Securitize Public Stock On/Off Ramp Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Jorge

Assisting Auditors

0ximmeas

Stalin

December 24, 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

Securitize is developing new functionality to enable blockchain-based trading of tokenized public stocks (TSLA, AAPL) through DS Token representations. This audit covers two repositories that together form the core infrastructure for pricing and investor onboarding/offboarding: the NAV Provider and the On/Off Ramp components.

4.0.1 NAV Provider

The new addition to NAV Provider is `SecuritizeAmmNavProvider`, a virtual Automated Market Maker (AMM) based on the Uniswap V2 constant product formula (`x × y = k`). Unlike traditional AMMs, this implementation uses virtual reserves that track "inventory pressure" without holding actual tokens. The system combines an external anchor price (from oracles representing real market prices) with AMM-derived price deviation to create execution prices that reflect both real-world value and supply/demand dynamics.

The virtual AMM serves as Securitize's inventory management mechanism. When investors heavily buy tokenized securities, the deviation multiplier increases, creating a premium above the oracle price that incentivizes selling. Conversely, heavy selling creates discounts that attract buyers. A configurable scale factor dampens these deviations to prevent excessive price impact. The system supports two market states (OPEN and CLOSED), with baseline resets occurring on market transitions to re-center the AMM around the current anchor price.

4.0.2 On/Off Ramp

The On/Off Ramp repository provides the entry and exit infrastructure for investors to acquire and redeem tokenized securities. The on-ramp contracts (`BaseOnRamp`, `PublicStockOnRamp`, `SecuritizeOnRamp`) handle the conversion of liquidity tokens (typically stablecoins such as USDC) into DS Tokens representing tokenized securities. The off-ramp contracts (`BaseOffRamp`, `PublicStockOffRamp`, `SecuritizeOffRamp`) handle the reverse process, allowing investors to redeem their DS Tokens for liquidity tokens.

The system implements a role-based architecture where operators submit transactions on behalf of investors, with investor authorization verified through EIP-712 signatures. Compliance features include investor registry validation through Securitize's DS Protocol services and country-based redemption restrictions. The contracts support

configurable liquidity providers (`AllowanceLiquidityProvider`, `CollateralLiquidityProvider`, `MintingAssetProvider`) and fee management, with both single-step and two-step transfer modes for different operational requirements. All contracts are upgradeable using the UUPS proxy pattern.

5 Audit Scope

The audit scope was limited to:

```
// primary scope - new & modified files
bc-on-off-ramp-sc/contracts/on-ramp/PublicStockOnRamp.sol
bc-on-off-ramp-sc/contracts/on-ramp/BaseOnRamp.sol
bc-on-off-ramp-sc/contracts/on-ramp/SecuritizeOnRamp.sol
bc-on-off-ramp-sc/contracts/off-ramp/BaseOffRamp.sol
bc-on-off-ramp-sc/contracts/off-ramp/PublicStockOffRamp.sol
bc-on-off-ramp-sc/contracts/off-ramp/SecuritizeOffRamp.sol
bc-nav-provider-sc/contracts/nav/SecuritizeAmmNavProvider.sol

// secondary scope - previously existed, no/few modifications
bc-on-off-ramp-sc/contracts/common/BaseContract.sol
bc-on-off-ramp-sc/contracts/fee/MbpsFeeManager.sol
bc-on-off-ramp-sc/contracts/on-ramp/provider/AllowanceAssetProvider.sol
bc-on-off-ramp-sc/contracts/on-ramp/provider/MintingAssetProvider.sol
bc-on-off-ramp-sc/contracts/off-ramp/provider/AllowanceLiquidityProvider.sol
bc-on-off-ramp-sc/contracts/off-ramp/provider/CollateralLiquidityProvider.sol
bc-on-off-ramp-sc/contracts/off-ramp/CountryValidator.sol
bc-on-off-ramp-sc/contracts/off-ramp/RedemptionManager.sol
bc-on-off-ramp-sc/contracts/off-ramp/RedemptionValidator.sol
bc-on-off-ramp-sc/contracts/off-ramp/TokenCalculator.sol
bc-nav-provider-sc/contracts/nav/RedStoneNavProvider.sol
bc-nav-provider-sc/contracts/nav/SecuritizeInternalNavProvider.sol
bc-nav-provider-sc/contracts/utils/Helper.sol
bc-nav-provider-sc/contracts/utils/BaseContract.sol
```

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Securitize Public Stock On/Off Ramp smart contracts provided by Securitize. In this period, a total of 41 issues were found.

The findings consist of 4 Medium and 15 Low severity issues with the remainder being informational and gas optimizations.

As part of the audit we developed an invariant fuzzing test suite provided to the client as an additional deliverable.

