering the safety requirements for every existing borrower. This can force perfectly valid positions into undercollateralized status or block withdrawals, even though borrowers acted in good faith under the original terms.

In `PoolConfig.sol`, both parameters are modified without delay or per-loan snapshots:

```
 function setLtv(address token, uint8 ltvPct) external reserveExists(token) on
lyOwner {
    if (ltvPct > DataTypes.ONE_HUNDRED_PCT) {
        revert Errors.LtvRange();
    }
    _reserves[token].ltvPct = ltvPct;
    emit LtvUpdate(token, ltvPct);
}
function setLiquidationThreshold(address token, uint8 liquidationThresholdPct
)
    external reserveExists(token) onlyOwner
{
    if (liquidationThresholdPct > DataTypes.ONE_HUNDRED_PCT) {
        revert Errors.LiquidationThresholdRange();
    }
    _reserves[token].liquidationThresholdPct = liquidationThresholdPct;
    emit LiquidationThresholdUpdate(token, liquidationThresholdPct);
}
```

Whenever a borrower takes out or adjusts a loan, the contract invokes:

```
 // Called after borrow() or withdraw()
_assertNotUnderCollateralized(borrowerHash, true);
// Internally:
function _assertNotUnderCollateralized(DataTypes.XrplAccountHash user, bool a
pplyLiquidationThreshold) internal view {
    if (!_isNotUndercollateralized(user, applyLiquidationThreshold)) {
        revert Errors.InsufficientCollateral();
    }
```

```
}
function _isNotUndercollateralized(DataTypes.XrplAccountHash user, bool apply
LiquidationThreshold)
    internal
    view
    returns (bool)
{
    // Fast track if no debt
    if (!userHasDebt(user)) {
        return true;
    }
    DataTypes.UserCollateralData memory data = calculateUserCollateralData(use
r, applyLiquidationThreshold);
    return data.collateralRequired <= data.collateralValue;
}
```

Here, `calculateUserCollateralData(...)` computes

```
collateralValue = ∑(user’s deposited USD value × current LTV)
collateralRequired = ∑(user’s debt USD value ÷ (liquidationThreshold / 100))
```

Both `ltvPct` and `liquidationThresholdPct` come directly from `_reserves[token]` rather than from a per-loan snapshot.

Impact:

- **Surprise Liquidations:**
  - A borrower deposits 1 000 TokensA at $100 each (collateral = $100 000) and borrows $75 000 at 75% LTV.
  - Later, the owner instantly lowers `ltvPct` to 50%. Without changing their collateral or debt, the borrower’s collateral now safely backs only $50 000, so they become critically undercollateralized. In the next block, the borrower can be liquidated—even though they believed they had a 75% safety buffer.
- **Blocked Withdrawals:**
  - Suppose a borrower tries to withdraw part of their collateral under the old threshold. If the owner raises `liquidationThresholdPct` from 80% to 90%, that same withdrawal will fail `InsufficientCollateral()`, locking collateral that was previously withdrawable.
- **Centralization & Trust Risk:**
  - Because **any** owner can call `setLtv(...)` or `setLiquidationThreshold(...)` at any time, borrowers have no guarantee their loan terms remain stable. An owner role could immediately force arbitrary liquidations or freeze collateral.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]
- core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Accepted

Classification

Impact Rate: 5/5

Likelihood Rate: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Store the `ltvPct` and `liquidationThresholdPct` *at the moment of borrowing* in each position struct. During collateral checks, use those stored values instead of the global ones.

Resolution: The finding is accepted and no further changes are applied to fix the issue.

F-2025-10715 - Zero Oracle Price Check is Missing - Medium

Description:

The `Pool` contract relies on the oracle price in multiple critical places—specifically, when computing a user's USD-valued debt and USD-valued collateral. However, there is **no guard** against the oracle returning a zero price. When `getPrice(token) == 0`, it breaks the collateral/debt logic :

Debt Value Drops to Zero → Free Collateral Removal

```
    function getUserDebtUsdValueForToken(DataTypes.XrplAccountHash user, addr
ess token)
        internal
        view
        returns (uint256)
    {
      // ... rest of the code

        uint256 debtPrice = _oracle.getPrice(token);

        uint256 debtValue = Math.mulDecimals(debtPrice, scaledUpDebtBalance,
decimals);

        return debtValue;
    }
```

If `debtPrice == 0`, then `debtValue == 0`. Later, in `_assertNotUnderCollateralized(...)`, it checks:

```
collateralRequired = debtValue.rdiv(liquidationThreshold)
                   = 0 / liquidationThreshold = 0
collateralValue    = the enabled amount user deposited converted to USD
require(collateralValue ≥ collateralRequired)  // always true since the colla
teralRequired = 0
```

As a result, a borrower with outstanding token B debt can suddenly "owe" zero USD, letting them call `withdrawAll(...)` and drain every token A collateral—despite still owing raw token B.

