hexens x Thesauros

Security Review Report for Thesauros

October 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review of Thesauros, a DeFi protocol for automated interest rebalancing. It supports ERC4626 and uses different liquidity providers to maximize yield.

Our security assessment was a full review of the code, spanning a total of 4 days.

During our review, we identified 2 medium severity vulnerabilities, which could have lead to temporary disruption of the protocol.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

▪ Review Led by

Jahyun Koo, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/Thesauros/contracts/tree/8ad87397735cf4148e41b4544aa9c7a99ab54911

The issues described in this report were fixed in the following commit:

🔗 https://github.com/Thesauros/contracts/commit/460c30b788e3d385e0c0e85158f963f2348d15c8

▪ Changelog

20 October 2025	Audit start
24 October 2025	Initial report
28 October 2025	Revision received
30 October 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

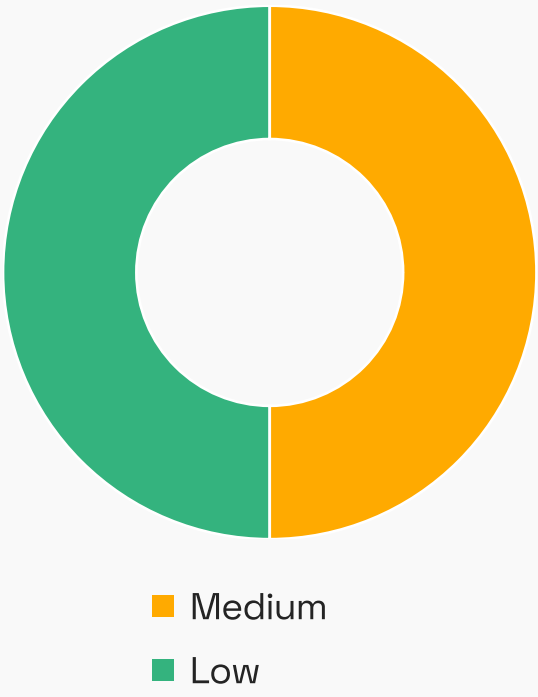
- **Issue Symbolic Codes**

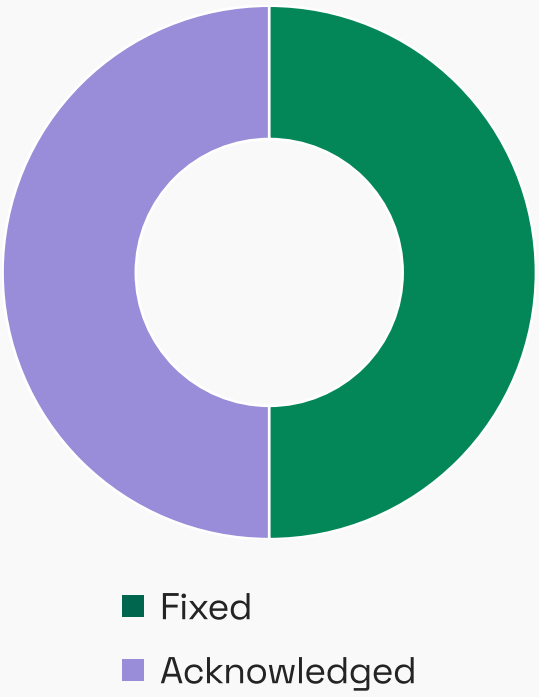
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	2
Low	1
Informational	3
Total:	**6**





6. Weaknesses

This section contains the list of discovered weaknesses.

THES1-4 | Active Provider Liquidity Mismatch Can Cause Withdrawal Failures

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/VaultManager.sol
contracts/base/Vault.sol

Description:

The **Vault** contract calculates **totalAssets** by aggregating the balances from all providers listed in the **_providers** array. However, the **_withdraw** function exclusively sources funds from the single designated **activeProvider**. This mismatch creates a scenario where withdrawals can fail despite the vault holding sufficient aggregate liquidity.

When the **rebalance()** function moves assets between providers without updating the **activeProvider** (when **activateToProvider** parameter is **false**), the active provider may retain insufficient funds to satisfy withdrawal requests. Users attempting to withdraw amounts that exceed the active provider's balance will experience transaction failures, even though **maxWithdraw()** indicates sufficient funds are available based on the total assets calculation.