Summary

Project Name	Securitize Public Stock On/Off Ramp
Repository	bc-on-off-ramp-sc
Commit	a2935dbffb53...
Fix Commit	f0fff37f39c2...
Repository 2	bc-nav-provider-sc
Commit	c31fc8d3621b...
Fix Commit	d05482b2d42a...
Audit Timeline	Dec 1st - Dec 10th, 2025
Methods	Manual Review, Invariant Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	14
Informational	10
Gas Optimizations	13
Total Issues	41

Summary of Findings

[M-1] `SecuritizeAmmNavProvider` missing `whenNotPaused` modifier on important state-changing functions	Resolved
[M-2] Incorrect use of `investorExists` modifier in `PublicStockOnRamp::swap`	Resolved
[M-3] Zero `curvePriceWad` from rounding causes incorrect pricing or denial of service	Resolved
[M-4] `SecuritizeOnRamp::swap` and `SecuritizeOffRamp::redeem` pass operator as investor address resulting in denial of service	Resolved
[L-01] `RedStoneNavProvider::rate` will return massively inflated value if underlying oracle returns a negative value and lacks common oracle validations	Resolved
[L-02] `SecuritizeAmmNavProvider` quote functions don't reflect execution behavior due to missing baseline reset logic	Resolved
[L-03] Lack of Price Feed Update Function in `RedStoneNavProvider`	Resolved
[L-04] Signatures used in `PublicStockOnRamp` and `PublicStockOffRamp` lack investor-specified deadline and nonce parameters so can be used multiple times by operators	Resolved

[L-05] `SecuritizeAmmNavProvider` virtual reserve rounding erosion can lead to denial of service	Resolved
[L-06] Missing zero output validation in `SecuritizeAmmNavProvider` quote and buy functions	Resolved
[L-07] `RedStoneNavProvider::rate` can return zero for non-zero oracle input due to rounding in `Helper::normalizeRate`	Resolved
[L-08] Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision	Resolved
[L-09] `SecuritizeOnRamp` doesn't provide a mechanism for investors to invalidate their nonces	Acknowledged
[L-10] Incorrect rounding direction in `SecuritizeAmmNavProvider::executeBuyBase` when scaling down `execPrice`	Resolved
[L-11] Incorrect rounding of rate in favor of users when they buy DS Tokens	Acknowledged
[L-12] Block number deadline is chain dependent and unreliable across L2s	Acknowledged
[L-13] `SecuritizeAmmNavProvider` violates core AMM invariant that `k` should never decrease	Resolved
[L-14] `SecuritizeAmmNavProvider` trades when pool price and anchor price differ can leak value	Resolved
[I-1] Use named mapping parameters to explicitly denote the purpose of keys and values	Resolved
[I-2] Consider enforcing minimum redemption amounts in `PublicStockOfframp, SecuritizeOffRamp::redeem`	Acknowledged
[I-3] Missing `liquidityToken` validation in `CollateralLiquidityProvider::initialize`	Resolved
[I-4] Not way to change the recipient in the liquidity provider in the offramp logic	Acknowledged
[I-5] Single step redemption and two step redemption not equivalent logic	Acknowledged
[I-6] Use `SafeERC20` approval and transfer functions instead of standard IERC20 functions for `liquidityToken`	Resolved
[I-7] Unused nonce Field in `ExecutePreApprovedTransaction` struct	Acknowledged
[I-8] Redundant Access Control Check in `SecuritizeInternalNavProvider::addRateUpdater, removeRateUpdater`	Resolved
[I-9] Refactor identical code from `SecuritizeAmmNavProvider::_pricingFromCurveBuy, _pricingFromCurveSell` into internal function	Acknowledged
[I-10] `MbpsFeeManager::setFeePercentageMBPS` allows setting fee greater than max	Acknowledged
[G-01] Better storage packing by changing declaration order	Resolved
[G-02] Emit events first to refactor away local variables storing previous values	Resolved
[G-03] Cache storage to prevent identical storage reads	Resolved
[G-04] When emitting events don't read known values from storage	Resolved
[G-05] In Solidity don't initialize to default values	Resolved

[G-06] Cache required storage slots in top-level functions then pass cached values to child functions	Acknowledged
[G-07] Use named return variables where this can refactor away local variables	Resolved
[G-08] Cache computation where it is performed multiple times	Acknowledged
[G-09] Cache decimals of underlying asset at initialization in `SecuritizeAmmNavProvider`	Acknowledged
[G-10] Use of modifier `nonZeroNavRate` in `SecuritizeOnRamp` and `SecuritizeOffRamp` results in duplicate external call with identical result	Acknowledged
[G-11] Fee calculation occurs twice	Acknowledged
[G-12] Refactor `PublicStockOnRamp::initializedNavProvider`, `BaseOffRamp::nonZeroLiquidityProvider` into internal functions to prevent identical storage reads	Acknowledged
[G-13] Refactor away unnecessary local variables in `SecuritizeAmmNavProvider::_curveBuy, _curveSell`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 `SecuritizeAmmNavProvider` missing `whenNotPaused` modifier on important state-changing functions

Description: `SecuritizeAmmNavProvider` inherits from `BaseContract` which inherits from `PausableUpgradeable`, but does not have the `whenNotPaused` modifier on important state-changing functions.

