Strata Tranches Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Stalin

Arno

InAllHonesty

October 8, 2025

Contents

1 About Cyfrin **3**

2 Disclaimer **3**

3 Risk Classification **3**

4 Protocol Summary **3**
4.1 Protocol summary . . . 3
4.2 Key Components . . . 3

5 Audit Scope **4**

6 Executive Summary **4**

7 Findings **7**
7.1 Critical Risk . . . 7
7.1.1 Withdrawers of `sUSDe` always incur a loss because parameters passed from `Tranche::_withdraw` to `CDO::withdraw` are inverted . . . 7
7.2 High Risk . . . 10
7.2.1 Users can get their withdrawal active requests DoSed by malicious users . . . 10
7.2.2 Mechanism to prevent donation attack can be gamed to cause withdrawals to revert causing assets to get stuck on the Strategy . . . 11
7.3 Medium Risk . . . 14
7.3.1 `Tranche::redeem` calls `super.withdraw` instead of `super.redeem` causing users to receive fewer assets . . . 14
7.3.2 Reducing reserves requesting `USDe` as the asset to receive causes the Strategy to release more `sUSDe` than necessary . . . 15
7.3.3 NAVs or APRs are updated using an outdated accounting on certain operations . . . 16
7.3.4 Proxy reuse without implementation check inside `UnstakeCooldown` leads to execution on outdated/vulnerable logic . . . 17
7.3.5 When Senior's TargetGain is negative, the tx will revert because the senior loss is not accounted for on the Junior Tranche as profit, causing the navs summation to not match the current nav . . . 18
7.3.6 `Tranche::maxMint` for Junior Tranches is at risk of overflow when the `jrNav` falls below `1:1` rate to `JR_Shares` . . . 19
7.4 Low Risk . . . 21
7.4.1 `Accounting::setMinimumJrtSrtRatio` sets `reserveBps` instead of `minimumJrtSrtRatio` making ratio configuration impossible . . . 21
7.4.2 Inconsistent APR boundary validation between `AprPairFeed` and `Accounting` . . . 21
7.4.3 Inconsistent Risk Premium Validation in `Accounting` Allows Future Underflows or Zero APR . . . 22
7.4.4 Frontrunning to Block Junior Tranche Withdrawals . . . 22
7.4.5 `DEFAULT_ADMIN_ROLE` can be mistakenly granted to an account when granting permission to call `fallback` on any contract . . . 24
7.5 Informational . . . 26
7.5.1 Incorrect Comment and Missing Lower Bound for `minimumJrtSrtRatio` in `Accounting` . . . 26
7.5.2 Unused Import of `OwnableUpgradeable` in `Accounting.sol` . . . 26
7.5.3 Missing Validation of Fallback APR Values in `AprPairFeed::latestRoundData` . . . 26
7.5.4 Missing `Unstaked` event for immediate unstake in `UnstakeCooldown::transfer` . . . 27
7.5.5 Use explicit unsigned integer sizing instead of `uint` . . . 27
7.5.6 Unreachable code inside `sUSDeStrategy::reduceReserve` . . . 27
7.5.7 Misleading variable name to set the `asset` for the `Tranche` . . . 28
7.5.8 Typos and Bad NATSPEC . . . 28
7.5.9 `AprPairFeed::getRoundData` can return data for a different round than the specified . . . 28
7.5.10 Cooldown contracts underreport the real balance of users because they only consider the balance of requests whose cooldown period is over . . . 29

7.5.11 `UnstakeCooldown::balance` requires a different token contract than the actual token that is reporting the balance for . . . 30
7.5.12 No helper functions for `maxDeposit`, `maxMint`, `maxRedeem`, `maxWithdraw` for `sUSDe` . . . 31
7.6 Gas Optimization . . . 32
7.6.1 Remove redundant checks . . . 32
7.6.2 Cache result of external calls when result can't change between calls and is used multiple times . . . 32

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.

- Senior Tranche offers a steady APR and additional coverage to deposits provided by the Junior Tranche.
- Junior Tranche offers the possibility to earn a higher yield in exchange for bearing the risk of offering coverage to the Senior's deposits.

The protocol allows deposits using USDe or sUSDe even though all the internal accounting is tracked in USDe units. All deposited assets from the two tranches is pooled into a single Strategy contract.

The Yield Distribution is as follows:

- Senior Tranche receives a stable yield floor benchmarked to Sky Savings Rate while retaining potential upside of sUSDe APY
- Junior Tranche earns all the excess yield after covering the Senior's Tranche target yield

4.2 Key Components

- **Tranches:** Tranches are Meta Vaults based on the ERC4626 standard. Allows users to deposit USDe and sUSDe into the system. Tranches are the entry-point for users into the system. Tranches receives users deposits as well as user's withdrawal requests and forwards the execution to the rest of the system.
- **CDO:** The CDO contract is at the hearth of the protocol. It is in charge of orchestrating asset flows between Tranches and the Strategy as well as handling the interactions with the Accounting contract to track the internal accounting in USDe units of all the TVL and yield distribution.

- **Strategy:** The Strategy contract is responsible for asset management. It receives assets from the CDO, stakes them, and must be able to return them when requested. At any point in time, the Strategy reports its total TVL (incl. yield).
- **Accounting:** The Accounting contract performs pure calculations to track the Jr and Sr TVL, as well as to handle updates to the senior's target APR. Yield distribution is also calculated by the Accounting contract.

5 Audit Scope

The scope of this audit was limited to:

```
contracts/governance/AccessControlled.sol
contracts/governance/AccessControlManager.sol
contracts/governance/StrataMasterChef.sol
contracts/tranches/base/cooldown/ERC20Cooldown.sol
contracts/tranches/base/cooldown/UnstakeCooldown.sol
contracts/tranches/base/CDOComponent.sol
contracts/tranches/base/Tranche.sol
contracts/tranches/oracles/AprTupleFeed.sol
contracts/tranches/strategies/ethena/sUSDeCooldownRequestImpl.sol
contracts/tranches/strategies/ethena/sUSDeStrategy.sol
contracts/tranches/strategies/Strategy.sol
contracts/tranches/utils/UD60x18Extra.sol
contracts/tranches/StrataCDO.sol
contracts/tranches/YieldAccounting.sol
contracts/tranches/StrataCDOStorage.sol
```

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Strata Tranches smart contracts provided by Strata Money. In this period, a total of 28 issues were found.

During the audit we identified 1 Critical, 2 High, 6 Medium and 5 Low severity issues with the remainder being informational and gas optimizations.

The single critical finding is caused by passing parameters in the inverse order resulting in users receiving less assets than what they should receive for their burnt shares.

One of the high findings allowed attackers to spam the system with fake withdrawal requests to fill up the user's active withdrawal request queue till the point of causing an out of gas error when legitimate users attempted to finalize their withdrawals.

The other high finding allowed manipulation of the shares<=>assets rate causing deposits to mint small amounts of Tranche Shares, which resulted in all withdrawal requests reverting because an explicit check that enforced the total number of shares to not fall below a predefined boundary, which was never reached due to the manipulated rate and small number of shares minted for deposits.

The medium findings were a combination of re-using proxies using old implementations even though the implementation had been updated, calling an incorrect parent's function to process redemptions, an overflow when calculating maxMint for Juniors when the exchange rate fell below 1:1, the Strategy releasing more sUSDe than it should because a missing conversion to calculate the actual amount of sUSDe that should have been released, updating an incorrect variable causing accounting to not sum up the current nav leading to tx revert, and incorrectly using and updating the accounting with outdated data.

Summary

Project Name	Strata Tranches
Repository	contracts-tranches
Commit	af5195b04712...
Fix Commit	1f82c6a45627...
Audit Timeline	Sep 22nd - Oct 1st, 2025
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	2
Medium Risk	6
Low Risk	5
Informational	12
Gas Optimizations	2
Total Issues	28

Summary of Findings

[C-1] Withdrawers of `sUSDe` always incur a loss because parameters passed from `Tranche::_withdraw` to `CDO::withdraw` are inverted	Resolved
[H-1] Users can get their withdrawal active requests DoSed by malicious users	Resolved
[H-2] Mechanism to prevent donation attack can be gamed to cause withdrawals to revert causing assets to get stuck on the Strategy	Resolved
[M-1] `Tranche::redeem` calls `super.withdraw` instead of `super.redeem` causing users to receive fewer assets	Resolved
[M-2] Reducing reserves requesting `USDe` as the asset to receive causes the Strategy to release more `sUSDe` than necessary	Resolved
[M-3] NAVs or APRs are updated using an outdated accounting on certain operations	Resolved
[M-4] Proxy reuse without implementation check inside `UnstakeCooldown` leads to execution on outdated/vulnerable logic	Resolved
[M-5] When Senior's TargetGain is negative, the tx will revert because the senior loss is not accounted for on the Junior Tranche as profit, causing the navs summation to not match the current nav	Resolved
[M-6] `Tranche::maxMint` for Junior Tranches is at risk of overflow when the `jrNav` falls below `1:1` rate to `JR_Shares`	Resolved
[L-1] `Accounting::setMinimumJrtSrtRatio` sets `reserveBps` instead of `minimumJrtSrtRatio` making ratio configuration impossible	Resolved

