Remora Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Stalin

July 4, 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

Remora is developing a Real-World Asset (RWA) protocol which enables physical real estate developers to tokenize and sell fractional shares of physical properties to investors worldwide. Investors can earn a share of passive rental income receiving regular payouts without the headache and burden of traditional landlord responsibility.

5 Audit Scope

The original audit scope at commit `0219ab16f594283ed245fb188ce70e4a9ef987ea` was:

```
contracts/RWAToken/BurnStateManager.sol
contracts/RWAToken/DividendManager.sol
contracts/RWAToken/DocumentManager.sol
contracts/RWAToken/LockUpManager.sol
contracts/RWAToken/RemoraToken.sol
contracts/AccessManager.sol
contracts/Allowlist.sol
contracts/PledgeManager.sol
contracts/ReferralManager.sol
contracts/RemoraIntermediary.sol
contracts/TokenBank.sol
```

A new contract `contracts/PaymentSettler.sol` was added and modifications made to existing contracts at commit `f929ff115f82e76c8eb497ce769792e8de99602b` during the audit; the new contract and the changes to existing contracts were reviewed as part of the audit.

6 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the Remora smart contracts provided by Remora. In this period, a total of 57 issues were found.

The findings consist of 3 Critical, 3 High, 17 Medium and 13 Low with the remainder being informational and gas optimizations.

2/3 Criticals were both related to using small unsigned integer sizes that resulted in Denial Of Service (DoS) overflow reverts. The protocol should strongly consider standardizing on at least `uint128` for all token amounts to prevent similar findings:

- 7.1.1 - `uint32` overflow that would DoS pledges and refunds
- 7.1.2 - `uint64` overflow that would DoS payout distributions
- 7.1.3 - payouts of holders forwarding to the same holder can be griefed by a single holder

Of the 3 Highs, 2 were related to changing the stable coin used for payouts to a stable coin which uses different decimal precision. The protocol should strongly consider standardizing an internal decimal precision for all internal protocol accounting then converting to native decimal precision for token transfers to prevent similar findings:

- 7.2.1 - attacker could make pledges on behalf of other users, spending their tokens
- 7.2.2 - `PaymentSettler` accounting became corrupted when changing the stable coin used to process payments
- 7.2.3 - `PaymentSettler` can change its stable coin but `RemoraToken` can't, resulting in a corrupted state with DoS of core functions

The 16 Medium and 13 Low findings involved a wide variety of issues.

Test Suite Analysis

The protocol has a basic hardhat test suite which has no fuzz or invariant testing and doesn't perform much state verification; via mutation testing we observed it was possible to comment out key lines of code such as token transfers and all unit tests would continue to pass.

We wrote a targeted Foundry test suite from scratch using fuzz testing to explore various scenarios which helped us get many of our findings, which we contributed to the protocol at the end of the audit as an additional deliverable.

Post Audit Recommendations

We strongly encourage the protocol to continue developing our Foundry test suite to:

- achieve as close as possible to full coverage using stateless fuzz tests which perform thorough pre/post state validation
- implement contract-specific and protocol-specific invariants using Foundry's powerful invariant testing capabilities

Due to the significant number of Critical & High findings it is possible other serious vulnerabilities still remain and were not able to be discovered during the 6-day audit window. Once a thorough test suite has been completed with full fuzz and invariant testing we encourage the client to return for a second audit during which no Critical or High severity vulnerabilities should be found.

Summary

Project Name	Remora
Repository	remora-smart-contracts
Commit	0219ab16f594...
Fix Commit	9051af840f92...
Audit Timeline	June 23rd - June 30th
Methods	Manual Review, Fuzz Testing

Issues Found

Critical Risk	3
High Risk	3
Medium Risk	17
Low Risk	13
Informational	9
Gas Optimizations	12
Total Issues	57

Summary of Findings

[C-1] `PledgeManager::pledge`, `refundTokens` will revert due to overflow when `pricePerToken * numTokens > type(uint32).max`	Resolved
[C-2] Distribution of payouts will revert due to overflow when payment is made using a stablecoin with high decimals	Resolved
[C-3] A single holder can grief the payouts of all holders forwarding their payouts to the same forwarder	Resolved
[H-1] Attacker can make pledge on behalf of users if those users have approved `PledgeManager` to spend their tokens	Resolved
[H-2] Accounting on `PaymentSettler` will be corrupted when changing `stablecoin` that is used to process payments	Resolved
[H-3] `PaymentSettler` can change `stablecoin` but `RemoraToken` can't resulting in corrupted state with DoS for core functions	Resolved
[M-01] `PledgeManager::refundTokens` doesn't decrement `tokensSold` when pledge hasn't concluded, preventing pledge from reaching its funding goal	Resolved
[M-02] `TokenBank::withdrawFunds` resets `memory` not `storage` fee and sale amounts allowing multiple withdraws for the same token	Resolved
[M-03] Fee should be calculated after first purchase discount is applied in `TokenBank::buy` to prevent over-charging users	Resolved
[M-04] Buyers can pledge for tokens without having signed all documents that are required to be signed	Resolved
[M-05] Hardcoding deadline for `permit()` will mess up the `structHash` leading to an invalid signature	Resolved
[M-06] `DocumentManager::hasSignedDocs` incorrectly returns `true` when there are no documents to sign	Resolved
[M-07] Don't add duplicate `documentHash` to `DocumentManager::DocumentStorage::_docHashes` when overwriting via `_setDocument` as this causes panic revert when calling `_removeDocument`	Resolved
[M-08] Check return value when calling `Allowlist::exchangeAllowed` and `RemoraToken::_exchangeAllowed` to prevent unauthorized transfers	Resolved
[M-09] `DividendManager::distributePayout` will always revert after 255 payouts, preventing any future payout distributions	Resolved

[M-10] Payout distributions to Remora token holders are diluted by initial token owner mint	Resolved
[M-11] Burning ALL PropertyTokens of a frozen holder results in the holder losing the payouts distribution while he was frozen	Resolved
[M-12] Overriding fees can't be switched back once set	Resolved
[M-13] Forwarders can lose payouts of the holders forwarding to them	Resolved
[M-14] Pledge can't successfully complete unless `RemoraToken` is paused	Resolved
[M-15] `RemoraToken` transfers are bricked when `from` is not whitelisted, has sufficient tokens to transfer but no tokens locked	Resolved
[M-16] Tokens that were locked when `lockUpTime > 0` will be impossible to unlock if `lockUpTime` is set to zero	Resolved
[M-17] Forwarders who aren't also holders are unable to claim forwarded payouts	Resolved
[L-01] Use `SafeERC20` functions instead of standard `ERC20` transfer functions	Resolved
[L-02] Fee refund can lose precision	Resolved
[L-03] `TokenBank::addToken` should revert if token has already been added	Resolved
[L-04] Changing stablecoin on TokenBank can mess up fees collection	Resolved
[L-05] `TokenBank::removeToken` reverts when token balance is zero, making it impossible to remove tokens from the `developments` array	Resolved
[L-06] Zero token transfers record receiving user as a holder in `DividendManager::HolderStatus` even if they have zero token balance	Resolved
[L-07] `DividendManager::distributePayout` records a new payout record increasing the current payout index for zero `payoutAmount`	Resolved
[L-08] `PaymentSettler::claimAllPayouts` doesn't validate input `tokens` addresses are legitimate contracts before calling `adminClaimPayout` on them	Resolved
[L-09] Minting new PropertyTokens close to the end of the distribution period will dilute rewards for holders who were holding for the full period	Acknowledged
[L-10] Forwarder can be frozen and still receive and claim payouts while frozen	Acknowledged
[L-11] Forwarder can be set to frozen address	Acknowledged
[L-12] Impossible to remove a document added with zero uri length	Resolved
[L-13] `PledgeManager::pricePerToken` can only support a maximum price of 4294	Acknowledged
[I-1] Emit missing events for storage changes	Resolved
[I-2] Rename `isAllowed` to `wasAllowed` in `Allowlist::allowUser`, `disallowUser`	Resolved
[I-3] Use constants instead of magic numbers	Resolved
[I-4] Using explicit unsigned integer sizing instead of `uint`	Resolved
[I-5] Retrieve and enforce token decimal precision	Resolved
[I-6] `LockUpManager::LockUpStorage::_regLockUpTime` is never used	Resolved

[I-7] Use `SignatureChecker` library and optionally support `EIP7702` accounts which use their private key to sign	Resolved
[I-8] Use `EIP712Upgradeable` library to simplify `DocumentManager`	Resolved
[I-9] Remove unnecessary imports and inheritance	Resolved
[G-01] Fail fast without performing unnecessary storage reads	Resolved
[G-02] Don't initialize to default values	Resolved
[G-03] Cache identical storage reads	Resolved
[G-04] Remove `< 0` comparison for unsigned integers	Resolved
[G-05] Variables in non-upgradeable contracts which are only set once in `constructor` should be declared `immutable`	Resolved
[G-06] Break out of loop once element has been deleted in `TokenBank::removeToken`	Resolved
[G-07] Use named returns where this eliminates a local variable and especially for `memory` returns	Resolved
[G-08] In `RemoraToken::adminClaimPayout`, `adminTransferFrom` don't call `hasSignedDocs` when `checkTC == false`	Resolved
[G-09] In `RemoraToken::transfer`, `transferFrom` and `_exchangeAllowed` perform all checks for each user together in order to prevent unnecessary work	Resolved
[G-10] `AllowList::hasTradeRestriction` mutability should be set to `view`	Resolved
[G-11] Remove `decimals` from initial `RemoraToken` mint	Resolved
[G-12] Use timestamp instead of uri length to test of existing document in `DocumentManager`	Resolved

7 Findings

7.1 Critical Risk

7.1.1 `PledgeManager::pledge`, `refundTokens` will revert due to overflow when `pricePerToken * numTokens > type(uint32).max`

Description: `PledgeManager::pledge` multiplies two `uint32` variables and stores the result into a `uint256`, attempting to account for when the multiplication returns a value greater than `type(uint32).max`:

```
uint256 stablecoinAmount = pricePerToken * numTokens; // account for overflow
```

`PledgeManager::refundTokens` does the same thing:

```
uint256 refundAmount = numTokens * pricePerToken; //TOOD: overflow check
```

However this won't work correctly since if the result of the multiplication is greater than `type(uint32).max` the function will revert.

Impact: The maximum value of `uint32` is 4294967295. Since `pricePerToken` uses 6 decimals, the maximum possible `stablecoinAmount` is $4294.96 which is very low; pledging will be revert for many reasonable amounts that users will want to do.

The contract is also not upgradeable so this can't be fixed via upgrading.

Proof of Concept: You can easily verify this behavior using chisel:

```
$ chisel
Welcome to Chisel! Type `!help` to show available commands.
 uint32 a = type(uint32).max;
 uint32 b = 10;
 uint256 c = a * b;
Traces:
  [401] 0xBd770416a3345F91E4B34576cb804a576fa48EB1::run()
     ← [Revert] panic: arithmetic underflow or overflow (0x11)

Error: Failed to execute REPL contract!
```

Recommended Mitigation: Firstly consider increasing the size of `pricePerToken` and `numTokens`, since the max value of `uint32` is 4,294,967,295 which means:

- for price with 6 decimals, the maximum `pricePerToken` is $4294 which may be too small
- the maximum token amount is 4.29B which may work or also be too small
- simple solution: standardize all protocol token amounts to `uint128`

Secondly instead of multiplying two smaller types such as `uint32`, cast one of them to `uint256`:

```
- uint256 stablecoinAmount = pricePerToken * numTokens; // account for overflow
+ uint256 stablecoinAmount = uint256(pricePerToken) * numTokens;
```

Verify the fix via chisel:

```
$ chisel
Welcome to Chisel! Type `!help` to show available commands.
 uint32 a = type(uint32).max;
 uint32 b = 10;
 uint256 c = uint256(a) * b;
 c
Type: uint256
 Hex: 0x9fffffff6
 Hex (full word): 0x00000000000000000000000000000000000000000000000000000009fffffff6
 Decimal: 42949672950
```

Consider these lines in `TokenBank::buyToken` whether a similar fix is needed there:

```
// @audit can `amount * curData.pricePerToken * curData.saleFee > type(uint64).max`? If so then
// consider making a similar fix here to prevent overflow revert
        uint64 stablecoinValue = amount * curData.pricePerToken;
        uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;
```

Remora: Fixed in commits a0b277f, ced21ba.

Cyfrin: Verified.

7.1.2 Distribution of payouts will revert due to overflow when payment is made using a stablecoin with high decimals

Description: Payouts are meant to be paid using a stablecoin, originally a stablecoin with 6 decimals (USDC). But the system has the capability of changing the stablecoin that is used for payments. Could be USDT (8 decimals), USDS (18 decimals).

As part of the changes made to introduce the PaymentSettler, the data type of the variable `calculatedPayout` was changed from a uint256 to a uint64. This change introduces a critical vulnerability that can cause an irreversible DoS to users to collect their payouts.

A uint64 would revert when distributing a payout of 20 USD using a stablecoin of 18 decimals.