Impact: Pausing `SecuritizeAmmNavProvider` has no effect; important state-changing functions can continue to be called even when the contract is paused. Looking at the other NAV providers this doesn't seem to be intended, eg `SecuritizeInternalNavProvider::setRate` has the `whenNotPaused` modifier.

Recommended Mitigation: Add the `whenNotPaused` modifier to important state-changing functions such as `SecuritizeAmmNavProvider::resetBaseline, setPriceScaleFactor, executeBuyBase, executeSellBase`.

Securitize: Fixed in commit f09cb9a.

Cyfrin: Verified.

7.1.2 Incorrect use of `investorExists` modifier in `PublicStockOnRamp::swap`

Description: `PublicStockOnRamp::swap` functions incorrectly validates investor registration. These functions use the `investorExists` modifier which checks if `msg.sender` is a registered wallet, but they also have the `onlyRole(OPERATOR_ROLE)` modifier, meaning `msg.sender` is always the operator, not the actual investor. The actual investor is passed as the `_investorWallet` , but this address is never validated against the registry:

```
function swap(
       uint256 _liquidityAmount,
       uint256 _minOutAmount,
       address _investorWallet,
       bytes memory _investorSignature,
       uint8 _marketStatus,
       uint256 _anchorPrice,
       uint256 _anchorPriceExpiresAt
   )
       public
       whenNotPaused
       investorExists <--------
       initializedNavProvider
       validateMinSubscriptionAmount(_liquidityAmount)
       nonZeroAnchorPrice(_anchorPrice)
       onlyRole(OPERATOR_ROLE)
   {...}
```

The `investorExists` modifier checks `_msgSender()` which resolves to `msg.sender`:

```
modifier investorExists() {
       IDSRegistryService registryService =
       ↪  IDSRegistryService(dsToken.getDSService(dsToken.REGISTRY_SERVICE()));
       if (!registryService.isWallet(_msgSender())) {
           revert InvestorNotRegisteredError();
       }
       _;
   }
```

Since the operator calls the function, the modifier validates whether the operator is registered, not whether `_investorWallet` (the actual investor receiving the tokens) is registered.

Impact: * If operators themselves had valid investor wallets, then they could execute swaps for completely unregistered investors, bypassing the entire investor registration system

- If operators don't themselves have valid investor wallets then calls to `PublicStockOnRamp::swap` will revert resulting in denial of service

Recommended Mitigation: Replace the `investorExists` modifier with an inline check that validates the `_investorWallet` parameter instead of `msg.sender`:

```
modifier investorWalletExists(address _wallet) {
    IDSRegistryService registryService = IDSRegistryService(
        dsToken.getDSService(dsToken.REGISTRY_SERVICE())
    );
    if (!registryService.isWallet(_wallet)) {
        revert InvestorNotRegisteredError();
    }
    _;
}
```

Securitize: Fixed in commit 090cd62.

Cyfrin: Verified.

7.1.3 Zero `curvePriceWad` from rounding causes incorrect pricing or denial of service

Description: The curve price calculation in `SecuritizeAmmNavProvider::_curveBuy` can round to zero when reserves become imbalanced:

```
curvePriceWad = (amountInQuote * WAD) / deltaBase;
```

Mathematically, this simplifies to:

```
curvePriceWad = WAD * (quoteReserves + amountInQuote) / baseReserves

// Proof of simplification:
// Step 1: Expand `deltaBase`
newQuote = Y + amountInQuote

newBase = k / newQuote
        = (X * Y) / (Y + amountInQuote)

deltaBase = X - newBase
          = X - (X * Y) / (Y + amountInQuote)

// Step 2: Find common denominator
deltaBase = X * (Y + amountInQuote) / (Y + amountInQuote) - (X * Y) / (Y + amountInQuote)
          = (X * Y + X * amountInQuote - X * Y) / (Y + amountInQuote)
          = (X * amountInQuote) / (Y + amountInQuote)

// Step 3: Substitute into curvePriceWad
curvePriceWad = (amountInQuote * WAD) / deltaBase
              = (amountInQuote * WAD) / [(X * amountInQuote) / (Y + amountInQuote)]
              = (amountInQuote * WAD) * (Y + amountInQuote) / (X * amountInQuote)
              = WAD * (Y + amountInQuote) / X

Since X = baseReserves, Y = quoteReserves therefore:
curvePriceWad = WAD * (quoteReserves + amountInQuote) / baseReserves
```

For `curvePriceWad` to round to zero:

```
// WAD = 1e18 gives:
1e18 * (quoteReserves + amountInQuote) < baseReserves
```

This state is reachable through repeated sell operations which increase `baseReserves` while depleting `quoteReserves`. For example, if `quoteReserves = 1` and `amountInQuote = 1`, the condition becomes `baseReserves > 2e18`.