Scenario:

1. The vault has two registered providers, **ProviderA** and **ProviderB**. **ProviderA** is the **activeProvider**.
2. The vault holds 1,000 USDC, all of which is deposited in **ProviderA**.
3. An operator executes a **rebalance**, moving 950 USDC from **ProviderA** to **ProviderB**, with the **activateToProvider** flag set to **false**.
4. The vault's **totalAssets()** function still correctly reports a balance of 1,000 USDC. However, the **activeProvider** (**ProviderA**) now holds only 50 USDC.
5. A user attempts to withdraw 100 USDC. The transaction fails because the **_withdraw** function attempts to pull funds from **ProviderA**, which has insufficient liquidity.

```
function rebalanceVault(
    IVault vault,
    uint256 assets,
    IProvider from,
    IProvider to,
    uint256 fee,
    bool activateToProvider
) external onlyExecutor returns (bool success) {
    uint256 assetsAtFrom = from.getDepositBalance(address(vault), vault);

    if (assets == type(uint256).max) {
        assets = assetsAtFrom;
    }
    if (assets == 0 || assets > assetsAtFrom) {
        revert VaultManager__InvalidAssetAmount();
    }

    vault.rebalance(assets, from, to, fee, activateToProvider);

    success = true;
}
```

```
function _getBalanceAtProviders()
    internal
    view
    returns (uint256 totalBalance)
{
    uint256 providerBalance;
    uint256 count = _providers.length;
    for (uint256 i; i < count; i++) {
        providerBalance = _providers[i].getDepositBalance(
            address(this),
            this
        );
        totalBalance += providerBalance;
    }
}
```

Remediation:

Align accounting and execution so that amounts ≤ maxWithdraw() never revert:

- Option A: Implement multi-provider withdrawal in the vault. Attempt from activeProvider first; if insufficient, withdraw the shortfall from other providers sequentially until assets is met. Apply the same strategy for redeem.
- Option B: Make limits liquidity-aware. Adjust maxWithdraw() (and maxRedeem() via convertToShares) to cap by the vault's immediately withdrawable liquidity under the current execution strategy.

THES1-7 | Deposit and withdrawal should be nonReentrant

Fixed

Severity:	Probability:	Impact:
Medium	Rare	Critical

Path:

Vault.sol:_deposit, _withdraw

Description:

In the Vault contract, the internal **_deposit** function uses **_delegateActionToProvider** to deposit the new assets to the provider protocol and the internal **_withdraw** function uses **_delegateActionToProvider** to withdraw the assets from the provider protocol.

The function **_delegateActionToProvider** is a delegate call to a provider that abstracts away the actual call to a supported protocol (Aave, Morpho, etc.). The external call is done before shares are minted in **deposit** and after shares are burned in **withdraw**: this means that there is an intermediate state where the share rate would be inflated from what it actually is.

This can be exploited to directly steal principal assets if one of the callbacks is reentrant. For example:

1. The vault has 200 WETH and 200 shares, 100 shares belong to user A.
2. User A withdraws 50 shares, calculated to 50 WETH.
3. The burn happens first, then the callback, so the vault temporarily has 200 WETH and 150 shares.
4. User A again withdraws another 50 shares inside of the callback, calculated to 66.67 WETH.
5. In the end, user A withdrew their 100 shares to 116.67 WETH.

This obviously depends on the protocol that is being integrated with on whether there is the possibility of a callback, but this is an unknown. If for example an integrated project uses a dynamic swap to deposit/withdraw, that could lead into a pool with an attacker-controlled hook, it could lead to a callback.

There is therefore a heavy risk on the possibility of a callback and a critical dependency on one of the integrated protocols (and the protocol that they further depend on).

```
function _deposit(
    address caller,
    address receiver,
    uint256 assets,
    uint256 shares
) internal {
    _asset.safeTransferFrom(caller, address(this), assets);
    _delegateActionToProvider(assets, "deposit", activeProvider);
    _mint(receiver, shares);

    emit Deposit(caller, receiver, assets, shares);
}

function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal {
    uint256 withdrawFee = assets.mulDiv(
        withdrawFeePercent,
        PRECISION_FACTOR
    );
    uint256 assetsToReceiver = assets - withdrawFee;

    _burn(owner, shares);
    _delegateActionToProvider(assets, "withdraw", activeProvider);

    address _treasury = treasury;

    _asset.safeTransfer(_treasury, withdrawFee);
    _asset.safeTransfer(receiver, assetsToReceiver);

    emit FeeCharged(_treasury, assets, withdrawFee);
    emit Withdraw(caller, receiver, owner, assetsToReceiver, shares);
}
```

Remediation:

We recommend not to create risk and add a reentrancy guard to **_deposit** and **_withdraw**.

Any other contract that relies on the Vault's share rate (e.g. functions such as **totalAssets**, **convertToShares**, **convertToAssets**, etc.) could still be vulnerable to read-only reentrancy in the case of a 3rd party callback, but at least the direct risk to the Vault should be mitigated.