Impact

- **Debt Escape:** If the oracle reports a debt token's price as zero, a borrower's `debtValue` becomes zero, allowing them to call `withdrawAll()` and remove all collateral despite still owing the raw debt.

- **Oracle Attack/DoS:** A malicious or faulty oracle can set any token price to zero, enabling theft of collateral or preventing legitimate withdrawals, undermining protocol safety.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Before using an oracle price, require it to be strictly positive. For example:

```
uint256 price = _oracle.getPrice(token);
if (price == 0) {
   revert Errors.InvalidPrice();
}
```

Resolution: The finding is fixed by implementing zero price checks in the out-of-scope oracle contracts; `BandProtocolConnector` and `OracleConnectorHub`.

(Revised commit: 78f82a2)

F-2025-10718 - Debt Calculation Logic is Broken for High-Index Reserves - Medium

Description:

The `DEBT_FLAG_FILTER` constant is designed to check if a user has debt in any reserve by masking the odd-numbered bits in the `userFlags` mapping. However, the bitmask is incorrectly constructed and **excludes bit 255**, which represents debt for reserve index 127 (the highest possible reserve index).

One `a` character is missing from the `DEBT_FLAG_FILTER`. As the protocol supports up to 127 reserves, this causes a significant problem for high-index reserves.

Simply, `DEBT_FLAG_FILTER` returns;

```
b001010101010..[REDACTED]..1010
```

The representation of reserve debts has a length of 252. Considering that the first two bytes are aimed for collateral reserve. The total length allocated to debt calculations is 250 (250/2 => 125 Reserves)

Therefore, this logic will work for the first R0, R1, R2, ..., and R124. But it will stop working on **Reserve125** as that extra character is missing from the mask value.

Pool.sol:

```
// b10101010...1010
    uint256 constant DEBT_FLAG_FILTER = 0x2aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa;
```

```
function userHasDebt(DataTypes.XrplAccountHash user) public view returns (boo
l) {
    uint256 map = userFlags[user];

    uint256 andResult = map & DEBT_FLAG_FILTER;
    return andResult != 0;
}
```

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider fixing the `DEBT_FLAG_FILTER` value by adding one extra `a` character.

Resolution: The finding was **fixed** by the Strobe team after they corrected the `DEBT_FLAG_FILTER` value in the commit **e2b31a1**.

Evidences

PoC

Reproduce:

Steps to Reproduce:

1. User deposits collateral in reserve 0
2. User borrows from reserve 127
3. `userHasDebt()` returns `false` (due to bug)
4. User calls `disableCollateral()` - NO checks performed
5. User withdraws all collateral while keeping debt
6. Protocol becomes insolvent

Test code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from '../lib/forge-std/src/Test.sol';
import {Pool} from '../src/core/Pool.sol';
import {AxelarPool} from '../src/axelar/AxelarPool.sol';
import {InterestRateStrategyOne} from '../src/core/irs/InterestRateStrategyOn
e.sol';
import {MockERC20} from './mocks/MockERC20.sol';
import {MockStdReference} from './mocks/MockStdReference.sol';
import {BandProtocolConnector} from '../src/oracles/BandProtocolConnector.sol
';
```

```
import {OracleConnectorHub} from '../src/oracles/OracleConnectorHub.sol';
import {DataTypes} from '../src/core/libraries/DataTypes.sol';

contract DebtFlagFilterVulnerabilityTest is Test {
    AxelarPool axelarPool;
    Pool pool;
    MockERC20 tokenA;

    address deployer = address(0xDEAD);
    DataTypes.XrplAccountHash treasury = DataTypes.XrplAccountHash.wrap(bytes
32(uint256(0x2222)));
    DataTypes.XrplAccountHash userHash = DataTypes.bytesToXrplAccountHash(abi
.encode("user"));

