
# SMART CONTRACTS REVIEW

zokyo

February 18th 2025 | v. 1.0

# Security Audit Score

## PASS

Zokyo Security has concluded that this smart contract passed a security audit.

# # ZOKYO AUDIT SCORING HYPERYIELD

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

# SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 1 Medium issue: 1 unresolved = - 10 points deducted
- 1 Low issue: 1 unresolved = - 5 points deducted
- 1 Informational issue: 1 unresolved = 0 points deducted

Thus, 100 - 10 - 5 = 85

# TECHNICAL SUMMARY

This document outlines the overall security of the HyperYield smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the HyperYield smart contract/s codebase for quality, security, and correctness.

## Contract Status


There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the HyperEVM network's fast-paced and rapidly changing environment, we recommend that the HyperYield team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

# Table of Contents

# AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the HyperYield repository:
Repo: https://github.com/hyperyieldevm/core-contracts/blob/master/contracts/deployments/ReservesSetupHelper.sol

Last commit - 6d6fa53d360b43f492ff5b3c7033f95aee4f1335

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ReservesSetupHelper.sol

**During the audit, Zokyo Security ensured that the contract:**

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of HyperYield smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

| | |
|---|---|
| 01 | Due diligence in assessing the overall code quality of the codebase. |
| 02 | Cross-comparison with other, similar smart contract/s by industry leaders. |
| 03 | Testing contract/s logic against common and uncommon attack vectors. |
| 04 | Thorough manual review of the codebase line by line. |

# Executive Summary

The `ReservesSetupHelper` contract is a deployment utility for configuring asset risk parameters in the Aave protocol's `PoolConfigurator` contract. It inherits from OpenZeppelin's `Ownable`, restricting function execution to the contract owner. The contract provides a batch configuration function, `configureReserves`, which allows the owner to set various risk parameters for multiple assets, including loan-to-value (LTV) ratios, liquidation thresholds, borrowing and flash loan permissions, and reserve factors.

# STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the HyperYield team and the HyperYield team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

- **Critical**

  The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

- **High**

  The issue affects the ability of the contract to compile or operate in a significant way.

- **Medium**

  The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

- **Low**

  The issue has minimal impact on the contract’s ability to operate.

- **Informational**

  The issue has no impact on the contract’s ability to operate.

# COMPLETE ANALYSIS

## FINDINGS SUMMARY

| # | Title | Risk | Status |
|---|---|---|---|
| 1 | Excessive Loop Iterations in configureReserves | Medium | Unresolved |
| 2 | Lack of Two-Step Ownership Transfer | Low | Unresolved |
| 3 | Floating Pragma | Informational | Unresolved |

MEDIUM-1 | UNRESOLVED

### Excessive Loop Iterations in `configureReserves`

The `configureReserves` function iterates over `inputParams` and performs multiple external contract calls within each iteration. If `inputParams.length` is large, this can result in excessive gas consumption, leading to transaction failures due to block gas limits.

```
function configureReserves(
  PoolConfigurator configurator,
  ConfigureReserveInput[] calldata inputParams
) external onlyOwner {
  for (uint256 i = 0; i < inputParams.length; i++) {
..
```

**Recommendation:**
Implement an upper bound on `inputParams.length` to limit the number of iterations.

LOW-1 | UNRESOLVED

### Lack of Two-Step Ownership Transfer

The ReservesSetupHelper.sol contract does not implement a two-step process for transferring ownership. In its current state, ownership can be transferred in a single step, which can be risky as it could lead to accidental or malicious transfers of ownership without proper verification.

**Recommendation:**
Implement a two-step process for ownership transfer where the new owner must explicitly accept the ownership. It is advisable to use OpenZeppelin's Ownable2Step.

INFORMATIONAL-1 | UNRESOLVED

**Floating Pragma**

The smart contract uses a floating pragma version `(^0.8.0)`. Contracts should be deployed using the same compiler version and settings as were used during development and testing. Locking the pragma version helps ensure that contracts are not inadvertently deployed with a different compiler version.