[L-2] Inconsistent APR boundary validation between `AprPairFeed` and `Accounting`	Resolved
[L-3] Inconsistent Risk Premium Validation in `Accounting` Allows Future Underflows or Zero APR	Resolved
[L-4] Frontrunning to Block Junior Tranche Withdrawals	Resolved
[L-5] `DEFAULT_ADMIN_ROLE` can be mistakenly granted to an account when granting permission to call `fallback` on any contract	Resolved
[I-01] Incorrect Comment and Missing Lower Bound for `minimumJrtSrtRatio` in `Accounting`	Resolved
[I-02] Unused Import of `OwnableUpgradeable` in `Accounting.sol`	Resolved
[I-03] Missing Validation of Fallback APR Values in `AprPairFeed::latestRoundData`	Resolved
[I-04] Missing `Unstaked` event for immediate unstake in `UnstakeCooldown::transfer`	Resolved
[I-05] Use explicit unsigned integer sizing instead of `uint`	Resolved
[I-06] Unreachable code inside `sUSDeStrategy::reduceReserve`	Acknowledged
[I-07] Misleading variable name to set the `asset` for the `Tranche`	Resolved
[I-08] Typos and Bad NATSPEC	Resolved
[I-09] `AprPairFeed::getRoundData` can return data for a different round than the specified	Resolved
[I-10] Cooldown contracts underreport the real balance of users because they only consider the balance of requests whose cooldown period is over	Resolved
[I-11] `UnstakeCooldown::balance` requires a different token contract than the actual token that is reporting the balance for	Acknowledged
[I-12] No helper functions for `maxDeposit`, `maxMint`, `maxRedeem`, `maxWithdraw` for `sUSDe`	Acknowledged
[G-1] Remove redundant checks	Resolved
[G-2] Cache result of external calls when result can't change between calls and is used multiple times	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Withdrawers of `sUSDe` always incur a loss because parameters passed from `Tranche::_withdraw` to `CDO::withdraw` are inverted

Description: Users can choose to withdraw either `sUSDe` or `USDe`. The system is in charge of making the proper calculations to determine how much `USDe` value will be withdrawn based on the requested asset and the `tokenAmount` of such an asset. Based on the calculated `USDe` value being withdrawn, the system burns the required `TrancheShares` for that amount of `USDe` being withdrawn from the system.

This issue reports a problem in which the `Tranche::_withdraw` passes two parameters in the inverse order to the `CDO::withdraw`. These parameters are `baseAssets` and `tokenAssets`. The CDO expects to receive `tokenAssets` first and then `baseAssets`, but the Tranche passes `baseAssets` first and then `tokenAssets`.

- The parameters in the inverse order make the system do calculations with the wrong amounts, resulting (on the `Strategy` contract) that the amount of `sUSDe` to release to the user is way lower than what it should be (especially when the `sUSDe <=> USDe` rate is high.

```
// Tranche::_withdraw() //
    function _withdraw(
        address token,
        address caller,
        address receiver,
        address owner,
        uint256 baseAssets,
        uint256 tokenAssets,
        uint256 shares
    ) internal virtual {
        ...
//@audit => Burn Trancheshares for the full requested sUSDe
@>      _burn(owner, shares);

//@audit-issue => Sends baseAssets first and then tokenAssets
@>      cdo.withdraw(address(this), token, baseAssets, tokenAssets, receiver);
        ...
    }

// StrataCDO::withdraw() //
    function withdraw(address tranche, address token, uint256 tokenAmount, uint256 baseAssets, address
    ↪  receiver) external onlyTranche nonReentrant {
        ...
//@audit => Because of the inverted parameters `tokenAmount` is actually `baseAssets`, and `baseAssets`
↪  is actually `tokenAssets`
 @>     strategy.withdraw(tranche, token, tokenAmount, baseAssets, receiver);
        ...
    }

// Strategy::withdraw() //
    function withdraw (address tranche, address token, uint256 tokenAmount, uint256 baseAssets, address
    ↪  receiver) external onlyCDO returns (uint256) {
//@audit => `baseAssets` should represent amount of `USDe` being withdrawn, but, because of the inverted
↪  parameters, here represents the actual requested amount of `sUSDe` to withdraw
        uint256 shares = sUSDe.previewWithdraw(baseAssets);
        if (token == address(sUSDe)) {
            uint256 cooldownSeconds = cdo.isJrt (tranche) ? sUSDeCooldownJrt : sUSDeCooldownSrt;
//@audit => transfers calculates `shares` of `sUSDe` to Cooldown to be sent to the user after cooldown.
            erc20Cooldown.transfer(sUSDe, receiver, shares, cooldownSeconds);
            return shares;
        }
        ...
```

```
    }
```

Impact: Withdrawers will always incur a loss in USDe value because more TrancheShares are burned compared to the received value in USDe terms.

Proof of Concept: In the next PoC, it is demonstrated how a user withdrawing sUSDe incurs a loss because he receives fewer sSUDe than the requested amount and the TrancheShares that were burned during the process.

As demonstrated on the next PoC, given a `sUSDe => USDe` rate of 1:1.5. A user requests to withdraw 100 sUSDe, which are worth 150 USDe

- 150 JRTranche will be burnt
- ERC20Cooldown should receive the requested 100 sUSDe

The withdrawer receives only ~66 `sUSDe`, which can withdraw only 100 `USDe` instead of transferring 100 `sUSDe`, which could withdraw 150 `USDe`

Add the next PoC to `CDO.t.sol` test file:

```
    function test_WithdrawingsUSDECausesLosesForUsers() public {
        address bob = makeAddr("Bob");
        USDe.mint(bob, 1000 ether);
        vm.startPrank(bob);
            //@audit => Bob initializes the exchange rate on sUSDe
            USDe.approve(address(sUSDe), type(uint256).max);
            sUSDe.deposit(1000 ether, bob);
        vm.stopPrank();

        address alice = makeAddr("Alice");
        uint256 initialDeposit = 150 ether;
        USDe.mint(alice, initialDeposit);

        //@audit-info => There are 1k sUSDe in circulation and 1k USDe deposited on the sUSDe contract
        //@audit-info => Exchange Rate is 1:1
        assertEq(USDe.balanceOf(address(sUSDe)), 1000 ether);
        assertEq(sUSDe.totalSupply(), 1000 ether);
        assertEq(sUSDe.convertToAssets(1e18), 1e18);

        // Simulate yield by adding USDe directly to sUSDe contract
        //@audit-info => Set sUSDe exchange rate to USDe (1:1.5)
        USDe.mint(address(sUSDe), 500 ether); // 50% yield
        assertApproxEqAbs(sUSDe.convertToAssets(1e18), 1.5e18, 1e6);

        //@audit-info => Bob deposits sUSDe when sUSDE rate to USDe is 1:1.5
        vm.startPrank(bob);
            sUSDe.approve(address(jrtVault), type(uint256).max);
            jrtVault.deposit(address(sUSDe), 100e18, bob);
            assertApproxEqAbs(jrtVault.balanceOf(bob), 150e18, 1e6);
        vm.stopPrank();

        vm.startPrank(alice);
            //@audit => Alice gets 100 sUSDe by staking 150 USDe
            USDe.approve(address(sUSDe), type(uint256).max);
            sUSDe.deposit(150e18, alice);
            assertApproxEqAbs(sUSDe.balanceOf(alice), 100e18, 1e6);

            //@audit => Alice deposits 100 sUSDe on the JRTranche and gets 150 JRTrancheShares
            sUSDe.approve(address(jrtVault), type(uint256).max);
            jrtVault.deposit(address(sUSDe), 100e18, alice);
            assertApproxEqAbs(jrtVault.balanceOf(alice), 150e18, 1e6);
            assertEq(sUSDe.balanceOf(alice), 0);

            //@audit-info => Requests to withdraw 100e18 sUSDe which are worth 150 USDe
            uint256 expected_sUSDeWithdrawn = 100e18;
```

```
            uint256 expected_USDe_valueWithdrawn = sUSDe.convertToAssets(expected_sUSDeWithdrawn);

            //@audit-issue => Alice withdraws 100 sUSDe but gets only ~66.6 sUSDe, all her
            ↪  JRTrancheShares are burnt
            jrtVault.withdraw(address(sUSDe), expected_sUSDeWithdrawn, alice, alice);
            uint256 alice_actual_sUSDeBalance = sUSDe.balanceOf(alice);
            uint256 alice_USDe_actualWithdrawn = sUSDe.convertToAssets(alice_actual_sUSDeBalance);

            assertEq(jrtVault.balanceOf(alice), 0);
            assertApproxEqAbs(alice_actual_sUSDeBalance, 66.5e18, 1e18);

            console2.log("Alice expected withdrawn sUSDe: ", expected_sUSDeWithdrawn);
            console2.log("Alice actual withdrawn sUSDe: ", alice_actual_sUSDeBalance);
            console2.log("====");
            console2.log("Alice expected withdrawn USDe value: ", expected_USDe_valueWithdrawn);
            console2.log("Alice actual withdrawn USDe value: ", alice_USDe_actualWithdrawn);
        vm.stopPrank();
    }
```

Recommended Mitigation: On the `Tranche::_withdraw`, make sure to pass the parameters in the correct order when calling the `CDO::withdraw`

```
    function _withdraw(
        address token,
        address caller,
        address receiver,
        address owner,
        uint256 baseAssets,
        uint256 tokenAssets,
        uint256 shares
    ) internal virtual {
        ...
-       cdo.withdraw(address(this), token, baseAssets, tokenAssets, receiver);
+       cdo.withdraw(address(this), token, tokenAssets, baseAssets, receiver);

    }
```

Strata: Fixed in commit 31d9b72 by passing the parameters in the correct order.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Users can get their withdrawal active requests DoSed by malicious users

Description: Users can opt to withdraw either `USDe` or `sUSDe`.