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

        // Deploy infrastructure (simplified from AxelarPool.t.sol pattern)
        tokenA = new MockERC20("Token A", "A", 18);
        MockStdReference mockRef = new MockStdReference();
        mockRef.setReferenceData("XRP", "USD", 1e18, block.timestamp, block.t
imestamp);

        BandProtocolConnector connector = new BandProtocolConnector(mockRef,
"XRP", 40 minutes);
        OracleConnectorHub oracleHub = new OracleConnectorHub();
        oracleHub.setTok
```

See more

Results:

```
 [PASS] test_PracticalImpact_DisableCollateralBypass() (gas: 13005)
Logs:
  Normal case - Reserve 0 debt detected: true
  Vulnerable case - Reserve 127 debt detected: false
```

F-2025-10721 - Inconsistency Between the Documentation and Code: Missing Base Rate Calculation - Medium

Description:

According to the official documentation, when the **Utilization Rate** is lower than **the Optimal Utilization Rate**, the **Borrow Rate** is calculated as follows:

$$U \leq U_{optimal} \Rightarrow R_{borrow} = R_0 + \frac{U}{U_{\text{optimal}}}(R_{\text{slope1}})$$

R0 is the base rate in that calculation. However, the actual code does not follow that specific invariant. The case when the **Utilization Rate** equals `zero` was not covered.

The `getInterestRates()` function returns a **0%** borrowing rate when utilization is **0%**, bypassing the configured base rate. This situation might be problematic for protocol users as they expect to see the actual Base Ratio for borrow rates. As it is more UI/UX bug, it was observed that the protocol **works as intended**.

Additionally, **inconsistency between the code and the official documentation** can lead to reliability issues.

InterestRateStrategyOne.sol:

```
 function getInterestRates(uint256 reserveBalance, uint256 totalDebt)
     external
     view
     returns (DataTypes.InterestRates memory interestRates)
{
    uint256 utilizationRate = calculateUtilizationRate(reserveBalance, totalD
ebt);
    if (utilizationRate > 0) { // @audit-issue : zero utilization returns 0%
instead of 5% base rate
        uint256 borrowingRate = calculateBorrowRate(utilizationRate);
        uint256 lendingRate = borrowingRate.rmul(utilizationRate);

        // Checked no overflow using validateMaxBorrowingRate already
        interestRates.borrowingRate = uint104(borrowingRate);
        interestRates.lendingRate = uint104(lendingRate);
    }

}
```

When it occurs:

- The first borrower in any new reserve
- The first borrower after all debt is repaid (therefore, can be repetitive)
- Other Zero-utilization scenarios

Assets:

- core/irs/InterestRateStrategyOne.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 4/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: To fix this issue, the base rate should be included in the **else-if-case** when the **utilization rate** is zero at a given time according to the official documentation.

Resolution: The given issue is fixed by considering also zero utilization rate possibility in the function:

```
    function getInterestRates(uint256 reserveBalance, uint256 totalDebt)
      external
      view
      returns (DataTypes.InterestRates memory interestRates)
  {
      uint256 utilizationRate = calculateUtilizationRate(reserveBalance, tot
alDebt);
      if (utilizationRate > 0) {
          uint256 borrowingRate = calculateBorrowRate(utilizationRate);
          uint256 lendingRate = borrowingRate.rmul(utilizationRate);
          interestRates.borrowingRate = uint104(borrowingRate);
          interestRates.lendingRate = uint104(lendingRate);
      }
      else{
          uint256 borrowingRate = strategyParams.baseRate;
          uint256 lendingRate = borrowingRate.rmul(utilizationRate);
```

```
            interestRates.borrowingRate = uint104(borrowingRate);
            interestRates.lendingRate = uint104(lendingRate);
        }

    }
```

(Revised commit: cabd7ce)

F-2025-10724 - Borrowing Functionality Will Be Lost When Loan-to-Value Greater Than Liquidation Threshold - Medium

Description:

The protocol allows configuring **LTV** (Loan-to-Value) higher than the **Liquidation Threshold**, breaking fundamental DeFi lending logic. Users cannot borrow at the advertised LTV percentage when `LTV > liquidation threshold`. There is no prevention in the `PoolConfig.sol` to prevent this situation from occurring.

When borrowing, the protocol uses a liquidation threshold for collateralization checks instead of LTV:

PoolConfig.sol:

```
function setLtv(address token, uint8 ltvPct) external reserveExists(token) on
lyOwner {
    if (ltvPct > DataTypes.ONE_HUNDRED_PCT) {
        revert Errors.LtvRange();
    }

    _reserves[token].ltvPct = ltvPct;

    emit LtvUpdate(token, ltvPct);
}
```

PoolConfig.sol:

```
function setLiquidationThreshold(address token, uint8 liquidationThresholdPct
)
    external
    reserveExists(token)
    onlyOwner
{
    if (liquidationThresholdPct > DataTypes.ONE_HUNDRED_PCT) {
        revert Errors.LiquidationThresholdRange();
    }

    _reserves[token].liquidationThresholdPct = liquidationThresholdPct;

    emit LiquidationThresholdUpdate(token, liquidationThresholdPct);
}
```

Pool.sol:

```
function borrow(DataTypes.XrplAccountHash borrowerHash, bytes memory borrower
, address token, uint256 amount)
    external
    reserveEnabled(token)
    onlyAxelarPool
{
    . . .

    // Confirm collateralization after all calculations
    _assertNotUnderCollateralized(borrowerHash, true); // root cause

    . . .
}
```

Due to this broken logic and missing check, borrow functionality will be completely lost when `ltvPct > liquidationThresholdPct`.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]
- core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Dependent

Complexity: Medium

Severity: Medium

Recommendations

Remediation: Consider implementing an extra check to prevent the case of `LTV > liquidation threshold`.

Resolution: This finding was **acknowledged** by the Strobe team with the following statement:

> The borrowing capacity would be 600, as per the ltv calculation finding. The liquidation threshhold would then be 600 * 0.75, so 450. It's actually ok for the liquidation thresh

hold to be higher than the LTV. After our discussion, I found an example in the documentation that actually has this:

[Overcollateralization and collateralization ratio | Strobe Protocol]

Evidences

PoC

Reproduce:

Steps to Reproduce:

1. User deposits: $1000
2. Protocol advertises: 80% LTV = $800 borrowing capacity
3. User can actually borrow: $600 (25% less than advertised)

Test code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from '../lib/forge-std/src/Test.sol';
import {Pool} from '../src/core/Pool.sol';
import {AxelarPool} from '../src/axelar/AxelarPool.sol';
import {InterestRateStrategyOne} from '../src/core/irs/InterestRateStrategyOne.sol';
import {MockERC20} from './mocks/MockERC20.sol';
import {MockStdReference} from './mocks/MockStdReference.sol';
import {BandProtocolConnector} from '../src/oracles/BandProtocolConnector.sol';
import {OracleConnectorHub} from '../src/oracles/OracleConnectorHub.sol';
import {DataTypes} from '../src/core/libraries/DataTypes.sol';

contract SimpleLTVIssuePoC is Test {
    AxelarPool axelarPool;
    Pool pool;
    MockERC20 token;
    DataTypes.XrplAccountHash userHash = DataTypes.bytesToXrplAccountHash(abi.encode("user"));

    function setUp() public {
        // Deploy protocol with standard setup
        token = new MockERC20("Token", "TKN", 18);
        MockStdReference mockRef = new MockStdReference();
        mockRef.setReferenceData("TKN", "USD", 1e18, block.timestamp, block.timestamp);
```

```
        BandProtocolConnector connector = new BandProtocolConnector(mockRef,
"TKN", 40 minutes);
        OracleConnectorHub oracleHub = new OracleConnectorHub();
        oracleHub.setTokenConnector(address(token), address(connector));

        DataTypes.InterestRateStrateyOneParams memory params = DataTypes.Inte
restRateStrateyOneParams({
            slope0: 8, slope1: 100, baseRate: 5, optimalRate: 65
        });
        InterestRateStrategyOne strategy = new InterestRateStrategyOne(params
);

        axelarPool = new AxelarPool(
            address(0x1a7580C2ef5D48
```

[See more]

Results:

```
Logs:
  User deposits: $1000
  Protocol advertises: 80% LTV = $800 borrowing capacity
  User can actually borrow: $600 (25% less than advertised)
```

F-2025-10727 - Wrong Comparison on ReserveFactor Percentage Makes The Upper Limit Inconsistent - Medium

Description:

A reserve factor is a fundamental economic parameter in DeFi lending protocols that determines what percentage of interest earned from borrowers goes to the protocol treasury (fees) versus being distributed to lenders.

```
Borrower pays 10% APY on a loan
Reserve factor = 15%
Protocol keeps: 15% × 10% = 1.5% APY as fees
Lenders receive: 10% - 1.5% = 8.5% APY
```

The protocol incorrectly validates reserve factor percentages by comparing them against `Math.RAY (1e27)` instead of `DataTypes.ONE_HUNDRED_PCT (100)`. This bug allows protocol administrators to accidentally configure economically impossible fee structures that could lead to protocol insolvency.

The `_setReserveFactor()` function accepts `uint8 reserveFactorPct` (0-255 range). It compares that parameter against `Math.RAY = 1,000,000,000,000,000,000,000,000,000`. The problem is, that no `uint8` value can exceed `1e27`.

All input values (0-255) pass validation, including invalid percentages. That makes the higher limit unusable.

Additionally, in case the malicious owner sets the `reserveFactorPct` to its highest possible limit (`255`), the protocol can send all assets to the treasury, or, the actual protocol logic can be completely broken.

PoolConfig.sol:

```
function _setReserveFactor(address token, uint8 reserveFactorPct) internal re
serveExists(token) {
    if (reserveFactorPct > Math.RAY) {
        revert Errors.ReserveFactorRange();
    }

    _reserves[token].reserveFactorPct = reserveFactorPct;
    emit ReserveFactorUpdate(token, reserveFactorPct);
}
```

Assets:

- core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 5/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Fix the aforementioned function as below:

```
function _setReserveFactor(address token, uint8 reserveFactorPct) internal re
serveExists(token) {
    if (reserveFactorPct > DataTypes.ONE_HUNDRED_PCT) {
        revert Errors.ReserveFactorRange();
    }
    _reserves[token].reserveFactorPct = reserveFactorPct;
    emit ReserveFactorUpdate(token, reserveFactorPct);
}
```

Resolution: The finding was resolved by the Strobe team in commit **748250d**. The suggested fix was implemented.

F-2025-10750 - Lending Limit Is Not Enforced - Medium

Description: Although the protocol defines a `lendingLimit` for each reserve, this value is never checked during deposit operations. As a result, users can deposit arbitrary amounts of tokens into the pool, even when the `lendingLimit` has been set.

The relevant function:

```
function deposit(...) external {
   ...
  reserve.rawTotalDeposit += scaledDownAmount;

}
```

Impact:

- Excessive deposits in one token may distort lending/borrowing dynamics.
- The `lendingLimit` is meant to cap exposure to volatile or manipulated assets, but its lack of enforcement nullifies that control.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Enforce the `lendingLimit` in deposit-related functions by checking that `rawTotalDeposit + amount <= lendingLimit`. Revert the transaction if the new deposit would exceed the cap.

Resolution:

The issue is fixed by enforcing the per-reserve `lendingLimit` by reverting deposits that would cause the total scaled-up deposits to exceed the configured cap:

```
    function _assetLendingLimitSatisfied(address token) internal view {
        uint256 scaledUpLend = IndexLogic.getScaledUpAmount(_poolConfig.getReserveRawTotalDeposit(token), _poolConfig.getReserveLendingIndex(token));

        if (_poolConfig.getReserveLendingLimit(token) < scaledUpLend) {
            revert Errors.LendingLimitExceeded();
        }
    }
```

(Revised commit: b098c9e)

F-2025-10679 - Missing Replay Protection for Cross-Chain Commands in AxelarPool - Low

Description: The `AxelarPool` contract inherits `executeWithInterchainToken(commandId, …)` from `InterchainTokenExecutable`, using `commandId` as a unique identifier for each cross-chain message. However, `AxelarPool` does **not** track or reject previously seen `commandId`'s, allowing a malicious or misbehaving relayer to replay the same message multiple times. This opens the protocol to unintended duplicate deposits, withdrawals, borrows, or repayments, each of which can be used to drain liquidity or inflate debt positions.

Impact:

- **Duplicate Value Flows:**
  - **Deposits**: A single deposit message replayed twice turns a 100-token deposit into 200, crediting unintended balance.
  - **Withdrawals**: A withdrawal command replay could drain more tokens than originally intended.
  - **Borrows/Repays**: Borrow or repay calls replayed can manipulate user debt and liquidity, potentially freezing or draining the pool.
- **Economic Manipulation:** Attackers can exploit duplicate borrows and repayments to push utilization, skew interest-rate logic, or withdraw collateral.