**Recommendation:**
Consider locking the pragma version to a specific, tested version to ensure consistent compilation and behavior of the smart contract.

| | ReservesSetupHelper.sol |
|---|---|
| Re-entrancy | Pass |
| Access Management Hierarchy | Pass |
| Arithmetic Over/Under Flows | Pass |
| Unexpected Ether | Pass |
| Delegatecall | Pass |
| Default Public Visibility | Pass |
| Hidden Malicious Code | Pass |
| Entropy Illusion (Lack of Randomness) | Pass |
| External Contract Referencing | Pass |
| Short Address/ Parameter Attack | Pass |
| Unchecked CALL Return Values | Pass |
| Race Conditions / Front Running | Pass |
| General Denial Of Service (DOS) | Pass |
| Uninitialized Storage Pointers | Pass |
| Floating Points and Precision | Pass |
| Tx.Origin Authentication | Pass |
| Signatures Replay | Pass |
| Pool Asset Security (backdoors in the underlying ERC-20) | Pass |

# CODE COVERAGE AND TEST RESULTS FOR ALL FILES

## Tests written by Zokyo Security

As a part of our work assisting HyperYield in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the HyperYield contract requirements for details about issuance amounts and how the system handles these.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "../contracts/deployments/
ReservesSetupHelper.sol";
import "../contracts/protocol/pool/
PoolConfigurator.sol";

contract PoolConfiguratorMock {
    event CollateralConfigured(address indexed asset,
uint256 ltv, uint256 lt, uint256 lb);
    event BorrowingSet(address indexed asset, bool
enabled);

    // Mock implementations
    function configureReserveAsCollateral(
        address asset,
        uint256 ltv,
        uint256 liquidationThreshold,
        uint256 liquidationBonus
    ) external {
        emit CollateralConfigured(asset, ltv,
liquidationThreshold, liquidationBonus);
    }

    function setReserveBorrowing(address asset, bool
enabled) external {
        emit BorrowingSet(asset, enabled);
    }
```

```
    function setBorrowCap(address, uint256) external {}
    function setReserveStableRateBorrowing(address, bool)
external {}
    function setReserveFlashLoaning(address, bool) external {}
    function setSupplyCap(address, uint256) external {}
    function setReserveFactor(address, uint256) external {}
}

contract ReservesSetupHelperTest is Test {
    ReservesSetupHelper public helper;
    PoolConfiguratorMock public mock;
    address public admin = address(1);

    function setUp() public {
        mock = new PoolConfiguratorMock();
        helper = new ReservesSetupHelper();
        helper.transferOwnership(admin);
    }

    function test_ConfigureReserves() public {
        ReservesSetupHelper.ConfigureReserveInput[] memory
inputs;

        // Initialize in scoped block to reduce stack depth
        {
            ReservesSetupHelper.ConfigureReserveInput memory
input = ReservesSetupHelper.ConfigureReserveInput({
                asset: address(0xABC),
                baseLTV: 5000,
                liquidationThreshold: 6000,
                liquidationBonus: 10750,

                reserveFactor: 1000,
                borrowCap: 1_000_000e18,
                supplyCap: 2_000_000e18,
                stableBorrowingEnabled: true,
                borrowingEnabled: true,
                flashLoanEnabled: false
            });
```

```
            inputs = new
ReservesSetupHelper.ConfigureReserveInput[](1);
            inputs[0] = input;
        }

        vm.startPrank(admin);

        // Collateral configuration check
        vm.expectEmit(true, true, true, true);
        emit PoolConfiguratorMock.CollateralConfigured(
            address(0xABC),
            5000,
            6000,
            10750
        );

        // Borrowing enablement check
        vm.expectEmit(true, true, true, true);
        emit PoolConfiguratorMock.BorrowingSet(address(0xABC),
true);

        // Execute test
        helper.configureReserves(PoolConfigurator(address(mock)),
inputs);
        vm.stopPrank();
    }
}
```

We are grateful for the opportunity to work with the HyperYield team.

**The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.**

Zokyo Security recommends the HyperYield team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo