Securitize Vault V2 and RWASegWrap Audit Report

Prepared by Cyfrin

Version 2.2

Lead Auditors

Hans

Rodion

August 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

4.1 System Overview

The Securitize RWA Segregated Vault system is a Phase 2 evolution that solves issues from Phase 1 by implementing a hybrid ERC-20/ERC-1155 like architecture. The system creates individual ERC-4626 compliant vaults per investor while maintaining DeFi fungibility through a wrapper orchestration layer.

4.1.1 Key Features

- Segregated Collateralization: Each investor gets their own vault, solving tracking and liquidation issues
- Hybrid Token Model: ERC-20 tokens for investor compatibility + ERC-1155 family tracking for DeFi protocols
- Dynamic Vault Deployment: Vaults are auto-deployed on first investor deposit
- Dual Liquidation Modes: Direct DSToken transfers or redemption to stablecoins
- Role-Based Access Control: Granular permissions across wrapper, deployer, and vault levels

4.1.2 Component Relationships

1. RWASegWrap (Wrapper Layer)
   - Purpose: Orchestrates individual investor vaults, provides ERC-1155 interface for DeFi
   - Key Functions: `deposit()`, `redeem()`, `liquidate()`
   - State Variables: `vaults[]`, `vaultIds[]`, `latestVaultId`, ...
2. VaultDeployer (Factory Layer)
   - Purpose: Factory pattern for deploying individual vaults using beacon proxy
   - Key Functions: `deploy()`

- State Variables: `upgradeableBeacon`, `admin` (VaultDeployer admin)

3. SegregatedVault/SecuritizeVaultV2 (Vault Layer)
   - Purpose: Individual ERC-4626 compliant vaults per investor
   - Key Functions: `deposit()`, `redeem()`, `liquidate()`
   - State Variables: `asset`, `liquidationToken`

4.1.3 Main Actors

1. Investors: Own individual vaults, deposit/redeem RWA assets
2. Administrators: Manage system configuration, upgrades, pausing
3. Fee Collectors: Receive deposit fees
4. NAV Providers: Supply pricing data for asset valuation
5. Liquidation Agents: Execute position liquidations

4.1.4 Key Processes

Deposit Process:

1. Investor calls `deposit()` on wrapper
2. If first deposit: Deploy new vault via VaultDeployer
3. Calculate and collect fees via FeeManager
4. Transfer net assets to vault
5. Vault mints shares to investor
6. Wrapper tracks balance via ERC-1155 interface

Liquidation Process:

1. DeFi protocol calls `liquidate()` with share amount and vault ID
2. Wrapper validates liquidator has LIQUIDATOR role for that vault
3. Vault converts shares to assets using NAV rate
4. Burn liquidator's shares
5. Transfer assets (direct mode) or convert via redemption contract (redemption mode)

Vault Management:

1. Wrapper acts as AGGREGATOR for all vaults it deploys
2. Investors are granted OPERATOR role on their vault
3. DeFi protocols are granted LIQUIDATOR role per vault basis
4. VaultDeployer admin becomes admin of all deployed vaults

4.2 System Invariants

4.2.1 Mathematical/Economic Invariants

1. Vault Asset Conservation:

```
vault.totalAssets() >= vault.totalSupply() * vault.convertToAssets(1)
```

2. Share-Asset Conversion Consistency:

```
convertToShares(convertToAssets(shares)) <= shares (allowing for rounding)
convertToAssets(convertToShares(assets)) <= assets (allowing for rounding)
```

4.2.2 Balance Equations

1. Wrapper Balance Consistency:

```
(balanceOf(investor, vaultId)) = totalShares in vault[vaultId]
```

2. Liquidation Asset Conservation:

```
If redemption mode: liquidationAsset != asset
If direct mode: liquidationAsset == asset
```

4.2.3 State Conditions

1. Vault ID Uniqueness: Each vault has a unique auto-incremented ID
2. Investor-Vault Mapping: One vault per investor (by wallet or investor ID)
3. Role Inheritance: Aggregator role on vaults is always held by the wrapper
4. Pause State Consistency: When wrapper is paused, critical operations are blocked

4.3 Security Considerations

4.3.1 Access Control Analysis

1. RWASegWrap Level:
   - `DEFAULT_ADMIN_ROLE`: Pause/unpause the wrapper, manage vaults, update URIs
2. VaultDeployer Level:
   - `DEFAULT_ADMIN_ROLE`: Add aggregators, update NAV providers, change admin
   - `AGGREGATOR_ROLE`: Deploy new vaults (typically held by RWASegWrap)
3. Beacon Level:
   - `OWNER`: Upgrade the beacon to a new implementation
4. Individual Vault Level:
   - `DEFAULT_ADMIN_ROLE`: Grant/revoke all roles (typically VaultDeployer admin)
   - `OPERATOR_ROLE`: Deposit, redeem, mint (typically investors)
   - `LIQUIDATOR_ROLE`: Liquidate positions (DeFi protocols)
   - `AGGREGATOR_ROLE`: Internal operations (RWASegWrap)

4.3.2 External Dependencies

1. NAV Provider Oracle Risk:
   - Must provide a rate to convert assets to shares
2. Redemption Contract Risk:
   - Controls conversion of assets to liquidation tokens
3. Fee Manager Risk:
   - Fee calculation is handled by the fee manager

4.3.3 Upgradeability Patterns

- All contracts (Vault, VaultDeployer, Aggregator) are UUPS upgradable and can be upgraded by their own admin with the `DEFAULT_ADMIN_ROLE`.
- Vault contracts are deployed via BeaconProxy and the beacon owner can upgrade all vaults to a new implementation simultaneously.