When `curvePriceWad` rounds down to zero and is subsequently passed to `_pricingFromCurveBuy`:

```
uint256 r0Wad = (quoteBaseline * WAD) / baseBaseline;
uint256 mWad = (curvePriceWad * WAD) / r0Wad;  // @audit 0

uint256 baseExecPriceWad = (anchorPriceWad * mWad) / WAD;  // @audit 0

// @audit Since 0 < anchorPriceWad, enter the `else` statement:
uint256 diff = anchorPriceWad - baseExecPriceWad;  // @audit = anchorPriceWad
execPriceWad = anchorPriceWad - (diff / priceScaleFactor);

// @audit With default `priceScaleFactor = 2`:
execPriceWad = anchorPriceWad - anchorPriceWad/2 = anchorPriceWad/2

// With `priceScaleFactor = 1`:
execPriceWad = anchorPriceWad - anchorPriceWad = 0
baseOut = (amountInQuote * WAD) / execPriceWad;  // Division by zero → REVERT
```

The same issue exists in `_curveSell`:

```
curvePriceWad = (deltaQuote * WAD) / amountInBase;
```

If `baseReserves` is small and `quoteReserves` is large (from repeated buys), a sell with large `amountInBase` produces tiny `deltaQuote`, rounding `curvePriceWad` to zero.

Impact: Two distinct failure modes depending on `priceScaleFactor`:

Condition	Result
`curvePriceWad = 0`, `scaleFactor  2`	Buyer receives tokens at ~50% of anchor price; seller receives ~50% premium
`curvePriceWad = 0`, `scaleFactor = 1`	Division by zero causes permanent DoS for affected trade direction

A router contract relying on these prices would transfer incorrect token amounts, potentially causing loss of funds for liquidity providers or the protocol.

Recommended Mitigation: Add minimum curve price validation in both curve functions to protect against the case where `curvePriceWad` rounds down to zero:

```
function _curveBuy(uint256 amountInQuote) internal view initialized returns (uint256 curvePriceWad,
↪  uint256 newBase, uint256 newQuote) {
    require(amountInQuote > 0, "amountInQuote=0");

    uint256 X = baseReserves;
    uint256 Y = quoteReserves;
    uint256 kLocal = k;

    newQuote = Y + amountInQuote;
    newBase = kLocal / newQuote;

    uint256 deltaBase = X - newBase;
    require(deltaBase > 0, "deltaBase=0");

    curvePriceWad = (amountInQuote * WAD) / deltaBase;
+   require(curvePriceWad > 0, "curvePriceWad=0");
}

function _curveSell(uint256 amountInBase) internal view initialized returns (uint256 curvePriceWad,
↪  uint256 newBase, uint256 newQuote) {
    require(amountInBase > 0, "amountInBase=0");

    uint256 X = baseReserves;
```

```
    uint256 Y = quoteReserves;
    uint256 kLocal = k;

    newBase = X + amountInBase;
    newQuote = kLocal / newBase;

    uint256 deltaQuote = Y - newQuote;
    require(deltaQuote > 0, "deltaQuote=0");

    curvePriceWad = (deltaQuote * WAD) / amountInBase;
+   require(curvePriceWad > 0, "curvePriceWad=0");
}
```

Additionally, consider enforcing a minimum `priceScaleFactor` of 2 to prevent division-by-zero in the pricing functions:

```
function setPriceScaleFactor(uint256 newScaleFactor) external onlyRole(DEFAULT_ADMIN_ROLE) {
-   require(newScaleFactor > 0, "scaleFactor = 0");
+   require(newScaleFactor >= 2, "scaleFactor must be >= 2");

    uint256 oldScaleFactor = priceScaleFactor;
    priceScaleFactor = newScaleFactor;

    emit PriceScaleFactorUpdated(oldScaleFactor, newScaleFactor);
}
```

Securitize: Fixed in commit bcd6e87.

Cyfrin: Verified.

7.1.4 `SecuritizeOnRamp::swap` and `SecuritizeOffRamp::redeem` pass operator as investor address resulting in denial of service

Description: `SecuritizeOnRamp::swap` and `SecuritizeOffRamp::redeem` have modifier `onlyRole(OPERATOR_ROLE)` meaning only an operator can call these functions.

But when calling child functions such as `BaseOnRamp::_swap` or `BaseOffRamp::_redeem`, they pass `_msgSender()` as the investor wallet which is incorrect since the operator and the investor are not the same entities.