- When withdrawing `sUSDe`, the assets can be put on a cooldown period depending on the Tranche from which the withdrawal is requested.
- When withdrawing `USDe`, regardless of the Tranche from which the withdrawal is requested, a new withdrawal request will be created on the `UnstakeCooldown` contract because to receive `USDe`, it is required to unstake `sUSDe` on the Ethena contract.

The problem this issue reports is a grief attack that allows malicious users to cause a DoS to other users' active withdrawal requests, effectively causing their assets to get stuck on the system.

The grief attack is achieved by withdrawing as low as 1 wei and setting the `receiver` as the victim user that the attacker wants to damage. Given that neither the `Tranche` nor the `CDO` nor the `Strategy` contracts validate if the withdrawer is authorized by the `receiver` to request withdrawals on their behalf, this allows anybody to make new withdrawal requests on behalf of anybody.

- Each new withdrawal request is pushed to an array on the `UnstakingContract`, which, once the cooldown period is over, the `UnstakingContract.finalize()` iterates over such an array to process all the ready requests. The attack inflates this array to a point that causes an `out of gas error` because of iterating over thousands of active requests (inflated by an attacker).

Since the unstaking contracts lack access control, an alternative approach is to directly call the `request()` method of the unstaking contracts. This allows circumventing the system's core contracts and directly inflating the user's withdrawal requests on the unstaking contracts.

```
    function transfer(IERC20 token, address to, uint256 amount) external {
@>      address from = msg.sender;
        ...
        SafeERC20.safeTransferFrom(token, from, address(proxy), amount);
        ...

@>      requests.push(TRequest(uint64(unlockAt), proxy));
        emit Requested(address(token), to, amount, unlockAt);
    }
```

Impact: Malicious users can cause a DoS for other users to complete the withdrawal of their assets when they are withdrawing USDe via the UnstakeCooldown contract.

Proof of Concept: Add the next PoC on the `CDO.t.sol` file. The PoC demonstrates how a malicious user can fully DoS the withdrawal of active requests for another user by requesting a huge amount of withdrawals for as low as one wei. As a result, a malicious user can fully DoS the active withdrawals for a small amount of resources, mostly covering the gas costs.

```
    function test_DoSUserActiveRequests() public {
        address alice = makeAddr("Alice");
        address bob = makeAddr("Bob");

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

        vm.startPrank(alice);
        USDe.approve(address(jrtVault), type(uint256).max);
        jrtVault.deposit(initialDeposit, alice);
        vm.stopPrank();

        vm.startPrank(bob);
        USDe.approve(address(jrtVault), type(uint256).max);
```

```
        jrtVault.deposit(initialDeposit, bob);
        vm.stopPrank();

        //@audit => Alice requests to withdraw its full USDe balance
        uint256 totalWithdrawableAssets = jrtVault.maxWithdraw(alice);
        vm.prank(alice);
        jrtVault.withdraw(totalWithdrawableAssets, alice, alice);

        //@audit => Bob does a huge amount of tiny withdrawals to inflate the activeRequests array of
        ↪  Alice
        vm.pauseGasMetering();
            for(uint i = 0; i < 35_000; i++) {
                vm.prank(bob);
                jrtVault.withdraw(1, alice, bob);
            }
        vm.resumeGasMetering();

        //@audit-info => Skip till a timestamp where the requests can be finalized
        skip(1_000_000);

        //@audit-issue => Alice gets DoS her tx to finalize the withdrawal of her USDe balance
        vm.prank(alice);
        unstakeCooldown.finalize(sUSDe, alice);
    }
```

Recommended Mitigation: A combination of a minimum withdrawal amount and a permission mechanism to allow users to specify who can request new withdrawals on their behalf.

In addition to the above mitigations, restrict who can call the `transfer()` on the `UnstakeCooldown` and `ERC20Cooldown` contracts. If anybody can call the `transfer()` function, the mitigations mentioned previously can be circumvented by directly calling the `transfer()` on any of the two Cooldown contracts.

Strata: Fixed in commits ea7371d, da327dc, and, 9b5ac62 by:

1. Adding access control to `UnstakeCooldown::transfer` and `ERC20Cooldown::transfer`.
2. Setting a soft limit for requests created by an account other than the receiver, and a hard limit for requests created by the actual receiver. Once the hard limit is reached, any subsequent request is added to the last request on the list.

Cyfrin: Verified. Implemented access control to prevent grief by calling functions directly. Implemented limits to prevent unauthorized users from spamming fake requests and filling up the withdrawer's requests queue.

7.2.2 Mechanism to prevent donation attack can be gamed to cause withdrawals to revert causing assets to get stuck on the Strategy

Description: Each time a withdrawal occurs, a check is performed to ensure that TrancheShares `totalSupply()` doesn't fall below a pre-defined boundary (`MIN_SHARES`). But, there is a way to game this mechanism such that it causes all withdrawals to revert because the `shares<=>assets ratio` is manipulated, which causes the Tranche to mint small amounts of shares, leading to the `totalSupply()` to not exceed the `MIN_SHARES` boundary, as a result, all withdrawals will revert.

```
    function _withdraw(
        ...
    ) internal override {
       ...
        _onAfterWithdrawalChecks();
        emit Withdraw(caller, receiver, owner, assets, shares);
    }

    function _onAfterWithdrawalChecks () internal view {
@>      if (totalSupply() < MIN_SHARES) {
```

```
            revert MinSharesViolation();
        }
    }
```

The attack is a first donation attack, in which an attacker donates `sUSDe` directly to the `Strategy`, this inflates the `totalAssets` in the system, and then the attacker makes a deposit for `1.1USDe`, which causes the share<=>assets rate to be manipulated, i.e. for a deposit of 1.1e18 USDe, the Tranche will mint 1 wei of shares. As a result, any subsequent deposit will mint shares at a manipulated rate, and the problem arises when withdrawals are attempted because of the `_onAfterWithdrawalChecks()`, the remaining `totalSupply()` won't exceed `MIN_SHARES`.

Impact: The withdrawal mechanism can be broken, causing user-deposited assets to get stuck on the `Strategy` contract.

Proof of Concept: Add the next PoC to the `CDO.t.sol` test file, and import the `IErrors` interface in the imports section. `import { IErrors } from "../contracts/tranches/interfaces/IErrors.sol";`

```
    function test_GameDonationAttackProtectionToTrapAssets() public {
        address alice = makeAddr("Alice");
        address bob = makeAddr("Bob");
        // Same value as in the Tranche contract
        uint256 MIN_SHARES = 0.1 ether;

        USDe.mint(bob, 1000 ether);
        vm.startPrank(bob);
            //@audit => Bob initializes the exchange rate on sUSDe
            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);

        vm.startPrank(alice);
            USDe.approve(address(sUSDe), type(uint256).max);
            sUSDe.deposit(1e18, alice);
            // Step 1 => Alice transfers 1 sUSDe directly to the strategy to inflate the exchange rate
            ↪  and deposits 1.1e18 USDe that results in minting 1 TrancheShare
            sUSDe.transfer(address(sUSDeStrategy), 1e18);
            USDe.approve(address(jrtVault), type(uint256).max);
            jrtVault.deposit(1.1e18, alice);
        vm.stopPrank();

        assertEq(jrtVault.totalSupply(), 1);

        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
            //Because of the manipulated exchange rate, the total minted TrancheShares for such a big
            ↪  deposits won't even be enough to reach the MIN_SHARES
            jrtVault.deposit(1_000_000e18, bob);
            assertLt(jrtVault.totalSupply(), MIN_SHARES);

            //Step 3 => Bob attempts to make a withdrawal, but the withdrawal reverts because the total
            ↪  shares on the Tranche don't reach 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:

1. Consider making a deposit of at least 1 full USDe on the same tx when the Tranche is deployed. Ideally, set the receiver of the deposit as `address(1)`, these shares should be considered unusable, so the minted TrancheShares are effectively used as the lower bound to not fall below the MIN_SHARES.
2. Take special care with the deployment script. To grant approvals to the strategy for pulling the deposited assets from the Tranche, you must call `Tranche::configure` (this function is required to call `StrataCDO::configure`, which requires the 2 Tranches and Strategy to have been deployed).

Strata: Fixed in commit f344885 by transferring any donations made to the strategy before the first deposit into the reserve.

Cyfrin: Verified. On the same transaction, when enabling deposits and making the first deposit, `cdo::reduceReserve` must be called before making the first deposit to sweep any donations to the strategy, as deposits were not enabled before this point.

7.3 Medium Risk

7.3.1 `Tranche::redeem` calls `super.withdraw` instead of `super.redeem` causing users to receive fewer assets

Description: The `Tranche::redeem` function contains a bug where it calls `super.withdraw(shares, receiver, owner)` instead of `super.redeem(shares, receiver, owner)`. This causes the function to incorrectly treat the shares parameter as an assets parameter, leading to wrong calculations.