5 Audit Scope

The review has been scoped to the following repository and contracts:

1. bc-rwa-seg-wrap-sc
    - RWASegWrap.sol
    - SecuritizeRWASegWrap.sol
    - SegregatedVault.sol
    - VaultDeployer.sol
    - SegregatedVaultDeployer.sol
    - SecritizeVaultDeployer.sol
2. bc-securitize-vault-sc
    - SecuritizeVaultV2.sol

6 Executive Summary

Over the course of 10 days, the Cyfrin team conducted an audit on the Securitize Vault V2 and RWASegWrap smart contracts provided by Securitize. In this period, a total of 16 issues were found.

A medium severity vulnerability was identified in the vault deployment architecture where shared configuration parameters across different asset types lead to incorrect pricing and fee calculations. The `VaultDeployer` and `SecuritizeVaultDeployer` contracts maintain unified `navProvider`, `feeManager`, and `redemptionAddress` configurations that are applied to all deployed vaults regardless of underlying asset type, causing different RWA assets to incorrectly share the same NAV provider and potentially resulting in wrong share calculations and economic losses for users.

The low severity findings include upgradeable contract security issues where implementation contracts lack constructor initializer disabling, allowing potential unauthorized initialization of implementation contracts. Incomplete mapping updates in the `setVault` function create bidirectional mapping inconsistencies between vault addresses and IDs. Unsafe ERC20 operations use direct function calls instead of SafeERC20 wrappers, causing compatibility issues with non-standard tokens like USDT. Transfer function logic incorrectly requires allowance checks even when users transfer their own tokens, violating standard ERC20 practices.

Informational and gas optimization findings encompass incorrect function documentation containing copy-paste errors from other functions, single-step admin transfer patterns creating permanent access loss risks, redundant balance checks and approval operations increasing gas consumption, code duplication in function overrides that only add modifiers, and various naming inconsistencies and missing validation modifiers during contract initialization.

Summary

Project Name	Securitize Vault V2 and RWASegWrap
Repository	bc-securitize-vault-sc
Commit	0c179554f39c...
Fix Commit	5105f03e9550...
Repository 2	bc-rwa-seg-wrap-sc
Commit	2cb4c9d990f9...
Fix Commit	6322e36c86c0...
Audit Timeline	Jul 21st - Aug 1st
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	7
Informational	5
Gas Optimizations	3
Total Issues	16

Summary of Findings

[M-1] Shared configuration parameters across different asset types in vault deployers leads to incorrect pricing and fee calculations	Resolved
[L-1] Missing storage gap in upgradeable parent contract causes storage slot collision risk	Resolved
[L-2] Upgradeable contract initializer not disabled in constructor allows implementation contract initialization	Resolved
[L-3] Incomplete mapping updates in `setVault` function cause vault address inconsistencies	Resolved
[L-4] Unsafe ERC20 operations can cause unexpected failures with non-standard tokens	Resolved
[L-5] Inconsistency between function name and documentation in `getUnderlyingAsset`	Resolved
[L-6] Incorrect allowance check in transfer functions prevents users from transferring their own tokens	Resolved
[L-7] Missing `notEmptyURI` modifier during initialization	Resolved
[I-1] Incorrect function documentation	Resolved

[I-2] Single-step admin transfer creates risk of permanent loss of administrative control	Acknowledged
[I-3] Redundant balance check in safeTransferFrom before calling underlying transfer function	Resolved
[I-4] `setApprovalForAll()` function is double initialized in the child contract	Resolved
[I-5] Code duplication in function overrides that only add modifiers	Resolved
[G-1] Unnecessary `_msgSender()` call in `_resolveVaultId` when `caller` parameter is available	Resolved
[G-2] Unnecessary gas consumption in deposit function due to redundant maximum deposit check	Resolved
[G-3] Gas inefficient approval pattern in liquidation function leads to unnecessary gas consumption	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Shared configuration parameters across different asset types in vault deployers leads to incorrect pricing and fee calculations

Description: The `VaultDeployer` and `SecuritizeVaultDeployer` contracts maintain shared configuration parameters (`navProvider`, `feeManager`, `redemptionAddress`) that are applied to all deployed vaults regardless of their underlying asset type. When `SegregatedVaultDeployer::deploy()` or `SecuritizeVaultDeployer::deploy()` is called with different `assetToken` and `liquidationToken` parameters to support various vault types, the same `navProvider` is used across all deployments. This creates a critical architectural flaw because different RWA assets require asset-specific NAV providers for accurate valuation.

In `SecuritizeVaultV2`, the `navProvider.rate()` is extensively used in critical functions like `_convertToShares()`, `_convertToAssets()`, and `getShareValue()` to determine share-to-asset conversion ratios. When vaults for different assets (e.g., real estate tokens vs commodity tokens) share the same NAV provider, the pricing calculations become incorrect for at least one of the asset types.

Similar issues exist with:

- `SecuritizeVaultDeployer.feeManager` - applies the same fee logic to all asset types
- `SecuritizeVaultDeployer.redemptionAddress` - uses the same redemption contract for different assets that may require different redemption mechanisms

Note that `SegregatedVault` does not use `navProvider` in its calculations, so it is not directly affected by this issue, but the architecture problem persists in the deployment pattern.

Impact: Users depositing assets into vaults with incorrect NAV providers will receive wrong share amounts, leading to economic losses and potential exploitation opportunities where attackers can deposit low-value assets but receive shares calculated using high-value asset NAV rates.

```
// In SecuritizeVaultDeployer::deploy()
BeaconProxy proxy = new BeaconProxy(
    upgradeableBeacon,
    abi.encodeWithSelector(
        SecuritizeVaultV2(payable(address(0))).initializeV2.selector,
        name,
        symbol,
        assetToken,       // Different per deployment
        redemptionAddress, // Same for all deployments - ISSUE
        liquidationToken,
        navProvider,      // Same for all deployments - ISSUE
        feeManager        // Same for all deployments - ISSUE
    )
);
```

Recommended Mitigation: We understand that these parameters are meant to be managed by only the admin, and that is why it's managed by the contract instead of allowing users to specify in the deploy function. We recommend the team consider either of below two solutions.