Assets:

- axelar/AxelarPool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation:

Implement a `processed` Mapping

```
mapping(bytes32 => bool) private processed;
```

Reject replays

At the start of `_executeWithInterchainToken`, add:

```
require(!processed[commandId], Errors.CommandAlreadyExecuted());
processed[commandId] = true;
```

Emit an Event

Optionally emit `CommandExecuted(bytes32 commandId, address executor)` to facilitate off-chain monitoring of command processing.

Resolution:

The issue is fixed by implementing `executedCommands` mapping that tracks all the command IDs:

```
function _executeWithInterchainToken(...) internal override {
  if(executedCommands[commandId]) {
      revert("Replay detected");
  }
  executedCommands[commandId] = true;
  // ...
}
```

(Revised commit: 3238965)

F-2025-10734 - Hardcoded Gas Value - Low

Description:

The AxelarPool contract contains a critical implementation flaw where all cross-chain token transfers use a hardcoded `gasValue` of `0`, potentially causing transaction failures on destination chains. This affects all major protocol operations including withdrawals, borrowing, and trapped token recovery.

In all `InterchainTokenService.interchainTransfer()` calls, hardcoded `gasValue: 0` parameter was used instead of proper gas estimation.

AxelarPool.sol:

```
InterchainTokenService(interchainTokenService).interchainTransfer(
                requestedTokenId, // bytes32 tokenId,
                acceptedSourceChain, // string calldata destinationChain,
                sourceAddress, // bytes calldata destinationAddress,
                requestedAmount, // uint256 amount,
                "", // bytes calldata metadata,
                // TODO: how do we estimate gas?
                0 // uint256 gasValue
            );
```

This situation creates confusion and reliability issues for protocol users.

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Consider fixing TODO messages from the code and implement functionally working gas estimations for inter-chain operations.

Resolution: The finding was fixed in commit **e322521**. The gas estimation design was completely changed. It will now be sent as `msg.value` rather than a function argument, and each `acceptedGasTokenId` will be checked.

F-2025-10713 - Unbounded, Costly Loop in Collateral Checks - Info

Description: The function `calculateUserCollateralData(...)` iterates over **all** configured reserves every time it needs to compute a user's total collateral value and required collateral. Because the number of reserves (`_reserveCount`) is unbounded up to a protocol-wide maximum (127 in this implementation), each call can consume significant—and growing—gas. Over time, as more reserves are added, the per-user collateral check becomes increasingly expensive. This can lead to:

High Gas Costs for Normal Operations

- Any action requiring a collateral check (borrow, withdraw, liquidation eligibility, etc.) invokes `calculateUserCollateralData(...)`. If there are, say, 100 reserves, the loop runs 100 iterations even though the user may only have deposited assets in 2 or 3 of them.
- As reserves grow, gas per operation scales linearly, making routine user interactions prohibitively expensive.

Potential Denial-of-Service or Revert

- In the worst case, a user with deposits may trigger a collateral check that exceeds the block gas limit, causing transactions (withdraw, borrow, repay, liquidation) to revert.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Instead of looping over every configured reserve, keep a simple list (or set) of which reserves each user has nonzero collateral in. Then have `calculateUserCollateralData(...)` only iterate that per-user list. This way, gas costs grow with a user's actual positions rather than the total number of reserves.

Resolution: The risk of the given finding is accepted and no further changes are applied to fix the issue.

F-2025-10716 - Missing Zero Value Checks for liquidationThresholdPct and ltvPct - Info

Description: The protocol allows setting the `liquidationThresholdPct` and `ltvPct` parameters to any value below 100%. However, setting either of them to **zero** can lead to unintended or unsafe behavior:

In the `getCollateralUsdValueRequiredForToken()` function, when `applyLiquidationThreshold == true`, the debt value is divided by the scaled `liquidationThresholdPct`:

```
uint256 liquidationThreshold = Math.scalePct(getReserveData(token).liquidationThresholdPct);
uint256 collateralRequired = debtValue.rdiv(liquidationThreshold); // potential division by zero
```

If `liquidationThresholdPct` is 0, the scaled value becomes zero, leading to a division-by-zero error and a revert.

Similarly, `ltvPct` is used to discount collateral in:

```
return collateralValue.rmul(Math.scalePct(reserve.ltvPct));
```

If ltvPct is 0, the function will always return 0, rendering all collateral valueless — which can break deposits, withdraws, and borrowing logic silently.