```
    function redeem(uint256 shares, address receiver, address owner) public override returns (uint256) {
        cdo.updateAccounting();
        uint256 assets = super.withdraw(shares, receiver, owner);
        return assets;
    }
```

According to EIP4626 standard:

- redeem(uint256 shares, ...) should burn exactly shares amount and return corresponding assets
- withdraw(uint256 assets, ...) should withdraw exactly assets amount and return shares burned

The bug causes redeem(100 shares) to internally call withdraw(100 assets), treating 100 as assets instead of shares.

Impact: Users receive significantly fewer assets than expected. Users can never redeem their exact intended share amount. ERC4626 Standard Violation: The function violates the ERC4626 specification by:

1. Not burning the exact number of shares specified
2. Returning wrong values (shares burned instead of assets transferred)
3. Breaking the fundamental redeem/withdraw distinction

Proof of Concept:

```
function test_RedeemBug() public {

    alice = makeAddr("alice");
    USDe.mint(alice, 10000 ether);

    vm.startPrank(alice);
    USDe.approve(address(jrtVault), type(uint256).max);


    uint256 initialDeposit = 1000 ether;
    uint256 sharesReceived = jrtVault.deposit(initialDeposit, alice);

    vm.stopPrank();

    // Simulate yield by adding USDe directly to sUSDe contract
    // This increases the asset-to-share ratio in sUSDe, creating yield
    USDe.mint(address(sUSDe), 200 ether); // 20% yield

    vm.startPrank(alice);

    console2.log("After yield simulation");
    console2.log("------------------------");
    console2.log("Alice shares balance:", jrtVault.balanceOf(alice));
    console2.log("Vault totalAssets:", jrtVault.totalAssets());
    console2.log("Alice convertToAssets:", jrtVault.convertToAssets(jrtVault.balanceOf(alice)));

    // Calculate exchange rate
    uint256 totalSupplyShares = jrtVault.totalSupply();
    uint256 totalAssetsVault = jrtVault.totalAssets();
    console2.log("Exchange rate (assets per share * 1e18):", totalSupplyShares > 0 ? totalAssetsVault *
    ↪  1e18 / totalSupplyShares : 1e18);
```

```
    uint256 sharesToRedeem = 100 ether;
    uint256 expectedAssets = jrtVault.previewRedeem(sharesToRedeem);
    uint256 sharesBeforeRedeem = jrtVault.balanceOf(alice);

    console2.log("Before redeem");
    console2.log("------------------------");
    console2.log("Shares to redeem:", sharesToRedeem);
    console2.log("Expected assets from previewRedeem:", expectedAssets);
    console2.log("Alice shares before:", sharesBeforeRedeem);

    // Call redeem - this calls super.withdraw instead of super.redeem
    uint256 actualAssetsReturned = jrtVault.redeem(sharesToRedeem, alice, alice);
    uint256 sharesAfterRedeem = jrtVault.balanceOf(alice);
    uint256 sharesBurned = sharesBeforeRedeem - sharesAfterRedeem;

    console2.log("After redeem");
    console2.log("------------------------");
    console2.log("Actual assets returned:", actualAssetsReturned);
    console2.log("Alice shares after:", sharesAfterRedeem);
    console2.log("Shares actually burned:", sharesBurned);

    // For comparison: what would happen if we called withdraw with same numerical value
    uint256 assetsToWithdraw = sharesToRedeem; // Same number as shares
    uint256 expectedSharesForWithdraw = jrtVault.previewWithdraw(assetsToWithdraw);

    console2.log("Comparison: withdraw behavior");
    console2.log("------------------------");
    console2.log("If we called withdraw(", assetsToWithdraw, "):");
    console2.log("Expected shares burned by withdraw:", expectedSharesForWithdraw);
    console2.log("Expected assets returned by withdraw:", assetsToWithdraw);

    console2.log("------------------------");
    console2.log("Expected assets from proper redeem:", expectedAssets);
    console2.log("Actual assets returned:", actualAssetsReturned);
    console2.log("Difference:", expectedAssets > actualAssetsReturned ? expectedAssets -
    ↪  actualAssetsReturned : 0);

    assertTrue(expectedAssets != actualAssetsReturned, "redeem returned wrong asset amount");

    vm.stopPrank();
}
```

Recommended Mitigation: Change the function call from `super.withdraw` to `super.redeem`:

```
function redeem(uint256 shares, address receiver, address owner) public override returns (uint256) {
    cdo.updateAccounting();
-   uint256 assets = super.withdraw(shares, receiver, owner);
+   uint256 assets = super.redeem(shares, receiver, owner);
    return assets;
}
```

Strata: Fixed in commit 31d9b72 by calling the correct function.

Cyfrin: Verified.

7.3.2 Reducing reserves requesting `USDe` as the asset to receive causes the Strategy to release more `sUSDe` than necessary

Description: When reducing reserves and asking for `USDe`, the Strategy incorrectly transfers sUSDe for the actual amount of asked `USDe` (`tokenAmount`) instead of only transferring the required `sUSDe` to cover the requested `tokenAmount` of `USDe`.

```
//sUSDeStrategy::reduceReserve()//
    function reduceReserve (address token, uint256 tokenAmount, address receiver) external onlyCDO {
        ...
        if (token == address(USDe)) {
            //@audit-issue => transfer sUSDe for `tokenAmount` which is in USDe.
            unstakeCooldown.transfer(sUSDe, receiver, tokenAmount);
            return;
        }
        revert UnsupportedToken(token);
    }
```

Impact: The strategy releases more sUSDe than necessary to cover the USDe requested to be withdrawn when reducing reserves and asking USDe. This means that depositors end up incurring a loss because the strategy is left with less USDe than it should have.

Proof of Concept: For example, if the `sUSDe <=> USDe` rate is 1:1.5, and it is requested to reduce reserves for 150 USDe.

- The Strategy will send 150 `sUSDe` (which are worth `225 USDe`) to the `treasury` **instead of only sending** `100 sUSDe` (worth `150 USDe`).

Recommended Mitigation: When reducing reserves asking for USDe, on `sUSDeStrategy::reduceReserve`, call `sUSDe::previewWithdraw` to get how much sUSDe is required to obtain the requested `tokenAmount` of `USDe`, and transfer that amount of `sUSDe` to the `UnstakeCooldown` contract.