1. Modify the vault deployer architecture to support asset-specific configurations. Below is an example implementation.

```
+ mapping(address => address) public assetNavProviders;
+ mapping(address => address) public assetFeeManagers;
+ mapping(address => address) public assetRedemptionAddresses;

+ function setAssetConfiguration(
+     address assetToken,
+     address navProvider,
+     address feeManager,
+     address redemptionAddress
```

```
+ ) external onlyRole(DEFAULT_ADMIN_ROLE) {
+     assetNavProviders[assetToken] = navProvider;
+     assetFeeManagers[assetToken] = feeManager;
+     assetRedemptionAddresses[assetToken] = redemptionAddress;
+ }
```

2. If the intention is to have one deployer for every pair of asset token ad liquid token, store the asset token address and liquid token address in the deployer with the nav provider together and remove the `assetToken` adn `liquidToken` parameters from the deploy function.

Securitize: Fixed in commit 05044b.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Missing storage gap in upgradeable parent contract causes storage slot collision risk

Description: The `VaultDeployer` abstract contract is designed to be upgradeable and inherited by child contracts `SegregatedVaultDeployer` and `SecuritizeVaultDeployer`. However, `VaultDeployer` lacks storage gap variables to reserve space for future upgrades.

Currently, `VaultDeployer` declares three state variables:

- `address public navProvider`
- `address internal admin`
- `address public upgradeableBeacon`

The child contracts add their own state variables after the parent's storage:

- `SecuritizeVaultDeployer` adds `redemptionAddress` and `feeManager`
- `SegregatedVaultDeployer` currently adds no additional state variables

In upgradeable contracts, when a parent contract adds new state variables in future versions, those variables are allocated to storage slots immediately following the existing parent variables. This will overwrite the storage slots currently occupied by child contract variables, leading to storage collision and data corruption.

While `BaseContract` (the grandparent) properly implements storage gaps with `uint256[50] private __gap`, the intermediate `VaultDeployer` contract breaks this pattern by not reserving space for its own future expansion.

Impact: If future versions of `VaultDeployer` add new state variables, they will overwrite child contract storage slots causing data corruption and potentially rendering deployed contracts unusable.

Recommended Mitigation: Add storage gap variables to `VaultDeployer` contract to reserve space for future upgrades. Choose either traditional storage gaps or ERC-7201 namespaced storage:

Option 1: Traditional Storage Gap

```
abstract contract VaultDeployer is IVaultDeployer, BaseContract {
    bytes32 public constant AGGREGATOR_ROLE = keccak256("AGGREGATOR_ROLE");

    address public navProvider;
    address internal admin;
    address public upgradeableBeacon;

+   // Reserve storage slots for future VaultDeployer upgrades
+   uint256[47] private __gap;

    // ... rest of contract
}
```

Option 2: ERC-7201 Namespaced Storage

```
abstract contract VaultDeployer is IVaultDeployer, BaseContract {
    /// @custom:storage-location erc7201:securitize.storage.VaultDeployer
    struct VaultDeployerStorage {
        address navProvider;
        address admin;
        address upgradeableBeacon;
    }

    // keccak256(abi.encode(uint256(keccak256("securitize.storage.VaultDeployer")) - 1)) &
    ↪  ~bytes32(uint256(0xff))
    bytes32 private constant VAULT_DEPLOYER_STORAGE_LOCATION = 0x...;

    function _getVaultDeployerStorage() private pure returns (VaultDeployerStorage storage $) {
        assembly {
```

```
            $.slot := VAULT_DEPLOYER_STORAGE_LOCATION
        }
    }

    // Update all variable access to use the storage struct
    // ... rest of contract
}
```

Securitize: Fixed in commit 3048c3.

Cyfrin: Verified.

7.2.2 Upgradeable contract initializer not disabled in constructor allows implementation contract initialization

Description: The `SegregatedVault` contract is designed as an upgradeable contract using the UUPS (Universal Upgradeable Proxy Standard) pattern, inheriting from `BaseContract` which extends `UUPSUpgradeable`. However, the implementation contract fails to disable its initializer in the constructor, creating a security vulnerability.

In UUPS upgradeable contracts, the implementation contract should have its initializer disabled in the constructor to prevent direct initialization of the implementation contract itself. Without this protection, an attacker could potentially call `SegregatedVault::initialize` directly on the implementation contract, granting themselves admin privileges and potentially disrupting the intended proxy-based upgrade mechanism.

The `SegregatedVault::initialize` function grants `DEFAULT_ADMIN_ROLE` to `msg.sender`, which would be the attacker if called directly on the implementation. This could allow unauthorized access to admin-only functions like role management and contract upgrades.

Similar issues exist in other upgradeable contracts in the codebase:

- `SecuritizeVaultV2` in the bc-securitize-vault-sc module lacks constructor initializer disabling
- `SecuritizeVault` in the bc-securitize-vault-sc module lacks constructor initializer disabling
- `RWASegWrap` lacks constructor initializer disabling
- `SecuritizeRWASegWrap` lacks constructor initializer disabling

All these contracts follow the same pattern of inheriting from `BaseContract` and implementing UUPS upgradeability without properly securing the implementation contract.

Impact: An attacker could initialize the implementation contract directly to gain unauthorized admin privileges and potentially compromise the upgrade mechanism for all proxy instances.

Recommended Mitigation: Add a constructor that disables the initializer to prevent direct initialization of the implementation contract:

```
contract SegregatedVault is ERC4626Upgradeable, ISegregatedVault, IVaultAccessControl, BaseContract {

+   /// @custom:oz-upgrades-unsafe-allow constructor
+   constructor() {
+       _disableInitializers();
+   }
```

Apply the same fix to other affected upgradeable contracts: `SecuritizeVaultV2`, `SecuritizeVault`, `RWASegWrap`, and `SecuritizeRWASegWrap`.

Securitize: Fixed in commits 1261ec and 1a2f4c.

Cyfrin: Verified.

7.2.3 Incomplete mapping updates in `setVault` function cause vault address inconsistencies

Description: The `RWASegWrap::setVault` function allows admins to update the vault address for a specific vault ID, but it fails to properly maintain the bidirectional mapping between vault addresses and vault IDs. The function only updates `vaults[id] = vault` but does not update the `vaultIds` mapping, which should map the new vault address to the vault ID and clear the mapping for the old vault address. This creates inconsistencies in the contract's state where the old vault address remains mapped to the vault ID in the `vaultIds` mapping, while the new vault address is not recognized by the system. The `vaultIds` mapping is critical for vault validation in functions like `isValidVault` and `getAssetId`, and is also used in `_addVault` to prevent duplicate vault registrations. The same issue exists in the `SecuritizeRWASegWrap` contract, which inherits from `RWASegWrap` and uses the same `setVault` implementation.

Impact: The incomplete mapping updates can cause vault operations to fail or behave unexpectedly, as the new vault address will not be recognized as valid by the system, while the old vault address may still appear valid even though it's no longer active.

Recommended Mitigation: Update the `setVault` function to properly maintain both mappings:

```
function setVault(
    address vault,
    uint256 id
) public virtual override onlyRole(DEFAULT_ADMIN_ROLE) idNotZero(id) recognizedVault(id)
↪   addressNotZero(vault) {
    address oldVault = vaults[id];
    address investorWallet = vaultIdOwnerWallets[id];
    vaults[id] = vault;
+   delete vaultIds[oldVault];
+   vaultIds[vault] = id;
    emit VaultUpdated(oldVault, vault, id, investorWallet);
}
```

Securitize: Fixed in commit 468bae.

Cyfrin: Verified.

7.2.4 Unsafe ERC20 operations can cause unexpected failures with non-standard tokens

Description: The protocol uses direct `IERC20.transferFrom` and `IERC20.approve` function calls instead of OpenZeppelin's `SafeERC20` library wrappers. This creates compatibility issues with tokens that do not return boolean values or have non-standard implementations.

The primary occurrence is in `RWASegWrap::_pullAndApprove`, which is a critical internal function used during deposit and mint operations. When users call `depositById` or `mintById`, the wrapper contract attempts to transfer assets from the caller and approve the vault for spending. With tokens like USDT, the `approve` function may fail when trying to approve from a non-zero allowance to another non-zero value, or the `transferFrom` may not return a boolean value, causing the transaction to revert unexpectedly.

Similar unsafe operations are found in:

- `SecuritizeVault::liquidate` - uses `IERC20Metadata(asset()).approve(address(redemption), assets)`
- `SecuritizeVaultV2::_liquidateTo` - uses `IERC20Metadata(asset()).approve(address(redemption), assets)`

These functions handle core protocol operations, including asset deposits, share minting, and liquidations, making them critical for normal protocol functionality.

Impact: Users may be unable to deposit assets or liquidate shares when using tokens with non-standard ERC20 implementations, leading to failed transactions and degraded user experience.

Recommended Mitigation: Replace all direct `IERC20` calls with `SafeERC20` equivalents. Add the SafeERC20 import to `RWASegWrap.sol` and update the unsafe operations:

```
// Add import to RWASegWrap.sol
+import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract RWASegWrap is ... {
+   using SafeERC20 for IERC20;

    function _pullAndApprove(address caller, uint256 assets, uint256 vaultId) internal {
        ISegregatedVault vault = ISegregatedVault(vaults[vaultId]);
-       bool success = IERC20(asset).transferFrom(caller, address(this), assets);
-       if (!success) {
-           revert AssetTransferFailed();
-       }
-       success = IERC20(asset).approve(address(vault), assets);
-       if (!success) {
-           revert AssetApprovalFailed();
-       }
+       IERC20(asset).safeTransferFrom(caller, address(this), assets);
+       IERC20(asset).forceApprove(address(vault), assets);
    }
}
```

For the SecuritizeVault contracts:

```
// In SecuritizeVault.liquidate
-IERC20Metadata(asset()).approve(address(redemption), assets);
+IERC20Metadata(asset()).forceApprove(address(redemption), assets);

// In SecuritizeVaultV2._liquidateTo
-bool success = IERC20Metadata(asset()).approve(address(redemption), assets);
-if (!success) {
-    revert AssetApprovalFailed();
-}
+IERC20Metadata(asset()).forceApprove(address(redemption), assets);
```

Securitize: Fixed in commit b64b27.

Cyfrin: Verified.

7.2.5 Inconsistency between function name and documentation in `getUnderlyingAsset`

Description: The `RWASegWrap::getUnderlyingAsset` function has inconsistent naming and documentation. The function name `getUnderlyingAsset` suggests it should return the underlying asset address (the RWA token), but the interface documentation states "Gets the vault address for a given asset ID", indicating it should return the vault contract address.

The current implementation returns `vaults[id]`, which is the vault contract address, aligning with the documentation but contradicting the function name. This creates confusion in the codebase where developers might expect the function to return the underlying asset address based on its name, when it actually returns the vault address.

```
// Interface documentation says: "Gets the vault address for a given asset ID"
// But the function name suggests it returns the underlying asset
function getUnderlyingAsset(uint256 id) external view returns (address);

// Implementation returns vault address, matching documentation but not the name
function getUnderlyingAsset(uint256 id) public virtual view override returns (address) {
    return vaults[id]; // Returns vault contract address
}
```

In the ERC4626 vault architecture, there's a clear distinction between:

- Vault address: The contract address of the ERC4626 vault (share token contract)

- Underlying asset address: The address of the token that the vault accepts as deposits (obtainable via `vault.asset()`)

The same issue exists in `SecuritizeRWASegWrap` as it inherits from `RWASegWrap` and doesn't override this function.

Impact: The inconsistency between the function name and documentation creates confusion about the function's intended behavior, potentially leading to integration errors where developers expect the underlying asset address but receive the vault address.

Recommended Mitigation: Choose one of the following approaches to resolve the inconsistency:

Option 1 - Rename function to match documentation:

```
/**
 * @notice Gets the vault address for a given asset ID.
 * @dev Returns zero address if not found.
 */
- function getUnderlyingAsset(uint256 id) external view returns (address);
+ function getVaultAddress(uint256 id) external view returns (address);
```

Option 2 - Update documentation to match function name and fix implementation:

```
/**
- * @notice Gets the vault address for a given asset ID.
+ * @notice Gets the underlying asset address for a given asset ID.
 * @dev Returns zero address if not found.
 */
function getUnderlyingAsset(uint256 id) external view returns (address);

// And update implementation:
function getUnderlyingAsset(uint256 id) public virtual view override returns (address) {
-   return vaults[id];
+   return vaults[id] != address(0) ? IERC4626(vaults[id]).asset() : address(0);
}
```

Securitize: Fixed in 23f879.

Cyfrin: Verified.

7.2.6 Incorrect allowance check in transfer functions prevents users from transferring their own tokens

Description: The protocol implements transfer functions in both `RWASegWrap` and `SecuritizeRWASegWrap` contracts that violate standard token transfer practices by always checking allowance, even when users are transferring their own tokens.

In `RWASegWrap::safeTransferFrom()`, lines 448-450 always check the allowance:

```
uint256 currentAllowance = allowance(from, _msgSender(), id);
if (currentAllowance < value) {
    revert ERC1155MissingApprovalForAll(_msgSender(), from);
}
```

Similarly, `RWASegWrap::transferFrom()` calls `ISegregatedVault(vaults[id]).internalTransferFrom(from, to, _msgSender(), value)` which internally calls `_spendAllowance(from, spender, value)` where the spender is `_msgSender()`, forcing an allowance check even for self-transfers.

According to standard token implementation practices, allowance checks should only occur when the sender is not the token owner. The correct behavior is demonstrated in OpenZeppelin's implementation, where approval is only checked when `from != sender`:

```
// OpenZeppelin's ERC1155Upgradeable
address sender = _msgSender();
if (from != sender && !isApprovedForAll(from, sender)) {
    revert ERC1155MissingApprovalForAll(sender, from);
```

```
}
```

`SecuritizeRWASegWrap` inherits from `RWASegWrap` and therefore has the same non-compliant behavior for both transfer functions.

Impact: Users cannot transfer their own tokens without first calling approve to grant themselves allowance, creating unnecessary friction and violating standard token transfer expectations.

Recommended Mitigation: Modify the allowance check to only occur when the sender is not the token owner, following standard token implementation practices:

```
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 value,
    bytes memory data
) public virtual override whenNotPaused idNotZero(id) recognizedVault(id) {
    if (to == address(0)) {
        revert ERC1155InvalidReceiver(address(0));
    }
    if (from == address(0)) {
        revert ERC1155InvalidSender(address(0));
    }
    uint256 vaultId = getVaultId(from);
    if (id != vaultId) {
        revert InvestorVaultMismatch(id, from);
    }

-   uint256 currentAllowance = allowance(from, _msgSender(), id);
-   if (currentAllowance < value) {
-       revert ERC1155MissingApprovalForAll(_msgSender(), from);
-   }
+   address sender = _msgSender();
+   if (from != sender) {
+       uint256 currentAllowance = allowance(from, sender, id);
+       if (currentAllowance < value) {
+           revert ERC1155MissingApprovalForAll(sender, from);
+       }
+   }

    uint256 currentBalance = balanceOf(from, id);
    if (currentBalance < value) {
        revert ERC1155InsufficientBalance(from, currentBalance, value, id);
    }

    emit TransferSingle(_msgSender(), from, to, id, value);
    ISegregatedVault(vaults[id]).internalTransferFrom(from, to, _msgSender(), value);
    ERC1155Utils.checkOnERC1155Received(_msgSender(), from, to, id, value, data);
}
```

The `internalTransferFrom` function should also be updated to conditionally call `_spendAllowance` only when `from != spender`.

Securitize: Fixed in commits dd0035 and 4f0722.

Cyfrin: Verified.

7.2.7 Missing `notEmptyURI` modifier during initialization

Description: Currently, there is no `notEmptyURI` modifier present in the `initialize()` function that checks for the empty URI and, if it's empty, reverts the transaction:

```
//RWASegWrap.sol#L98-103
    modifier notEmptyUri(string memory newUri) {
        if (bytes(newUri).length == 0) {
            revert EmptyUriInvalid();
        }
        _;
    }
```

```
//RWASegWrap.sol#L131-147
   function initialize(
        string memory baseNameArg,
        string memory baseSymbolArg,
        string memory uriArg,
        address liquidationAssetArg,
        address assetArg,
        address vaultDeployerArg
    )
    public
    virtual
    override
    onlyProxy
    initializer
    addressNotZero(liquidationAssetArg)
    addressNotZero(assetArg)
    addressNotZero(vaultDeployerArg)
    {
```

Impact: Insufficient validation, `projectURI` may not be set during the initialization process.

Recommended Mitigation: Add the `notEmptyUri` modifier that checks for the empty URI.

Securitize Fixed in commit 0946fb.

Cyfrin: Verified.

7.3 Informational

7.3.1 Incorrect function documentation

Description: The `SegregatedVault::addAggregator` function contains incorrect documentation that mentions "Operators" instead of "Aggregators" in both the function description and parameter documentation. The comment states "Operators can deposit and redeem. Emits AggregatorAdded event" when it should describe aggregator capabilities, and the parameter description says "The address to which the Operator role will be granted" when it should reference the Aggregator role.

This appears to be a copy-paste error from the `addOperator` function documentation. The same issue exists in `SecuritizeVaultV2::addAggregator`.

Additionally, there is another inconsistency in `SegregatedVault::revokeAggregator` where the comment states "Revokes the Operator role from an account. Emits a OperatorRevoked event" when it should reference the Aggregator role and AggregatorRevoked event.

Impact: Incorrect documentation may confuse developers and auditors about the intended role permissions and capabilities, potentially leading to integration errors or security misunderstandings.

Recommended Mitigation: Update the documentation to correctly describe aggregator role capabilities and parameters:

```
/**
 * @dev Grants the aggregator role to an account.
- * Operators can deposit and redeem. Emits AggregatorAdded event
+ * Aggregators can manage vault operations and perform deposits/redeems on behalf of the protocol.
↪  Emits AggregatorAdded event
 *
- * @param account The address to which the Operator role will be granted.
+ * @param account The address to which the Aggregator role will be granted.
 */
function addAggregator(address account) external addressNotZero(account) onlyRole(DEFAULT_ADMIN_ROLE) {
    _grantRole(AGGREGATOR_ROLE, account);
    emit AggregatorAdded(account);
}
```

Apply similar fixes to `SegregatedVault::revokeAggregator` and `SecuritizeVaultV2::addAggregator` functions.

Securitize: Fixed in commits 0e881e and 402daa.

Cyfrin: Verified.

7.3.2 Single-step admin transfer creates risk of permanent loss of administrative control

Description: The `SegregatedVault::changeAdmin` function implements a single-step admin transfer mechanism that immediately grants admin privileges to the new address and revokes them from the current admin in a single transaction. This pattern creates a risk where administrative control can be permanently lost if an incorrect address is provided, as there is no validation that the new admin can actually control the contract.

The same unsafe single-step admin transfer pattern is implemented across multiple contracts in the codebase:

- `VaultDeployer::changeAdmin` performs immediate admin role transfer
- `SecuritizeVaultV2::changeAdmin` uses the same single-step approach
- `SecuritizeVault::changeAdmin` also implements immediate transfer

The admin role has extensive privileges including the ability to add/revoke operators, liquidators, and aggregators, as well as control over vault upgrades and configuration. Losing admin access would render these administrative functions permanently inaccessible, potentially freezing protocol operations and preventing emergency responses.

Impact: A typographical error or incorrect address during admin transfer would result in permanent loss of administrative control over the contract, rendering critical management functions inaccessible and potentially requiring expensive redeployment procedures.

Recommended Mitigation: Implement a two-step admin transfer pattern where the new admin must explicitly accept the role:

```
+ address public pendingAdmin;

+ function transferAdmin(address newAdmin) external addressNotZero(newAdmin)
↪  onlyRole(DEFAULT_ADMIN_ROLE) {
+     pendingAdmin = newAdmin;
+     emit AdminTransferStarted(msg.sender, newAdmin);
+ }

+ function acceptAdmin() external {
+     if (msg.sender != pendingAdmin) {
+         revert NotPendingAdmin();
+     }
+     address oldAdmin = msg.sender;
+     _grantRole(DEFAULT_ADMIN_ROLE, pendingAdmin);
+     _revokeRole(DEFAULT_ADMIN_ROLE, oldAdmin);
+     delete pendingAdmin;
+     emit AdminChanged(pendingAdmin);
+ }

- function changeAdmin(address newAdmin) external addressNotZero(newAdmin) onlyRole(DEFAULT_ADMIN_ROLE)
↪  {
-     _grantRole(DEFAULT_ADMIN_ROLE, newAdmin);
-     _revokeRole(DEFAULT_ADMIN_ROLE, msg.sender);
-     emit AdminChanged(newAdmin);
- }
```

Apply similar changes to `VaultDeployer`, `SecuritizeVaultV2`, and `SecuritizeVault` contracts.

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.3.3 Redundant balance check in safeTransferFrom before calling underlying transfer function

Description: The `RWASegWrap::safeTransferFrom()` function performs an unnecessary balance check on lines 452-454 before calling the underlying transfer function:

```
uint256 currentBalance = balanceOf(from, id);
if (currentBalance < value) {
    revert ERC1155InsufficientBalance(from, currentBalance, value, id);
}
```

This balance check is redundant because the subsequent call to `ISegregatedVault(vaults[id]).internalTransferFrom(fro to, _msgSender(), value)` internally calls the ERC20 `_transfer()` function, which already performs the same balance validation. The ERC20 `_update()` function (which `_transfer()` calls) contains the exact same check:

```
uint256 fromBalance = $._balances[from];
if (fromBalance < value) {
    revert ERC20InsufficientBalance(from, fromBalance, value);
}
```

When the balance is insufficient, the ERC20 mechanism will automatically revert with `ERC20InsufficientBalance`, making the wrapper-level balance check redundant.

Impact: The redundant balance check results in unnecessary gas consumption and code complexity without providing additional safety.

Recommended Mitigation: Remove the redundant balance check:

```
function safeTransferFrom(
```

```
    address from,
    address to,
    uint256 id,
    uint256 value,
    bytes memory data
) public virtual override whenNotPaused idNotZero(id) recognizedVault(id) {
    if (to == address(0)) {
        revert ERC1155InvalidReceiver(address(0));
    }
    if (from == address(0)) {
        revert ERC1155InvalidSender(address(0));
    }
    uint256 vaultId = getVaultId(from);
    if (id != vaultId) {
        revert InvestorVaultMismatch(id, from);
    }

    uint256 currentAllowance = allowance(from, _msgSender(), id);
    if (currentAllowance < value) {
        revert ERC1155MissingApprovalForAll(_msgSender(), from);
    }

-   uint256 currentBalance = balanceOf(from, id);
-   if (currentBalance < value) {
-       revert ERC1155InsufficientBalance(from, currentBalance, value, id);
-   }

    emit TransferSingle(_msgSender(), from, to, id, value);
    ISegregatedVault(vaults[id]).internalTransferFrom(from, to, _msgSender(), value);
    ERC1155Utils.checkOnERC1155Received(_msgSender(), from, to, id, value, data);
}
```

Securitize: Fixed in commit 13955e.

Cyfrin: Verified.

7.3.4 `setApprovalForAll()` function is double initialized in the child contract

Description: At the moment, there is a double function initialization of the ERC1155 `setApprovalForAll()` function both in the parent `RWASegWrap` and the child `SecuritizeRWASegWrap` contracts:

```
    // @inheritdoc IERC1155
    function setApprovalForAll(address, bool) public virtual pure override {
        revert FeatureNotSupported();
    }
```

```
    // @inheritdoc IERC1155
    function setApprovalForAll(address, bool) public virtual pure override {
        revert FeatureNotSupported();
    }
```

Impact: Increased deployment costs.

Recommended Mitigation: Remove `setApprovalForAll()` implementation from the `SecuritizeRWASegWrap`.

Securitize Fixed in commit b78f30.

Cyfrin: Verified.

7.3.5 Code duplication in function overrides that only add modifiers

Description: The `SecuritizeRWASegWrap` contract overrides four functions from its parent `RWASegWrap` contract (`deposit`, `redeem`, `redeemById`, and `depositById`) with identical implementation logic, only adding the `receiverIsSender` modifier. Instead of duplicating the entire function body, these functions should call their parent implementations using `super` after applying the additional modifier.

The current implementations in `SecuritizeRWASegWrap::deposit`, `SecuritizeRWASegWrap::redeem`, `SecuritizeRWASegWrap::redeemById`, and `SecuritizeRWASegWrap::depositById` repeat the exact same business logic as their parent functions in `RWASegWrap`, including variable declarations, vault resolution, validation checks, and internal function calls.

```
// Current implementation in SecuritizeRWASegWrap
function deposit(uint256 assets, address receiver)
    public override whenNotPaused amountNotZero(assets) addressNotZero(receiver)
    ↪  receiverIsSender(receiver)
    returns (uint256) {
    address caller = _msgSender();
    uint256 vaultId = getVaultId(caller);
    if (vaultId == 0) {
        vaultId = ++latestVaultId;
        vault = _deployVault(vaultId);
        _addVault(address(vault), vaultId, caller);
    }
    return _doDeposit(caller, assets, receiver, vaultId);
}

// Parent implementation in RWASegWrap (identical logic except missing receiverIsSender modifier)
function deposit(uint256 assets, address receiver)
    external virtual override whenNotPaused amountNotZero(assets) addressNotZero(receiver)
    returns (uint256) {
    address caller = _msgSender();
    uint256 vaultId = _resolveVaultId(caller);
    return _doDeposit(caller, assets, receiver, vaultId);
}
```

Impact: This code duplication increases the maintenance burden while creating a risk of inconsistencies.

Recommended Mitigation: Refactor the overridden functions to use `super` calls instead of duplicating logic:

```
function deposit(uint256 assets, address receiver)
    public override whenNotPaused amountNotZero(assets) addressNotZero(receiver)
    ↪  receiverIsSender(receiver)
    returns (uint256) {
-   address caller = _msgSender();
-   uint256 vaultId = getVaultId(caller);
-   if (vaultId == 0) {
-       vaultId = ++latestVaultId;
-       vault = _deployVault(vaultId);
-       _addVault(address(vault), vaultId, caller);
-   }
-   return _doDeposit(caller, assets, receiver, vaultId);
+   return super.deposit(assets, receiver);
}

function redeem(uint256 shares, address receiver, address owner)
    external override whenNotPaused amountNotZero(shares) addressNotZero(receiver)
    ↪  receiverIsSender(receiver) addressNotZero(owner)
    returns (uint256) {
-   address caller = _msgSender();
-   uint256 vaultId = getVaultId(caller);
-   if (vaultId == 0) {
-       revert VaultNotFound();
```

```
-   }
-   return _doRedeem(caller, shares, receiver, owner, vaultId);
+   return super.redeem(shares, receiver, owner);
}

function redeemById(uint256 shares, address receiver, address owner, uint256 id)
    external override whenNotPaused amountNotZero(shares) addressNotZero(receiver) addressNotZero(owner)
    ↪  receiverIsSender(receiver) idNotZero(id) recognizedVault(id)
    returns (uint256) {
-   address caller = _msgSender();
-   uint256 vaultId = getVaultId(caller);
-   if (id != vaultId) {
-       revert InvestorVaultMismatch(id, caller);
-   }
-   return _doRedeem(caller, shares, receiver, owner, id);
+   return super.redeemById(shares, receiver, owner, id);
}

function depositById(uint256 assets, address receiver, uint256 id)
    external override whenNotPaused amountNotZero(assets) receiverIsSender(receiver) idNotZero(id)
    ↪  recognizedVault(id)
    returns (uint256) {
-   address caller = _msgSender();
-   uint256 vaultId = getVaultId(caller);
-   if (id != vaultId) {
-       revert InvestorVaultMismatch(id, caller);
-   }
-   return _doDeposit(caller, assets, receiver, vaultId);
+   return super.depositById(assets, receiver, id);
}
```

Securitize: Fixed in commit 6322e3.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Unnecessary `_msgSender()` call in `_resolveVaultId` when `caller` parameter is available

Description: In `RWASegWrap::_resolveVaultId()`, the function receives a `caller` parameter representing the address for which a vault ID should be resolved, but when calling `_addVault()`, it uses `_msgSender()` instead of the provided `caller` parameter. This creates an unnecessary function call and potential inconsistency since the `caller` parameter already contains the correct address.

```
function _resolveVaultId(address caller) internal returns (uint256) {
    uint256 vaultId = getVaultId(caller);
    if (vaultId == 0) {
        vaultId = ++latestVaultId;
        ISegregatedVault vault = _deployVault(vaultId);
        _addVault(address(vault), vaultId, _msgSender()); // <- use caller directly
    }
    return vaultId;
}
```

Impact: The unnecessary `_msgSender()` call results in additional gas consumption and reduces code clarity by using different variables that should represent the same address.

Recommended Mitigation: Replace `_msgSender()` with the `caller` parameter in the `_addVault()` call:

```
function _resolveVaultId(address caller) internal returns (uint256) {
    uint256 vaultId = getVaultId(caller);
    if (vaultId == 0) {
        vaultId = ++latestVaultId;
        ISegregatedVault vault = _deployVault(vaultId);
-       _addVault(address(vault), vaultId, _msgSender());
+       _addVault(address(vault), vaultId, caller);
    }
    return vaultId;
}
```

Securitize: Fixed in commit 3e16e8.

Cyfrin: Verified.

7.4.2 Unnecessary gas consumption in deposit function due to redundant maximum deposit check

Description: The `SecuritizeVaultV2::_depositWithFee` function performs an unnecessary check against `maxDeposit(from)` to validate that the deposit amount doesn't exceed the maximum allowed deposit limit. However, `SecuritizeVaultV2` does not override the `maxDeposit` function from its parent `ERC4626Upgradeable` contract, which means `maxDeposit` always returns `type(uint256).max`.

```
// In SecuritizeVaultV2::_depositWithFee
function _depositWithFee(uint256 assets, address from) private returns (uint256) {
    address caller = _msgSender();
    uint256 maxAssets = maxDeposit(from); // Returns type(uint256).max
    if (assets > maxAssets) { // This condition will never be true
        revert ERC4626ExceededMaxDeposit(from, assets, maxAssets);
    }
    // ... rest of function
}

// In ERC4626Upgradeable (not overridden by SecuritizeVaultV2)
function maxDeposit(address) public view virtual returns (uint256) {
    return type(uint256).max; // Always returns maximum uint256 value
}
```

This creates a redundant comparison where `assets > type(uint256).max` will never be true for any realistic deposit amount, making the check pointless and wasteful of gas. The condition on line 346-348 will never trigger the revert `ERC4626ExceededMaxDeposit` because no uint256 value can exceed `type(uint256).max`.

The function call `maxDeposit(from)` and the subsequent comparison are executed on every deposit operation, unnecessarily consuming gas for a check that serves no purpose in the current implementation.

Impact: This unnecessary computation increases gas costs for every deposit operation without providing any functional benefit, resulting in higher transaction costs for users.

Recommended Mitigation: Remove the unnecessary maximum deposit check since `SecuritizeVaultV2` doesn't implement custom deposit limits:

```
function _depositWithFee(uint256 assets, address from) private returns (uint256) {
    address caller = _msgSender();
-   uint256 maxAssets = maxDeposit(from);
-   if (assets > maxAssets) {
-       revert ERC4626ExceededMaxDeposit(from, assets, maxAssets);
-   }

    uint256 fee = 0;
    if (address(feeManager) != address(0)) {
        fee = IFeeManager(feeManager).computeFee(IFeeManager.FeeApplicableOperation.Deposit, assets);
    }

    uint256 shares = previewDeposit(assets - fee);
    _depositAndSendFees(caller, from, assets, fee, shares);
    return shares;
}
```

Alternatively, if deposit limits are intended to be implemented in the future, override the `maxDeposit` function with the appropriate logic.

Securitize: Fixed in commits 5105f0 and 57805a.

Cyfrin: Verified.

7.4.3 Gas inefficient approval pattern in liquidation function leads to unnecessary gas consumption

Description: The `SecuritizeVaultV2::_liquidateTo` function performs an approval operation to the redemption contract on every liquidation call when a redemption contract is configured. On line 489, the function calls `IERC20Metadata(asset()).approve(address(redemption), assets)` for each liquidation operation, approving only the exact amount of assets to be redeemed.

This approach is gas inefficient because:

1. **Repetitive SSTORE operations**: Each `approve` call requires an expensive SSTORE operation to update the allowance mapping
2. **Fixed redemption contract**: The redemption contract is set during initialization and cannot be changed afterward, making it safe to grant a permanent approval

The current implementation unnecessarily consumes gas on every liquidation when a more efficient approach would be to grant `type(uint256).max` approval to the redemption contract during initialization.

Impact: Every liquidation operation consumes additional gas due to redundant approval calls, increasing transaction costs for users performing liquidations.

Recommended Mitigation: Grant maximum approval to the redemption contract during initialization instead of approving on each liquidation:

```
function _initialize(
    string memory _name,
    string memory _symbol,
    address _securitizeToken,
```

```
    address _redemptionAddress,
    address _liquidationToken,
    address _navProvider
) private {
    // ... existing initialization logic ...

    redemption = ISecuritizeOffRamp(_redemptionAddress);
    liquidationToken = IERC20Metadata(_liquidationToken);
    navProvider = ISecuritizeNavProvider(_navProvider);

+   // Grant maximum approval to redemption contract if it exists
+   if (address(redemption) != address(0)) {
+       bool success = IERC20Metadata(_securitizeToken).approve(address(redemption), type(uint256).max);
+       if (!success) {
+           revert AssetApprovalFailed();
+       }
+   }

    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
}

function _liquidateTo(address to, uint256 shares) internal {
    if (balanceOf(to) < shares) {
        revert NotEnoughShares();
    }
    uint256 assets = convertToAssets(shares);
    _burn(to, shares);

    emit Liquidate(to, assets, shares);

    if (address(0) != address(redemption)) {
-       bool success = IERC20Metadata(asset()).approve(address(redemption), assets);
-       if (!success) {
-           revert AssetApprovalFailed();
-       }
        uint256 balanceBefore = liquidationToken.balanceOf(address(this));
        redemption.redeem(assets, 0);
        uint256 receivedAmount = liquidationToken.balanceOf(address(this)) - balanceBefore;
        liquidationToken.safeTransfer(to, receivedAmount);
    } else {
        liquidationToken.safeTransfer(to, assets);
    }
}
```

Securitize: Acknowledged.

Cyfrin: Acknowledged.