THES1-1 | validatedShares Rounds Down to Zero, Causing 0-Share Burn on Withdraw

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

Vault.sol:_deposit, _withdraw

Description:

```
uint256 _maxWithdraw = maxWithdraw(owner);
if (assets > _maxWithdraw) {
  validatedAssets = _maxWithdraw;
  validatedShares = validatedAssets.mulDiv(shares, assets);
} else {
  validatedAssets = assets;
  validatedShares = shares;
}
```

When **assets > _maxWithdraw**, the process of recalculating **validatedShares** uses rounding CEIL when computing shares, but the code above uses **FLOOR**. As a result, **validatedShares** may end up being smaller than expected.

Remediation:

Consider checking that **validatedShares** and **validatedAssets** are not zero after the max-withdraw calculation. You can also combine this with **ceiling rounding** (**Math.Rounding.Ceil**) in **mulDiv** to reduce the chance of rounding down to zero.

```
if (assets > _maxWithdraw) {
  validatedAssets = _maxWithdraw;
  validatedShares = _convertToShares(validatedAssets, Math.Rounding.Ceil);
```

Proof-of-concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import {IProvider} from "../../contracts/interfaces/IProvider.sol";
import {IVault} from "../../contracts/interfaces/IVault.sol";
import {Vault} from "../../contracts/base/Vault.sol";
import {MockingUtilities} from "../utils/MockingUtilities.sol";
import {console2} from "forge-std/console2.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

/**
 * @title Line 408 Rounding Inconsistency Proof
 * @notice Mathematical proof of CEIL vs FLOOR rounding difference
 *
 * @dev Core proof:
 * 1. shares = previewWithdraw(assets) uses CEIL rounding
 * 2. validatedShares = validatedAssets.mulDiv(shares, assets) uses FLOOR rounding
 * 3. Proves that CEIL != FLOOR in certain cases
 */
contract VulnerabilityRoundingProofTest is MockingUtilities {

    address public user = makeAddr("user");

    function setUp() public {
        initializeVault(vault, MIN_AMOUNT, initializer);
    }

    /**
     * @notice Proof of Line 408's logical inconsistency
     */
    function test_ProofLine408LogicalInconsistency() public view {
        console2.log("\n=== PROOF: Line 408 Logical Inconsistency ===\n");

        // Simulation values
        uint256 totalSupply = 1000 ether;
        uint256 totalAssets = 1500 ether; // 1.5x ratio
        uint256 maxWithdraw = 100 ether;
        uint256 requestedAssets = 300 ether; // 3x request

        console2.log("Scenario:");
        console2.log("  Total supply:", totalSupply / 1 ether, "ether");
```

```
console2.log("  Total assets:", totalAssets / 1 ether, "ether");
console2.log("  Max withdraw:", maxWithdraw / 1 ether, "ether");
console2.log("  Requested:   ", requestedAssets / 1 ether, "ether");
console2.log("");

// === CORRECT WAY (previewWithdraw) ===
console2.log("CORRECT: previewWithdraw(maxWithdraw) with CEIL");

// shares = CEIL(maxWithdraw * totalSupply / totalAssets)
uint256 correctShares = Math.mulDiv(
    maxWithdraw,
    totalSupply,
    totalAssets,
    Math.Rounding.Ceil
);
console2.log("  Formula: CEIL(100 * 1000 / 1500)");
console2.log("  Result:", correctShares);
console2.log("");

// === LINE 408 WAY (mulDiv default = FLOOR) ===
console2.log("LINE 408: validatedAssets.mulDiv(shares, assets)");

// Step 1: shares = CEIL(requestedAssets * totalSupply / totalAssets)
uint256 sharesFromPreview = Math.mulDiv(
    requestedAssets,
    totalSupply,
    totalAssets,
    Math.Rounding.Ceil
);
console2.log("  Step 1 - shares from preview (CEIL):", sharesFromPreview);

// Step 2: validatedShares = FLOOR(maxWithdraw * shares / requestedAssets)
uint256 line408Shares = Math.mulDiv(
    maxWithdraw,
    sharesFromPreview,
    requestedAssets,
    Math.Rounding.Floor // Default value
);
console2.log("  Step 2 - recalc with FLOOR:", line408Shares);
console2.log("");

// === COMPARISON ===
console2.log("COMPARISON:");
```

```
        console2.log("  Correct (CEIL):  ", correctShares);
        console2.log("  Line 408 (FLOOR):", line408Shares);

        if (correctShares != line408Shares) {
            console2.log("  DIFFERENCE:      ", correctShares - line408Shares, "wei");
            console2.log("");
            console2.log("PROVEN: Line 408 uses different rounding!");
        }
        console2.log("");