- As we can see on chisel, 20e18 is > the max value a uint64 can fit

```
 bool a = type(uint64).max > 20e18;
 a
Type: bool
 Value: false
```

For example, there is a user who has 5 distributions pending to be calculated, and in the most recent distribution, the distribution is paid with a stablecoin of 18 decimals. (assume the user is earning 50USD on each distribution)

- When the user attempts to calculate its payout, the tx will revert because the last distribution will take the user's payout beyond the value that can fit in a uint64, so, when safeCasting the payout down to a uint64, an overflow will occur, and tx will blow up, resulting in this user getting DoS from claiming not only the most recent payout, but all the previous payouts that haven't been calculated yet.

```
    function payoutBalance(address holder) public returns (uint256) {
        ...
        for (uint16 i = payRangeStart; i >= payRangeEnd; --i) {
            ...

            PayoutInfo memory pInfo = $._payouts[i];
//@audit => `pInfo.amount` set using a stablecoin with high decimals will bring up the payoutAmount
↪  beyond the limit of what can fit in a uint64
            payoutAmount +=
                (curEntry.tokenBalance * pInfo.amount) /
                pInfo.totalSupply;
            if (i == 0) break; // to prevent potential overflow
        }
        ...
        if (payoutForwardAddr == address(0)) {
//@audit-issue => overflow will blow up the tx
            holderStatus.calculatedPayout += SafeCast.toUint64(payoutAmount);
        } else {
//@audit-issue => overflow will blow up the tx
            $._holderStatus[payoutForwardAddr].calculatedPayout += SafeCast
                .toUint64(payoutAmount);
        }
```

Impact: Irreversible DoS to holders' payouts distribution.

Recommended Mitigation: To solve this issue, the most straightforward fix is to change the data type of `calculatePayout` to at least `uint128` & consider standardizing all token amounts to `uint128`.

But, this time it is recommended to go one step further and normalize the internal accounting of the system to a fixed number of decimals in such a way that it won't be affected by the decimals of the actual stablecoin that is being used to process the payments.

As part of this change, the `PaymentSettler` contract must be responsible for converting the values sent and received from the RemoraToken to the actual decimals of the current configured stablecoin.

Remora: Fixed in commits a0b277f, ced21ba.

Cyfrin: Verified.

7.1.3 A single holder can grief the payouts of all holders forwarding their payouts to the same forwarder

Description: This grief attack is similar to [issue *Forwarders can lose payouts of the holders forwarding to them*](`https://github.com/remora-projects/remora-smart-contracts/issues/49`). The main difference is that this attack does not need the forwarder to gain holder status and zero out his balance on the same distributionIndex. This grief attack can be executed at any index while the forwarder has no balance.

The steps that allows the grief attack to occur are:

1. forwarder has balance, it is a holder
2. various holders set the same address as their designated forwarder
3. payouts for holders are computed and credited to forwarder
4. forwarder claims payouts, and gets computed all pending payouts
    - At this point, payoutBalance of forwarder would be 0
5. forwarder zeros out his balance, and gets removed the `isHolder` status (no longer a holder)
6. distributions passes
7. One of the holders removes the forwarder as his designated forwarder
    - Because the forwarder has no balance, and is not a holder, the data of the forwarder will be deleted, including any outstanding calculatedPayout that has been accumulated for the holders who set the forwarder as their forwarder.
8. As a result of step 7, the unclaimed payouts earned by the holder get lost

Impact:

- Payouts of holders forwarding to the same forwarder can be grief by a single holder.
- Holders forwarding their payouts to a non-holder account will lose their payouts if they remove the forwarder while he is still a non-holder.

Proof of Concept: Run the following test to reproduce the scenario described in the Description section.

```
    function test_holderForcesForwarderToLosePayouts() public {
        address user1 = users[0];
        address user2 = users[1];
        address forwarder = users[2];

        uint256 amountToMint = 1;

        _whitelistAndMintTokensToUser(user1, amountToMint * 8);
        _whitelistAndMintTokensToUser(user2, amountToMint);
        _whitelistAndMintTokensToUser(forwarder, amountToMint);

        // both users sets the same forwarder as their forwardAddress
        remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);
        remoraTokenProxy.setPayoutForwardAddress(user2, forwarder);
        // fund total payout amount to funding wallet
        uint64 payoutDistributionAmount = 100e6;

        // Distribute payouts for the first 5 distributions
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // user1 must have 0 payout because it is forwarding to `forwarder`
        uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");

        // user2 must have 0 payout because it is forwarding to `forwarder`
        uint256 user2PayoutBalance = remoraTokenProxy.payoutBalance(user2);
        assertEq(user2PayoutBalance, 0, "Forwarding payout is not working as expected");

        //forwarder must have the full payout for the 5 distributions because both users are forwarding
        ↪  to him
        uint256 forwarderPayoutBalance = remoraTokenProxy.payoutBalance(forwarder);
        assertEq(forwarderPayoutBalance, payoutDistributionAmount * 5, "Forwarding payout is not working
        ↪  as expected");

        // forwarder claims all the outstanding payout
        vm.startPrank(forwarder);
        remoraTokenProxy.claimPayout();
        assertEq(stableCoin.balanceOf(forwarder), forwarderPayoutBalance);

        // forwarder zeros out his PropertyToken's balance
        remoraTokenProxy.transfer(user2, remoraTokenProxy.balanceOf(forwarder));
        vm.stopPrank();

        assertEq(remoraTokenProxy.balanceOf(forwarder), 0);

        (bool isHolder) = remoraTokenProxy.getHolderStatus(forwarder).isHolder;
        assertEq(isHolder, false);

        // Distribute payouts for distributions 5 - 10
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // user1 must have 0 payout because it is forwarding to `forwarder`
        user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");

        // user2 must have 0 payout because it is forwarding to `forwarder`
        user2PayoutBalance = remoraTokenProxy.payoutBalance(user2);
        assertEq(user2PayoutBalance, 0, "Forwarding payout is not working as expected");

        (uint64 calculatedPayout) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        assertEq(calculatedPayout, payoutDistributionAmount * 5, "Forwarder did not receive payout for
        ↪  holder forwarding to him");

        // user2 gets forwarder removed as its forwardedAddress
        remoraTokenProxy.removePayoutForwardAddress(user2);

        //@audit => When this vulnerability is fixed, we expect finalCalculatedPayout to be equals than
        ↪  calculatedPayout!
        (uint64 finalCalculatedPayout) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        //@audit-issue => user2 causes the payout of user1 to be lost, which is 4x the payout lose by
        ↪  him
        assertEq(finalCalculatedPayout, 0, "Forwarder did not lose payout of holder");
```

```
    }
```

There is a second scenario similar to the one explained in the description section. In this other scenario, the forwarder is a non-holder, and, after a couple of distributions, the holder decides to remove or change the current forwarder to a different address, which leads to unclaimed payouts being lost.

- Run the next test to demonstrate the previous scenario

```
    function test_HolderLosesPayout_HolderRemovesForwarderWhoWasNeverAHolder() public {
        address user1 = users[0];
        address forwarder = users[1];

        uint256 amountToMint = 1;

        _whitelistAndMintTokensToUser(user1, amountToMint);

        remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);

        // fund total payout amount to funding wallet
        uint64 payoutDistributionAmount = 100e6;

        // Distribute payouts for the first 5 distributions
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // user1 must have 0 payout because it is forwarding to `forwarder`
        uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");

        (uint64 forwarderPayoutBalance) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        assertEq(forwarderPayoutBalance, payoutDistributionAmount * 5, "Forwarding payout is not working
        ↪  as expected");

        // forwarder attempts to claim all his payout while he is not a holder
        (bool isHolder) = remoraTokenProxy.getHolderStatus(forwarder).isHolder;
        assertEq(isHolder, false);
        // claiming reverts because forwarder is not a holder
        vm.prank(forwarder);
        vm.expectRevert();
        remoraTokenProxy.claimPayout();

        // user1 gets forwarder removed as its forwardedAddress
        remoraTokenProxy.removePayoutForwardAddress(user1);

        // validate forwarder and holder have lost the payouts for the past 5 distributions
        (forwarderPayoutBalance) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        assertEq(forwarderPayoutBalance, 0, "Forwarding payout is not working as expected");

        (uint256 finalForwarderPayoutBalance) = remoraTokenProxy.payoutBalance(forwarder);
        assertEq(finalForwarderPayoutBalance, 0, "Forwarding payout is not working as expected");

        // user1 must have 0 payout because it is forwarding to `forwarder`
        user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");
    }
```

Recommended Mitigation: On `_removePayoutForwardAddress()`, validate that the holderStatus of the forwardedAddress is 0, if it is not, don't call `deleteUser()`

```
function _removePayoutForwardAddress(
        HolderManagementStorage storage $,
        address holder,
        address forwardedAddress
    ) internal {
        if (forwardedAddress != address(0)) {
            ...
            if (
                balanceOf(forwardedAddress) == 0 &&
                payoutBalance(forwardedAddress) == 0
+              && $._holderStatus[forwardedAddress].calculatedPayout == 0
            ) deleteUser(forwardedHolder);
        }
```

Remora: Fixed in commit 7bd2691.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Attacker can make pledge on behalf of users if those users have approved `PledgeManager` to spend their tokens

Description: `PledgeManager` requires users to approve it to spend their tokens in order to make pledges. Users can do this by either:

1) using `IERC20Permit::permit` which enforces a nonce for the signer, deadline and domain separator
2) manually by calling `IERC20::approve`

If users use the manual method 2) and leave an open token approval, an attacker can call `PledgeManager::pledge` to make a pledge on their behalf since this function never enforces that `msg.sender == data.signer`.

Impact: Attacker can make pledges on behalf of innocent users which spends those users' tokens. It is common for users to have max approvals for protocols they use often, even though they don't intend to spend all their tokens with that protocol.

Recommended Mitigation: In `PledgeManager::pledge`, when not using `IERC20Permit::permit` enforce that `msg.sender == data.signer`:

```
        if (data.usePermit) {
            IERC20Permit(stablecoin).permit(
                signer,
                address(this),
                finalStablecoinAmount,
                block.timestamp + 300,
                data.permitV,
                data.permitR,
                data.permitS
            );
        }
+       else if(msg.sender != signer) revert MsgSenderNotSigner();
```

Alternatively always use `msg.sender` similar to how `PledgeManager::refundTokens` works.

Remora: Fixed in commit e3bda7c by always using `msg.sender` and also removed the permit method.

Cyfrin: Verified.

7.2.2 Accounting on `PaymentSettler` will be corrupted when changing `stablecoin` that is used to process payments

Description: The accounting on the `PaymentSettler` will be initialized based on the decimals of the initial stablecoin that is used at the beginning of the system. The system is capable of changing the stablecoin that is used for the payments, and, when the stablecoin is changed for a stablecoin with different decimals, all the existing accounting will be messed up because the new amounts will vary from the existing values on the system.

This problem was introduced on the last change when the `PaymentSettler` was introduced to the system. On the previous version, the system correctly handled the decimals of the internal accounting to the decimals of the active stablecoin used for payments.

For example, 100 USD of fees that were generated while the stablecoin had 6 decimals would be only 1 USD if the stablecoin were changed to a stablecoin with 8 decimals.

Impact: Accounting on `PaymentSettler` will be corrupted when changing `stablecoin` to different decimals.

Recommended Mitigation: See recommendation for C-2.

Remora: Fixed in commits a0b277f, ced21ba.

Cyfrin: Verified.

7.2.3 `PaymentSettler` can change `stablecoin` but `RemoraToken` can't resulting in corrupted state with DoS for core functions

Description: `RemoraToken` has a `stablecoin` member with a comment that indicates it must match `PaymentSettler`:

```
address public stablecoin; //make sure same stablecoin is used here that is used in payment settler
```

But in the updated code there is no way to update `RemoraToken::stablecoin`; previously `DividendManager` which `RemoraToken` inherits from had a `changeStablecoin` function but this was commented out with the introduction of `PaymentSettler`.

`PaymentSettler` has a `stablecoin` member and a function to change it:

```
address public stablecoin;

function changeStablecoin(address newStablecoin) external restricted {
    if (newStablecoin == address(0)) revert InvalidAddress();
    stablecoin = newStablecoin;
}
```

Impact: When `PaymentSettler` changes its `stablecoin` it will now be different to `RemoraToken::stablecoin` which can't be changed, corrupting the state causing key functions to revert.

Proof Of Concept:

```
function test_changeStablecoin_inconsistentState() external {
    address newStableCoin = address(new Stablecoin("USDC", "USDC", 0, 6));

    // change stablecoin on PaymentSettler
    paySettlerProxy.changeStablecoin(newStableCoin);
    assertEq(paySettlerProxy.stablecoin(), newStableCoin);

    // now inconsistent with RemoraToken
    assertEq(remoraTokenProxy.stablecoin(), address(stableCoin));
    assertNotEq(paySettlerProxy.stablecoin(), remoraTokenProxy.stablecoin());

    // no way to update RemoraToken::stablecoin
}
```

Recommended Mitigation: Enforce that `RemoraToken` and `PaymentSettler` must always refer to the same `stablecoin`. When implementing this consider our other findings where changing the `stablecoin` to one with different decimals corrupts protocol accounting.

The simplest solution may be to remove `stablecoin` from `RemoraToken` completely and have `PaymentSettler` perform all the necessary transfers.

Remora: Fixed in commit ced21ba by removing `stablecoin` from `RemoraToken`, moving the transfer fee logic into `PaymentSettler` and having `RemoraToken` call `PaymentSettler::settleTransferFee`.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 `PledgeManager::refundTokens` doesn't decrement `tokensSold` when pledge hasn't concluded, preventing pledge from reaching its funding goal

Description: `PledgeManager::refundTokens` doesn't decrement `tokensSold` when pledge hasn't concluded.

Impact: The pledge will be prevented from reaching its funding goal since the refunded tokens can't be purchased by other users.