```
    function reduceReserve (address token, uint256 tokenAmount, address receiver) external onlyCDO {
       ...
        if (token == address(USDe)) {
+           uint256 shares = sUSDe.previewWithdraw(tokenAmount);
+           unstakeCooldown.transfer(sUSDe, receiver, shares);
-           unstakeCooldown.transfer(sUSDe, receiver, tokenAmount);
            return;
        }
        revert UnsupportedToken(token);
    }
```

Strata: Fixed in commit 953c3bc by converting `tokenAmount` to `shares` in `sUSDe` units.

Cyfrin: Verified.

7.3.3 NAVs or APRs are updated using an outdated accounting on certain operations

Description: The next list of functions makes changes either to the NAVs or APRs, or both of them, but none of those variables are updated before the update is reflected on the storage, which means, the accounting is updated using outdated values that don't reflect the current value.

- `Accounting::setReserveBps()`
- `Accounting::setRiskParameters()`
- `Accounting::reduceReserve()`
- `Accounting::updateAprs()`
- `Accounting::onAprChanged()`

For example, when reducing reserves, the `reserveNav` and `nav` are updated, but those variables were not previously updated, which means that the update to `reserveNav` and `nav` is made on outdated values since the last time those navs were updated.

- Since the last update, the `sUSDe <=> USDe rate` could've changed, but that difference is not reflected on the navs prior to updating them to subtract the amount by which the reserves are been reduced.

Another example is when updating APRs:

- If the APR is lowered, SR Tranche will loss on rewards because the lower APR will be applied to a higher time period.
- If the APR is increased, JR Tranche will loss on profits because the higher APR will be applied to a higher time period benefiting the SR Tranche at the expense of the JR Tranche.

Impact: Update of accounting with outdated data can result in unexpected outcomes and potentially messing up the accounting for operations afterwards.

Updating APRs without updating the internal accounting causes that the new APR is applied to calculations until an update occurs.

- If the APR was lowered, SR Tranche will loss on rewards because the lower APR will be applied to a higher time period.
- If the APR was increased, JR Tranche will loss on profits because the higher APR will be applied to a higher time period benefiting the SR Tranche at the expense of the JR Tranche.

Recommended Mitigation: Consider updating the internal NAVs and APRs prior to making any changes to them on the listed functions in the Description section.

Strata: Fixed in commits 7b2354, c3c927, cc1172a, 90a7ad, 2cfe69, and, 674505 by refactoring the update to indexes and aprs in two separate actions as well as triggering updates to the accounting prior to use any of the NAVs, and recalculating the aprs when changes to `jrtNav` or `srtNav` occurs.

Cyfrin: Verfified. The system now has a symmetric relationship across operations where:

- internal accounting its updated at the top of the execution.
- whenever a change to `jrtNav` or `srtNav` happens, the apr is re-calculated at the end of the tx.

7.3.4 Proxy reuse without implementation check inside `UnstakeCooldown` leads to execution on outdated/vulnerable logic

Description: The contract reuses old proxies from the user's `UnstakeCooldown::proxiesPool` without verifying whether those proxies were created from the current implementation. Since a clone's target implementation is permanently embedded in its bytecode, if the owner updates `implementations[token]` using the available `UnstakeCooldown::setImplementations`, any proxies already in a user's pool will still delegate to the old implementation.

Impact: Users may continue operating through outdated or vulnerable implementations even after the owner updates `implementations`. This can cause:

- Inconsistent behavior across requests (some proxies use the new implementation, others use the old one).
- Security risks if the previous implementation contains a bug or vulnerability.
- Accounting or logic mismatches if old and new implementations are not compatible.

Proof of Concept:

1. Owner sets `implementations[token] = ImplV1`.
2. Alice makes two transfers, creating two proxies that point to `ImplV1`.
3. Owner later calls `setImplementations(token, ImplV2)`.
4. Some time passes, the two proxies pointing to `ImplV1` are available.
5. Alice makes another transfer. The contract pops a proxy from her pool and reuses it.
6. That proxy still delegates to `ImplV1`, even though `implementations[token]` is now `ImplV2`.

Recommended Mitigation: When reusing proxies, verify that the proxy's implementation matches the current `implementations[token]`. If not, discard the old proxy and create a new one.

Strata: Fixed in commit ffbedf48d by implementing a validation to check if the current implementation for a token is different than the implementation that was used to create a proxy being reused. If so, then a new proxy is made with the new implementation, and the old proxy is discarded.

Cyfrin: Verified.

7.3.5 When Senior's TargetGain is negative, the tx will revert because the senior loss is not accounted for on the Junior Tranche as profit, causing the navs summation to not match the current nav

Description: When `srtGainTarget < 0`, the code transfers senior loss to `jrtNavT0` *after* initializing `jrtNavT1 = jrtNavT0 + gain_dTAbs`. This fails to propagate the junior "profit" to `jrtNavT1`, leaving it unchanged.

```
 if (srtGainTarget < 0) {
            // Should never happen, jic: transfer the loss to Juniors as profit
            uint256 loss = uint256(-srtGainTarget);
            uint256 srtLoss = Math.min(srtNavT0, loss);

            srtNavT0 -= srtLoss;
//@audit-issue => Updates jrtNavT0 instead of T1
@>          jrtNavT0 += srtLoss;
            srtGainTarget = 0;
        }
        uint256 srtGainTargetAbs = Math.min(
            uint256(srtGainTarget),
            Math.saturatingSub(jrtNavT1, 1e18)
        );

          // [*Users can get their withdrawal active requests DoSed by malicious
          ↪  users*](#users-can-get-their-withdrawal-active-requests-dosed-by-malicious-users) Final
          ↪  new Jrt
        jrtNavT1 = jrtNavT1 - srtGainTargetAbs;
        // [*Withdrawers of `sUSDe` always incur a loss because parameters passed from
        ↪  `Tranche::_withdraw` to `CDO::withdraw` are inverted*](#withdrawers-of-susde-always-incur-a
        ↪  -loss-because-parameters-passed-from-tranchewithdraw-to-cdowithdraw-are-inverted) Final new
        ↪  Srt
        srtNavT1 = srtNavT0 + srtGainTargetAbs;

//@audit-issue => sum of navs won't match because the senior loss is missing on jrtNavT1
        if (navT1 != (jrtNavT1 + srtNavT1 + reserveNavT1)) {
            revert InvalidNavSpit(navT1, jrtNavT1, srtNavT1, reserveNavT1);
        }
```

Impact: Tx will revert with error `InvalidNavSpit()` because the `srtGainTarget` was discounted from `srtNavT0` but was not accounted on `jrtNavT1`

Proof of Concept: **Recommended Mitigation:** Make sure to update `jrtNavT1` instead of T0.

```
 if (srtGainTarget < 0) {
            // Should never happen, jic: transfer the loss to Juniors as profit
            uint256 loss = uint256(-srtGainTarget);
            uint256 srtLoss = Math.min(srtNavT0, loss);

            srtNavT0 -= srtLoss;
-           jrtNavT0 += srtLoss;
+           jrtNavT1 += srtLoss;
            srtGainTarget = 0;
        }
        ...
```

Strata: Fixed in commit 5332b3 by updating the correct variable `jrtNavT1` instead of `jrtNavT0`

Cyfrin: Verified.

7.3.6 `Tranche::maxMint` for Junior Tranches is at risk of overflow when the `jrNav` falls below `1:1` rate to `JR_Shares`

Description: Given that the system is composed of two tranches (Senior and Junior), and all the assets deposited among the two Tranches are polled together on the Strategy, the actual `totalAssets()` for each Tranche is calculated using the corresponding NAVs (`srtNav` and `jrtNav`). The Junior Tranche has the peculiarity that can be used to:

1. Fund the Senior's target APR when the generated APR is not enough.
2. Cover losses by taking the hit first and covering as much as possible to limit/reduce the loss for the Seniors.

Any of the above two events results in a decrease in the `jrtNav`, which is translated into the `totalAssets()` for the Junior Tranche being decremented. As a result, the JR Shares to assets decrements and can cause the `share<=>assets` rate to fall below 1:1.

When the `JR_Shares<=>assets` falls below 1:1, `maxMint()` results in overflow because the underlying Math methods implemented for the conversions of shares to assets, and the fact that `assets` to be converted is set as `type(uint256).max`

Impact: DoS of `Tranche::mint` because `Tranche::maxMint` reverts due to an overflow when converting shares to assets.

Proof of Concept: As demonstrated on the next PoC, when the JR_Shares<=>assets rate falls below 1:1, any calls to `Tranche::maxMint` result in overflow, effectively reverting the tx.

Add the following PoC to `CDO.t.sol` test file:

```
    function test_MaxMintOverflowsInJrTranche() public {
        address alice = makeAddr("Alice");

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

        vm.startPrank(alice);
        USDe.approve(address(jrtVault), type(uint256).max);
        jrtVault.deposit(initialDeposit, alice);
        vm.stopPrank();

//@audit-info => Simulate 10% losses on the Jr Strategy
//@audit => This would be akin to JR Tranche covering losses or making Senior's APR whole.
        vm.prank(address(sUSDeStrategy));
        sUSDe.transfer(alice, initialDeposit / 10);

        vm.expectRevert();
        jrtVault.maxMint(alice);
    }
```

Recommended Mitigation: Given that the max deposits for the Jr Tranche are unlimited, it's okay to return an unlimited max shares too.

- Skip the convertion of assets to shares when `CDO::maxDeposit` returns `type(uint256).max` and instead return the same value.

```
// Tranche::maxMint //

    function maxMint(address owner) public view override returns (uint256) {
        uint256 assets = cdo.maxDeposit(address(this));
+       if (assets == type(uint256).max) {
+          return type(uint256).max;
+       }
        return convertToShares(assets);
    }
```

Strata: Fixed in commit 5748b2f by not converting `type(uint256).max` onto shares and instead returning that value as max shares to mint for JR Tranche.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 `Accounting::setMinimumJrtSrtRatio` sets `reserveBps` instead of `minimumJrtSrtRatio` making ratio configuration impossible

Description: The `Accounting::setMinimumJrtSrtRatio` function contains a an implementation error where it modifies the wrong state variable. Instead of setting `minimumJrtSrtRatio`, the function incorrectly sets `reserveBps`, making it impossible to configure the minimum Junior-to-Senior tranche ratio.