        // Verification
        assertTrue(correctShares >= line408Shares, "Correct shares should be >= Line 408 shares");
    }
}
```

THES1-2 | There is no function to claim Extra Incentives

Acknowledged

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/providers/AaveV3Provider.sol

Description:

Aave offers incentives - such as staking or liquidity mining rewards (see: Incentives) - to users who supply assets to the protocol. These rewards are typically distributed as additional tokens (e.g., AAVE or other governance tokens) and can be claimed by users through Aave's incentive mechanisms.

However, the current **AaveV3Provider** contract lacks functionality to claim these incentives. This omission prevents users from fully benefiting from their supplied assets on Aave.

The similar issue applies to Compound and Morpho, Dolomite either:

- Compound III Docs | Protocol Rewards
- Rewards on Morpho
- Token Mechanics | Dolomite

Remediation:

To resolve this, we should introduce a function that enables users to claim their Aave incentives. This would require integrating with Aave's **Incentives Controller** or **Rewards Distributor** contracts.

Commentary from the client:

"Our protocol operates as an automated yield optimizer, and we have designed the system to capture only core lending yields."

THES1-5 | Inconsistent Accrual Projection Across Providers (Projected vs Last-Updated)

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

contracts/providers/MorphoProvider.sol

Description:

The provider implementations use different accrual models when exposing yield/balance views:

- MorphoProvider computes rates using projected accrual and projected weights (via expectedMarketBalances/expectedSupplyAssets) and applies a wrapper fee.
- AaveV3Provider, CompoundV3Provider, and DolomiteProvider return values based on last-updated protocol state/parameters without projection, which can be stale until a state-changing transaction occurs.

As a result, cross-provider outputs are not directly comparable. This can skew off-chain decisions (e.g., rebalancing) and, in mixed views, may introduce minor, transient drift when aggregating across providers.

Example scenario:

- A rebalancing routine compares providers using getDepositRate. Morpho's rate reflects projected accrual, while another provider returns a stale rate from hours earlier. The routine chooses a suboptimal destination and misses higher yield until the next update.

```
function _getMarketRate(
    MarketParams memory marketParams,
    Market memory market
) internal view returns (uint256 marketRate) {
    IMorpho morpho = _getMorpho();

    uint256 borrowRate;
    if (marketParams.irm == address(0)) {
        return 0;
    } else {
        borrowRate = IIrm(marketParams.irm)
            .borrowRateView(marketParams, market)
            .wTaylorCompounded(365 days);
    }
```

```
        (uint256 totalSupplyAssets, , uint256 totalBorrowAssets, ) = morpho
            .expectedMarketBalances(marketParams);

        uint256 utilization = totalBorrowAssets == 0
            ? 0
            : totalBorrowAssets.wDivUp(totalSupplyAssets);

        marketRate = borrowRate.wMulDown(1e18 - market.fee).wMulDown(
            utilization
        );
    }
```

Remediation:

Standardize the accrual model across providers and implement it consistently in code:

- Either project accrual at read time for all providers before returning rates/balances, or
- Return only last-updated values everywhere without projection.

Commentary from the client:

"The different accrual models across providers reflect intentional design choices optimized for each protocol's specific mechanics."

THES1-8 | Consider Adding Slippage Protection to Core ERC4626 Functions

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

contracts/base/Vault.sol

Description:

The vault's core ERC4626 functions (**deposit()**, **mint()**, **withdraw()**, and **redeem()**) do not include slippage protection parameters, which could expose users to receiving unfavorable conversion rates in certain scenarios.

- What is protected: The vault reverts when a deposit would mint zero shares, preventing “0 shares” outcomes and principal loss on-chain. Deposits are also paused until setup completes and enforce a minimum asset amount (**minAmount**)
- What is not protected: The vault does not bound user-expected outcomes for ERC4626 actions. All conversions rely on **totalAssets()** which aggregates provider balances. The vault's **totalAssets()** calculation relies on **_getBalanceAtProviders()**, which queries provider token balances (e.g., aTokens). This design could potentially be susceptible to manipulation through direct token transfers or through front-running of large transactions.

```
function totalAssets() public view override returns (uint256 assets) {
    return _getBalanceAtProviders();
}
```

```
function _getBalanceAtProviders()
    internal
    view
    returns (uint256 totalBalance)
{
    uint256 providerBalance;
    uint256 count = _providers.length;
    for (uint256 i; i < count; i++) {
        providerBalance = _providers[i].getDepositBalance(
            address(this),
            this
        );
        totalBalance += providerBalance;
```

```
    }
}
```

Remediation:

Consider adding optional slippage protection parameters to all four core ERC4626 functions. For **deposit()** and **redeem()**, add a minimum output parameter to ensure users receive at least the expected amount. For **mint()** and **withdraw()**, add a maximum input parameter to cap the amount users must provide. The functions should revert if the actual execution falls outside these user-defined bounds, providing defense-in-depth against both malicious front-running and unexpected state changes.

Commentary from the client:

"Our protocol only accepts deposits from whitelisted addresses, which significantly reduces the attack surface for front-running and manipulation scenarios."

hexens x Thesauros