Recommended Mitigation: `PledgeManager::refundTokens` should always decrement `tokensSold`:

```
        if (
            !pledgeRoundConcluded &&
            SafeCast.toUint32(block.timestamp) < deadline
        ) {
            refundAmount -= (refundAmount * earlySellPenalty) / 1e6;
            _propertyToken.adminTransferFrom(
                signer,
                holderWallet,
                numTokens,
                false,
                false
            );
            emit TokensUnPledged(signer, numTokens);
        } else {
            fee = (userPay.fee / userPay.tokensBought) * numTokens;
            _propertyToken.burnFrom(signer, numTokens);
            emit TokensRefunded(signer, numTokens);
-           tokensSold -= numTokens;
        }

+      tokensSold -= numTokens;
```

Remora: Fixed in commit 6be4660.

Cyfrin: Verified.

7.3.2 `TokenBank::withdrawFunds` resets `memory` not `storage` fee and sale amounts allowing multiple withdraws for the same token

Description: `TokenBank::withdrawFunds` resets `memory` not `storage` fee and sale amounts allowing multiple withdraws for the same token:

```
    function withdrawFunds(
        address tokenAddress,
        bool fee
    ) public nonReentrant restricted {
        TokenData memory curData = tokenData[tokenAddress];
        address to;
        uint64 amount;
        if (fee) {
            to = custodialWallet;
            amount = curData.feeAmount;
            curData.feeAmount = 0; // @audit resets memory not storage
        } else {
            to = curData.withdrawTo;
            amount = curData.saleAmount;
            curData.saleAmount = 0; // @audit resets memory not storage
        }
        if (amount != 0) IERC20(stablecoin).transfer(to, amount);

        if (fee) emit FeesClaimed(tokenAddress, amount);
        else emit FundsClaimed(tokenAddress, amount);
    }
```

Impact: The admin can make multiple fee and sale amount withdraws for the same token address. This will work as long as there are sufficient fee and sale tokens from other sales.

Recommended Mitigation: Reset `storage` not `memory`:

```
    function withdrawFunds(
        address tokenAddress,
        bool fee
    ) public nonReentrant restricted {
        TokenData memory curData = tokenData[tokenAddress];
        address to;
        uint64 amount;
        if (fee) {
            to = custodialWallet;
            amount = curData.feeAmount;
-           curData.feeAmount = 0;
+           tokenData[tokenAddress].feeAmount = 0;
        } else {
            to = curData.withdrawTo;
            amount = curData.saleAmount;
-           curData.saleAmount = 0;
+           tokenData[tokenAddress].saleAmount = 0;
        }
        if (amount != 0) IERC20(stablecoin).transfer(to, amount);

        if (fee) emit FeesClaimed(tokenAddress, amount);
        else emit FundsClaimed(tokenAddress, amount);
    }
```

Remora: Fixed in commit 571bfe4.

Cyfrin: Verified.

7.3.3 Fee should be calculated after first purchase discount is applied in `TokenBank::buy` to prevent overcharging users

Description: `TokenBank::buy` calculates the total purchase amount as being composed of:

- `stablecoinValue` : value of the tokens
- `feeValue` : fee calculated off `stablecoinValue`

```
        uint64 stablecoinValue = amount * curData.pricePerToken;
        uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;
```

Afterwards if this is the user's first purchase, they receive a discount on the `stablecoinValue`:

```
        IReferralManager refManager = IReferralManager(referralManager);
        bool firstPurchase = refManager.isFirstPurchase(to);
        if (firstPurchase) stablecoinValue -= refManager.referDiscount();
```

However the fee is not updated so was still calculated from the initial higher `stablecoinValue` amount.

Impact: Users will pay higher fees than they should when receiving the first purchase discount.

Recommended Mitigation: Only calculate `feeValue` once the discount has been applied:

```
        address to = msg.sender;
        uint64 stablecoinValue = amount * curData.pricePerToken;
-       uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;

        IReferralManager refManager = IReferralManager(referralManager);
        bool firstPurchase = refManager.isFirstPurchase(to);
        if (firstPurchase) stablecoinValue -= refManager.referDiscount();

+       uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;
        curData.saleAmount += stablecoinValue;
        curData.feeAmount += feeValue;
```

Remora: Fixed in commits 4aea246, 5510920 - changed the way fee calculation works for regulatory reasons.

Cyfrin: Verified.

7.3.4 Buyers can pledge for tokens without having signed all documents that are required to be signed

Description: When buyers pledge() during the pledgeRound, it is verified that they have signed all the documents that are required to be signed. If at least one document is not signed, instead of reverting the tx, the execution will verify a signature on behalf of the signer, and, if this signature is legit, the execution continues.

```
    function _verifyDocumentSignature(
        PledgeData calldata data,
        address signer
    ) internal {
        (bool res, ) = IRemoraRWAToken(propertyToken).hasSignedDocs(signer);
        if (!res)
            IRemoraRWAToken(propertyToken).verifySignature(
                signer,
                data.docHash,
                data.signature
            );
    }
```

The problem is that this implementation allows buyers to bypass the requirement to have signed all the documents by signing only one. For example: There are 3 documents that need to be signed, and the user has not signed any of them. The user calls pledge() and provides the signature's data to sign 1 document. Here is what will happen:

PropertyToken::hasSignedDocs() will return false because the user has not signed any of the 3 documents

```
    function hasSignedDocs(address signer) public view returns (bool, bytes32) {
        ...

        for (uint256 i = 0; i < numDocs; ++i) {
            bytes32 docHash = $._docHashes[i];
//@audit => If one document that needs signature is not signed, returns false
            if (
                $._documents[docHash].needSignature &&
                $._signatureRecords[signer][docHash] == 0
            ) return (false, docHash);
        }

//@audit => returns true only if all documents that requires signature are signed
        return (true, 0x0);
    }
```

PropertyToken::verifySignature() won't revert because it will sign one of the 3 documents

```
    function verifySignature(
        address signer,
        bytes32 docHash,
        bytes memory signature
    ) external returns (bool result) {
        ...
        if (signer.code.length == 0) {
            //signer is EOA
```

```
      (address returnedSigner, , ) = ECDSA.tryRecover(digest, signature);
      result = returnedSigner == signer;
    } else {
      //signer is SCA
      (bool success, bytes memory ret) = signer.staticcall(
        ...
      );
      result = (success && ret.length == 32 && bytes4(ret) == MAGICVALUE);
    }

    if (!result) revert InvalidSignature();
    if ($._signatureRecords[signer][docHash] == 0) {
      ...
    }
//@audit => if the verification of the provided signature succeeds, execution continues
  }
```

The problem is that the execution will continue even though the user has only signed 1 of the 3 documents that have to be signed because (as previously explained), _verifyDocumentSignature() will be bypassed to only enforce one signature, and, when transferring from the holderWallet to the signer, the `checkTC` is set as false.

PledgeManager::pledge()

```
  function pledge(PledgeData calldata data) external nonReentrant {
    ...

    _verifyDocumentSignature(data, signer);

    ...

//@audit => checkTC is set as false
    //this address should be whitelisted in property token
    IRemoraRWAToken(propertyToken).adminTransferFrom(
      holderWallet,
      signer,
      numTokens,
      false,  // <====> checkTC //
      true
    );
    ...
  }
```

RemoraToken::adminTransferFrom()

```
  function adminTransferFrom(
    address from,
    address to,
    uint256 value,
    bool checkTC,
    bool enforceLock
  ) external restricted returns (bool) {
    ...
//@audit => checkTC as false effectively bypass the verification of TC to be signed
    (bool res, ) = hasSignedDocs(to);
    if (checkTC && !res) revert TermsAndConditionsNotSigned(to);

    ...
  }
```

Impact: Buyers can purchase tokens even though they have not signed all the documents that have to be signed

Recommended Mitigation: Revert the execution if the call to `PropertyToken.hasSignedDocs()` returns false.

```
function _verifyDocumentSignature(
        PledgeData calldata data,
        address signer
    ) internal {

        (bool res, ) = IRemoraRWAToken(propertyToken).hasSignedDocs(signer);

-      if (!res)
-          IRemoraRWAToken(propertyToken).verifySignature(
-             signer,
-             data.docHash,
-              data.signature
-         );

+     if (!res) revert NotAllDocumentsAreSigned();

    }
```

Remora: Fixed in commit 5510920.

Cyfrin: Verified.

7.3.5 Hardcoding deadline for `permit()` will mess up the `structHash` leading to an invalid signature

Description: Functions allowing users to grant ERC20 approvals via Permit incorrectly hardcode the deadline passed to permit().

```
function pledge(PledgeData calldata data) external nonReentrant {
        ...

        //5 minute deadline
        if (data.usePermit) {
            IERC20Permit(stablecoin).permit(
                signer,
                address(this),
                finalStablecoinAmount,
//@audit-issue => hardcoded deadline
                block.timestamp + 300,
                data.permitV,
                data.permitR,
                data.permitS
            );
        }
        ...
}
```

The problem is that the recovered signature will be invalid because the deadline is a parameter of the structHash, and, if the signed deadline differs even by 1 second, that hashed structHash will be different than the one that was actually signed by the user.

```
function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        if (block.timestamp > deadline) {
            revert ERC2612ExpiredSignature(deadline);
        }

//@audit-issue => A hardcoded deadline will mess up the structHash
        bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
        ↪  _useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

//@audit-issue => A different hash than the actual hash signed by the signer will recover a != signer
↪  (even address(0)
        address signer = ECDSA.recover(hash, v, r, s);
        if (signer != owner) {
            revert ERC2612InvalidSigner(signer, owner);
        }

        _approve(owner, spender, value);
    }
```

Impact: Functions like `pledge()`, `refundToken()` that allows to authorize ERC20Tokens via permit won't work because the structHash will be different than the actual hash signed by the user.

Recommended Mitigation: Receive the deadline as a parameter instead of hardcoding it.

```
function pledge(PledgeData calldata data) external nonReentrant {
        ...

        //5 minute deadline
        if (data.usePermit) {
            IERC20Permit(stablecoin).permit(
                signer,
                address(this),
                finalStablecoinAmount,
-               block.timestamp + 300,
+               data.deadline,
                data.permitV,
                data.permitR,
                data.permitS
            );
        }
        ...
}
```

Remora: Fixed in 5510920 by removing the permit functionality.

Cyfrin: Verified.

7.3.6 `DocumentManager::hasSignedDocs` incorrectly returns `true` when there are no documents to sign

Description: `DocumentManager::hasSignedDocs` incorrectly returns `true` when there are no documents to sign:

```
function hasSignedDocs(address signer) public view returns (bool, bytes32) {
    DocumentStorage storage $ = _getDocumentStorage();
    uint256 numDocs = $._docHashes.length;

    // @audit when numDocs = 0, the `for` loop is bypassed
    // skipping to the `return (true, 0x0);` statement
    for (uint256 i = 0; i < numDocs; ++i) {
        bytes32 docHash = $._docHashes[i];
        if (
            $._documents[docHash].needSignature &&
            $._signatureRecords[signer][docHash] == 0
        ) return (false, docHash);
    }
    return (true, 0x0);
}
```

Impact: Upstream contracts incorrectly assume users have signed docs and allow user actions which should be prohibited.

Proof Of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import {DocumentManager} from "../../../contracts/RWAToken/DocumentManager.sol";

import {UnitTestBase} from "../UnitTestBase.sol";

contract DocumentManagerTest is UnitTestBase, DocumentManager {
    function setUp() public override {
        UnitTestBase.setUp();
        initialize();
    }

    function initialize() public initializer {
        __RemoraDocuments_init("NAME", "VERSION");
    }

    // this is incorrect and should be changed once bug is fixed
    function test_hasSignedDocs_TrueWhenNoDocs() external {
        // verify no docs
        assertEq(_getDocumentStorage()._docHashes.length, 0);

        // hasSignedDocs returns true even though no docs to sign
        (bool hasSigned, ) = hasSignedDocs(address(0x1337));
        assertTrue(hasSigned);
    }
}
```

Recommended Mitigation: When no docs exist, it is impossible for users to have signed them. Hence in this case `DocumentManager::hasSignedDocs` should either revert with a specific error such as `EmptyDocument` or `return (false, 0x0)`.

Remora: Fixed in commit 7454e55.

Cyfrin: Verified.

7.3.7 Don't add duplicate `documentHash` to `DocumentManager::DocumentStorage::_docHashes` when overwriting via `_setDocument` as this causes panic revert when calling `_removeDocument`

Description: `DocumentManager::_setDocument` intentionally allows overwriting but when overwriting it adds an additional duplicate `documentHash` to `_docHashes`:

```
function _setDocument(
    bytes32 documentName,
    string calldata uri,
    bytes32 documentHash,
    bool needSignature
) internal {
    DocumentStorage storage $ = _getDocumentStorage();
    $._documents[documentHash] = DocData({
        needSignature: needSignature,
        docURI: uri,
        docName: documentName,
        timestamp: SafeCast.toUint32(block.timestamp)
    });

    // @audit duplicate if overwriting
    $._docHashes.push(documentHash);
    emit DocumentUpdated(documentName, uri, documentHash);
}
```

Impact: Once the hash has been duplicated in `_docHashes`, it is impossible to remove the document by calling `_removeDocument` as it panic reverts.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import {DocumentManager} from "../../../contracts/RWAToken/DocumentManager.sol";

import {UnitTestBase} from "../UnitTestBase.sol";

contract DocumentManagerTest is UnitTestBase, DocumentManager {
    function setUp() public override {
        UnitTestBase.setUp();
        initialize();
    }

    function initialize() public initializer {
        __RemoraDocuments_init("NAME", "VERSION");
    }

    // this is incorrect and should be changed once bug is fixed
    function test_setDocumentOverwrite(string calldata uri) external {
        bytes32 docName = "0x01234";
        bytes32 docHash = "0x5555";
        bool needSignature = true;

        // add the document
        _setDocument(docName, uri, docHash, needSignature);

        // verify its hash has been added to `_docHashes`
        DocumentStorage storage $ = _getDocumentStorage();
        assertEq($._docHashes.length, 1);
        assertEq($._docHashes[0], docHash);

        // ovewrite it
        _setDocument(docName, uri, docHash, needSignature);
        // this duplicates the hash in `_docHashes`
        assertEq($._docHashes.length, 2);
        assertEq($._docHashes[0], docHash);
        assertEq($._docHashes[1], docHash);

        // now attempt to remove it, reverts with
        // panic: array out-of-bounds access
        _removeDocument(docHash);
    }
}
```

Recommended Mitigation: In `DocumentManager::_setDocument` check if the `documentHash` already exists and if so, don't add it to `_docHashes`.