```
    function setMinimumJrtSrtRatio (uint256 bps) external onlyOwner {
        require(bps <= RESERVE_BPS_MAX, "ReserveBpsMax");
        reserveBps = bps;
        emit ReservePercentageChanged(reserveBps);
    }
```

Impact: Impossible Risk Parameter Configuration: The `minimumJrtSrtRatio` variable can only be set during initialization (currently hardcoded to 5%) and cannot be updated afterward, preventing proper risk management adjustments.

Accidental Reserve Configuration: Calling `setMinimumJrtSrtRatio` thinking it will adjust tranche ratios will instead modify the reserve percentage, leading to unintended reserve allocation changes.

Recommended Mitigation: Perform the following changes inside the `Accounting` contract:

```
++event MinimumJrtSrtRatioChanged(uint256 minimumJrtSrtRatio);

function setMinimumJrtSrtRatio (uint256 bps) external onlyOwner {
-   require(bps <= RESERVE_BPS_MAX, "ReserveBpsMax");
-   reserveBps = bps;
-   emit ReservePercentageChanged(reserveBps);
+   require(bps <= 1e18, "InvalidRatio"); // Max 100%
+   minimumJrtSrtRatio = bps;
+   emit MinimumJrtSrtRatioChanged(minimumJrtSrtRatio);
}
```

Strata: Fixed in commit 657bde by updating the correct variable `minimumJrtSrtRatio`.

Cyfrin: Verfified.

7.4.2 Inconsistent APR boundary validation between `AprPairFeed` and `Accounting`

Description: There is a mismatch between APR boundary validation constants in `AprPairFeed` and `Accounting` contracts. The `AprPairFeed` accepts negative APRs down to -50%, but the `Accounting` contract rejects any negative APR values, thus data deemed valid by the oracle is rejected during normalization.

```
// AprPairFeed
int64 private constant APR_BOUNDARY_MAX =    2e12; // 200%
int64 private constant APR_BOUNDARY_MIN = -0.5e12; // -50%

     /// @dev Validates that the given APR is within acceptable bounds
    function ensureValid(int64 answer) internal pure {
        require(
            APR_BOUNDARY_MIN <= answer && answer <= APR_BOUNDARY_MAX,
            "INVALID_APR"
        );


// Accounting
int64   private constant APR_BOUNDARY_MAX = 200e12;
int64   private constant APR_BOUNDARY_MIN = 0;
```

```
    function normalizeAprFromFeed (/* SD7x12 */ int64 apr) internal pure returns (UD60x18) {
        require(
            APR_BOUNDARY_MIN <= apr && apr <= APR_BOUNDARY_MAX,
            "invalid apr"
        );
```

Impact: Protocol DoS: When the feed contains valid negative APR data (between -50% and 0%), the Accounting.normalizeAprFromFeed() function will revert, preventing:

- APR updates via `updateAprs()`;
- Index calculations in `updateIndexes()`;
- Proper accounting updates during deposit/withdrawal flows;

Recommended Mitigation: Align the two contracts:

```
// Accounting.sol
- int64   private constant APR_BOUNDARY_MIN = 0;
+ int64   private constant APR_BOUNDARY_MIN = -0.5e12; // -50%
```

Strata: Fixed in commit c80308 by enforcing APR range on the `Accounting` to not be negative, instead, negative APRs reported from the feed will be considered, Seniors won't have negative APRs.

Cyfrin: Verified.

7.4.3 Inconsistent Risk Premium Validation in `Accounting` Allows Future Underflows or Zero APR

Description: `Accounting::calculateRiskPremium` computes `risk = riskX + riskY * pow(tvlRatio, riskK)`. `Accounting::setRiskParameters` checks that `risk < 1e18` (i.e., less than 100%) immediately after updating the parameters, using the current TVL. However, TVL can change later, so `risk` may become `>= 1e18`. In `Accounting::updateIndexes`, the expression `UD60x18.wrap(1e18) - risk` will yield `0` if `risk == 1e18` (making `aprSrt1` zero) and will revert due to underflow if `risk > 1e18`. This creates an inconsistency between the functions and can produce unintended zeros or reverts at runtime.

Impact:

- If `risk > 1e18` after TVL changes, `Accounting::updateIndexes` reverts on underflow, blocking accounting updates, tranche deposits/withdrawals, and NAV calculations.
- If `risk == 1e18`, `aprSrt1` becomes zero, potentially setting senior APR (`aprSrt`) to low values, leading to incorrect NAV splits and no yield for seniors.

Recommended Mitigation: In `Accounting::updateIndexes`, cap `risk` or revert explicitly if `risk >= 1e18`.

Strata: Fixed in commit 151661 by calculating risk with the maximum TVLsrt ratio: 1 instead of using the current TVLsrt.

Cyfrin: Verified.

7.4.4 Frontrunning to Block Junior Tranche Withdrawals

Description: In `Accounting.sol`, the junior tranche's `maxWithdraw` is capped to ensure the post-withdrawal Junior NAV (`jrtNav`) remains at least `srtNav * minimumJrtSrtRatio / 1e18` (default 5%). This check uses the current (post-any-updates) NAVs from storage and is called during withdrawals in `Tranche.sol` (via `withdraw` function).

An attacker can frontrun a victim's junior withdrawal transaction in the mempool:

- The attacker deposits a calculated amount into the senior tranche (SRT), inflating `srtNav` via `updateBalanceFlow` (called internally during deposit).
- This raises the `minJrt` threshold, causing the victim's withdrawal to fail the `maxWithdraw` check and revert.

It is not neccessary to deposit max amount just need to deposit enough to make this condition revert

```
uint256 maxAssets = maxWithdraw(owner);
if (baseAssets > maxAssets) {
            revert ERC4626ExceededMaxWithdraw(owner, baseAssets, maxAssets);
        }
```

Impact:

- **DoS for Withdrawals**: Victims' legitimate junior withdrawals are blocked, trapping liquidity. In the PoC, a 40e18 withdrawal fails after the attack inflates `srtNav`, setting max to 0.

Proof of Concept: The following Foundry test (`test_FrontrunJRTWithdrawal` from `test/CDO.t.sol`)

```
function test_FrontrunJRTWithdrawal() public {
    // Setup initial state: Mint and deposit to JRT and SRT
    address victim = address(0x1234);
    address attacker = address(0x5678);
    address initialDepositor = address(0x9999);

    uint256 initialJRTDeposit = 100 ether; // jrtNav  100 ether
    uint256 initialSRTDeposit = 1000 ether; // srtNav  1000 ether
    uint256 victimWithdrawalAmount = 40 ether; // Should be valid pre-attack
    uint256 attackDepositAmount = 1000 ether; // Enough to push JRT maxWithdraw to 0

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

    // Initial depositor to SRT (could be anyone)
    vm.startPrank(initialDepositor);
    USDe.mint(initialDepositor, initialSRTDeposit);
    USDe.approve(address(srtVault), initialSRTDeposit);
    srtVault.deposit(initialSRTDeposit, initialDepositor);
    vm.stopPrank();

    // Verify initial state: JRT maxWithdraw should allow the withdrawal
    uint256 preMaxWithdraw = accounting.maxWithdraw(true); // isJrt=true
    assertGt(preMaxWithdraw, victimWithdrawalAmount, "Initial maxWithdraw too low");

    // Simulate attacker frontrunning with large SRT deposit
    vm.startPrank(attacker);
    USDe.mint(attacker, attackDepositAmount);
    USDe.approve(address(srtVault), attackDepositAmount);
    srtVault.deposit(attackDepositAmount, attacker);
    vm.stopPrank();

    // Now simulate victim's withdrawal attempt (should fail due to updated cap)
    vm.startPrank(victim);
    vm.expectRevert(
        abi.encodeWithSelector(
            ERC4626ExceededMaxWithdraw.selector,
            victim,
            victimWithdrawalAmount,
            0 // Post-attack maxWithdraw should be 0
        )
    );
    jrtVault.withdraw(victimWithdrawalAmount, victim, victim);
    vm.stopPrank();

    // Verify post-attack state
```

```
    uint256 postMaxWithdraw = accounting.maxWithdraw(true);
    assertEq(postMaxWithdraw, 0, "Post-attack maxWithdraw not zeroed");
}
```

Steps to Reproduce:

1. Run the test in Foundry: `forge test --match-test test_FrontrunJRTWithdrawal`.

Strata: Fixed in commit 23777d by implementing a soft limit for SRTranche deposits to prevent reaching `minimumJrtSrtRatio` by incrementing the SR deposits.

Cyfrin: Verified.

7.4.5 `DEFAULT_ADMIN_ROLE` can be mistakenly granted to an account when granting permission to call `fallback` on any contract

Description: `AccessControlManager` is in charge of managing roles and permissions for accounts to determine what users can call on which contracts.

Permissions are granted at the selector level, and is possible to grant permissions to only one contract at a time, or authorize an account to call the same selector on any contract.

There is an edge case when permitting an account to call the `fallback()` function whose selector is `bytes4(0)` on any account (address(0)).

- The combination of those two inputs results in calculating the role as bytes(0), which is the exact value assigned for the DEFAULT_ADMIN_ROLE.