Impact: The most likely impact is temporary denial of service; these functions will revert as the operator is not also an investor. But the contracts are upgradeable so can be fixed or in a worst-case scenario re-deployed.

Recommended Mitigation: If these functions are designed to be called by investors then remove the modifier `onlyRole(OPERATOR_ROLE)`.

Otherwise if they are supposed to be called by an operator, then add an input parameter to specify the investor address.

Also see the related issue M-1 "Incorrect use of `investorExists` modifier in `PublicStockOnRamp::swap`" which may also be relevant here.

Securitize: Fixed in commit 5cbd6d8 by removing the `onlyRole` as these functions are intended to be directly called by investors.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 `RedStoneNavProvider::rate` will return massively inflated value if underlying oracle returns a negative value and lacks common oracle validations

Description: `RedStoneNavProvider::rate` has an underlying oracle price feed which returns an `int256`, then performs an unsafe cast to `uint256`:

```
int256 rsRate = priceFeed.latestAnswer();
require(rsRate != 0, 'Rate must not be zero');
return Helper.normalizeRate(uint256(rsRate), oracleDecimals, assetDecimals);
```

Impact: If the underlying oracle returned a negative number, performing an unsafe cast to `uint256` will return an absurdly high rate.

Recommended Mitigation: Enforce that the rate returned by the underlying oracle must be greater than zero:

```
    function rate() external view override returns (uint256) {
        uint8 oracleDecimals = priceFeed.decimals();
        uint8 assetDecimals = asset.decimals();
        int256 rsRate = priceFeed.latestAnswer();
-       require(rsRate != 0, 'Rate must not be zero');
+       require(rsRate > 0, 'Rate must be greater than zero');
        return Helper.normalizeRate(uint256(rsRate), oracleDecimals, assetDecimals);
    }
```

Also consider adding other common oracle-related checks such as:

- staleness
- min/max price thresholds

Securitize: Fixed in commit ec23faf.

Cyfrin: Verified.

7.2.2 `SecuritizeAmmNavProvider` quote functions don't reflect execution behavior due to missing baseline reset logic

Description: `SecuritizeAmmNavProvider::quoteBuyBase, quoteSellBase` are supposed to provide price quotes, however they don't simulate the behavior of `_checkAndResetBaseline` which is executed by actual buys and sells performed via `executeBuyBase, executeSellBase`.

Impact: Actual price execution can differ from the quoted price; the quotes may not be accurate because they don't simulate the `_checkAndResetBaseline` logic.

Recommended Mitigation: Either document this limitation clearly or implement the baseline reset simulation in quote functions (though this adds complexity to `view` functions).

Securitize: Fixed in commits 71e40b8, d05482b.

Cyfrin: Verified.

7.2.3 Lack of Price Feed Update Function in `RedStoneNavProvider`

Description: The `RedStoneNavProvider` contract sets the RedStone price feed address during initialization but provides no mechanism to update it afterwards. The priceFeed is set once in the `RedStoneNavProvider::initialize` function and becomes immutable for the lifetime of the contract:

```
function initialize(address _priceFeed, address _asset) public onlyProxy initializer {
        __BaseDSContract_init();
        priceFeed = IPriceFeed(_priceFeed);
        asset = IERC20Metadata(_asset);
    }
```

The contract declares the `priceFeed` as a public state variable but offers no admin function to update it, f the RedStone oracle address needs to be changed due to:

- Oracle migration to a new address
- Oracle deprecation or compromise
- Need to switch to a different price feed
- Oracle upgrade or maintenance

The only solution is to:

1. Deploy a new `RedStoneNavProvider` contract with the new feed address
2. Call `updateNavProvider()` on all `OnRamp/OffRamp` contracts using this provider
3. Potentially requiring governance votes or multi-sig operations

Impact: Requires complete contract redeployment instead of a simple parameter update in case of the `priceFeed` need to be changed.

Recommended Mitigation: Add an admin-controlled function to update the price feed address.

Securitize: Fixed in commit 4146a77.

Cyfrin: Verified.

7.2.4 Signatures used in `PublicStockOnRamp` and `PublicStockOffRamp` lack investor-specified deadline and nonce parameters so can be used multiple times by operators

Description: Both `PublicStockOnRamp` and `PublicStockOffRamp` contracts use EIP-712 signatures to authorize investor transactions. However, these signatures do not include a deadline/expiration parameter in their signed data structure, making them valid indefinitely until executed. In `PublicStockOnRamp`, the signature only includes the liquidity amount and minimum output amount:

```
bytes32 private constant TXTYPE_HASH = keccak256("Swap(uint256 liquidityAmount,uint256 minOutAmount)");

  function hashTx(uint256 _liquidityAmount, uint256 _minOutAmount) private view returns (bytes32) {
        bytes32 structHash = keccak256(
            abi.encode(TXTYPE_HASH, _liquidityAmount, _minOutAmount)
        );

        return _hashTypedDataV4(structHash);
    }
```

Similarly in `PublicStockOffRamp`:

```
bytes32 private constant TXTYPE_HASH = keccak256("Redeem(uint256 assetAmount,uint256 minOutputAmount)");

function hashTx(uint256 _assetAmount, uint256 _minOutputAmount) private view returns (bytes32) {
        bytes32 structHash = keccak256(
            abi.encode(TXTYPE_HASH, _assetAmount, _minOutputAmount)
        );

        return _hashTypedDataV4(structHash);
    }
```

While both contracts have `_anchorPriceExpiresAt` as a function parameter to ensure the price feed isn't stale, this expiration is not part of the signed message. An investor's signature remains valid indefinitely and can be executed at any future time by an operator, as long as they provide a valid (non-expired) anchor price. Additionally, there is no nonce mechanism in either contract to allow investors to invalidate/cancel previously signed transactions.

Impact: Once an investor signs a transaction, they cannot invalidate it even if market conditions change significantly, An operator could hold onto a signature for days/weeks/months and execute it at an inopportune time for

the investor. Because there is no nonce used the operator could use the investor's signature to execute multiple transactions.

Recommended Mitigation: Include a deadline parameter in the signed message structure and implement a nonce perhaps using NoncesUpgradeable or similar to how `SecuritizeOnRamp` does it via the mapping `noncePerInvestor`. Since `PublicStockOnRamp` and `SecuritizeOnRamp` both inherit from `BaseOnRamp`, it may be ideal to move the common functionality into there.

Securitize: Fixed in commit 85142ed.

Cyfrin: Verified.

7.2.5 `SecuritizeAmmNavProvider` virtual reserve rounding erosion can lead to denial of service

Description: The virtual AMM's constant-product math erodes `k` through integer division truncation on every trade:

```
// In `_curveBuy`:
newQuote = Y + amountInQuote;
newBase = kLocal / newQuote;  // Rounds down

// After trade execution:
baseReserves = newBase;
quoteReserves = newQuote;
k = newBase * newQuote;  // New k  old k due to truncation
```

Each trade can lose up to `(denominator - 1)` from k. Over many trades, `baseReserves` (or `quoteReserves` for sells) progressively decreases.

The `_checkAndResetBaseline` function can accelerate this by locking in depleted reserves when market status changes from OPEN → CLOSED:

```
if (shouldReset) {
    uint256 newBase = baseReserves;  // Uses current depleted value
    uint256 newQuote = (newBase * anchorPriceWad) / WAD;
    _resetBaseline(newBase, newQuote);  // k = newBase * newQuote (tiny)
}
```

Once k becomes sufficiently small, a single trade can round reserves to zero:

```
// If k = 1, quoteReserves = 1, and amountInQuote = 2:
newQuote = 1 + 2 = 3;
newBase = 1 / 3 = 0;  // Rounds to zero

// Trade completes (deltaBase = 1 passes the check), then:
baseReserves = 0;
k = 0 * 3 = 0;
```

All subsequent trades revert at the `initialized` modifier:

```
modifier initialized() {
    require(baseReserves > 0 && quoteReserves > 0, "uninitialized");
    // ...
}
```

Impact: Denial of service for all trade execution until admin manually calls `resetBaseline` to restore valid reserves.

Recommended Mitigation: Add minimum reserve thresholds to prevent reserves from falling into the danger zone. This could be done using new admin-configurable storage slots and should be based on the decimals of the `asset`:

```
+ uint256 public minReserves;
```

```
function initialize(uint256 _baseReserves, uint256 _quoteReserves, address _asset) public onlyProxy
↪  initializer {
    // ... existing checks ...

    asset = IERC20Metadata(_asset);
+   uint8 d = asset.decimals();
+   uint256 minReservesTemp = 10 ** d;  // 1 whole token minimum

+   require(_baseReserves >= minReservesTemp, "baseReserves too small");
+   require(_quoteReserves >= minReservesTemp, "quoteReserves too small");
+   minReserves = minReservesTemp;

    // ... rest of initialization
}

function _resetBaseline(uint256 newBase, uint256 newQuote) internal {
    require(newBase > 0, "newBase=0");
    require(newQuote > 0, "newQuote=0");

+   uint256 minReservesCache = minReserves;
+   require(newBase >= minReservesCache, "newBase too small");
+   require(newQuote >= minReservesCache, "newQuote too small");

    // ... rest of function
}

function _curveBuy(uint256 amountInQuote) internal view initialized returns (uint256 curvePriceWad,
↪  uint256 newBase, uint256 newQuote) {
    require(amountInQuote > 0, "amountInQuote=0");

    uint256 X = baseReserves;
    uint256 Y = quoteReserves;
    uint256 kLocal = k;

    newQuote = Y + amountInQuote;
    newBase = kLocal / newQuote;

+   require(newBase >= minReserves, "base reserves too low");

    uint256 deltaBase = X - newBase;
    require(deltaBase > 0, "deltaBase=0");

    curvePriceWad = (amountInQuote * WAD) / deltaBase;
}

function _curveSell(uint256 amountInBase) internal view initialized returns (uint256 curvePriceWad,
↪  uint256 newBase, uint256 newQuote) {
    require(amountInBase > 0, "amountInBase=0");

    uint256 X = baseReserves;
    uint256 Y = quoteReserves;
    uint256 kLocal = k;

    newBase = X + amountInBase;
    newQuote = kLocal / newBase;

+   require(newQuote >= minReserves, "quote reserves too low");

    uint256 deltaQuote = Y - newQuote;
    require(deltaQuote > 0, "deltaQuote=0");

    curvePriceWad = (deltaQuote * WAD) / amountInBase;
```

```
}
```

Another benefit of enforcing minimum reserves is a consistent pattern of high-profile mainnet hacks have involved attackers manipulating pool reserves to very low wei amounts; enforcing minimum reserves acts as defensive programming technique helping to reduce the attack surface available to hackers.

Securitize: Fixed in commit 1919a89.

Cyfrin: Verified.

7.2.6 Missing zero output validation in `SecuritizeAmmNavProvider` quote and buy functions

Description: `SecuritizeAmmNavProvider::quoteBuyBase, quoteSellBase, executeBuyBase, executeSellBase` lack validation that output values are non-zero. Apart from issues already mentioned where rounding down to zero can occur, there are multiple other calculation points which can result in the final return values of these functions round down to zero:

1. `baseOut` **in buy operations:**

```
baseOut = (amountInQuote * WAD) / rawExecPriceWad;
```

Rounds to zero when `amountInQuote < rawExecPriceWad / 1e18`. For example, if `rawExecPriceWad = 100e18` (100 quote per base), any `amountInQuote < 100` produces `baseOut = 0`.

2. `quoteOut` **in sell operations:**

```
quoteOut = (amountInBase * rawExecPriceWad) / WAD;
```

Rounds to zero when `amountInBase * rawExecPriceWad < 1e18`. For example, if `rawExecPriceWad = 1e16` (0.01 quote per base), any `amountInBase < 100` produces `quoteOut = 0`.

3. `execPrice` **in both directions:**

```
uint256 scaleDown = 10 ** (18 - d);
execPrice = rawExecPriceWad / scaleDown;
```

Rounds to zero when `rawExecPriceWad < scaleDown`. For a 6-decimal asset, `scaleDown = 1e12`, so any `rawExecPriceWad < 1e12` produces `execPrice = 0`.

These conditions can occur through:

- Small trade amounts relative to price
- Extreme price deviations from curve imbalance (as described in another issue)
- Low-decimal assets with unfavorable price scaling

Impact: When zero outputs are returned the potential negative outcomes include:

1. **Silent fund loss** — A router calling `executeBuyBase` or `executeSellBase` may transfer real tokens. The NAV provider returns `baseOut = 0` or `quoteOut = 0`, but the router may not distinguish this from a legitimate trade. In a worst-case scenario the user could send tokens and receive nothing, though this is unlikely.
2. **State corruption** — Virtual reserves update based on a trade that produced zero output:

```
  baseReserves = newBase;
  quoteReserves = newQuote;
  k = newBase * newQuote;
```

The AMM state reflects input that was "absorbed" without corresponding output.

3. **Broken invariants** — The constant-product invariant assumes balanced input/output. Zero-output trades violate this assumption and skew future pricing.

4. **Misleading quotes** — The view functions `quoteBuyBase` and `quoteSellBase` return zero outputs without reverting, causing off-chain integrations to display incorrect expectations.

