Size Solidity

Security Review

Cantina Managed review by:

Hyh, Lead Security Researcher
Elhaj, Associate Security Researcher

June 14, 2025

Contents

DRAFT

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

DRAFT

2 Security Review Summary

Size Credit is the first protocol where lenders and borrowers can fully customize offers around any rates and dates.

From May 27th to Jun 6th the Cantina team conducted a review of size-solidity on commit hash daf1d1d8. The team identified a total of **16** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	5	5	0
Low Risk	4	4	0
Gas Optimizations	0	0	0
Informational	5	3	2
Total	**16**	**14**	**2**

DRAFT

3 Findings

3.1 Critical Risk

3.1.1 No-op on zero balance in `_transferFrom` function enables asset theft

Severity: Critical Risk

Context: NonTransferrableRebasingTokenVault.sol#L430,442, NonTransferrableRebasingToken-Vault.sol#L260,266

Description: A vulnerability exists in the `setVault()` function that allows users to steal protocol assets by exploiting a no-op condition in the `_transferFrom` function. When a user has shares in a vault but zero balance (`sharesOf[user] > 0` but `balanceOf(user) == 0`), they can effectively copy their shares to a different vault with higher share price, enabling theft of assets. The issue occurs because `setVault()` unconditionally updates the user's vault even when no actual transfer happens:

```
function setVault(address user, address vault) external onlyMarket {
    if (user == address(0)) {
        revert Errors.NULL_ADDRESS();
    }
    if (vaultToIdMap.contains(vault) && vaultOf[user] != vault) {
        // Transfer happens based on balanceOf(user), which might be 0
        _transferFrom(vaultOf[user], vault, user, user, balanceOf(user));

        emit VaultSet(user, vaultOf[user], vault);
        vaultOf[user] = vault; // Always sets the new vault regardless of transfer success
    }
}
```

The problem is in the `_transferFrom` function which does nothing when `value == 0`:

```
function _transferFrom(address vaultFrom, address vaultTo, address from, address to, uint256 value) private {
    if (value > 0) { // No-op when value is 0
        // Transfer logic here...
    }
}
```

This creates an accounting mismatch where the user's shares from one vault are effectively copied to another vault without proper asset transfer. This desynchronization happens because:

- `setVault()` updates `vaultOf[user]` to the new vault unconditionally.
- `_transferFrom()` does nothing when `balanceOf(user) == 0` (no-op condition).
- But `sharesOf[user]` values remain unchanged.
- The mismatch allows users to claim assets from a vault they never properly deposited into.

This can happen in multiple scenarios:

1. Malicious Vault: A vault can issue many shares for minimal assets, report zero asset for any amount of shares.
2. Rounding Issues: Small deposits in low-priced vaults can result in shares with zero asset value due to rounding. Moving these shares to a higher-priced vault enables value extraction.
3. Bankrupt Vault: A vault with bad debt will have shares with zero asset value. Users can transfer these worthless shares to healthy vaults and withdraw assets they never deposited.

All the cases can be exploited straightforwardly, while share price discrepancy between honest and non-bankrupt vaults can occur naturally over time.

Recommendation: Modify the `setVault` function so it always sets user shares to zero before updating user vault.

Size: Fixed in PR 177 (see discussion 1, discussion 2, discussion 3, discussion 4, discussion 5).

Cantina Managed: Fix confirmed, `_setSharesOf(user, 0)` is now invoked in all the logic branches of `Set-Vault`. Keep in mind that all new adapter contracts now need to have `fullWithdraw(...)` implementation that performs `tokenVault.setSharesOf(from, 0)` unconditionally.

DRAFT

3.2 High Risk

3.2.1 `AaveAdapter` precision loss can cause withdrawals reverts and cross-vault transfers failures

Severity: High Risk

Context: AaveAdapter.sol#L94-L97, NonTransferrableRebasingTokenVault.sol#L434-L443

Description: The `withdraw` function in the `AaveAdapter` contract can revert due to precision when interacting with Aave. the issue affects both direct withdrawals and cross-vault transfers,causing DoS to alot of core protocol functionality.

- Aave uses rebasing tokens (aTokens) that track user balances using two values:
  1. A raw balance amount visible to users (the aToken balance).
  2. An internal "scaled balance" that gets multiplied by a constantly increasing `liquidityIndex` to calculate the raw balance.
- The relationship is: `rawBalance = rayMul(scaledBalance , liquidityIndex)`.

  The current withdrawal flow has these steps:

  ```
  // First the aToken is transferred from tokenVault to adapter
  IERC20Metadata(address(aToken)).safeTransferFrom(address(tokenVault), address(this), amount);
  // Then the adapter tries to withdraw the underlying token from Aave
  aavePool.withdraw(address(underlyingToken), amount, to);
  ```

  The problem occurs because:

  1. When transferring aTokens using `safeTransferFrom`, Aave internally converts the raw token amount to a `scaledBalance` and send the `scaledBalance`:

  ```
  newScaledBalance = rayDiv(amount ,liquidityIndex)
  ```

  2. When calling `withdraw` function on Aave pool, it performs the reverse calculation and validates that the result is sufficient:

  ```
  // Inside Aave's withdraw function
  calculatedAmount = rayMul(newScaledBalance ,liquidityIndex)
  assert(calculatedAmount >= amount)
  ```

  3. Due to precision loss (division followed by multiplication), the calculated amount can be slightly less than the original requested amount, causing the assertion to fail and the withdrawal to revert.

Since this function can be called from `NonTransferrableRebasingTokenVault._transferFrom` and `NonTransferrableRebasingTokenVault.withdraw`, it can block all core transfer flows leading to:

1. Direct user withdrawals will revert when attempting to withdraw assets from Aave vaults.
2. Cross-vault transfers will fail when Aave is the source vault blocking: *claims, sell/buy credits, debt repayments, etc...*
3. User vault migration will be blocked through `setVault()` function if the src vault is Aave.

Recommendation: Create a direct withdrawal helper in the `NonTransferrableRebasingTokenVault` contract to bypass the extra transfer.

Size: Fixed in PR 177 (see discussion 1, discussion 2).

Cantina Managed: Fix confirmed, `requestAaveWithdraw` function was added to make a direct withdrawal from `NonTransferrableRebasingTokenVault` which eliminates the rounding issue in `transferFrom`.

3.3 Medium Risk

3.3.1 Incorrect liquidity check in `ERC4626Adapter` allows borrowing without guaranteed withdrawal

Severity: Medium Risk

Context: ERC4626Adapter.sol#L104-L107

DRAFT

Description: The `transferFrom` function in the `ERC4626Adapter` incorrectly checks for sufficient liquidity using `totalAssets()` instead of `maxWithdraw()`:

```
function transferFrom(address vault, address from, address to, uint256 value) external onlyOwner {
    if (IERC4626(vault).totalAssets() < value) {
        revert InsufficientTotalAssets(address(vault), IERC4626(vault).totalAssets(), value);
    }
    // ...
}
```

`totalAssets()` returns the total amount of underlying assets managed by the vault, including assets that might be locked, invested elsewhere, or otherwise unavailable for immediate withdrawal. Meanwhile, `maxWithdraw(address)` returns the maximum amount of underlying assets that can be actually withdrawn by a specific owner. Checking liquidity is meant to prevent users from borrowing assets that are not available for withdrawal, but now it's not achieved. This vulnerability allows users to borrow assets that may not be immediately withdrawable. A user could borrow an amount against their collateral, but later be unable to withdraw the borrowed funds for an arbitrary time.

Recommendation: Replace the check in `transferFrom` with a check on the maximum withdrawable amount:

```
  function transferFrom(address vault, address from, address to, uint256 value) external onlyOwner {
-     if (IERC4626(vault).totalAssets() < value) {
+     if (IERC4626(vault).maxWithdraw(address(tokenVault)) < value) {
          revert InsufficientTotalAssets(address(vault), IERC4626(vault).totalAssets(), value);
      }
      // ...
  }
```

This prevents situations where users are unable to withdraw borrowed funds.

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `ERC4626Adapter` now checks liquidity using `maxWithdraw` instead of `totalAssets`.

3.3.2 Any ERC-4626 vault can drain protocol liquidity via reentrancy

Severity: Medium Risk

Context: ERC4626Adapter.sol#L54-L67

Description: A vault can reenter the Size protocol during the deposit operation and change user configuration vault by calling `setUserConfigurationOnBehalfOf` on the `sizeMarket` before the share accounting is updated. This creates a mismatch between the vault assignment (`vaultOf` mapping) and share accounting (`sharesOf` mapping), allowing an attacker to drain all liquidity from any vault in the protocol. The vulnerability exists in the `ERC4626Adapter.deposit` implementation, which makes an external call to a vault before updating the user's share accounting:

```
function deposit(address vault, address to, uint256 amount) external onlyOwner returns (uint256 assets) {
    // Read state before external call
    vars.sharesBefore = IERC4626(vault).balanceOf(address(tokenVault));
    vars.userSharesBefore = tokenVault.sharesOf(to);

    // External call that can lead to reentrancy
    IERC4626(vault).deposit(amount, address(tokenVault));

    // State updates AFTER external call (too late)
    uint256 shares = IERC4626(vault).balanceOf(address(tokenVault)) - vars.sharesBefore;
    assets = IERC4626(vault).convertToAssets(shares);
    tokenVault.setSharesOf(to, vars.userSharesBefore + shares);
}
```

Scenario:

1. A user deposits funds into a whitelisted ERC4626 vault through the Size protocol.
2. During the `deposit` call, the vault's implementation is executed.

DRAFT

3. If the vault is malicious or has callbacks(erc721 transfer ..etc), an attacker can make a call to `size.setUserConfigurationOnBehalfOf(user, newVaultAddress)` while inside the deposit transaction.
4. This changes the user's vault from the original vault to a different vault in the `vaultOf[user]` mapping with higher share price.
5. When control returns to the adapter, it updates `sharesOf[user]` with shares from the original vault.
6. This creates a critical accounting mismatch:

- `vaultOf[user]` points to VaultB.
- But `sharesOf[user]` are from VaultA.

7. The user can now call `withdraw()`, which will withdraw funds from VaultB based on the number of shares from VaultA.
8. This allows the attacker to withdraw funds they never deposited.

Another attack path is calling deposit again inside the deposit callback , this will double account user shares ,since the function uses :

```
uint256 shares = IERC4626(vault).balanceOf(address(tokenVault)) - vars.sharesBefore;
```

The protocol is especially vulnerable because:

- The state update happens after the external call.
- Multiple Size market contracts share the same token vault, allowing cross-contract reentrancy.
- The vault adapter doesn't validate that the user's vault configuration remains unchanged during deposit.

This attack if successful could drain all liquidity from any whitelisted vault in the protocol, leading to a complete loss of user funds.

Recommendation: Add reentrancy guards to all state-changing functions in `NonTransferrableRebasing-TokenVault` and in `Size` contracts.

Size: Fixed in PR 177 (see discussion 1, discussion 2).

Cantina Managed: Fix confirmed, all state changing functions in both contracts now control for reentrancy.

3.3.3 Users with removed or compromised vaults can be locked out of protocol

Severity: Medium Risk

Context: NonTransferrableRebasingTokenVault.sol#L256-L266

Description: The current implementation of `setVault()` always attempts to interact with a user's current vault when switching to a new vault. This creates a deadlock situation when a vault is compromised and subsequently removed from the whitelist. When a vault is removed from the whitelist (via `removeVault()`), any user whose `vaultOf[user]` still points to that removed vault becomes permanently locked.

```
function setVault(address user, address vault) external onlyMarket {
    if (user == address(0)) {
        revert Errors.NULL_ADDRESS();
    }
    if (vaultToIdMap.contains(vault) && vaultOf[user] != vault) {
        _transferFrom(vaultOf[user], vault, user, user, balanceOf(user));
        emit VaultSet(user, vaultOf[user], vault);
        vaultOf[user] = vault;
    }
}
```

When a user with a removed vault tries to use `setVault()` to switch to a valid vault, the function will attempt to call `_transferFrom()`, which tries to interact with the removed vault. Since the vault is no longer in `vaultToIdMap`, the call to `getWhitelistedVaultAdapter(vaultFrom)` will revert, blocking the user from switching to a new vault.

DRAFT

This creates a deadlock where users can never escape from a compromised vault, with the most severe consequences for claim processing:

1. If a vault is compromised and drained, it must be removed to prevent attackers from stealing unclaimed funds.
2. But once removed, users with pending claims who used that vault cannot execute their claims.
3. The `executeClaim()` function will revert when trying to interact with the removed vault.
4. These users have no way to forfeit their association with the compromised vault.

This effectively means that if a vault is compromised, users with pending claims may permanently lose access to those funds, even if the claims themselves are valid and shouldn't be affected.

Recommendation: Implement an option to forfeit shares from the old vault when switching to a new vault. This would allow users to abandon a compromised vault and move to a valid one, accepting the loss of any assets in the compromised vault.

```
function setVault(address user, address vault, bool forfeitOldShares) external onlyMarket {
    if (user == address(0)) {
        revert Errors.NULL_ADDRESS();
    }
    if (vaultToIdMap.contains(vault) && vaultOf[user] != vault) {
        if (forfeitOldShares || !vaultToIdMap.contains(vaultOf[user])) {
            // Record dust shares if any
            vaultDust[vaultOf[user]] += sharesOf[user];
            sharesOf[user] = 0;

            emit VaultSet(user, vaultOf[user], vault);
            vaultOf[user] = vault;
            // ...
            // ...
        }
    }
}
```

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `forfeitOldShares` option was added to the `SetVault` operation.

3.3.4 Incorrect new borrow APR validation in `LiquidateWithReplacement`

Severity: Medium Risk

Context: LiquidateWithReplacement.sol#L84

Description: In the `validateLiquidateWithReplacementParams` function, the APR validation uses an incorrect rate calculation function. The validation uses `getUserDefinedBorrowOfferAPR()` which only considers the user's default rate, while the actual liquidation operation uses `getBorrowOfferRatePerTenor()` which considers the collection-specific rate provider.

```
// In validateLiquidateWithReplacementParams
uint256 borrowAPR = state.getUserDefinedBorrowOfferAPR(params.borrower, tenor);
if (borrowAPR < params.minAPR) {
    revert Errors.APR_LOWER_THAN_MIN_APR(borrowAPR, params.minAPR);
}
```

But later in `executeLiquidateWithReplacement`, a different function is used:

```
uint256 ratePerTenor =
    state.getBorrowOfferRatePerTenor(params.borrower, params.collectionId, params.rateProvider, tenor);
uint256 issuanceValue = Math.mulDivDown(debtPositionCopy.futureValue, PERCENT, PERCENT + ratePerTenor);
```

This inconsistency means that the APR validation may pass when it should fail (or vice versa), leading to liquidations occurring at unexpected rates. The validation is only correct when `params.collectionId == RESERVED_ID`, but incorrect for all other collection IDs. The primary risk is that liquidations could be executed at rates lower than the minimum specified by the liquidator (`params.minAPR`), potentially resulting in financial losses. It also means that liquidations with replacement is not accessible for replacement borrowers with no user defined curve set. Since `LiquidateWithReplacement` access is limited to the `KEEPER_ROLE` the main impact of such incorrect validation is reducing their total profit from this activity.

Recommendation: Replace the APR validation in `validateLiquidateWithReplacementParams` to use the same rate calculation as in the execution function:

```
  function validateLiquidateWithReplacementParams(
      State storage state,
      LiquidateWithReplacementParams calldata params
  ) external view {
      // ... existing code ...

      // validate minAPR
-     uint256 borrowAPR = state.getUserDefinedBorrowOfferAPR(params.borrower, tenor);
+     uint256 borrowAPR = state.getBorrowOfferAPR(params.borrower, params.collectionId, params.rateProvider,
↪  tenor);
      if (borrowAPR < params.minAPR) {
          revert Errors.APR_LOWER_THAN_MIN_APR(borrowAPR, params.minAPR);
      }

      // ... remaining code ...
  }
```

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `getBorrowOfferAPR(params.borrower, params.collectionId, params.rateProvider, tenor)` is now used for `params.minAPR` validation.

3.3.5 It can be impossible to migrate from an `ERC4626` vault with fees or imposing slippages

Severity: Medium Risk

Context: NonTransferrableRebasingTokenVault.sol#L437-L440

Description: The protocol's `setVault()` function may revert when used with ERC4626 vaults that have fees on withdrawal or slippage, rendering users unable to switch vaults. This happens because, from the `EIP-4626` specification, `convertToAssets`:

> "MUST NOT be inclusive of any fees" and "MUST NOT reflect slippage"

However, the protocol uses `convertToAssets` in the `balanceOf()` function to calculate the full withdrawal amount transferring between vaults:

```
// In ERC4626Adapter.sol
function balanceOf(address vault, address account) public view returns (uint256) {
    return IERC4626(vault).convertToAssets(tokenVault.sharesOf(account));
}
```

```
// In NonTransferrableRebasingTokenVault.sol
function setVault(address user, address vault) external onlyMarket {
    // ...
    // Uses the fee-excluding balance calculation to determine withdrawal amount
    _transferFrom(vaultOf[user], vault, user, user, balanceOf(user));
    // ...
}
```

This can lead to withdrawal call in the `_transferFrom` function to revert when calling `ERC4626Adapter.withdraw` function , due to requiring more shares than the total user shares:

```
function _transferFrom(address vaultFrom, address vaultTo, address from, address to, uint256 value) private {
    if (value > 0) {
        if (vaultFrom == vaultTo) {
            IAdapter adapter = getWhitelistedVaultAdapter(vaultFrom);
            adapter.transferFrom(vaultFrom, from, to, value);
        } else {
            IAdapter adapterFrom = getWhitelistedVaultAdapter(vaultFrom);
            IAdapter adapterTo = getWhitelistedVaultAdapter(vaultTo);
              // slither-disable-next-line unused-return
            adapterFrom.withdraw(vaultFrom, from, address(adapterTo), value); // <<<
              // slither-disable-next-line unused-return
            adapterTo.deposit(vaultTo, to, value);
        }
    }
}
```

DRAFT

This issue effectively locks users into their current vaults when using ERC4626 implementations with fees or slippage mechanisms, trapping user funds in vaults that might be underperforming or deprecated.

Recommendation: Consider redeeming all the user shares on vault change, i.e. basing the logic on shares amount instead of underlying funds amount in this case.

Size: Fixed in PR 177 (see discussion 1, discussion 2, discussion 3).

Cantina Managed: Fix confirmed, shares balance utilizing `fullWithdraw` function was added and is now used in `SetVault`.

3.4 Low Risk

3.4.1 Incorrect `totalSupply` calculation in `ERC4626Adapter`

Severity: Low Risk

Context: ERC4626Adapter.sol#L44-L46

Description: The `totalSupply` function in the `ERC4626Adapter` incorrectly uses `maxWithdraw()` instead of calculating the total assets owned by the vault:

```
function totalSupply(address vault) external view returns (uint256) {
    return IERC4626(vault).maxWithdraw(address(tokenVault));
}
```

The `maxWithdraw()` function returns the maximum amount that can be withdrawn from the vault at a particular moment, which can be less than the total assets the `tokenVault` actually owns. In many `ERC4626` implementations, there may be assets that are temporarily locked, invested, or subject to withdrawal limitations.

`totalSupply` should represent the full balance of assets the vault owns, not just what's currently withdrawable. The inconsistency with how `balanceOf` is calculated (using `convertToAssets`) breaks the fundamental *ERC20* invariant where the sum of all account balances should be less than or equal to the total supply.

Recommendation: Align the `totalSupply` implementation with how individual balances are calculated:

```
  function totalSupply(address vault) external view returns (uint256) {
-     return IERC4626(vault).maxWithdraw(address(tokenVault));
+     return IERC4626(vault).convertToAssets(IERC4626(vault).balanceOf(address(tokenVault)));
  }
```

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed. The `totalSupply` function now accurately accounts for all vault balances and ensures that the called adapter is the whitelisted adapter for the vault.

3.4.2 `balanceOf` of both Adapters may return incorrect values for mismatched vaults

Severity: Low Risk

Context: ERC4626Adapter.sol#L49-L51, AaveAdapter.sol#L61-L63

Description: The `balanceOf` function in both the `ERC4626Adapter` and `AaveAdapter` can be called directly to retrieve a user's balance for a specific vault:

```
// ERC4626Adapter.sol
function balanceOf(address vault, address account) public view returns (uint256) {
    return IERC4626(vault).convertToAssets(tokenVault.sharesOf(account));
}

// AaveAdapter.sol
function balanceOf(address vault, address account) public view returns (uint256) {
    return _unscale(vault, tokenVault.sharesOf(account));
}
```

The core issue is that these functions use the user's share amount from the `tokenVault` but don't verify if those shares correspond to the vault being queried. In the Size protocol, users have shares in specific vaults tracked by `tokenVault.sharesOf(account)` and `tokenVault.vaultOf(account)`, but the adapter's `balanceOf` function doesn't check this relationship.

DRAFT

Example scenario:

1. User Alice has 100 shares in `vaultA` (tracked by `tokenVault.sharesOf(Alice) == 100` and `tokenVault.vaultOf(Alice) == vaultA`).
2. Alice has no shares in `vaultB`.
3. If a third-party integration calls `adapter.balanceOf(vaultB, Alice)`, it will receive:

- `vaultB.convertToAssets(100)`.

4. This incorrectly suggests Alice has tokens in `vaultB` when she actually has none.

The protocol itself is not affected by this issue because when the protocol interacts with adapters, it always passes the correct vault corresponding to the user (via `vaultOf[account]`). However, third-party integrations could receive incorrect balance report.

Recommendation: Modify the `balanceOf` function in both adapters to verify that the requested vault matches the user's actual vault, and that the adapter is correctly whitelisted for the vault.

Size: Fixed in PR 177 (see discussion 1, discussion 2).

Cantina Managed: Fix confirmed, now the `balanceOf` functions in both adapters ensure that the vault is the user configured vault and it is whitelisted for the Adapter.

3.4.3 Arbitrageable APR configuration is possible in `CopyLimitOrders`

Severity: Low Risk

Context: CollectionsManagerCuratorActions.sol#L84-L97, CollectionManager.sol#L1-L50.

Description: Collection curators or users can create arbitrageable configurations in copy limit orders by setting up incompatible APR ranges between borrow and loan offers. Specifically, when:

- `copyBorrowOfferConfigs[i].minAPR > copyLoanOfferConfigs[i].maxAPR` AND tenor ranges overlap (`copyBorrowOfferConfigs[i].maxTenor >= copyLoanOfferConfigs[i].minTenor && copyLoanOfferConfigs[i].maxTenor >= copyBorrowOfferConfigs[i].minTenor`).

  This combination creates a guaranteed arbitrage opportunity where attackers can borrow at `copyLoanOfferConfigs[i].maxAPR` (or lower) and immediately lend at `copyBorrowOfferConfigs[i].minAPR` (or higher), earning risk-free profit.

  Users without their own `CopyLimitOrderConfig` who are subscribed to a collection with this misconfiguration are particularly vulnerable, as they automatically inherit these exploitable settings.

  The protocol currently validates that each individual configuration's min/max values are consistent (e.g., minAPR maxAPR), but it doesn't check for cross-configuration arbitrage opportunities:

  ```
  // Current validation in setCollectionMarketConfigs (CollectionManager.sol)
  if (copyLoanOfferConfigs[i].minAPR > copyLoanOfferConfigs[i].maxAPR) {
      revert Errors.INVALID_APR_RANGE(copyLoanOfferConfigs[i].minAPR, copyLoanOfferConfigs[i].maxAPR);
  }
  if (copyBorrowOfferConfigs[i].minAPR > copyBorrowOfferConfigs[i].maxAPR) {
      revert Errors.INVALID_APR_RANGE(copyBorrowOfferConfigs[i].minAPR, copyBorrowOfferConfigs[i].maxAPR);
  }
  ```

Recommendation: Add validation in both collection-level and user-level configuration methods to detect and prevent arbitrageable setups:

```
// Add this check in setCollectionMarketConfigs and when users set their configs
if (
    copyBorrowOfferConfigs[i].minAPR > copyLoanOfferConfigs[i].maxAPR &&
    copyBorrowOfferConfigs[i].maxTenor >= copyLoanOfferConfigs[i].minTenor &&
    copyLoanOfferConfigs[i].maxTenor >= copyBorrowOfferConfigs[i].minTenor
) {
    revert Errors.ARBITRAGEABLE_CONFIGURATION(
        copyLoanOfferConfigs[i].maxAPR,
        copyBorrowOfferConfigs[i].minAPR
    );
}
```

DRAFT

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `validateCopyLimitOrderConfigs()` was updated with the control for the state described.

3.4.4 `SetUserConfiguration` events can be inconsistent with actual state

Severity: Low Risk

Context: NonTransferrableRebasingTokenVault.sol#L260-L266

Description: The `setVault` function silently ignores invalid vault addresses without feedback. When `vault != address(0) && !vaultToIdMap.contains(vault)` the function exits without updates. This causes upstream events to contain incorrect information, as `executeSetUserConfiguration()` emits events with the requested vault address rather than the actual vault that was set:

```
function executeSetUserConfiguration(
    State storage state,
    SetUserConfigurationOnBehalfOfParams memory externalParams
) external {
    // ...
    emit Events.SetUserConfiguration(
        msg.sender,
        onBehalfOf,
        params.vault,
        params.openingLimitBorrowCR,
        params.allCreditPositionsForSaleDisabled,
        params.creditPositionIdsForSale,
        params.creditPositionIds
    );
}
```

Recommendation: Return the current `vaultOf[user]` when `vault == address(0)` and revert when provided with invalid addresses. This ensures events emitted upstream correctly reflect actual vault changes.

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `setVault` function now reverts when `!vaultToIdMap.contains(vault)`, it's either valid update or revert.

DRAFT

3.5 Informational

3.5.1 Events emitted regardless of set operations results

Severity: Informational

Context: CollectionsManagerCuratorActions.sol#L134-L135, CollectionsManagerCuratorActions.sol#L150-L151, NonTransferrableRebasingTokenVault.sol#L401-L404

Description: A number of functions emit events after adding or removing items from sets regardless of whether the operation actually modified the set. The `add()` and `remove()` functions from `EnumerableSet` return a boolean value indicating whether the set was actually changed, but this return value is ignored.

- In `CollectionsManagerCuratorActions`:

```
collections[collectionId][market].rateProviders.add(rateProviders[i]);
emit AddRateProviderToMarket(collectionId, address(market), rateProviders[i]);
```

```
collections[collectionId][market].rateProviders.remove(rateProviders[i]);
emit RemoveRateProviderFromMarket(collectionId, address(market), rateProviders[i]);
```

- Similarly in `NonTransferrableRebasingTokenVault` function:

```
function _removeAdapter(bytes32 id) private {
    address adapter = IdToAdapterMap.get(id);
    IdToAdapterMap.remove(id);
    adapterToIdMap.remove(adapter);
    emit AdapterRemoved(id);
}
```

This could lead to misleading event logs where events indicate changes that didn't actually occur, complicating off-chain monitoring and state tracking.

Recommendation: Consider emitting events only when the set is actually modified by checking the return value of the add/remove operations.

Size: Fixed in PR 177 (see discussion 1, discussion 2, discussion 3, discussion 4).

Cantina Managed: Fix confirmed, now events are emitted conditionally on the corresponding set operation success.

3.5.2 Any rate provider can block several market operations for subscribed users

Severity: Informational

Context: CollectionsManagerView.sol#L174

Description: When a rate provider within a collection sets `borrowOfferApr >= lendOfferApr`, it will block several market operations for all users subscribed to a collection containing this rate provider. This effectively prevents:

1. `sellCreditMarket`.
2. `buyCreditMarket`.
3. `LiquidateWithReplacement`.

While this constraint is part of the design and trust assumptions for rate providers, it creates a situation where a single rate provider in any collection can impact the available operations for all its subscribers, so users should be extra careful when subscribing to collections.

Size: Acknowledged in PR 177 (see discussion).

Cantina Managed: Acknowledged in `README.md` under `Collections, curators and rate providers`.

DRAFT

3.5.3 `vaultDust` is accumulated, but cannot be used

Severity: Informational

Context: NonTransferrableRebasingTokenVault.sol#L317-L321

Description: The `vaultDust` mapping and `setVaultDust` function are implemented but not used anywhere . There is no defined workflow for utilizing these dust shares.

```
function setVaultDust(address vault, uint256 dust) public onlyAdapter {
    vaultDust[vault] = dust;
}
```

Keeping unused state variables and functions increases gas costs due to unnecessary storage writes.

Recommendation: Remove the `vaultDust` mapping and `setVaultDust` function from the contract to avoid unnecessary storage writes.

Size: Fixed in PR 177 (see discussion).

Cantina Managed: Fix confirmed, `vaultDust` removed.

3.5.4 `copyLimitOrders` and `copyLimitOrdersOnBehalfOf` function names are misleading

Severity: Informational

Context: Size.sol#L418-L431

Description: The functions `copyLimitOrders` and `copyLimitOrdersOnBehalfOf` have misleading names that don't accurately reflect their actual functionality. These functions don't perform any direct copying of limit orders; instead, they only set configuration parameters such as limits and offsets for the copying process.

```
function copyLimitOrders(CopyLimitOrdersParams calldata params) external payable override(ISize) {
    copyLimitOrdersOnBehalfOf(CopyLimitOrdersOnBehalfOfParams({params: params, onBehalfOf: msg.sender}));
}

function copyLimitOrdersOnBehalfOf(CopyLimitOrdersOnBehalfOfParams memory externalParams)
    public
    payable
    override(ISizeV1_7)
    whenNotPaused
{
    state.validateCopyLimitOrders(externalParams);
    state.executeCopyLimitOrders(externalParams);
}
```

The actual subscription and copying functionality is handled through SizeFactory's `subscribeToCollections()` and `unsubscribeFromCollections()` functions.

Recommendation: Rename these functions to better reflect their purpose, for example `setUserCopyLimitOrdersConfig` and `setUserCopyLimitOrdersConfigOnBehalfOf`.

Size: Fixed in PR 177 (see discussion 1, discussion 2).

Cantina Managed: Fix confirmed, names updated.

3.5.5 Missing functionality for Aave reward claims

Severity: Informational

Context: NonTransferrableRebasingTokenVault.sol#L40

Description: Aave's `ATokens` can accrue rewards over time for holders, and `NonTransferrableRebasingTokenVault` contract holds Aave `aTokens`, but the current implementation lacks functionality to claim these rewards. As a result, any rewards accrued to `NonTransferrableRebasingTokenVault` contract remains inaccessible.

Recommendation: Consider implementing functionality to allow claims of Aave rewards.

Size: We'll likely not implement this straight away unless there's user demand.

Cantina Managed: Acknowledged.

DRAFT