```
    function grantCall(address contractAddress, bytes4 sel, address accountToPermit) public {
//@audit-issue => The calculated role for `address(0)` and `bytes4(0)` is bytes32(0)`
        bytes32 role = roleFor(contractAddress, sel);
//@audit-issue => Granting bytes32(0) to the account results in granting the DEFAULT_ADMIN_ROLE
        grantRole(role, accountToPermit);
        emit PermissionGranted(accountToPermit, contractAddress, sel);
    }

    function roleFor(address contractAddress, bytes4 sel) internal pure returns (bytes32 role) {
//@audit-issue => The calculated role for `address(0)` and `bytes4(0)` is bytes32(0)`
        role = (bytes32(uint256(uint160(contractAddress))) << 96) | bytes32(uint256(uint32(sel)));
    }
```

Impact: Users can be mistakenly granted the DEFAULT_ADMIN_ROLE, which they can then use to authorize other users to call restricted functions.

Proof of Concept: Add the next PoC to `CDO.t.sol` test file:

1. Alice gets DEFAULT_ADMIN_ROLE by mistake when she was meant to receive permission to call fallback() on any contract
2. Once Alice has DEFAULT_ADMIN_ROLE, he authorizes Bob to call other functions on a different contract.

```
    function test_grantsDefaultAdminByMisstake() public {
        bytes32 DEFAULT_ADMIN_ROLE = acm.DEFAULT_ADMIN_ROLE();

        address contractAddress = address(0);
        bytes4 selector = bytes4(0);

        address alice = makeAddr("Alice");

        assertFalse(acm.hasRole(DEFAULT_ADMIN_ROLE, alice));

        //@audit-issue => Granting permission to alice to call fallback function on any contract results
        ↪  on granting Alice the DEFAULT_ADMIN_ROLE
```

```
        acm.grantCall(contractAddress, selector, alice);
        assertTrue(acm.hasRole(DEFAULT_ADMIN_ROLE, alice));

        address contractA = makeAddr("contractA");
        address bob = makeAddr("bob");
        bytes4 withdrawSelector = bytes4(keccak256(bytes("withdraw(address,uint256)")));

        assertFalse(acm.hasPermission(bob, contractA, withdrawSelector));

        //@audit-info => Alice w/ DEFAULT_ADMIN can grant permissions to other accounts
        vm.startPrank(alice);
        acm.grantCall(contractA, withdrawSelector, bob);
        assertTrue(acm.hasPermission(bob, contractA, withdrawSelector));
    }
```

Recommended Mitigation: Validate that the computed role is not the DEFAULT_ADMIN_ROLE; otherwise, revert the tx.

```
    function grantCall(address contractAddress, bytes4 sel, address accountToPermit) public {
        bytes32 role = roleFor(contractAddress, sel);
+       require(role != DEFAULT_ADMIN_ROLE, "Granting DEFAULT_ADMIN_ROLE");
        grantRole(role, accountToPermit);
        emit PermissionGranted(accountToPermit, contractAddress, sel);
    }
```

Strata: Fixed in commit e6ad2d by adding a check to revert when `contractAddress` is `address(0)` or `selector` is `bytes(0)`

Cyfrin: Verified. New change prevents from mistakenly assign DEFAULT_ADMIN_ROLE. Permissions are granted on a per-contract per-selector basis.

7.5 Informational

7.5.1 Incorrect Comment and Missing Lower Bound for `minimumJrtSrtRatio` in `Accounting`

Description:

```
/// @dev minimum TVL ratio: TVLjrt/TVLsrt, e.g. >= 0.05%
```

```
minimumJrtSrtRatio = 0.05e18;
```

The comment says "0.05%" (0.0005e18) but the value is 5% (0.05e18). and there is no check to prevent setting value <0.05% and it may be intended to start with 5% ratio

Recommended Mitigation:

- Update comment to reflect intended 5% (e.g., ">= 5%").
- In `Accounting::setMinimumJrtSrtRatio`, add `require(bps >= 0.0005e18, "RatioTooLow");` for a min bound.

Strata: Fixed in commit eefd73 and c1afee2. Updated comment and added check to validate lower bound for `minimumJrtSrtRatio`

Cyfrin: Verified.

7.5.2 Unused Import of `OwnableUpgradeable` in `Accounting.sol`

Description: `OwnableUpgradeable` is imported but not used directly in this file.

Recommended Mitigation: Remove the unused import to improve code cleanliness:

```
- import { OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
```

Strata: Fixed in commit cfc5117b.

Cyfrin: Verified.

7.5.3 Missing Validation of Fallback APR Values in `AprPairFeed::latestRoundData`

Description: `AprPairFeed::latestRoundData` fetches APRs from a preferred source (feed or strategy provider). If the feed is stale, it falls back to the strategy provider via `provider.getAprPair()` but does not validate the returned values (e.g., via `ensureValidAprs` or bounds checks), unlike potential validations in the feed path. This allows potentially invalid values to be used

```
function latestRoundData() external view returns (TRound memory) {
        TRound memory round = latestRound;

        if (sourcePref == ESourcePref.Feed) {
            uint256 deltaT = block.timestamp - uint256(round.updatedAt);
            if (deltaT < roundStaleAfter) {
                return round;
            }
            // falls back to strategy ↓
        }

        (int64 aprTarget, int64 aprBase, uint64 t1) = provider.getAprPair();
        return TRound({
            aprTarget: aprTarget,
            aprBase: aprBase,
            updatedAt: t1,
            answeredInRound: latestRoundId + 1
        });
    }
```

Recommended Mitigation: Add validation after fallback fetch, similar to feed bounds:

```
        (int64 aprTarget, int64 aprBase, uint64 t1) = provider.getAprPair();
+       // Add validation, e.g.:
+       ensureValid(aprTarget);
+       ensureValid(aprBase);
        return TRound({
            aprTarget: aprTarget,
            aprBase: aprBase,
            updatedAt: t1,
            answeredInRound: latestRoundId + 1
        });
```

Strata: Fixed in commit 1c4009a by validating `aprTarget` and `aprBase`.

Cyfrin: Verified.

7.5.4 Missing `Unstaked` event for immediate unstake in `UnstakeCooldown::transfer`

Description: When `UnstakeCooldown::transfer` triggers an immediate unstake (i.e. the handler's call to `proxy.request()` returns `unlockAt <= block.timestamp`), the function returns early after returning the proxy to the pool — but it does not emit the `Unstaked` event for that immediate completion. As a result an on-chain event is missing for the flow where the redemption happened immediately (no cooldown).

Impact: Off-chain systems relying on events for tracking withdrawals may miss these immediate unstake operations. This does not affect on-chain balances or security, and users still receive their funds correctly.

Recommended Mitigation: Emit an `Unstaked` (or create a new `ImmediateUnstake`) event in the `transfer()` immediate branch using the `amount` parameter passed to `transfer()`.

Strata: Fixed in commit c08784 by unifying the event being emitted in both cooldown contracts as `Finalized`

Cyfrin: Verified.

7.5.5 Use explicit unsigned integer sizing instead of `uint`

Description: In Solidity `uint` automatically maps to `uint256` but it is considered good practice to specify the exact size when declaring variables:

```
Accounting.sol
200:    ) public view returns (uint jrtNavT1, uint srtNavT1, uint reserveNavT1) {

StrataCDO.sol
86:     uint totalAssetsOverall = strategy.totalAssets();
128:    uint jrtAssetsIn = isJrt_ ? baseAssets : 0;
129:    uint srtAssetsIn = isJrt_ ? 0          : baseAssets;
143:    uint jrtAssetsOut = isJrt_ ? baseAssets : 0;
144:    uint srtAssetsOut = isJrt_ ? 0          : baseAssets;
```

Strata: Fixed in commit 506c4c.

Cyfrin: Verified.

7.5.6 Unreachable code inside `sUSDeStrategy::reduceReserve`

Description: The last line inside `sUSDeStrategy::reduceReserve`: `revert UnsupportedToken(token);` is unreachable.

`sUSDeStrategy::reduceReserve` can be called only via `StrataCDO::reduceReserve`, which before calling `strategy.reduceReserve(token, tokenAmount, treasury);` will call `uint256 baseAssets = strategy.convertToAssets(token, tokenAmount, Math.Rounding.Floor);`.

`sUSDeStrategy::convertToAssets` already reverts for unsupported tokens:

```
function convertToAssets (address token, uint256 tokenAmount, Math.Rounding rounding) external view
↪  returns (uint256) {
    if (token == address(sUSDe)) { // if sUSDe, use previewRedeem or previewMint
        return rounding == Math.Rounding.Floor
            ? sUSDe.previewRedeem(tokenAmount) // aka convertToAssets(tokenAmount)
            : sUSDe.previewMint(tokenAmount); // aka convertToAssetsCeil(tokenAmount)
    }
    if (token == address(USDe)) { // if USDe, return the input amount
        return tokenAmount;
    }
    revert UnsupportedToken(token);
}
```

Strata: Acknowledged; we prefer to leave the current behaviour. You are absolutely right that this revert is unreachable under the current conditions, but to keep it consistent with the other methods, the Strategy must check every operation on tokens to see if it is supported.

Cyfrin: Acknowledged.

7.5.7 Misleading variable name to set the `asset` for the `Tranche`

Description: The variable that is used to set the `asset` the `Tranche` will work with is named `stakedAsset`. This is misleading because the two assets the system works with are `USDe` and `sUSDe`. ' sUSDe is the staked version of `USDe`, so the variable name can mislead people into thinking that the `asset` of the `Tranche` is expected to be `sUSDe` instead of `USDe`.

Recommended Mitigation: Rename the variable name to `baseAsset` or another name that doesn't have the word `stake` in the name.

Strata: Fixed in commit 2d7f5a17.

Cyfrin: Verified.

7.5.8 Typos and Bad NATSPEC

Description: We have identified the following typos/NATSPEC problems:

```
AccessControlled.sol
29: `NewAccessControlManager` should be `NewAccessControllManager`

Accounting.sol:
66: `InvalidNavSpit` should be `InvalidNavSplit`

ERC20Cooldown.sol:
10: `allows to store` should be `allows storing`

StrataCDO.sol:
42: `Sinior` should be `Senior`
62: `@notice` is empty

UnstakeCooldown.sol:
144: `for a tokens` should be `for tokens`
```

Strata: Fix in commit 506c4c by correcting Typos and updating Natspec.

Cyfrin: Verified.

7.5.9 `AprPairFeed::getRoundData` can return data for a different round than the specified

Description: When updating round data on the `AprPairFeed`, the data is saved in the `rounds` mapping, which is accessed via the calculated `roundIdx`. The way in which the `roundIdx` is derived causes it to be repeated

every 20 updates, from 0 to 20, over and over. This means that an older `roundIdx` would eventually calculate the same `roundIdx` as a newer one. This can cause the problem that when retrieving data for a specific roundId, `AprPairFeed::getRoundData` returns data for a newer roundId.

```
    function updateRoundDataInner(int64 aprTarget, int64 aprBase, uint64 t) internal {
        ...
        uint64 roundId = (latestRoundId + 1);
@>      uint64 roundIdx = roundId % roundsCap;

        latestRoundId = roundId;
        latestRound = TRound({
            aprTarget: aprTarget,
            aprBase: aprBase,
            updatedAt: t,
            answeredInRound: roundId
        });
@>      rounds[roundIdx] = latestRound;

        emit AnswerUpdated(aprTarget, aprBase, roundId, t);
    }

    function getRoundData(uint64 roundId) public view returns (TRound memory) {
@>      uint64 roundIdx = roundId % roundsCap;
        TRound memory round = rounds[roundIdx];
        require(round.updatedAt > 0, "No data present");
@>      return round;
    }
```

Recommended Mitigation: Consider validating that the data read from the `rounds` mapping matches the specified roundId.

```
    function getRoundData(uint64 roundId) public view returns (TRound memory) {
        uint64 roundIdx = roundId % roundsCap;
        TRound memory round = rounds[roundIdx];
        require(round.updatedAt > 0, "No data present");
+       require(round.answeredInRound == roundId, "old round");
        return round;
    }
```

Strata: Fixed in commit 233e3d by verifying the queried data corresponds to the same requested `roundId`.

Cyfrin: Verified.

7.5.10 Cooldown contracts underreport the real balance of users because they only consider the balance of requests whose cooldown period is over

Description: Cooldown contracts underreport the real balance of users because not all active requests are accounted for; only those requests whose cooldown period has expired are considered part of a user's balance.

This implementation doesn't accurately show the actual information about the user's balance at all times, only until requests are finalized (cooldown period is over).

Recommended Mitigation: Consider refactoring the `balanceOf()` method to return all balances, including both available and locked balances.

Strata: Fixed in commit 949cb4 and 1f82c6 to return more detailed data about the active requests, such as `pending` and `claimable` amounts, based on the lock periods.

Cyfrin: Verified.

7.5.11 `UnstakeCooldown::balance` requires a different token contract than the actual token that is reporting the balance for

Description: The `UnstakeCooldown` contract is used when withdrawing `USDe`. It is meant to allow the system to request the cooldown of `sUSDe` on the Ethena contract and then `unstake` the underlying `USDe` to be sent to the user. When a cooldown is requested on the `sUSDe` contract, the underlying amount of `USDe` that will be given to the user is tracked on `cooldown.underlyingAmount`.

So, `cooldowns` on `sUSDe` returns the amount of `USDe`, which means, querying `proxy.getPendingAmount()` returns amounts of `USDe`, not `sUSDe`.

- This means, `UnstakeCooldown::balanceOf` is querying amounts of `USDe` for the `user`, not `sUSDe`, but the request on the `activeRequests` mapping was associated with `sUSDe` instead of `USDe`. This forces the users to call `balanceOf()` specifying `sUSDe` as the token when in reality they are querying amounts of `USDe`.

```
//sUSDeStrategy::withdraw//
    function withdraw (address tranche, address token, uint256 tokenAmount, uint256 baseAssets, address
    ↪  receiver) external onlyCDO returns (uint256) {
        ...
        if (token == address(USDe)) {
@>          unstakeCooldown.transfer(sUSDe, receiver, shares);
            return baseAssets;
        }
        revert UnsupportedToken(token);
    }

//UnstakeCooldown::transfer//
function transfer(IERC20 token, address to, uint256 amount) external {
    ...

@>  TRequest[] storage requests = activeRequests[address(token)][to];
    IUnstakeHandler[] storage proxies = proxiesPool[address(token)][to];

    ...

    requests.push(TRequest(uint64(unlockAt), proxy));
    emit Requested(address(token), to, amount, unlockAt);
}

function balanceOf (IERC20 token, address user, uint256 at) public view returns (uint256) {
@>  TRequest[] storage requests = activeRequests[address(token)][user];
    uint256 l = requests.length;
    uint256 balance = 0;
    for (uint256 i = 0; i < l; i++) {
        TRequest memory req = requests[i];
        if (req.unlockAt <= at) {
@>          balance += req.proxy.getPendingAmount();
        }
    }
    return balance;
}

//sUSDeCooldownRequestImpl//

    function getPendingAmount () external view returns (uint256 amount) {
@>      amount = sUSDe.cooldowns(address(this)).underlyingAmount;
        return amount;
    }
```

Recommended Mitigation: When registering a new request, associate it with `USDe` instead of `sUSDe`. The rest of the code would work fine, and the users would now input `USDe` as the token and get amounts of `USDe`. Additionally, it might be worth documenting that the reported balance corresponds to `USDe` units.

Strata: Acknowledged; changing the accepted token in balanceOf to the underlying token could break the finalization logic, since the underlying token may later be used in different staked tokens. Here we want to stay focused on the specific staked asset.

7.5.12 No helper functions for `maxDeposit`, `maxMint`, `maxRedeem`, `maxWithdraw` for `sUSDe`

Description: The Tranches allow deposits using `USDe` or `sUSDe`, but there aren't helper functions to calculate how much `sUSDe` an operation can be made with. The implemented functions only work for `USDe` amounts.

Specifically for withdrawing/redeeming `sUSDe`, it is not a straightforward process for a user to determine how much `sUSDe` they can request to be withdrawn.

The fact that the withdrawal flows for `USDe` or `sUSDe` vary considerably is mainly due to the unstaking time window for withdrawing USDe.

Recommended Mitigation: Consider adding helper functions to allow users to calculate max values when operating using `sUSDe`.

- Note: These functions don't need to be called anywhere in the code per se; they should just be available for a user to call them off-chain.

Strata: Acknowledged; the user can use the `Strategy::convertToTokens`. To get "maxWithdraw" for sUSDe, the user calls the basic `maxWithdraw` to get amount in `USDe` and converts the value into `sUSDe` with `convertToTokens`. Later on we will extract those helper functions into additional "Lens" contract.

7.6 Gas Optimization

7.6.1 Remove redundant checks

Description: The function below behaves the same whether or not that check is present.

- `StrataCDO.sol`

```
// isJrt
154:         if (tranche == address(0)) {
155:             revert InvalidTranche(tranche);
156:         }
```

Strata: Removed redundant check in commit 934be5.

Cyfrin: Verified.

7.6.2 Cache result of external calls when result can't change between calls and is used multiple times

Description: * `Tranche.sol`

```
// configure
212:         IERC20[] memory tokens = cdo.strategy().getSupportedTokens();
213:         uint256 len = tokens.length;
214:         address strategy = address(cdo.strategy());
```

Recommended Mitigation:

```
++import { IStrategy } from "./interfaces/IStrategy.sol";
[...]
    function configure () external onlyCDO {
--      IERC20[] memory tokens = cdo.strategy().getSupportedTokens();
--      uint256 len = tokens.length;
--      address strategy = address(cdo.strategy());
++      address strategy = address(cdo.strategy());
++      IERC20[] memory tokens = IStrategy(strategy).getSupportedTokens();
++      uint256 len = tokens.length;
```

Strata: Fixed in commit 732b1a8 by caching the result of the call `cdo::strategy` and re-using it.

Cyfrin: Verified.