



---

# Securitize Audit Report

---

Prepared by [Cyfrin](#)

Version 2.0

**Lead Auditors**

[Hans](#)

July 18, 2024

# Contents

<b>1 About Cyfrin</b>	<b>2</b>
<b>2 Disclaimer</b>	<b>2</b>
<b>3 Risk Classification</b>	<b>2</b>
<b>4 Protocol Summary</b>	<b>2</b>
4.1 Architecture . . . . .	2
4.1.1 Securitize NAV Provider . . . . .	2
4.1.2 Securitize Redemption Protocol . . . . .	2
4.1.3 Securitize Swap Contract . . . . .	3
4.2 Actors . . . . .	3
<b>5 Audit Scope</b>	<b>3</b>
<b>6 Executive Summary</b>	<b>3</b>
<b>7 Findings</b>	<b>6</b>
7.1 Critical Risk . . . . .	6
7.1.1 Wrong decimals in the calculation of liquidity amount during redemption . . . . .	6
7.1.2 Missing validation on the <code>msg.sender</code> in <code>SecuritizeSwap::buy</code> function . . . . .	6
7.2 High Risk . . . . .	8
7.2.1 Wrong validation on the <code>availableLiquidity</code> in function <code>SecuritizeRedemption::redeem()</code> . . . . .	8
7.2.2 Missing access control in <code>CollateralLiquidityProvider::setCollateralProvider</code> . . . . .	8
7.2.3 Missing access control in <code>CollateralLiquidityProvider::setExternalCollateralRedemption</code> . . . . .	9
7.3 Medium Risk . . . . .	10
7.3.1 No validation check of <code>_investorWallet</code> when buying <code>dsToken</code> . . . . .	10
7.3.2 No storage gap for upgradeable contract might lead to storage slot collision . . . . .	10
7.3.3 Lack of a feature to allow investors to increase their nounce . . . . .	11
7.3.4 Missing validation of <code>msg.value</code> in the function <code>SecuritizeSwap::executePreApprovedTransaction()</code> . . . . .	11
7.4 Low Risk . . . . .	13
7.4.1 Precision loss in the <code>SecuritizeSwap.calculateDsTokenAmount()</code> function. . . . .	13
7.4.2 The <code>SecuritizeRedemption.updateLiquidityProvider</code> function emits the wrong information. . . . .	14
7.4.3 Bad practice to have multiple initializers . . . . .	14
7.4.4 Missing slippage control for investors during the buy . . . . .	14
7.4.5 Missing zero check for the result of <code>ecrecover()</code> . . . . .	15
7.4.6 Missing input validation in <code>CollateralLiquidityProvider::setExternalCollateralRedemption</code> . . . . .	15
7.5 Informational . . . . .	17
7.5.1 Emission of wrong value for some events . . . . .	17
7.5.2 Inconsistent use of <code>override</code> keyword . . . . .	17
7.5.3 Wrong/misleading comments . . . . .	17

# 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](https://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

	<b>Impact: High</b>	<b>Impact: Medium</b>	<b>Impact: Low</b>
<b>Likelihood: High</b>	Critical	High	Medium
<b>Likelihood: Medium</b>	High	Medium	Low
<b>Likelihood: Low</b>	Medium	Low	Low

## 4 Protocol Summary

The audited part of the Securitize protocol consists of three main components: the NAV Provider, the Redemption Protocol, and the Swap Contract. These components work together to allow investors to redeem their digital securities for stable coins or swap stable coins for securities tokens.

### 4.1 Architecture

#### 4.1.1 Securitize NAV Provider

- Defines a common interface (`ISecuritizeNavProvider`) to get the current NAV rate for redeem or swap operations involving DSToken and stable coins.
- The NAV rate is used to calculate the amount of stable coins to be supplied during redemption and the amount of securities tokens to be supplied during a swap.
- The rate is interpreted as the number of stable coins that can be exchanged for one DSToken. (asset:liquidity ratio)
- The rate is supposed to be expressed in the same decimals as the stable coin.
- UUPS upgradeable contract pattern is used.

#### 4.1.2 Securitize Redemption Protocol

- Allows investors to redeem their digital securities for stable coins.
- Designed to support multiple implementations for supplying stable coins, but currently only one implementation exists.
- Future extensions can be made by extending the `ILiquidityProvider` interface.
- UUPS upgradeable contract pattern is used.

#### **4.1.3 Securitize Swap Contract**

- Works with the DS Protocol to enable investors to swap their stable coins for securities tokens.
- During the swap operation, the investor and their wallet can also be registered.
- The contract is supposed to be used via a proxy contract, but the proxy contract is not in scope.

### **4.2 Actors**

- **NAV Provider Owner**
  - Sets the rate.
  - Upgrades the contract.
- **Redemption Admin**
  - Sets the liquidity provider and NAV rate provider.
  - Upgrades the contract.
- **Swap Admin**
  - Pauses and unpauses the contract.
  - Update the NAV rate provider.
- **Investor**
  - Engages in redemption and swap operations.
  - Holds and manages digital securities and stable coins.

There are some other roles like ROLE\_ISSUER, ROLE\_MASTER that are not in scope.

## **5 Audit Scope**

For all 3 repositories, all Solidity files in \contracts\ directory are included in the scope of the audit.

## **6 Executive Summary**

Over the course of 8 days, the Cyfrin team conducted an audit on the [Securitize](#) smart contracts provided by [Securitize](#). In this period, a total of 18 issues were found.

Summary of the audit findings and any additional executive comments.

## Summary

Project Name	Securitize
Repository	<a href="#">bc-redemption-sc</a>
Commit	<a href="#">32e23d5318be...</a>
Repository	<a href="#">securitize-swap</a>
Commit	<a href="#">47704357c2da...</a>
Repository	<a href="#">bc-nav-provider-sc</a>
Commit	<a href="#">dce942a8a54a...</a>
Audit Timeline	July 1st - July 10th
Methods	Manual Review

## Issues Found

Critical Risk	2
High Risk	3
Medium Risk	4
Low Risk	6
Informational	3
Gas Optimizations	0
Total Issues	18

## Summary of Findings

[C-1] Wrong decimals in the calculation of liquidity amount during redemption	Resolved
[C-2] Missing validation on the msg.sender in SecuritizeSwap::buy function	Resolved
[H-1] Wrong validation on the availableLiquidity in function SecuritizeRedemption::redeem()	Resolved
[H-2] Missing access control in CollateralLiquidityProvider::setCollateralProvider	Resolved
[H-3] Missing access control in CollateralLiquidityProvider::setExternalCollateralRedemption	Resolved
[M-1] No validation check of _investorWallet when buying dsToken.	Resolved
[M-2] No storage gap for upgradeable contract might lead to storage slot collision	Resolved
[M-3] Lack of a feature to allow investors to increase their nounce	Acknowledged
[M-4] Missing validation of msg.value in the function SecuritizeSwap::executePreApprovedTransaction()	Resolved
[L-1] Precision loss in the SecuritizeSwap.calculateDsTokenAmount() function.	Resolved

[L-2] The SecuritizeRedemption.updateLiquidityProvider function emits the wrong information.	Resolved
[L-3] Bad practice to have multiple initializers	Resolved
[L-4] Missing slippage control for investors during the buy	Resolved
[L-5] Missing zero check for the result of <code>ecrecover()</code>	Resolved
[L-6] Missing input validation in <code>CollateralLiquidityEngineProvider::setExternalCollateralRedemption</code>	Resolved
[I-1] Emission of wrong value for some events	Acknowledged
[I-2] Inconsistent use of <code>override</code> keyword	Resolved
[I-3] Wrong/misleading comments	Resolved

## 7 Findings

### 7.1 Critical Risk

#### 7.1.1 Wrong decimals in the calculation of liquidity amount during redemption

**Description:** The contract SecuritizeRedemption provides a public function `redeem()` so that investors can redeem their asset (DS Token) for liquidity token. (Stable Coin) The parameter `_amount` represents the amount of asset to be redeemed and it is in decimals of the asset token. The function utilizes the rate provided by the `navProvider` and the rate is in decimals of the stable coin.

```
SecuritizeRedemption.sol
77:     function redeem(uint256 _amount) whenNotPaused external override {
78:         uint256 rate = navProvider.rate();
79:         require(rate != 0, "Rate should be defined");
80:         require(asset.balanceOf(msg.sender) >= _amount, "Redeemer has not enough balance");
81:         require(address(liquidityProvider) != address(0), "Liquidity provider should be defined");
82:         require(liquidityProvider.availableLiquidity() >= _amount, "Not enough liquidity");
83:
84:         ERC20 stableCoin = ERC20(address(liquidityProvider.liquidityToken()));
85:         uint256 liquidity = _amount * rate / (10 ** stableCoin.decimals());
86:
87:         liquidityProvider.supplyTo(msg.sender, liquidity);
88:         asset.transferFrom(msg.sender, liquidityProvider.recipient(), _amount);
89:
90:         emit RedemptionCompleted(msg.sender, _amount, liquidity, rate);
91:     }
```

Looking at the L85 where the returning liquidity amount is calculated, we can see that the liquidity will be in the decimals of asset token which can be different from the decimals of liquidity token (stable coin). We can verify that the liquidity value here must be in the decimals of stable coin because `CollateralLiquidityProvider::supplyTo()` function transfers `liquidityToken` using the provided amount as is. Assuming a realistic scenario, where the `liquidityToken=USDC` with 6 decimals and `asset` is a standard ERC20 with 18 decimals, this vulnerability will transfer 1e12 times of the actually required value to the redeemer.

**Impact:** We evaluate the impact to be CRITICAL because the vulnerability can cause a severe financial loss to the protocol in a realistic scenario.

#### Recommended Mitigation:

```
-     uint256 liquidity = _amount * rate / (10 ** stableCoin.decimals());
+     uint256 liquidity = _amount * rate / (10 ** asset.decimals());
```

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

#### 7.1.2 Missing validation on the `msg.sender` in `SecuritizeSwap::buy` function

**Description:** The function `SecuritizeSwap::buy` is supposed to be used by investors to swap `stableCoinToken` for `dsToken` and the key difference from the function `SecuritizeSwap::swap` is that `buy` can be called by anyone while the `swap` function can only be called by `Issuer` or `Master`. The problem is the function `buy()` does not validate if the caller is actually the investor of `_senderInvestorId`.

```
SecuritizeSwap.sol
104:     function buy(//@audit-info public call that investors can directly call
105:         string memory _senderInvestorId,
106:         address _investorWallet,
107:         uint256 _stableCoinAmount,
108:         uint256 _blockLimit,
```

```

109:     uint256 _issuanceTime
110:     ) public override whenNotPaused {
111:         require(_blockLimit >= block.number, "Transaction too old");
112:         require(stableCoinToken.balanceOf(_investorWallet) >= _stableCoinAmount, "Not enough
→ stable tokens balance");
113:         require(IDSRegistryService(dsToken.getDSService(DS_REGISTRY_SERVICE)).isInvestor(_senderIn ]
→ vestorId), "Investor not
→ registered");
114:         require(navProvider.rate() > 0, "NAV Rate must be greater than 0");
115:         (uint256 dsTokenAmount, uint256 currentNavRate) =
→ calculateDsTokenAmount(_stableCoinAmount);
116:         stableCoinToken.transferFrom(_investorWallet, issuerWallet, _stableCoinAmount);
117:
118:         dsToken.issueTokensCustom(_investorWallet, dsTokenAmount, _issuanceTime, 0, "", 0);
119:
120:         emit Buy(msg.sender, _stableCoinAmount, dsTokenAmount, currentNavRate, _investorWallet);
121:     }

```

As we can see in the implementation, the function only checks if the provided `_senderInvestorId` is actually registered and do not check if the `msg.sender` is an investor assigned to that ID. This vulnerability allows anyone to force any investor to buy `dsToken` with `stableCoinToken` as long as the investor granted allowance to the `SecuritizeSwap` contract.

**Impact:** We evaluate the impact to be CRITICAL because anyone can change the investors economical status at the will.

**Recommended Mitigation:** Validate the `msg.sender` to be an actual investor utilizing the registry service.

**Securitize:** Fixed in commit [b09460](#)

**Cyfrin:** Verified.

## 7.2 High Risk

### 7.2.1 Wrong validation on the availableLiquidity in function SecuritizeRedemption::redeem()

**Description:** The function SecuritizeRedemption::redeem(uint256 \_amount) is supposed to be used by investors to be able to redeem their assets (DSToken) for liquidity (stable coin). The parameter \_amount is the amount of asset that the caller wants to redeem.

```
SecuritizeRedemption.sol
77:     function redeem(uint256 _amount) whenNotPaused external override {
78:         uint256 rate = navProvider.rate();
79:         require(rate != 0, "Rate should be defined");
80:         require(asset.balanceOf(msg.sender) >= _amount, "Redeemer has not enough balance");
81:         require(address(liquidityProvider) != address(0), "Liquidity provider should be defined");
82:         require(liquidityProvider.availableLiquidity() >= _amount, "Not enough
→ liquidity"); //audit-issue WRONG
83:
84:         ERC20 stableCoin = ERC20(address(liquidityProvider.liquidityToken()));
85:         uint256 liquidity = _amount * rate / (10 ** stableCoin.decimals());
86:
87:         liquidityProvider.supplyTo(msg.sender, liquidity);
88:         asset.transferFrom(msg.sender, liquidityProvider.recipient(), _amount);
89:
90:         emit RedemptionCompleted(msg.sender, _amount, liquidity, rate);
91:     }
```

Looking at the implementation, there is a line to check if the liquidityProvider has enough liquidity. But it is checking against a wrong value \_amount that is in fact the amount of asset, not liquidity. Instead, it must be checked against the liquidity value that is calculated afterwards at L85. This wrong check can lead to a situation where the check passes while the actual availableLiquidity() is less than required (if the rate is greater than 1) or the check fails while there are enough liquidity available (if the rate is less than 1). The second case is more severe than the former one.

**Impact:** We evaluate the risk to be HIGH because the redemption will fail most of the time when the rate is much less than 1 and there is a difference in the number of decimals for asset and liquidity tokens.

**Recommended Mitigation:** Move the validation below the calculation of required liquidity and fix it to compare with a correct value.

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

### 7.2.2 Missing access control in CollateralLiquidityProvider::setCollateralProvider

**Description:** The contract CollateralLiquidityProvider relies on collateralProvider to pull the collateral and process redemption in liquidity token. This state variable can be changed via the function setCollateralProvider() as below.

```
CollateralLiquidityProvider.sol
79:     function setCollateralProvider(address _collateralProvider) external { //audit-issue CRITICAL
→ missing access control
80:         collateralProvider = _collateralProvider;
81:     }
```

Looking at the implementation, the function is not protected by any access control and anyone can change collateralProvider to whatever they want. Note that with a malicious collateralProvider , it is possible to make the function availableLiquidity to return zero all the time.

**Impact:** We evaluate the impact to be HIGH because any attacker can change the core state variable collateralProvider as they want that can lead to various critical outcomes including permanent DoS.

**Recommended Mitigation:** Add a onlyOwner modifier to the function setCollateralProvider().

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

### 7.2.3 Missing access control in CollateralLiquidityProvider::setExternalCollateralRedemption

**Description:** The contract CollateralLiquidityProvider relies on externalCollateralRedemption to provide liquidity and this state variable can be changed via the function setExternalCollateralRedemption() as below.

```
CollateralLiquidityProvider.sol
75:     function setExternalCollateralRedemption(address _externalCollateralRedemption) external
→    override { // @audit-issue CRITICAL missing access control
76:         externalCollateralRedemption = IRedemption(_externalCollateralRedemption);
77:     }
```

Looking at the implementation, the function is not protected by any access control and anyone can change externalCollateralRedemption to whatever they want. Note that with a malicious externalCollateralRedemption, it is possible to make the core function supplyTo useless.

**Impact:** We evaluate the impact to be HIGH because any attacker can change the core state variable externalCollateralRedemption as they want that can lead to various critical outcomes including permanent DoS.

**Recommended Mitigation:** Add a onlyOwner modifier to the function setExternalCollateralRedemption().

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

## 7.3 Medium Risk

### 7.3.1 No validation check of \_investorWallet when buying dsToken.

**Description:** In the SecuritizeSwap.buy() function, there is no validation check to verify if \_investorWallet has already been added to the wallet list of \_senderInvestorId, unlike in the swap() function. Consequently, users can purchase dsToken using wallets that have not been added.

The following code snippet demonstrates the wallet validation part within the swap() function. It first verifies if the wallet has already been added. If the wallet hasn't been added, it adds the wallet. If the wallet has already been added, it verifies if the wallet belongs to the investor. However, the buy() function does not include these steps.

```
File: securitize_dev-securitize-swap-47704357c2da\contracts\swap\SecuritizeSwap.sol

88:     //Check if new wallet should be added
89:     string memory investorWithNewWallet =
→  IDSRegistryService(dsToken.getDSService(DS_REGISTRY_SERVICE)).getInvestor(_newInvestorWallet);
90:     if(CommonUtils.isEmptyString(investorWithNewWallet)) {
91:
→  IDSRegistryService(dsToken.getDSService(DS_REGISTRY_SERVICE)).addWallet(_newInvestorWallet,
→  _senderInvestorId);
92:     } else {
93:         require(CommonUtils.isEqualString(_senderInvestorId, investorWithNewWallet), "Wallet
→  does not belong to investor");
94:     }
```

**Impact:** Users can buy dsToken using wallets that have not been added.

**Recommended Mitigation:** A validation check for the wallet should be included in the buy() function.

**Securitize:** Fixed in commit [b09460](#)

**Cyfrin:** Verified.

### 7.3.2 No storage gap for upgradeable contract might lead to storage slot collision

**Description:** For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments" (quote OpenZeppelin). Otherwise it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts, potentially causing loss of user fund or cause the contract to malfunction completely.

Refer to the bottom part of this article: <https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable>

**Impact:** No storage gap for upgradeable contract might lead to storage slot collision

**Proof of Concept:** Several contracts are intended to be upgradeable contracts in the code base, including securitize\_dev-bc-redemption-sc-32e23d5318be\contracts\utils\BaseContract.sol securitize\_dev-bc-nav-provider-sc-dce942a8a54a\contracts\utils\BaseContract.sol

However, none of these contracts contain storage gap. The storage gap is essential for upgradeable contract because "It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments".

**Recommended Mitigation:** Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

**Securitize:** Fixed in commit [3977ca](#) and [334f49](#)

**Cyfrin:** Verified.

### 7.3.3 Lack of a feature to allow investors to increase their nounce

**Description:** The function `SecuritizeSwap::executePreApprovedTransaction()` allows anyone to execute a pre-approved transaction by providing a valid signature. This function relies on an internal function `doExecuteByInvestor()` and the hash is calculated based on various attributes including the investor's nonce. This nonce mechanism is in place to prevent executing the same transaction. Generally, nonce mechanism comes with an invalidation mechanism together so that the nonce owner can invalidate the current nonce by increasing the nonce proactively. The current design does not provide a way to increase the nonce and this means an investor can not invalidate an already signed hash.

**Impact:** The investors do not have a way to invalidate already signed hashes.

**Recommended Mitigation:** Add a function where an investor can increase their nonce to invalidate the already signed hashes.

**Securitize:** We will not apply this suggestion because the investor cannot modify their nonce. This will only be done with the contract and our backend. Additionally, NAV Rate changes will be controlled.

**Cyfrin:** Acknowledged.

### 7.3.4 Missing validation of `msg.value` in the function `SecuritizeSwap::executePreApprovedTransaction()`

**Description:** `SecuritizeSwap` allows anyone to execute pre-approved transactions using the function `SecuritizeSwap::executePreApprovedTransaction`. This function validates that the protocol is not paused and the length of `params` parameter is two and then calls `doExecuteByInvestor()`.

```
SecuritizeSwap.sol
191:     uint256 value = _params[0];
192:     uint256 gasLimit = _params[1];
193:     assembly {
194:         let ptr := add(_data, 0x20)
195:         let result := call(gasLimit, _destination, value, ptr, mload(_data), 0, 0)
196:         let size := returndatasize()
197:         returndatacopy(ptr, 0, size)
198:         switch result
199:             case 0 {
200:                 revert(ptr, size)
201:             }
202:             default {
203:                 return(ptr, size)
204:             }
205:     }
```

Looking at the implementation of `doExecuteByInvestor`, `params[0]` is used as a value to make a call to `_destination`. But the `params[0]` was NOT validated to be same to the actual `msg.value` and there is no mechanism to refund the excessive value either. Due to this problem, if the provided `msg.value` is greater than the `params[0]` (which is included in the signed hash), the residual amount of native token is permanently locked in the contract.

**Impact:** We evaluate the impact to be MEDIUM given that it is unlikely the caller makes a call with excessive `msg.value`.

**Recommended Mitigation:** Require the `msg.value` to be the same to `params[0]` or return the excessive amount to the caller.

**Securitize:** Revised the function in commit [b09460](#). In this case we used OpenZeppelin `Address.functionCall(_destination, _data)` function for non-payable transactions. The `msg.value` is 0. Also the `SecuritizeSwap::executePreApprovedTransaction()` function is not payable.

**Cyfrin:** Verified.

## 7.4 Low Risk

### 7.4.1 Precision loss in the `SecuritizeSwap.calculateDsTokenAmount()` function.

**Description:** When `stableCoinDecimals` is greater than `dsTokenDecimals`, the calculation is divided into two parts: dividing by  $(10^{** (stableCoinDecimals - dsTokenDecimals)})$  at L242 and then multiplying by  $10^{** stableCoinDecimals}$  at L245. This can result in a loss of precision.

```
if (stableCoinDecimals <= dsTokenDecimals) {
    adjustedStableCoinAmount = _stableCoinAmount * (10 ** (dsTokenDecimals -
    ↪ stableCoinDecimals));
} else {
242     adjustedStableCoinAmount = _stableCoinAmount / (10 ** (stableCoinDecimals -
→ dsTokenDecimals));
}
// The InternalNavSecuritizeImplementation uses rate expressed with same number of decimals as
↪ stableCoin
245     uint256 dsTokenAmount = adjustedStableCoinAmount * 10 ** stableCoinDecimals / currentNavRate;
```

**Impact:** The calculated `dsTokenAmount` is less than it should be, resulting in a loss of funds for users.

#### Proof Of Concept:

In fact, the correct formula is

```
dsTokenAmount = _stableCoinAmount * 10 ** dsTokenDecimals / currentNavRate;
```

Let's consider the following scenario:

1. `stableCoinDecimals = 18`
2. `dsTokenDecimals = 6`
3. `currentNavRate = 1e12`(implying `stableCoin : dsToken = 1e6 : 1`)
4. `_stableCoinAmount = 1e18 - 1`

The current logic calculates as follows:

- L242:  $\text{adjustedStableCoinAmount} = (1e18 - 1) / (10^{** (18 - 6)}) = 1e6 - 1$
- L245:  $\text{dsTokenAmount} = (1e6 - 1) * 10^{** 18} / 1e12 = 1e12 - 1e6$

However, the actual correct `dsTokenAmount` should be  $(1e18 - 1) * 1e6 / 1e12 = 1e12 - 1$ , resulting in a difference of  $1e6 - 1$ .

**Recommended Mitigation:** The intermediate variable `adjustedStableCoinAmount` is unnecessary.

```
function calculateDsTokenAmount(uint256 _stableCoinAmount) internal view returns (uint256, uint256)
{
    uint256 stableCoinDecimals = ERC20(address(stableCoinToken)).decimals();
    uint256 dsTokenDecimals = ERC20(address(dsToken)).decimals();
    uint256 currentNavRate = navProvider.rate();

    uint256 adjustedStableCoinAmount;
    if (stableCoinDecimals <= dsTokenDecimals) {
        adjustedStableCoinAmount = _stableCoinAmount * (10 ** (dsTokenDecimals -
    ↪ stableCoinDecimals));
    } else {
        adjustedStableCoinAmount = _stableCoinAmount / (10 ** (stableCoinDecimals -
→ dsTokenDecimals));
    }
    // The InternalNavSecuritizeImplementation uses rate expressed with same number of decimals as
↪ stableCoin
```

```

-
  uint256 dsTokenAmount = adjustedStableCoinAmount * 10 ** stableCoinDecimals / currentNavRate;

+
  uint256 dsTokenAmount = _stableCoinAmount * 10 ** dsTokenDecimals / currentNavRate;

  return (dsTokenAmount, currentNavRate);
}

```

**Securitize:** Fixed in commit [b09460](#)

**Cyfrin:** Verified.

#### 7.4.2 The SecuritizeRedemption.updateLiquidityProvider function emits the wrong information.

**Description:** The SecuritizeRedemption.updateLiquidityProvider function emits the information that oldProvider is updated to liquidityProvider. But the function allocates \_liquidityProvider to oldProvider instead of liquidityProvider

In the updateLiquidityProvider function, it allocates new variable \_liquidityProvider to oldProvider instead of liquidityProvider from L66.

```

File: securitize_dev-bc-redemption-sc-32e23d5318be\contracts\redemption\SecuritizeRedemption.sol
65:   function updateLiquidityProvider(address _liquidityProvider) onlyOwner external override {
66:     address oldProvider = address(_liquidityProvider);
67:     liquidityProvider = ILiquidityProvider(_liquidityProvider);
68:     emit LiquidityProviderUpdated(oldProvider, address(liquidityProvider));
69:   }

```

From L67, liquidityProvider is also same as \_liquidityProvider. From L68, it emits same variables and this is wrong.

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

#### 7.4.3 Bad practice to have multiple initializers

**Description:** We found that the contract SecuritizeSwap has two functions named initialize(). Both functions are public and behind the modifier initializer and forceInitializeFromProxy and the main initializer (assuming the test suite shows the typical usecase) is the function with four params and it calls the other one. This is not a good practice and we strongly recommend fix it. We recommend changing the initialize() function with three parameters into an internal function with a proper name (maybe \_initialize()) and the modifiers to be applied to only the main public initializer.

**Securitize:** Fixed in commit [b09460](#)

**Cyfrin:** Verified.

#### 7.4.4 Missing slippage control for investors during the buy

**Description:** The protocol allows investors to buy dsToken with stableCoinToken using the function SecuritizeSwap::buy(). The function calculates the amount of dsToken to send to the caller based on the navProvider.rate() and there is no way for an investor to predict what rate will be applied to his purchase.

```

SecuritizeSwap.sol
104:   function buy(
105:     string memory _senderInvestorId,
106:     address _investorWallet,
107:     uint256 _stableCoinAmount,
108:     uint256 _blockLimit,

```

```

109:     uint256 _issuanceTime
110:     ) public override whenNotPaused {
111:         require(_blockLimit >= block.number, "Transaction too old");//@audit-ok
112:         require(stableCoinToken.balanceOf(_investorWallet) >= _stableCoinAmount, "Not enough
→ stable tokens balance");
113:         require(IDSRegistryService(dsToken.getDSService(DS_REGISTRY_SERVICE)).isInvestor(_senderIn ]
→ vestorId), "Investor not
→ registered");
114:         require(navProvider.rate() > 0, "NAV Rate must be greater than 0");
115:         (uint256 dsTokenAmount, uint256 currentNavRate) =
→ calculateDsTokenAmount(_stableCoinAmount);
116:         stableCoinToken.transferFrom(_investorWallet, issuerWallet, _stableCoinAmount);
117:
118:         dsToken.issueTokensCustom(_investorWallet, dsTokenAmount, _issuanceTime, 0, "", "
→ 0);//@audit-issue missing protection against change in the NAV rate
119:
120:         emit Buy(msg.sender, _stableCoinAmount, dsTokenAmount, currentNavRate, _investorWallet);
121:     }
122:

```

If the navProvider.rate() is changed during a specific timeframe by whatever reason, the investors do not have other ways but to accept the loss due to the rate change.

**Recommended Mitigation:** Add a parameter \_minDsTokenAmount in the function so that the caller can specify the minimum expected amount. It is also recommended to make the function calculateDsTokenAmount() public so that users can anticipate the receiving amounts beforehand.

**Securitize:** Fixed in commit [b09460](#). We modified the definition of the buy function so that the dsTokenAmount is fixed, and the liquidity tokens vary according to the NAV rate. We also added the parameter \_maxStableCoinAmount to prevent price changes with sudden variations.

**Cyfrin:** Verified.

#### 7.4.5 Missing zero check for the result of ecrecover()

**Description:** The function SecuritizeSwap::doExecuteByInvestor() validates the provided signature was actually signed by an actor with a role ROLE\_ISSUER or ROLE\_MASTER. In the validation, ecrecover() is used but the result is not validated to be non-zero. Although, it is understood that the following line validates the role of the recovered, it is strongly recommended to implement this check for unexpected problems in the future.

**Securitize:** Fixed in commit [b09460](#)

**Cyfrin:** Verified.

#### 7.4.6 Missing input validation in CollateralLiquidityProvider::setExternalCollateralRedemption

**Description:** Note that this finding assumes the other finding about access control is mitigated. The contract CollateralLiquidityProvider relies on externalCollateralRedemption to process redemption. The function supplyTo() is the core function of this contract.

```

CollateralLiquidityProvider.sol
61:     function supplyTo(address _redeemer, uint256 _amount) whenNotPaused onlySecuritizeRedemption
→ public override {
62:         //take collateral funds from collateral provider
63:         IERC20(externalCollateralRedemption.asset()).transferFrom(collateralProvider,
→ address(this), _amount);
64:
65:         //approve external redemption
66:         IERC20(externalCollateralRedemption.asset()).approve(address(externalCollateralRedemption),
→ _amount);
67:

```

```
68:     //get liquidity
69:     externalCollateralRedemption.redeem(_amount);
70:
71:     //supply _redeemer
72:     liquidityToken.transfer(_redeemer, _amount);
73: }
```

Looking at the L69 with L72, the function is assuming that `liquidityToken = externalCollateralRedemption.liquidity`. But this is not validated in the function `setExternalCollateralRedemption` and there is a risk to cause an inconsistency.

**Impact:** We evaluate the impact to be LOW assuming the function will be protected by a proper access control mitigating the other finding.

**Recommended Mitigation:** Validate that `liquidityToken = externalCollateralRedemption.liquidity` in the function `setExternalCollateralRedemption()`.

**Securitize:** Fixed in commit [3977ca](#)

**Cyfrin:** Verified.

## 7.5 Informational

### 7.5.1 Emission of wrong value for some events

**Description:** In the `SecuritizeSwap::swap()` function, an event is emitted after a successful swap. But the event is emitting `msg.sender` in the place of `from` value while the `msg.sender` for the function `swap()` is only issuer or master.

```
SecuritizeSwap.sol
101: emit Swap(msg.sender, _valueDsToken, _valueStableCoin, _newInvestorWallet);
```

**Securitize:** We won't fix. The first argument of Buy event is address `_from`, in this case the `msg.sender` is correct.

**Cyfrin:** Acknowledged.

### 7.5.2 Inconsistent use of `override` keyword

**Description:** We found that `override` keyword is being used inconsistently in several places. It is recommended to use the `override` keyword consistently and reasonably.

```
CollateralLiquidityProvider.sol
75:     function setExternalCollateralRedemption(address _externalCollateralRedemption) external
→   override { // @audit override keyword
76:       externalCollateralRedemption = IRedemption(_externalCollateralRedemption);
77:   }
78:
79:     function setCollateralProvider(address _collateralProvider) external { // @audit no override
→   keyword
80:       collateralProvider = _collateralProvider;
81:   }
```

**Securitize:** Fixed in commit [b09460](#) and [334f49](#).

**Cyfrin:** Verified.

### 7.5.3 Wrong/misleading comments

**Description:** In multiple locations, comments are wrong or misleading. These wrong/misleading comments can lead to unexpected interpretation of the contract and potential security issues. We recommend fix these wrong or misleading comments.

```
ISecuritizeNavProvider.sol
6:  * @dev Defines a common interface to get NAV (Native Asset Value) Rate to // @audit Incomplete
→   comment

SecuritizeInternalNavProvider.sol
9:    * @dev rate: NAV rate expressed with 6 decimals // @audit-issue INFO incomplete comment
14:    * @dev Throws if called by any account other than the owner. // @audit-issue INFO wrong comment

SecuritizeRedemption.sol
26: * @dev Emitted when the Settlement address changes // @audit-issue INFO wrong comment

BaseSecuritizeSwap.sol
98: * @param sigR R signature // @audit-issue INFO wrong comment, should be sigS

BaseSecuritizeSwap.sol
102: * @param params array of params. params[0] = value, params[1] = gasLimit, params[2] =
→   blockLimit // @audit-issue INFO wrong comment, params length is two
```

```
SecuritizeSwap.sol
131: * @param sigR R signature //audit-issue INFO wrong comment, should be sigS

IDSToken.sol
16: * @param _cap address The address which is going to receive the newly issued tokens //audit-issue
→ INFO wrong comment
```

**Securitize:** Fixed in commit [3977ca](#), [b09460](#), and [334f49](#).

**Cyfrin:** Verified.