Alternatively use `EnumerableSet` for `DocumentManager::DocumentStorage::_docHashes` which doesn't allow duplicates.

In `_removeDocument` break out of the loop once the element has been deleted:

```
        uint256 dHLen = $._docHashes.length;
        for (uint i = 0; i < dHLen; ++i) {
            if ($._docHashes[i] == documentHash) {
                $._docHashes[i] = $._docHashes[dHLen - 1];
                $._docHashes.pop();
+               break;
            }
        }
```

Remora: Fixed in commit 1218d18.

Cyfrin: Verified.

7.3.8 Check return value when calling `Allowlist::exchangeAllowed` and `RemoraToken::_exchangeAllowed` to prevent unauthorized transfers

Description: `Allowlist::exchangeAllowed` will revert if the users are not allowed but when the users are both allowed, it will return a boolean determined by whether the `domestic` field of both users match:

```
function exchangeAllowed(
    address from,
    address to
) external view returns (bool) {
    HolderInfo memory fromUser = _allowed[from];
    HolderInfo memory toUser = _allowed[to];
    if (from != address(0) && !fromUser.allowed)
        revert UserNotRegistered(from);
    if (to != address(0) && !toUser.allowed) revert UserNotRegistered(to);
    return fromUser.domestic == toUser.domestic; //logic to be edited later on
}
```

But `RemoraToken::adminTransferFrom` doesn't check the boolean return when calling `Allowlist::exchangeAllowed`:

```
function adminTransferFrom(
    address from,
    address to,
    uint256 value,
    bool checkTC,
    bool enforceLock
) external restricted returns (bool) {
    // @audit boolean return not checked
    IAllowlist(allowlist).exchangeAllowed(from, to);
```

Similary `RemoraToken::_exchangeAllowed` returns the boolean output of `Allowlist::exchangeAllowed`, but this is never checked in `RemoraToken::transfer`, `transferFrom`.

Impact: Transfers are allowed even when `Allowlist::exchangeAllowed` returns `false`.

Recommended Mitigation: Check the boolean return of `Allowlist::exchangeAllowed`, `RemoraToken::_exchangeAllowed` and only allow transfers when they return `true`.

Alternatively change `Allowlist::exchangeAllowed` and `RemoraToken::_exchangeAllowed` to not return anything but to always revert.

Remora: Fixed in commit 13cf261.

Cyfrin: Verified.

7.3.9 `DividendManager::distributePayout` will always revert after 255 payouts, preventing any future payout distributions

Description: `DividendManager::HolderManagementStorage::_currentPayoutIndex` is declared as `uint8`:

```
/// @dev The current index that is yet to be paid out.
uint8 _currentPayoutIndex;
```

`_currentPayoutIndex` is incremented every time `DividendManager::distributePayout` is called:

```
$._payouts[$._currentPayoutIndex++] = PayoutInfo({
    amount: payoutAmount,
    totalSupply: SafeCast.toUint128(totalSupply())
});
```

Impact: The maximum value of `uint8` is 255 so `DividendManager::distributePayout` can only be called 255 times; any further calls will always revert meaning no more payout distributions are possible.

Recommended Mitigation: If requiring more than 255 payout distributions:

- use a larger size to store `DividendManager::HolderManagementStorage::_currentPayoutIndex`
- change the `uint8` key in this mapping to match the larger size:

```
mapping(address => mapping(uint8 => TokenBalanceChange)) _balanceHistory;
```

- change the `uint256` key in this mapping to match:

```
mapping(uint256 => PayoutInfo) _payouts;
```

Consider using named mappings to explicitly show that these indexes all refer to the same entity, the payout index.

In Solidity a storage slot is 256 bits and an address uses 160 bits. Examining the relevant storage layout of `struct HolderManagementStorage` shows that `_currentPayoutIndex` could be declared as large as `uint56` without using any additional storage slots:

```
IERC20 _stablecoin; // 160 bits
uint8 _stablecoinDecimals; // 8 bits
uint32 _payoutFee; // 32 bits
// 200 bits have been used so 56 bits available in the current storage slot
// _currentPayoutIndex could be declared as large as `uint56` with no
// extra storage requirements
uint8 _currentPayoutIndex;
```

Remora: Fixed in commit f929ff1 by increasing `_currentPayoutIndex` to `uint16` which will be sufficient.

Cyfrin: Verified.

7.3.10 Payout distributions to Remora token holders are diluted by initial token owner mint

Description: `RemoraToken::initialize` takes a parameter `_initialSupply` and mints this to `tokenOwner`:

```
_mint(tokenOwner, _initialSupply * 10 ** decimals());
```

`DividendManager::distributePayout` records the payout amount together with the current total supply:

```
$._payouts[$._currentPayoutIndex++] = PayoutInfo({
    amount: payoutAmount,
    totalSupply: SafeCast.toUint128(totalSupply())
});
```

`DividendManager::payoutBalance` calculates user payout amount dividing by the recorded total supply:

```
payoutAmount +=
    (curEntry.tokenBalance * pInfo.amount) /
    pInfo.totalSupply;
```

Impact: Payout distributions for Remora token holders are diluted by the initial token owner mint. If the token owner is minted a significant percentage of the supply, normal users will have their rewards significantly diluted.

Recommended Mitigation: Exclude the token owner's tokens from the payout distribution by subtracting them from the total supply. The token owner could get around this though by transferring their tokens to other addresses the control.

Another way is to have a variable in `RemoraToken` which only the admin can set that records the amount of tokens the owner holds, and this gets subtracted from the total supply for purposes of payout distribution. The owner would need to update this variable so there is still trust required in the owner.

Alternatively the finding can be acknowledged as long as the protocol is aware that users will have their rewards diluted by the owner's holdings.

Remora: The plan is to send all of the initial token mint to the `TokenBank`; the intended `tokenOwner` is actually the token bank. These tokens then go on sale for the investors so the protocol admin won't hold any tokens themselves; the tokens will either be in the token bank awaiting sale or with the investors who bought them.

Unsold tokens held by `TokenBank` are owned by the protocol; we will use the forwarding mechanism to claim our share of the payout distributions for unsold tokens still held by the token bank.

7.3.11 Burning ALL PropertyTokens of a frozen holder results in the holder losing the payouts distribution while he was frozen

Description: Burning PropertyTokens from frozen holders has an edge case when all the tokens owned by a frozen holder get burned. This is how burning PropertyTokens impacts the payouts for distributions of frozen holders:

- If not all the PropertyTokens owned by a frozen holder are burned, the holder can still have access to the payouts for the distributions while he was frozen.
- If all the PropertyTokens owned by a frozen holder are burned, the holder would lose the payouts for the distributions while he was frozen.

The discrepancy in the behavior when burning PropertyTokens of Frozen Holders demonstrates an edge case that can result in frozen holders losing payouts.

For example - (Assume holders were frozen at the same index and got their tokens burned at the same index too):

- UserA has 1 PropertyToken and is frozen
- UserB has 2 PropertyTokens and is frozen too
- UserA and B get burned each a PropertyToken.
  - UserA loses the payouts distributions while he was frozen, whilst UserB still has access to the payouts for the 2 PropertyTokens that he owned during those distributions.

Impact: Burning ALL the PropertyTokens owned by a frozen holder causes the holder to lose the payouts for the distributions while he was frozen.

Proof of Concept: Run the following test to reproduce the issue described on the Description section

```
    function test_frozenHolderLosesPayoutsBecauseItsTokensGotBurnt() public {
        address user1 = users[0];
        uint256 amountToMint = 2;

        // fund total payout amount to funding wallet
        uint64 payoutDistributionAmount = 100e6;
        // Distribute payouts for the first 5 distributions
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        _whitelistAndMintTokensToUser(user1, amountToMint);

        vm.prank(user1);
        remoraTokenProxy.approve(address(this), amountToMint);

        paySettlerProxy.initiateBurning(address(remoraTokenProxy), address(this), 0);

        // verify increased current payout index twice
        assertEq(remoraTokenProxy.getCurrentPayoutIndex(), 5);

        // Distribute payouts for distributions 5 - 10
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // Freze user 2 at distributionIndex 10
        remoraTokenProxy.freezeHolder(user1);

        uint256 user1HolderBalanceAfterBeingFrozen = remoraTokenProxy.payoutBalance(user1);

        // Distribute payouts for distributions 10 - 15
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // verify increased current payout index twice
        assertEq(remoraTokenProxy.getCurrentPayoutIndex(), 15);

        assertEq(user1HolderBalanceAfterBeingFrozen, remoraTokenProxy.payoutBalance(user1), "Frozen
        ↪  holder earned payout while being frozen");

        uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        // 5 distributions of 100e6 all for user1
        assertEq(user1PayoutBalance, 500e6, "User1 Payout is incorrect");

        vm.prank(user1);
        remoraTokenProxy.claimPayout();
        assertEq(stableCoin.balanceOf(user1), user1PayoutBalance, "Error while claiming payout");
        assertEq(remoraTokenProxy.payoutBalance(user1), 0);

        //@audit-info => When the holder is unfrozen, he gets access to the payouts for al the
        ↪  distributions while he was frozen
        uint256 snapshotBeforeUnfreezing = vm.snapshotState();
        // unfreeze user1 and validate it gets access to all the distributions while it was frozen
        remoraTokenProxy.unFreezeHolder(user1);
        // After being unfrozen there were pending only 5 distributions of 100e6 all for user1
        assertEq(remoraTokenProxy.payoutBalance(user1), 500e6, "User1 Payout is incorrect");
        vm.revertToState(snapshotBeforeUnfreezing);

        uint256 snapshotBeforeBurningAllFrozenHolderTokens = vm.snapshotState();
        //@audit-info => Burning ALL PropertyTokens from a holder while is frozen results in the holder
        // NOT being able to access the payouts of the distributions while he was frozen
        remoraTokenProxy.burnFrom(user1, 2, false);
        assertEq(remoraTokenProxy.balanceOf(user1), 0);

        assertEq(remoraTokenProxy.payoutBalance(user1), 0);
        // unfreeze user1 and validate it loses the payouts of the distributions while it was frozen
        remoraTokenProxy.unFreezeHolder(user1);
        assertEq(remoraTokenProxy.payoutBalance(user1), 0);
        vm.revertToState(snapshotBeforeBurningAllFrozenHolderTokens);

        //@audit-info => Burning NOT ALL PropertyTokens from a holder while is frozen results in the
        ↪   holder
        // being able to access the payouts of the distributions while he was frozen
        assertEq(remoraTokenProxy.payoutBalance(user1), 0);
        remoraTokenProxy.burnFrom(user1, 1, false);
        assertEq(remoraTokenProxy.balanceOf(user1), 1);
        remoraTokenProxy.unFreezeHolder(user1);
        assertEq(remoraTokenProxy.payoutBalance(user1), 500e6);
    }
```

Recommended Mitigation: The least disruptive mitigation to prevent this issue is to add a check on the `burnFrom` to revert the tx if the account has been left without more propertyTokens and the account is frozen

```
function burnFrom(
        address account,
        uint256 value
    ) external restricted whenBurnable {
        _spendAllowance(account, _msgSender(), value);
        _burn(account, value);
+      if(balanceOf(account) == 0 && isHolderFrozen(account)) revert UserIsFrozen(account);
    }
```

Alternatively, similar to the burn(), revert if the account if frozen.

```
    function burnFrom(
        address account,
        uint256 value
    ) external restricted whenBurnable {
+       if (isHolderFrozen(account)) revert UserIsFrozen(account);
        _spendAllowance(account, _msgSender(), value);
        _burn(account, value);
    }
```

Remora: Fixed in commit 6008aec by preventing burning on frozen users.

Cyfrin: Verified.

7.3.12 Overriding fees can't be switched back once set

Description: A fee is charged when buying a PropertyToken via the TokenBank. The system allows for charging a personalized fee on a per-token basis or a baseFee for all tokens. If the variable `overrideFees` is set as true, the TokenBank will charge the baseFee that is charged to all tokens instead of charging the configured per-token fee.

The problem is that once the `overrideFees` variable is set to true, it is not possible to set it back to false to allow charging fees on a per-token basis.

```
    function setBaseFee(
        bool updateFee,
        bool overrideFee,
        uint32 newFee
    ) external restricted {
        ...
//@audit => enters only when it is true.
//@audit => So, once set to true, it can't be changed back to false
        if (overrideFee) {
            overrideFees = overrideFee;
            emit FeeOverride(overrideFee);
        }
    }
```

Impact: Not possible to charge fees on a per-token basis once it has been configured to charge the baseFee.

Recommended Mitigation: Directly update `overrideFees` with the value of the parameter `overrideFee`.

```
    function setBaseFee(
        bool updateFee,
        bool overrideFee,
        uint32 newFee
    ) external restricted {
        ...
-       if (overrideFee) {
            overrideFees = overrideFee;
            emit FeeOverride(overrideFee);
-       }
    }
```

Remora: Fixed in commit c38787f.

Cyfrin: Verified.

7.3.13 Forwarders can lose payouts of the holders forwarding to them

Description: A holder can have a designated forwarder who will accumulate the payouts earned by the holder. The vulnerability identified in this report is when the designated forwarder has no pending payouts to claim and zeroes out his PropertyToken's balance, and, time passes (while forwarder is still accumulating the payouts of the holder) and then it becomes a holder again, but on the same distributionIndex the forwarder zeroes out his balance again.

- The combination of these steps leads the contract's state to an inconsistent state that ends up causing the unclaimed accumulated payouts to be lost.

The steps that lead the contracts to the inconsistent state are:

1. forwarder has balance, it is a holder
2. holder sets a forwarder
3. payouts for holder are computed and credited to forwarder
4. forwarder claims payouts, and gets computed all pending payouts
    - At this point, payoutBalance of forwarder would be 0
5. forwarder zeros out his balance, and gets removed the `isHolder` status (no longer a holder)
6. distributions passes
7. payouts for holder are computed and credited to the forwarder
8. forwarder gets a balance and regains holder status
    - lastPayoutIndexCalculated = currentPayoutIndex
9. forwarder zeros out his balance again
    - payoutBalance(forwarder) is called but returns 0 because `lastPayoutIndexCalculated == currentPayoutIndex`
    - so, balance == 0 and payoutBalance() returns 0, deleteUser(forwarder) gets called
10. As a result of step 10, the payouts earned by the holder get lost

Impact: Forwarders can lose holders' payouts

Proof of Concept: Run the following test to reproduce the scenario described on the Description section.

```
    function test_forwarderLosesHolderPayouts() public {
        address user1 = users[0];
        address forwarder = users[1];
        address anotherUser = users[2];

        uint256 amountToMint = 1;

        _whitelistAndMintTokensToUser(user1, amountToMint);
        _whitelistAndMintTokensToUser(forwarder, amountToMint);
        // Only whitelist and allow anotherUser
        _whitelistAndMintTokensToUser(anotherUser, 0);

        remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);

        // fund total payout amount to funding wallet
        uint64 payoutDistributionAmount = 100e6;

        // Distribute payouts for the first 5 distributions
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // user1 must have 0 payout because it is forwarding to `forwarder`
        uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");

        //forwarder must have the full payout for the 5 distributions because user1 is forwarding to him
        uint256 forwarderPayoutBalance = remoraTokenProxy.payoutBalance(forwarder);
        assertEq(forwarderPayoutBalance, payoutDistributionAmount * 5, "Forwarding payout is not working
        ↪  as expected");

        // forwarder claims all his payout
        vm.startPrank(forwarder);
        remoraTokenProxy.claimPayout();
        assertEq(stableCoin.balanceOf(forwarder), forwarderPayoutBalance);

        // forwarder zeros out his PropertyToken's balance
        remoraTokenProxy.transfer(anotherUser, remoraTokenProxy.balanceOf(forwarder));
        vm.stopPrank();

        assertEq(remoraTokenProxy.balanceOf(forwarder), 0);

        (bool isHolder) = remoraTokenProxy.getHolderStatus(forwarder).isHolder;
        assertEq(isHolder, false);

        // Distribute payouts for distributions 5 - 10
        for(uint i = 1; i <= 5; i++) {
            _fundPayoutToPaymentSettler(payoutDistributionAmount);
        }

        // user1 must have 0 payout because it is forwarding to `forwarder`
        user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0, "Forwarding payout is not working as expected");

        (uint64 calculatedPayout) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        assertEq(calculatedPayout, (payoutDistributionAmount * 5) / 2, "Forwarder did not receive payout
        ↪  for holder forwarding to him");

        vm.startPrank(anotherUser);
        // forwarder becomes a holder again
        remoraTokenProxy.transfer(forwarder, remoraTokenProxy.balanceOf(anotherUser));
        vm.stopPrank();
        vm.startPrank(forwarder);
        // forwarder zeroues out his balance again
        remoraTokenProxy.transfer(anotherUser, remoraTokenProxy.balanceOf(forwarder));
        vm.stopPrank();

        //@audit => When this vulnerability is fixed, we expect finalCalculatedPayout to be equals than
        ↪   calculatedPayout!
        (uint64 finalCalculatedPayout) = remoraTokenProxy.getHolderStatus(forwarder).calculatedPayout;
        assertEq(finalCalculatedPayout, 0, "Forwarder did not lose payout of holder");
    }
```

Recommended Mitigation: In `_updateHolders()`, check `holderStatus.calculatedPayout` to be 0, if it is not 0, don't call deleteUser()

```
function _updateHolders(address from, address to) internal {
        ...

        if (from != address(0)) {
            ...
            HolderStatus storage fromHolderStatus = $._holderStatus[from];
-           if (fromBalance == 0 && payoutBalance(from) == 0) {
+           if (fromBalance == 0 && payoutBalance(from) == 0 && fromHolderStatus.calculatedPayout == 0)
↪   {
                deleteUser(fromHolderStatus);
                return;
            }
            ...
            }
        }
    }
```

Remora: Fixed in commit d379e89.

Cyfrin: Verified.

7.3.14 Pledge can't successfully complete unless `RemoraToken` is paused

Description: When the funding goal has been reached, `PledgeManager::checkPledgeStatus` calls `RemoraToken::unpause`:

```
function checkPledgeStatus() public returns(bool pledgeNowConcluded) {
    if (pledgeRoundConcluded) return true;

    uint32 curTime = SafeCast.toUint32(block.timestamp);
    if (tokensSold >= fundingGoal) {
        pledgeRoundConcluded = true;
        IRemoraRWAToken(propertyToken).unpause();
        emit PledgeHasConcluded(curTime);
        return true;
```

But if `RemoraToken` is not paused, this reverts since `PausableUpgradeable::_unpause` has the `whenPaused` modifier.

Impact: Pledge can't successfully complete unless `RemoraToken` is paused.

Proof of Concept:

```
function test_pledge(uint256 userIndex, uint32 numTokensToBuy) external {
    address user = _getRandomUser(userIndex);
    numTokensToBuy = uint32(bound(numTokensToBuy, 1, DEFAULT_FUNDING_GOAL));

    // fund buyer with stablecoin
    (uint256 finalStablecoinAmount, uint256 fee)
        = pledgeManager.getCost(numTokensToBuy);
    stableCoin.transfer(user, finalStablecoinAmount);

    // fund this with remora tokens
    remoraTokenProxy.mint(address(this), numTokensToBuy);
    assertEq(remoraTokenProxy.balanceOf(address(this)), numTokensToBuy);

    // allow PledgeManager to spend our remora tokens
    remoraTokenProxy.approve(address(pledgeManager), numTokensToBuy);

    PledgeManagerState memory pre = _getState(address(this), user);

    remoraTokenProxy.pause();

    vm.startPrank(user);
    stableCoin.approve(address(pledgeManager), finalStablecoinAmount);
    pledgeManager.pledge(numTokensToBuy, bytes32(0x0), bytes(""));
    vm.stopPrank();

    PledgeManagerState memory post = _getState(address(this), user);

    // verify remora token balances
    assertEq(post.holderRemoraBal, pre.holderRemoraBal - numTokensToBuy);
    assertEq(post.buyerRemoraBal, pre.buyerRemoraBal + numTokensToBuy);

    // verify stablecoin balances
    assertEq(post.pledgeMgrStableBal, pre.pledgeMgrStableBal + finalStablecoinAmount);
    assertEq(post.buyerStableBal, pre.buyerStableBal - finalStablecoinAmount);

    // verify PledgeManager storage
    assertEq(post.pledgeMgrTokensSold, pre.pledgeMgrTokensSold + numTokensToBuy);
    assertEq(post.pledgeMgrTotalFee, pre.pledgeMgrTotalFee + fee);
    assertEq(post.pledgeMgrBuyerFee, pre.pledgeMgrBuyerFee + fee);
    assertEq(post.pledgeMgrTokensBought, pre.pledgeMgrTokensBought + numTokensToBuy);
}
```

Recommended Mitigation: The `RemoraToken` contract should remain in the `paused` state until the pledge completes, though this may not be convenient. Alternatively change `PledgeManager::checkPledgeStatus` to only unpause `RemoraToken` if it is paused.

Remora: Fixed in commit dddde02.

Cyfrin: Verified.

7.3.15 `RemoraToken` transfers are bricked when `from` is not whitelisted, has sufficient tokens to transfer but no tokens locked

Description: `RemoraToken::adminTransferFrom`, `transfer` and `transferFrom` always check if `from` is on the whitelist and if not, call `_unlockTokens`:

```
    bool fromWL = whitelist[from];
    bool toWL = whitelist[to];

    if (!fromWL) _unlockTokens(from, value, false);
    if (!toWL) _lockTokens(to, value);
```

But a scenario can occur where:

1) `from` has nothing to unlock

2) `from` has tokens to fulfill the transfer

In this case transfer would be bricked. Another scenario to consider is when:

1) `from` has 10 tokens
2) only 5 of those tokens are locked
3) `from` is attempting to transfer 10 tokens

In this case the call to `_unlockTokens` should have `amount = 5` since `from` only needs to unlock 5 tokens in order to send the 10 total.

But with the current code the call to `_unlockTokens` will have `amount = 10` (since it just uses the transfer amount) which makes no sense and causes a revert.

Impact: `RemoraToken` transfers are bricked when `from` is not whitelisted, has sufficient tokens to transfer but no tokens locked since the call to `_unlockTokens` will revert.

Proof Of Concept:

```
function test_transferBricked_fromNotWhitelisted_ButHasTokensToTransfer() external {
    address from = users[0];
    address to = users[1];
    uint256 amountToTransfer = 1;

    // fund `from` with remora tokens
    remoraTokenProxy.mint(from, amountToTransfer);
    assertEq(remoraTokenProxy.balanceOf(from), amountToTransfer);

    // remove `from` from whitelist
    remoraTokenProxy.removeFromWhitelist(from);

    // set lock time
    remoraTokenProxy.setLockUpTime(3600);

    vm.expectRevert(); // reverts with InsufficientTokensUnlockable
    vm.prank(from);
    remoraTokenProxy.transfer(to, amountToTransfer);
}
```

This also totally bricks transfers for users who received tokens when they were whitelisted, then are removed from the whitelist:

```
    function test_transferBricked_whitelistedHolderIsRemovedFromWhitelist() external {
        address from = users[0];
        address to = users[1];
        uint256 amountToTransfer = 10;

        _whitelistAndMintTokensToUser(from, amountToTransfer);

        // set lock time
        remoraTokenProxy.setLockUpTime(3600);

        // remove `from` from whitelist
        remoraTokenProxy.removeFromWhitelist(from);

        // fund `from` with remora tokens once it is not whitelisted
        remoraTokenProxy.mint(from, amountToTransfer);

        // 10 when was whitelisted and 10 when from was not whitelisted
        assertEq(remoraTokenProxy.balanceOf(from), amountToTransfer * 2);

        // forward beyond the lockup time to demonstrate the removed whitelisted holder can't do
        ↪  transfers
        vm.warp(3600 + 1);
        // reverts because from has only 10 tokens locked
        vm.expectRevert(); // reverts with InsufficientTokensUnlockable
        vm.prank(from);
        remoraTokenProxy.transfer(to, 11);

        // verify from can only transfer the 10 tokens that he received after he was removed from
        ↪   whitelist
        vm.prank(from);
        remoraTokenProxy.transfer(to, 10);
    }
```

Recommended Mitigation: A simple and elegant solution may be:

1) check `from` balance; if smaller than `amount` required for transfer revert
2) in transfer functions if the user is not whitelisted, calculate their `uint256 unlockedBalanceToSend = balance - getTokensLocked(sender);` then if `unlockedBalanceToSend < amount` call `_unlockTokens(sender, value - unlockedBalanceToSend..);`

This solution only attempts to unlock the exact amount needed to fulfill a transfer, and doesn't attempt unlock if nothing to unlock or not required as the user has enough unlocked tokens to fulfill the transfer.

Remora: Fixed in commits 67c5e8e, 5db7f11.

Cyfrin: Verified.

7.3.16 Tokens that were locked when `lockUpTime > 0` will be impossible to unlock if `lockUpTime` is set to zero

Description: `LockUpManager::_unlockTokens` returns if `lockUpTime == 0`:

```
function _unlockTokens(
    address holder,
    uint256 amount,
    bool disregardTime
) internal {
    LockUpStorage storage $ = _getLockUpStorage();
    uint32 lockUpTime = $._lockUpTime;
    // @audit returns if `lockUpTime == 0`
    if (lockUpTime == 0 || amount == 0) return;
```

Impact: Tokens that were locked when `lockUpTime > 0` will be impossible to unlock if `lockUpTime` is subsequently set to zero. Initially this won't cause any problems and users will be able to transfer tokens as normal, but if `lockUpTime` is changed to be greater than zero it will start to cause accounting-related problems as one of the protocol invariants is that the amount of tokens a user has locked should be <= to the token balance of the user.

This invariant would be violated since the lockups would still be present but the user could have transferred their tokens, causing underflow reverts in transfers when determine unlocked balance: `uint256 unlockedBalanceToSend = balance - getTokensLocked(sender);`

Recommended Mitigation: Even if `lockUpTime == 0`, proceed through to the `for` loop iterating over all token locks to unlock them. This maintains the protocol invariant that the amount of tokens a user has locked is <= the user's token balance.

Remora: Fixed in commit 5db7f11.

Cyfrin: Verified.

7.3.17 Forwarders who aren't also holders are unable to claim forwarded payouts

Description: Forwarders who aren't also holders are unable to claim forwarded payouts due to this check in `DividendManager::payoutBalance`:

```
    function payoutBalance(address holder) public returns (uint256) {
        HolderManagementStorage storage $ = _getHolderManagementStorage();
        HolderStatus memory rHolderStatus = $._holderStatus[holder];
        uint16 currentPayoutIndex = $._currentPayoutIndex;

        if (
             // @audit must be a holder to claim payouts, prevents forwarders who aren't
             // also holders from claiming their forwarded payouts
            (!rHolderStatus.isHolder) || //non-holder calling the function
            (rHolderStatus.isFrozen && rHolderStatus.frozenIndex == 0) || //user has been frozen from
            ↪  the start, thus no payout
            rHolderStatus.lastPayoutIndexCalculated == currentPayoutIndex // user has already been paid
            ↪  out up to current payout index
        ) return 0;
```

Impact: Forwarders who aren't also holders are unable to claim forwarded payouts.

Recommended Mitigation: Remove the `(!rHolderStatus.isHolder)` check in `DividendManager::payoutBalance`.

Remora: Fixed in commit 82fd5d1.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Use SafeERC20 functions instead of standard ERC20 transfer functions

Description: Use SafeERC20 functions instead of standard ERC20 transfer functions:

```
$ rg "transferFrom" && rg "transfer\("
RWAToken/DividendManager.sol
317:        $._stablecoin.transferFrom(

RWAToken/RemoraToken.sol
220:     * @dev Calls OpenZeppelin ERC20Upgradeable transferFrom function.
251:        return super.transferFrom(from, to, value);
344:            $._stablecoin.transferFrom(sender, $._wallet, _transferFee);
352:     * @dev Calls OpenZeppelin ERC20Upgradeable transferFrom function.
358:    function transferFrom(
376:            $._stablecoin.transferFrom(sender, $._wallet, _transferFee);
379:        return super.transferFrom(from, to, value);

TokenBank.sol
261:        IERC20(stablecoin).transferFrom(

PledgeManager.sol
196:        IERC20(stablecoin).transferFrom(

RemoraIntermediary.sol
172:        IERC20(data.assetReceived).transferFrom(
177:        IERC20(data.assetSold).transferFrom(
197:        IERC20(data.assetReceived).transferFrom(
238:        IERC20(data.paymentToken).transferFrom(
269:            IERC20(data.paymentToken).transferFrom(
296:        IERC20(data.paymentToken).transferFrom(
331:        IERC20(token).transferFrom(payer, recipient, amount);
RWAToken/DividendManager.sol
409:            $._stablecoin.transfer(holder, payoutAmount);
429:        stablecoin.transfer($._wallet, valueToClaim);

RWAToken/RemoraToken.sol
401:        $._stablecoin.transfer(account, burnPayout);

TokenBank.sol
185:        IERC20(tokenAddress).transfer(to, amount);
206:        if (amount != 0) IERC20(stablecoin).transfer(to, amount);
237:        IERC20(stablecoin).transfer(custodialWallet, totalValue);
266:        IERC20(tokenAddress).transfer(to, amount);

PledgeManager.sol
237:                _stablecoin.transfer(feeWallet, feeValue);
239:            _stablecoin.transfer(destinationWallet, amount);
299:        IERC20(stablecoin).transfer(signer, _fixDecimals(refundAmount + fee));
```

Remora: Fixed in commit f2f3f7e.

Cyfrin: Verified.

7.4.2 Fee refund can lose precision

Description: PledgeManager::refundTokens calculates the user fee refund as:

```
fee = (userPay.fee / userPay.tokensBought) * numTokens;
```

Impact: The fee refund will be less than it should be due to division before multiplication

Recommended Mitigation: Perform multiplication before division:

```
fee = userPay.fee * numTokens / userPay.tokensBought;
```

Remora: Fixed in commit b69836f.

Cyfrin: Verified.

7.4.3 `TokenBank::addToken` should revert if token has already been added

Description: `TokenBank::addToken` should revert if token has already been added.

Impact: Token data such as `saleAmount` and `feeAmount` will be reset to zero causing other core functions to malfunction.

Remora: Fixed as of latest commit 2025/07/02 though exact commit unknown.

Cyfrin: Verified.

7.4.4 Changing stablecoin on TokenBank can mess up fees collection

Description: The feeAmount on each token is computed with the decimals of the current stablecoin (initially, a stablecoin of 6 decimals). If the stablecoin is changed to another one that uses decimals != than 6, if there are any pending fees before changing the stablecoin, those pending fees will then be paid with the new stablecoin, causing the actual collected money to be different than expected.

It is possible that by normal operations, a tx to buyTokens gets executed in between fees were claimed and the stablecoin is changed, if the purchased of new tokens generates fees, those new fees will be computed based on the current stablecoin, but will be paid out in the new stablecoin. For example: if 10USDC (10e6) are as pending fees, and the new stablecoin is USDT (10e8), when those fees are collected, they will represent 0.1USDT.

```
    function buyToken(
        address tokenAddress,
        uint32 amount
    ) external nonReentrant {
        ...
        uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;

        ...
        curData.feeAmount += feeValue;

        IERC20(stablecoin).transferFrom(
            to,
            address(this),
            stablecoinValue + feeValue
        );
        ...
    }

    function claimAllFees() external nonReentrant restricted {
       ...
        for (uint i = 0; i < developments.length; ++i) {
//@audit => Pending fees before stablecoin was changed were computed with the decimals of the old
↪   stablecoin
            totalValue += tokenData[developments[i]].feeAmount;
            tokenData[developments[i]].feeAmount = 0;
        }
        IERC20(stablecoin).transfer(custodialWallet, totalValue);
    }
```

Impact: Collected fees can be different from expected if the stablecoin is changed to a stablecoin that has different decimals than 6

Recommended Mitigation: Similar to how values are normalized to the decimals of the current stablecoin on the PledgeManager, implement the same logic on the TokenBank.

- Normalize the amounts of stablecoin before doing the actual transfers.

```
    function claimAllFees() external nonReentrant restricted {
        ...
-       IERC20(stablecoin).transfer(custodialWallet, totalValue);
+       IERC20(stablecoin).transfer(custodialWallet, _fixDecimals(totalValue));
        ...
    }

//@audit => Add this function to normalize values to decimals of the current stablecoin
    function _fixDecimals(uint256 value) internal view returns (uint256) {
        return
            stablecoinDecimals < 6
                ? value / (10 ** (6 - stablecoinDecimals))
                : value * (10 ** (stablecoinDecimals - 6));
    }
```

Remora: Fixed in commit afd07fb.

Cyfrin: Verified.

7.4.5 `TokenBank::removeToken` reverts when token balance is zero, making it impossible to remove tokens from the `developments` array

Description: When removing a token from the TokenBank, the removal attempt reverts if it has zero tokens on the TokenBank balance. As time passes the `developments` array will grow having unnecessary tokens inside it that are impossible to remove.

Impact: The `developments` array will continue to grow leading to unnecessary waste of gas when claiming fees.

Recommended Mitigation: Before calling `withdrawTokens()`, check if the tokenAddress' balance is 0, if so, skip the call (it would anyways revert):

```
    function removeToken(address tokenAddress) external restricted {
        ...
-        withdrawTokens(tokenAddress, custodialWallet, 0);
+       if(IERC20(tokenAddress).balanceOf(address(this)) > 0) withdrawTokens(tokenAddress,
↪   custodialWallet, 0);
        ...
    }
```

Remora: Fixed in commit 2e797fc.

Cyfrin: Verified.

7.4.6 Zero token transfers record receiving user as a holder in `DividendManager::HolderStatus` even if they have zero token balance

Description: Zero token transfers record receiving user as a holder in `DividendManager::HolderStatus` even if they have zero token balance.

Proof of Concept: First add these two functions in `DividendManager`:

```
function getCurrentPayoutIndex() view external returns(uint8 currentPayoutIndex) {
    currentPayoutIndex = _getHolderManagementStorage()._currentPayoutIndex;
}

function getHolderStatus(address holder) view external returns(HolderStatus memory status) {
    status = _getHolderManagementStorage()._holderStatus[holder];
}
```

Then the PoC:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;

import {RemoraToken, DividendManager} from "../../../contracts/RWAToken/RemoraToken.sol";
import {Stablecoin} from "../../../contracts/ForTestingOnly/Stablecoin.sol";
import {AccessManager} from "../../../contracts/AccessManager.sol";
import {Allowlist} from "../../../contracts/Allowlist.sol";

import {UnitTestBase} from "../UnitTestBase.sol";

import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract RemoraTokenTest is UnitTestBase {
    // contract being tested
    RemoraToken remoraTokenProxy;
    RemoraToken remoraTokenImpl;

    // required support contracts & variables
    Stablecoin internal stableCoin;
    AccessManager internal accessMgrProxy;
    AccessManager internal accessMgrImpl;
    Allowlist internal allowListProxy;
    Allowlist internal allowListImpl;
    address internal withdrawalWallet;
    uint32 internal constant DEFAULT_PAYOUT_FEE = 100_000; // 10%

    function setUp() public override {
        // test harness setup
        UnitTestBase.setUp();

        // support contracts / variables setup
        stableCoin = new Stablecoin("USDC", "USDC", type(uint256).max/1e6, 6);
        assertEq(stableCoin.balanceOf(address(this)), type(uint256).max/1e6*1e6);

        // contract being tested setup
        accessMgrImpl = new AccessManager();
        ERC1967Proxy proxy1 = new ERC1967Proxy(address(accessMgrImpl), "");
        accessMgrProxy = AccessManager(address(proxy1));
        accessMgrProxy.initialize(address(this));

        allowListImpl = new Allowlist();
        ERC1967Proxy proxy2 = new ERC1967Proxy(address(allowListImpl), "");
        allowListProxy = Allowlist(address(proxy2));
        allowListProxy.initialize(address(accessMgrProxy), address(this));

        withdrawalWallet = makeAddr("WITHDRAWAL_WALLET");

        // contract being tested setup
        remoraTokenImpl = new RemoraToken();
        ERC1967Proxy proxy3 = new ERC1967Proxy(address(remoraTokenImpl), "");
        remoraTokenProxy = RemoraToken(address(proxy3));
        remoraTokenProxy.initialize(
            address(this), // tokenOwner
            address(accessMgrProxy), // initialAuthority
            address(stableCoin),
            withdrawalWallet,
            address(allowListProxy),
            "REMORA",
            "REMORA",
            0
        );
        assertEq(remoraTokenProxy.authority(), address(accessMgrProxy));
    }

    function test_transferZeroTokens_RegistersHolderWithDividendManager() external {
        address user1 = users[0];
        address user2 = users[1];

        uint256 user1RemoraTokens = 1;

        // whitelist both users
        remoraTokenProxy.addToWhitelist(user1);
        assertTrue(remoraTokenProxy.isWhitelisted(user1));
        remoraTokenProxy.addToWhitelist(user2);
        assertTrue(remoraTokenProxy.isWhitelisted(user2));

        // mint user1 their tokens
        remoraTokenProxy.mint(user1, user1RemoraTokens);
        assertEq(remoraTokenProxy.balanceOf(user1), user1RemoraTokens);

        // allowlist both users
        allowListProxy.allowUser(user1, true, true, false);
        assertTrue(allowListProxy.allowed(user1));
        allowListProxy.allowUser(user2, true, true, false);
        assertTrue(allowListProxy.allowed(user2));
        assertTrue(allowListProxy.exchangeAllowed(user1, user2));

        // user1 transfers zero tokens to user2
        vm.prank(user1);
        remoraTokenProxy.transfer(user2, 0);

        // fetch user2's HoldStatus from DividendManager
        DividendManager.HolderStatus memory user2Status = remoraTokenProxy.getHolderStatus(user2);

        // user2 is listed as a holder even though they have no tokens!
        assertEq(user2Status.isHolder, true);
        assertEq(remoraTokenProxy.balanceOf(user2), 0);
    }
}
```

Recommended Mitigation: Either revert on zero token transfers inside `RemoraToken::_update` or change `DividendManager::_updateHolders` to not set `to` as a holder if their balance is zero.

Remora: Fixed in commit 0a2dea2.

Cyfrin: Verified.

7.4.7 `DividendManager::distributePayout` records a new payout record increasing the current payout index for zero `payoutAmount`

Description: `DividendManager::distributePayout` records a new payout record increasing the current payout index for zero `payoutAmount`.

Proof of Concept:

```
function test_distributePayout_ZeroPayout() external {
    // setup one user
    address user = users[0];
    uint256 userRemoraTokens = 1;

    // whitelist user
    remoraTokenProxy.addToWhitelist(user);
    assertTrue(remoraTokenProxy.isWhitelisted(user));
    // mint user their tokens
    remoraTokenProxy.mint(user, userRemoraTokens);
    assertEq(remoraTokenProxy.balanceOf(user), userRemoraTokens);

    // allowlist user
    allowListProxy.allowUser(user, true, true, false);
    assertTrue(allowListProxy.allowed(user));

    // zero payout distribution - should revert here
    remoraTokenProxy.distributePayout(0);
    // didn't revert but increased current payout index
    assertEq(remoraTokenProxy.getCurrentPayoutIndex(), 1);
}
```

Recommended Mitigation: `DividendManager::distributePayout` should revert when `payoutAmount == 0`.

Remora: Fixed in commit c2002fb.

Cyfrin: Verified.

7.4.8 `PaymentSettler::claimAllPayouts` doesn't validate input `tokens` addresses are legitimate contracts before calling `adminClaimPayout` on them

Description: In `PaymentSettler::initiateBurning` and `distributePayment`, before calling any functions on the input `token` address this check occurs to ensure it is a legitimate address:

```
if (!tokenData[token].active) revert InvalidTokenAddress();
```

But in `PaymentSettler::claimAllPayouts` this check does not occur:

```
function claimAllPayouts(address[] calldata tokens) external nonReentrant {
    address investor = msg.sender;
    uint256 totalPayout = 0;
    for (uint i = 0; i < tokens.length; ++i) {
        TokenData storage curToken = tokenData[tokens[i]];
        uint256 amount = IRemoraToken(tokens[i]).adminClaimPayout(
            investor,
            true
        );
```

Impact: An attacker can deploy their own contract which implements the `adminClaimPayout` function interface but this function can contain arbitrary code; execution flow is transferred to the attacker's contract. We have not found a way to further abuse this but it isn't a good practice to allow an attacker to hijack execution flow into their own custom contracts.

Recommended Mitigation: Verify that the input `tokens` are valid `RemoraToken` contracts prior to calling any functions on them.

Remora: Fixed in commit 4ba903e.

Cyfrin: Verified.

7.4.9 Minting new PropertyTokens close to the end of the distribution period will dilute rewards for holders who were holding for the full period

Description: Holder's balanceHistory is updated each time a transfer of tokens occurs (mint or burn too). The accounting saves the holders' balance during a certain index, which is used to determine the payout earned from the distributed amount of stablecoin for the index.

```
    function _updateHolders(address from, address to) internal {
        ...
```

```
            } else {
                // else update status data and create new entry
                tHolderStatus.mostRecentEntry = payoutIndex;
//@audit => saves the holder balance for the current payout, regardless of how much time is left for the
↪   distribution to end
                $._balanceHistory[to][payoutIndex] = TokenBalanceChange({
                    isValid: true,
                    tokenBalance: toBalance
                });
            }
        }
```

New mintings of PropertyTokens mean that the same amount of payout distributed for all holders will give less payout to each PropertyToken. The problem is that new mintings that occur close to the end of the distribution period will dilute payouts for holders who have held their PropertyTokens for the full period.

Impact: Holders who have held their PropertyTokens for the full distribution period will get their rewards diluted by new PropertyTokens that get minted close to the end of the period.

Recommended Mitigation: On the mint() calculate how much time has passed on the current distribution period, and if a certain threshold (maybe 75-80%) has passed, don't allow new mintings until the next distribution period.

Remora: Acknowledged.

7.4.10 Forwarder can be frozen and still receive and claim payouts while frozen

Description: If the forwarder is frozen before the first payout then they don't receive and can't claim any payouts while frozen.

But if the forwarder is frozen after the first payout, then they do receive and can claim payouts while frozen.

Proof of Concept:

```
function test_forwarderFrozenBeforeFirstPayout_noPayoutBalanceWhileFrozen() external {
    address user1 = users[0];
    address forwarder = users[1];
    uint256 amountToMint = 1;

    _whitelistAndMintTokensToUser(user1, amountToMint);
    _whitelistAndMintTokensToUser(forwarder, amountToMint);

    remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);

    // forwarder frozen before first distribution payout
    remoraTokenProxy.freezeHolder(forwarder);

    uint64 payoutDistributionAmount = 100e6;
    _fundPayoutToPaymentSettler(payoutDistributionAmount);

    // user1 must have 0 payout because it is forwarding to `forwarder`
    uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
    assertEq(user1PayoutBalance, 0);

    // forwarder is frozen yet so receives no forwarded payout
    uint256 forwarderPayoutBalancePreUnfreeze = remoraTokenProxy.payoutBalance(forwarder);
    assertEq(forwarderPayoutBalancePreUnfreeze, 0);
}

function test_forwarderFrozenAfterFirstPayout_validPayoutBalanceWhileFrozen_claimPayoutWhileFrozen()
↪   external {
    _fundPayoutToPaymentSettler(1);
    address user1 = users[0];
    address forwarder = users[1];
    uint256 amountToMint = 1;

    _whitelistAndMintTokensToUser(user1, amountToMint);
    _whitelistAndMintTokensToUser(forwarder, amountToMint);

    remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);

    remoraTokenProxy.freezeHolder(forwarder);

    uint64 payoutDistributionAmount = 100e6;
    _fundPayoutToPaymentSettler(payoutDistributionAmount);

    // user1 must have 0 payout because it is forwarding to `forwarder`
    uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
    assertEq(user1PayoutBalance, 0);

    // forwarder is frozen yet still receives forwarded payout
    uint256 forwarderPayoutBalance = remoraTokenProxy.payoutBalance(forwarder);
    assertEq(forwarderPayoutBalance, payoutDistributionAmount/2);

    // forwarder claims all their payout
    vm.prank(forwarder);
    remoraTokenProxy.claimPayout();
    assertEq(stableCoin.balanceOf(forwarder), forwarderPayoutBalance);
}
```

Recommended Mitigation: The mitigation depends on what the protocol wants to happen in this case, and whether it plans to mitigate L-11. If the protocol wants to allow frozen address to also serve as forwarding addresses and have a payout balance, this could be achieved by:

```
    function payoutBalance(address holder) public returns (uint256) {
        HolderManagementStorage storage $ = _getHolderManagementStorage();
        HolderStatus memory rHolderStatus = $._holderStatus[holder];
        uint16 currentPayoutIndex = $._currentPayoutIndex;

+       if ((rHolderStatus.isFrozen && rHolderStatus.frozenIndex == 0) && rHolderStatus.calculatedPayout
↪   > 0) return rHolderStatus.calculatedPayout;

        if (
            (!rHolderStatus.isHolder) || //non-holder calling the function
            (rHolderStatus.isFrozen && rHolderStatus.frozenIndex == 0) || //user has been frozen from
            ↪   the start, thus no payout
            rHolderStatus.lastPayoutIndexCalculated == currentPayoutIndex // user has already been paid
            ↪   out up to current payout index
        ) return 0;
```

Remora: Acknowledged; the forwarding mechanism is only intended to be used by Remora protocol-owned address to forward payout distributions from tokens held by the `TokenBank` and our liquidity pools. So it is unlikely that the freezing and forwarding mechanisms will ever interact.

7.4.11 Forwarder can be set to frozen address

Description: Forwarder can be set to frozen address; this is not ideal and can result in lost tokens in a worst-case scenario.

Proof of Concept:

```
    function test_freezeHolder_setPayoutForwardAddress_toFrozenForwarder() external {
        address user1 = users[0];
        address forwarder = users[1];
        uint256 amountToMint = 1;

        _whitelistAndMintTokensToUser(user1, amountToMint);
        _whitelistAndMintTokensToUser(forwarder, amountToMint);

        // freeze forwarder
        remoraTokenProxy.freezeHolder(forwarder);

        // forward user1 payouts to frozen address
        remoraTokenProxy.setPayoutForwardAddress(user1, forwarder);

        uint64 payoutDistributionAmount = 100e6;
        _fundPayoutToPaymentSettler(payoutDistributionAmount);

        // user1 must have 0 payout because it is forwarding to `forwarder`
        uint256 user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0);

        // forwarder is frozen  so receives no forwarded payout
        uint256 forwarderPayoutBalance = remoraTokenProxy.payoutBalance(forwarder);
        assertEq(forwarderPayoutBalance, 0);

        // remove the forwarding while forwarder still frozen
        remoraTokenProxy.removePayoutForwardAddress(user1);

        // user1 can't claim any tokens - user1 has lost their payouts
        user1PayoutBalance = remoraTokenProxy.payoutBalance(user1);
        assertEq(user1PayoutBalance, 0);
    }
```

Recommended Mitigation: `DividendManager::setPayoutForwardAddress` should revert if `forwardingAddress` is frozen. When an address is frozen if that address has users forwarding to it, consider cancelling all those forwards.

Remora: Acknowledged; the forwarding mechanism is only intended to be used by Remora protocol-owned address to forward payout distributions from tokens held by the `TokenBank` and our liquidity pools. So it is unlikely that the freezing and forwarding mechanisms will ever interact.

7.4.12 Impossible to remove a document added with zero uri length

Description: Impossible to remove a document added with zero uri length.

Proof of Concept: After fixing the bug "Don't add duplicate documentHash to DocumentManager::DocumentStorage::_docHashes when overwriting via _setDocument as this causes panic revert when calling _removeDocument", run this fuzz test:

```
    function test_setDocumentOverwrite(string calldata uri) external {
        bytes32 docName = "0x01234";
        bytes32 docHash = "0x5555";
        bool needSignature = true;

        // add the document
        _setDocument(docName, uri, docHash, needSignature);

        // verify its hash has been added to `_docHashes`
        DocumentStorage storage $ = _getDocumentStorage();
        assertEq($._docHashes.length, 1);
        assertEq($._docHashes[0], docHash);

        // ovewrite it
        _setDocument(docName, uri, docHash, needSignature);
        // verify overwriting doesn't duplicate the hash in `_docHashes`
        assertEq($._docHashes.length, 1);
        assertEq($._docHashes[0], docHash);

        // now attempt to remove it
        _removeDocument(docHash);
    }
```

It reverts with `[FAIL: EmptyDocument();` when calling `_removeDocument` at the end.

Recommended Mitigation: Don't allowing adding documents with empty uri.

Remora: Fixed in commit 1218d18.

Cyfrin: Verified.

7.4.13 `PledgeManager::pricePerToken` can only support a maximum price of `$4294`

Description: `PledgeManager::pricePerToken` uses `uint32` and indicates in the comment it represents USD price using 6 decimals of precision:

```
    uint32 public pricePerToken; //in usd (6 decimals)
```

Impact: Since the maximum value of `uint32` is 4294967295, with 6 decimals of precision the maximum USD `pricePerToken` is limited to `$4294`.

This may be insufficient because the goal is to tokenize real-estate which can be worth many millions of dollars.

Recommended Mitigation: Use a larger size to store `pricePerToken` if supporting a large USD price is required.

Remora: Acknowledged; we plan to keep the price low around say $50 per token.

7.5 Informational

7.5.1 Emit missing events for storage changes

Description: Emit missing events for storage changes:

- `Allowlist::changeUserAccreditation`, `changeAdminStatus`
- `Allowlist::UserAllowed` should be expanded to contain and emit the `HolderInfo` boolean flags
- `ReferralManager::addReferral`
- `RemoraIntermediary::setFundingWallet`, `setFeeRecipient`
- `TokenBank::changeReferralManager`, `changeStablecoin`, `changeCustodialWallet`
- `TokenBank::TokensWithdrawn` should include withdrawn `amount`
- `DividendManager::setPayoutForwardAddress`, `changeWallet`
- `RemoraToken::addToWhitelist`, `removeFromWhitelist`, `updateAllowList`
- `PaymentSettler::withdraw`, `withdrawAllFees` should emit amount withdrawn, `addToken`, `changeCustodian`, `changeStablecoin`

Remora: Fixed in commit 9051af8.

Cyfrin: Verified.

7.5.2 Rename `isAllowed` to `wasAllowed` in `Allowlist::allowUser`, `disallowUser`

Description: `Allowlist::allowUser`, `disallowUser` return the existing `allowed` status into a named return variable called `isAllowed`, before potentially modifying the `allowed` status.

Since these functions can modify the `allowed` status, the named return variable should be renamed to `wasAllowed` to explicitly indicate the returned status may not be current.

Remora: Fixed in commit 06d17a6.

Cyfrin: Verified.

7.5.3 Use constants instead of magic numbers

Description: Use constants instead of magic numbers; 1000000, 1e6 and 10 ** 6 are all identical and should declared in a constant that is imported into the various files.

```
TokenBank.sol
111:        if (newFee > 1000000) revert InvalidValuePassed();
252:        uint64 feeValue = (stablecoinValue * curData.saleFee) / 1e6;

PledgeManager.sol
151:        require(newPenalty <= 1000000);
157:        require(newFee <= 1000000);
180:        uint256 fee = (stablecoinAmount * pledgeFee) / 1e6;
283:            refundAmount -= (refundAmount * earlySellPenalty) / 1e6;

RWAToken/DividendManager.sol
230:        require(newFee <= 1e6);
416:        payoutAmount -= (payoutAmount * fee) / (10 ** 6);

RWAToken/RemoraToken.sol
306:            require(newBurnFee <= 1e6);
398:        if (burnFee != 0) burnPayout -= (burnPayout * burnFee) / 1e6;
```

Remora: Fixed in commit aaaab45.

Cyfrin: Verified.

7.5.4 Using explicit unsigned integer sizing instead of `uint`

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

```
PaymentSettler.sol
124:        for (uint i = 0; i < len; ++i) {
174:        for (uint i = 0; i < tokens.length; ++i) {
227:        for (uint i = 0; i < tokenList.length; ++i) {

RWAToken/DocumentManager.sol
224:        for (uint i = 0; i < dHLen; ++i) {

TokenBank.sol
185:        for (uint i = 0; i < len; ++i) {
260:        for (uint i = 0; i < developments.length; ++i)
267:        for (uint i = 0; i < developments.length; ++i) {
```

Remora: Fixed in commit 6602423.

Cyfrin: Verified.

7.5.5 Retrieve and enforce token decimal precision

Description: Retrieve and enforce token decimal precision using `IERC20Metadata`. For example:

1) `PledgeManager::initialize`

```
    constructor(
        address authority,
        address _holderWallet,
        address _propertyToken,
        address _stablecoin,
-       uint16 _stablecoinDecimals,
        uint32 _fundingGoal,
        uint32 _deadline,
        uint32 _withdrawDuration,
        uint32 _pledgeFee,
        uint32 _earlySellPenalty,
        uint32 _pricePerToken
    ) AccessManaged(authority) ReentrancyGuardTransient() {
        holderWallet = _holderWallet;
        propertyToken = _propertyToken;
        stablecoin = _stablecoin;
-       stablecoinDecimals = _stablecoinDecimals;
+       stablecoinDecimals = IERC20Metadata(_stablecoin).decimals();
        fundingGoal = _fundingGoal;
        deadline = _deadline;
        postDeadlineWithdrawPeriod = _withdrawDuration;
        pledgeFee = _pledgeFee;
        earlySellPenalty = _earlySellPenalty;
        pricePerToken = _pricePerToken;
        tokensSold = 0;
    }
```

2) `TokenBank::initialize`

```
        stablecoin = _stablecoin; //must be 6 decimal stablecoin
+       require(IERC20Metadata(stablecoin).decimals() == 6, "Wrong decimals");
```

Remora: This was resolved by adding the `remoraToNativeDecimals` which always converts from internal Remora precision to external stablecoin precision, so the protocol can now work with different decimal stablecoins.

7.5.6 `LockUpManager::LockUpStorage::_regLockUpTime` is never used

Description: `LockUpManager::LockUpStorage::_regLockUpTime` is never used:

```
$ rg "_regLockUpTime"
RWAToken/LockUpManager.sol
36:        uint32 _regLockUpTime; //lock up time for foreign to domestic trades
```

Either remove it or add a comment noting it will be used in the future but is currently not used.

Remora: Fixed in commit 91aed23 by adding a note noting it is intended for future use.

Cyfrin: Verified.

7.5.7 Use `SignatureChecker` library and optionally support `EIP7702` accounts which use their private key to sign

Description: In `DocumentManager::verifySignature` use the SignatureChecker library:

```
-        if (signer.code.length == 0) {
-            //signer is EOA
-            (address returnedSigner, , ) = ECDSA.tryRecover(digest, signature);
-            result = returnedSigner == signer;
-        } else {
-            //signer is SCA
-            (bool success, bytes memory ret) = signer.staticcall(
-                abi.encodeWithSelector(
-                    bytes4(keccak256("isValidSignature(bytes32,bytes)")),
-                    digest,
-                    signature
-                )
-            );
-            result = (success && ret.length == 32 && bytes4(ret) == MAGICVALUE);
-        }

-        if (!result) revert InvalidSignature();
+        if(!SignatureChecker.isValidSignatureNow(signer, digest, signature)) revert InvalidSignature();
```

Additionally with EIP7702, it is now possible for addresses to have `code.length > 0` but still use their private keys to sign; so with the current code or the above recommendation this scenario won't be supported.

To support this scenario check out this finding from our recent audit where this scenario is also supported by first calling ECDSA.tryRecover then if that didn't work calling `SignatureChecker::isValidERC1271SignatureNow` as the backup option:

```
+        (address recovered, ECDSA.RecoverError error,) = ECDSA.tryRecover(digest, signature);

+        if (error == ECDSA.RecoverError.NoError && recovered == signer) result = true;
+        else result = SignatureChecker.isValidERC1271SignatureNow(signer, digest, signature);

+        if (!result) revert InvalidSignature();
```

Remora: Fixed in commit b545498.

Cyfrin: Verified.

7.5.8 Use `EIP712Upgradeable` library to simplify `DocumentManager`

Description: Use `EIP712Upgradeable` library to simplify `DocumentManager` as this library provides the domain separator and the helpful function `_hashTypedDataV4`.

Inherit from `EIP712Upgradeable`, remove all the duplicate code which it provides then in `verifySignature` do this:

```
-        bytes32 digest = keccak256(
-            abi.encodePacked("\x19\x01", _DOMAIN_SEPARATOR, structHash)
-        );
+        bytes32 digest = _hashTypedDataV4(structHash);
```

Remora: Fixed in commit b545498.

Cyfrin: Verified.

7.5.9 Remove unnecessary imports and inheritance

Description: Remove unnecessary imports and inheritance:

- `BurnStateManager` should only import and inherit from `Initializable`

Remora: Fixed in commit 8419903.

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Fail fast without performing unnecessary storage reads

Description: Fail fast without performing unnecessary storage reads:

- `Allowlist::exchangeAllowed` - don't read `_allowed[to]` from storage if the transaction will fail since `from` is not allowed:

```
function exchangeAllowed(
    address from,
    address to
) external view returns (bool) {
    HolderInfo memory fromUser = _allowed[from];
    if (from != address(0) && !fromUser.allowed) revert UserNotRegistered(from);

    HolderInfo memory toUser = _allowed[to];
    if (to != address(0) && !toUser.allowed) revert UserNotRegistered(to);

    return fromUser.domestic == toUser.domestic; //logic to be edited later on
}
```

- `Allowlist::hasTradeRestriction` - don't read `_allowed[user2]` from storage if the transaction will fail since `_allowed[user1]` is not allowed:

```
function hasTradeRestriction(
    address user1,
    address user2
) public returns (bool) {
    HolderInfo memory u1Data = _allowed[user1];
    if (!u1Data.allowed) revert UserNotRegistered(user1);

    HolderInfo memory u2Data = _allowed[user2];
    if (!u2Data.allowed) revert UserNotRegistered(user2);
```

Remora: Fixed in commits 81faadb, 760469f.

Cyfrin: Verified.

7.6.2 Don't initialize to default values

Description: Don't initialize to default values:

```
RWAToken/DividendManager.sol
186:      $._currentPayoutIndex = 0;

RWAToken/DocumentManager.sol
118:        for (uint256 i = 0; i < numDocs; ++i) {
222:        for (uint i = 0; i < dHLen; ++i) {

RWAToken/DividendManager.sol
349:                for (uint256 i = 0; i < len; ++i) {
463:        for (uint256 i = 0; i < rHolderStatus.forwardedPayouts.length; ++i) {

TokenBank.sol
152:        for (uint i = 0; i < developments.length; ++i) {
225:        uint64 totalValue = 0;
226:        for (uint i = 0; i < developments.length; ++i)
232:        uint64 totalValue = 0;
233:        for (uint i = 0; i < developments.length; ++i) {

PledgeManager.sol
119:        tokensSold = 0;
278:        uint256 fee = 0;
```

```
RemoraIntermediary.sol
258:        for (uint256 i = 0; i < len; ++i) {

PaymentSettler.sol
124:        for (uint i = 0; i < len; ++i) {
174:        for (uint i = 0; i < tokens.length; ++i) {
226:        uint256 totalFees = 0;
227:        for (uint i = 0; i < tokenList.length; ++i) {
```

Remora: Fixed in commit 6602423.

Cyfrin: Verified.

7.6.3 Cache identical storage reads

Description: Reading from storage is expensive, cache identical storage reads to prevent re-reading the same storage values multiple times.

`PledgeManager.sol`:

```
// cache `stablecoinDecimals` in `_fixDecimals`
307:            stablecoinDecimals < 6
308:                ? value / (10 ** (6 - stablecoinDecimals))
309:                : value * (10 ** (stablecoinDecimals - 6));

// cache `propertyToken` in `_verifyDocumentSignature`
316:        (bool res, ) = IRemoraRWAToken(propertyToken).hasSignedDocs(signer);
318:            IRemoraRWAToken(propertyToken).verifySignature(
```

`TokenBank.sol`:

```
// cache `developments.length` in `removeToken`
152:        for (uint i = 0; i < developments.length; ++i) {
154:            address end = developments[developments.length - 1];

// cache `developments.length` in `viewAllFees`, claimAllFees
226:        for (uint i = 0; i < developments.length; ++i)
233:        for (uint i = 0; i < developments.length; ++i) {
```

`LockUpManager.sol`:

```
// cache `userData.endInd` in `availableTokens`, `_unlockTokens`
117:        for (uint16 i = userData.startInd; i < userData.endInd; ++i) {
146:        for (uint16 i = userData.startInd; i < userData.endInd; ++i) {
```

Remora: Fixed in commit 6602423.

Cyfrin: Verified.

7.6.4 Remove `< 0` comparison for unsigned integers

Description: Unsigned integers can't be `< 0` so this comparison should be removed: `PledgeManager.sol`:

```diff
-        if (numTokens <= 0 || _propertyToken.balanceOf(signer) < numTokens)
+        if (numTokens == 0 || _propertyToken.balanceOf(signer) < numTokens)
```

Remora: Fixed in commit 6be4660.

Cyfrin: Verified.

7.6.5 Variables in non-upgradeable contracts which are only set once in `constructor` should be declared `immutable`

Description: Variables in non-upgradeable contracts which are only set once in `constructor` should be declared `immutable`:

- `PledgeManager::holderWallet`, `propertyToken`, `stablecoin`, `stablecoinDecimals`, `deadline`, `postDeadlineWithdrawPeriod`, `pricePerToken`

Remora: Fixed in commit afd07fb.

Cyfrin: Verified.

7.6.6 Break out of loop once element has been deleted in `TokenBank::removeToken`

Description: Break out of loop once element has been deleted in `TokenBank::removeToken`:

```
    function removeToken(address tokenAddress) external restricted {
        for (uint i = 0; i < developments.length; ++i) {
            if (developments[i] == tokenAddress) {
                address end = developments[developments.length - 1];
                developments[i] = end;
                developments.pop();
+               break;
            }
        }
```

Remora: Fixed as of latest commit 2025/07/02 but exact commit unknown.

Cyfrin: Verified.

7.6.7 Use named returns where this eliminates a local variable and especially for `memory` returns

Description: Use named returns where this eliminates a local variable and especially for `memory` returns:

- `TokenBank::viewAllFees`

Remora: Fixed in commit 6602423.

Cyfrin: Verified.

7.6.8 In `RemoraToken::adminClaimPayout`, `adminTransferFrom` don't call `hasSignedDocs` when `checkTC == false`

Description: In `RemoraToken::adminClaimPayout` don't call `hasSignedDocs` when `checkTC == false` to prevent doing unnecessary work:

```
function adminClaimPayout(
    address investor,
    bool useStablecoin,
    bool useCustomFee,
    bool checkTC,
    uint256 feeValue
) external nonReentrant restricted {
    if(checkTC) {
        (bool res, ) = hasSignedDocs(investor);
        if (!res) revert TermsAndConditionsNotSigned(investor);
    }

    _claimPayout(investor, useStablecoin, useCustomFee, feeValue);
}
```

Apply similar fix to `adminTransferFrom`.

Remora: Fixed in commit a5c03d4.

Cyfrin: Verified.

7.6.9 In `RemoraToken::transfer`, `transferFrom` and `_exchangeAllowed` perform all checks for each user together in order to prevent unnecessary work

Description: In `RemoraToken::transfer`, `transferFrom` and `_exchangeAllowed` perform all checks for each user together in order to prevent unnecessary work.

`transfer`:

```
        bool fromWL = _whitelist[sender];
        if (!fromWL) _unlockTokens(sender, value, false);

        bool toWL = _whitelist[to];
        if (!toWL) _lockTokens(to, value);
```

`transferFrom`:

```
        bool fromWL = _whitelist[from];
        if (!fromWL) _unlockTokens(from, value, false);

        bool toWL = _whitelist[to];
        if (!toWL) _lockTokens(to, value);
```

`_exchangeAllowed`:

```
        (bool resFrom, ) = hasSignedDocs(from);
        if (!resFrom && !_whitelist[from]) revert TermsAndConditionsNotSigned(from);

        (bool resTo, ) = hasSignedDocs(to);
        if (!resTo && !_whitelist[to]) revert TermsAndConditionsNotSigned(to);
```

Remora: Fixed in commit 59cf2eb.

Cyfrin: Verified.

7.6.10 `AllowList::hasTradeRestriction` mutability should be set to `view`

Description: `AllowList::hasTradeRestriction` does not make any changes to storage, it simply reads some variables and makes some checks, but the function's mutability is not marked as view.

Recommended Mitigation: Change mutability to view.

Remora: Fixed in commit 81faadb.

Cyfrin: Verified.

7.6.11 Remove `decimals` from initial `RemoraToken` mint

Description: `RemoraToken` overrides the `decimals` function to return 0 as its tokens are non-fractional:

```
    /**
     * @notice Defines the number of decimal places for the token.
     * RWA tokens are non-fractional and operate in whole units only.
     * @return The number of decimals (always `0`).
     */
    function decimals() public pure override returns (uint8) {
        return 0;
    }
```

So remove the call to `decimals` inside `RemoraToken::initialize` since `10 ** decimals()` will always evaluate to 1:

```
    function initialize(
        address tokenOwner,
        address initialAuthority,
        address stablecoin,
        address wallet,
        address _allowList,
        string memory _name,
        string memory _symbol,
        uint256 _initialSupply
    ) public initializer {
        // does this order matter?, open zeppelin upgrades giving errors here
        __ERC20_init(_name, _symbol);
        __ERC20Permit_init(_name);
        __Pausable_init();
        __RemoraBurnable_init();
        __RemoraLockUp_init(0); //start with 0 lock up time
        __RemoraDocuments_init(_name, "1");
        __RemoraHolderManagement_init(
            initialAuthority,
            stablecoin,
            wallet,
            0 //starts at zero, will need to update it later
        );
        __UUPSUpgradeable_init();

        allowlist = _allowList;
        _whitelist[tokenOwner] = true; //whitelist owner to be able to send tokens freely
-       _mint(tokenOwner, _initialSupply * 10 ** decimals());
+       _mint(tokenOwner, _initialSupply);
    }
```

Remora: Fixed in commit 462d2de.

Cyfrin: Verified.

7.6.12 Use timestamp instead of uri length to test of existing document in `DocumentManager`

Description: Use timestamp instead of uri length to test of existing document in `DocumentManager`:

```
-       if (bytes($._documents[docHash].docURI).length == 0)
-           revert EmptyDocument();
+       if ($._documents[docHash].timestamp == 0) revert EmptyDocument();
```

Remora: Fixed in commit 77af634.

Cyfrin: Verified.