Impact:

- **Division-by-Zero Reverts**: Any on-chain calculation relying on `liquidationThresholdPct` may revert if it's `0`, disrupting core functions like collateral checks and debt assessments.
- **Silent Logic Failures**: An `ltvPct` of `0` will silently invalidate all collateral usage, preventing borrowing and potentially blocking deposits or undercollateralization checks.
- **Inconsistent Protocol Configuration**: Zero values for these parameters don’t appear to have a practical use case and suggest a misconfiguration that should be restricted.

Assets:

- core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Enforce a non-zero minimum (e.g., `> 0`) for both `ltvPct` and `liquidationThresholdPct` in reserve setup functions.

Resolution: The required zero value checks are implemented for both `ltvPct` and `liquidationThresholdPct` variables. (Revised commit: 78f82a2)

F-2025-10717 - Redundant applyLiquidationThreshold Parameter in Collateral Assertion Functions - Info

Description: In `Pool` contract, both `_assertNotUnderCollateralized()` and `_assertNotOvercollateralized()` accept a `bool applyLiquidationThreshold` parameter, allowing collateral checks to optionally apply the `liquidationThresholdPct`. However:

- In **all usages of** `_assertNotUnderCollateralized()`, the parameter is always passed as `true`.
- In **all usages of** `_assertNotOvercollateralized()`, the parameter is always passed as `false`.

This effectively turns the parameter into a hardcoded constant per function, making its presence misleading. Despite being designed to support conditional behavior, the functions are never used in a dynamic way.

Dead or redundant logic branches add complexity and room for silent failure. It may be assumed conditional logic exists where it does not, potentially leading to incorrect assumptions or misuses.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: If there are no plans to pass varying values in the future, consider removing the parameter from both functions and hardcoding the respective logic (`true` or `false`).

F-2025-10751 - Missing Zero-Address Validation - Info

Description: Throughout the system contracts, there are several functions and constructor parameters that do not enforce a check against the zero address (`address(0)`). Specifically:

- `setTreasury()` function allows setting the treasury field to `bytes32(0)` (or the zero address, when converted), with no validation. This could inadvertently disable treasury-based fees or redirect rewards to a null location.
- `Pool`'s constructor takes three addresses (`poolConfigManager`, `oracleConnectorHub`, and `axelarPool`) and a `treasury` hash, but there is no check to ensure any of these inputs are non-zero. As a result, deploying with a zero address would break owner-based logic, price feeds, Axelar routing, or fee collection immediately.
- `_addReserve(address token, IInterestRateStrategy strategy, ...)` registers a new reserve without verifying that `token` or `strategy` are non-zero. Allowing either to be zero corrupts reserve mappings and undermines reserve existence checks.
- `_setInterestRateStrategy(address token, address newStrategy)` updates a reserve's interest-rate strategy but does not verify that `token` or `newStrategy` are non-zero. Setting either to zero breaks rate calculations or marks a non-existent reserve as valid.

Assets:

- core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]
- core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]

Status: Fixed

Classification

Impact Rate: 4/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Insert `require(... != address(0))` checks at the beginning of every constructor, public, and external function that accepts an address parameter. This guarantees protocol invariants and prevents any zero-address from entering critical state.

Resolution: Required missing zero-address validations are implemented for the mentioned functions. (Revised commit: 78f82a2)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/strobe-protocol/strobe-v1-core
Commit	c3281219a141c0575692d34c8c8d1c7ce0a16b40
Retest	78f82a280b1ae9d52167259d83e5f05e452a2ef7
Whitepaper	Strobe Protocol Gitbook
Requirements	README.md
Technical Requirements	README.md

Asset	Type
axelar/AxelarPool.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/irs/InterestRateStrategyOne.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/libraries/Constants.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/libraries/DataTypes.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/libraries/IndexLogic.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/Pool.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
core/PoolConfig.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
math/Math.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
oracles/BandProtocolConnector.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
oracles/BaseConnector.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract
oracles/OracleConnectorHub.sol [https://github.com/strobe-protocol/strobe-v1-core]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of the **Strobe Protocol**, Hacken followed its methodology by performing invariant testing on the project's main functions. Foundry, a tool used in the Solidity testing framework, was employed to check how the protocol behaves under various input conditions. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use invariant testing to ensure that several system invariants hold true in all situations.

Invariant testing enables the input of numerous random data points into the system, facilitating the identification of issues that regular testing may overlook. **25** invariants were tested with significant runs. This thorough testing identified some broken invariants.

Invariant	Test Case	Description	Test Result
Total Reserve Non-Negative	`invariant_CR01_totalReserveAmountsNonNegative()`	Total reserve amounts remain non-negative	Passed
Lending Index Monotonicity	`invariant_CR02_lendingIndexMonotonicity()`	Lending index always >= RAY, never decreases	Passed
Borrowing Index Monotonicity	`invariant_CR03_borrowingIndexMonotonicity()`	Borrowing index always >= RAY, never decreases	Passed
User Deposits Non-Negative	`invariant_AC04_userDepositsNonNegative()`	User deposit balances must always be ≥ 0 to prevent negative balances	Passed
User Debts Non-Negative	`invariant_AC05_userDebtsNonNegative()`	User debt balances must always be ≥ 0 to prevent negative debt	Passed
Total Debt Within Limit	`invariant_PR06_totalDebtWithinBorrowingLimit()`	Total protocol debt must not exceed configured borrowing limit	Passed
Reserve Accounting Consistency	`invariant_AC07_reserveAccountingConsistency()`	Available reserves must be ≤ total deposits for accounting integrity	Passed
LTV Parameter Validation	`invariant_PM08_ltvWithinValidRange()`	Loan-to-Value ratio must be ≤ 100% as per protocol specification	Passed
Liquidation Threshold Validation	`invariant_PM09_liquidationThresholdWithinValidRange()`	Liquidation threshold must be ≤ 100% for valid liquidation logic	Passed
Reserve Factor Validation	`invariant_PM10_reserveFactorWithinValidRange()`	Reserve factor validation broken (accepts 203% > 100%)	**Failed**
Utilization Rate Bounds	`invariant_RT11_utilizationRateWithinBounds()`	Utilization rate must be ≤ 100% for mathematical consistency	Passed
Interest Rate Relationship	`invariant_RT12_lendingRateLowerThanBorrowingRate`	Lending rate must be ≤ borrowing rate for economic viability	Passed

Invariant	Test Case	Description	Test Result
	`()`		
Interest Rate Reasonableness	`invariant_RT13_interestRatesWithinReasonableBounds()`	Interest rates must be ≤ 1000% APY to prevent unreasonable rates	Passed
Index Monotonic Growth	`invariant_MT14_indicesMonotonicallyIncreasing()`	Both indices must only increase over time for mathematical consistency	Passed
Scaling Operation Consistency	`invariant_MT15_scalingOperationsConsistent()`	Scale down → scale up operations must be consistent within rounding tolerance	Passed
Reserve Enabled Status	`invariant_CF16_reserveAlwaysEnabled()`	Active reserves must always remain enabled for protocol operations	Passed
Valid Interest Rate Strategy	`invariant_CF17_reserveHasValidInterestRateStrategy()`	Reserves must have valid (non-zero) interest rate strategy addresses	Passed
Strategy Parameter Immutability	`invariant_CF18_strategyParametersRemainConstant()`	Interest rate strategy parameters must remain constant post-deployment	Passed
Deposit Traceability	`invariant_AC19_totalDepositsTraceableViaRawDeposits()`	Total deposits must equal raw deposits × lending index for audit trail	Passed
Zero Utilization Base Rate	`invariant_BD20_zeroUtilizationBaseRateValidation()`	Zero utilization returns 0% instead of 5% base rate	**Failed**
LTV-Liquidation Relationship	`invariant_HF21_ltvLessThanLiquidationThreshold()`	LTV vs liquidation threshold logic broken (21% > 7%)	**Failed**
Health Factor Consistency	`invariant_CL22_healthFactorConsistency()`	Users with debt must maintain health factor > 1 for liquidation safety	Passed
Protocol Solvency	`invariant_EC23_protocolSolvencyMaintained()`	Protocol must remain solvent: available liquidity + debts ≥ deposit obligations	Passed
Index Precision Maintenance	`invariant_PR24_indexPrecisionMaintained()`	Indices must not exceed 10x RAY to prevent precision degradation	Passed
Liquidation Threshold Logic	`invariant_LQ25_liquidationThresholdExceedsLTV()`	Same as HF21, liquidation threshold logic broken (1% < 10%)	**Failed**

All detected findings were **addressed** in the report.

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.