Recommended Mitigation: Add zero-output validation to all four functions:

```
function executeBuyBase(
    uint256 amountInQuote,
    uint256 anchorPriceWad,
    uint8 marketStatus
) external onlyRole(EXECUTOR_ROLE) returns (uint256 baseOut, uint256 execPrice) {
    // ... existing logic ...
    if (marketStatus == CLOSED_MARKET) {
        (baseOut, rawExecPriceWad) = _pricingFromCurveBuy(amountInQuote, curvePriceWad, anchorPriceWad);
    } else if (marketStatus == OPEN_MARKET) {
        rawExecPriceWad = anchorPriceWad;
        baseOut = (amountInQuote * WAD) / rawExecPriceWad;
    } else {
        revert("invalid market status");
    }

+   require(baseOut > 0, "baseOut=0");

    baseReserves = newBase;
    quoteReserves = newQuote;
    k = newBase * newQuote;

    _recordTrade(marketStatus, anchorPriceWad);

    uint8 d = asset.decimals();
    require(d <= 18, "decimals > 18");
    uint256 scaleDown = 10 ** (18 - d);

    execPrice = rawExecPriceWad / scaleDown;

+   require(execPrice > 0, "execPrice=0");

    emit ExecuteBuy(msg.sender, amountInQuote, baseOut, rawExecPriceWad);
}

function executeSellBase(
    uint256 amountInBase,
    uint256 anchorPriceWad,
    uint8 marketStatus
) external onlyRole(EXECUTOR_ROLE) returns (uint256 quoteOut, uint256 execPrice) {
    // ... existing logic ...
    if (marketStatus == CLOSED_MARKET) {
        (quoteOut, rawExecPriceWad) = _pricingFromCurveSell(amountInBase, curvePriceWad,
        ↪  anchorPriceWad);
    } else if (marketStatus == OPEN_MARKET) {
        rawExecPriceWad = anchorPriceWad;
        quoteOut = (amountInBase * rawExecPriceWad) / WAD;
    } else {
        revert("invalid market status");
    }

+   require(quoteOut > 0, "quoteOut=0");

    baseReserves = newBase;
    quoteReserves = newQuote;
    k = newBase * newQuote;

    _recordTrade(marketStatus, anchorPriceWad);
```

```
    uint8 d = asset.decimals();
    require(d <= 18, "decimals > 18");
    uint256 scaleDown = 10 ** (18 - d);

    execPrice = rawExecPriceWad / scaleDown;

+   require(execPrice > 0, "execPrice=0");

    emit ExecuteSell(msg.sender, amountInBase, quoteOut, rawExecPriceWad);
}

function quoteBuyBase(
    uint256 amountInQuote,
    uint256 anchorPriceWad,
    uint8 marketStatus
) external view returns (uint256 baseOut, uint256 execPrice) {
    // ... existing logic ...
+   require(baseOut > 0, "baseOut=0");

    execPrice = rawExecPriceWad / scaleDown;
+   require(execPrice > 0, "execPrice=0");
}

function quoteSellBase(
    uint256 amountInBase,
    uint256 anchorPriceWad,
    uint8 marketStatus
) external view returns (uint256 quoteOut, uint256 execPrice) {
    // ... existing logic ...
+   require(quoteOut > 0, "quoteOut=0");

    execPrice = rawExecPriceWad / scaleDown;
+   require(execPrice > 0, "execPrice=0");
}
```

Securitize: Fixed in commit affb350.

Cyfrin: Verified.

7.2.7 `RedStoneNavProvider::rate` can return zero for non-zero oracle input due to rounding in `Helper::normalizeRate`

Description: In `RedStoneNavProvider::rate`, the zero check validates the raw oracle value but not the normalized result:

```
function rate() external view override returns (uint256) {
    uint8 oracleDecimals = priceFeed.decimals();
    uint8 assetDecimals = asset.decimals();
    int256 rsRate = priceFeed.latestAnswer();
    require(rsRate != 0, 'Rate must not be zero');  // @audit Checks raw value only

    // @audit normalized rate never zero checked
    return Helper.normalizeRate(uint256(rsRate), oracleDecimals, assetDecimals);
}
```

The `Helper::normalizeRate` function divides when `fromDecimals > toDecimals`:

```
function normalizeRate(uint256 value, uint8 fromDecimals, uint8 toDecimals) internal pure returns
↪  (uint256) {
    if (fromDecimals == toDecimals) {
        return value;
    } else if (fromDecimals > toDecimals) {
        return value / (10**(fromDecimals - toDecimals));  // @audit Can round to zero
```

```
    } else {
        return value * (10**(toDecimals - fromDecimals));
    }
}
```

When the raw oracle value is smaller than the divisor, integer division truncates to zero:

```
oracleDecimals = 18 (common for RedStone)
assetDecimals = 6 (like USDC)
rsRate = 5e11 (passes the != 0 check)
divisor = 10^(18 - 6) = 1e12
normalizedRate = 5e11 / 1e12 = 0
```

The `require(rsRate != 0)` passes because `5e11 != 0`, but the function returns `0`.

This could occur with:

- Severely devalued assets approaching zero value
- Misconfigured oracle/asset decimal pairing
- Oracle malfunction returning unexpectedly small values
- Exotic assets with very low unit prices

Impact: Any protocol consuming `RedStoneNavProvider::rate` receives zero, potentially causing a number of errors if the protocol doesn't revert such as:

- Incorrect NAV calculations valuing assets at zero
- Potential division-by-zero errors in downstream calculations
- Users trading at incorrect prices, either losing funds or extracting value from the protocol

Recommended Mitigation: Add a zero check on the normalized result:

```
function rate() external view override returns (uint256 normalizedRate) {
    uint8 oracleDecimals = priceFeed.decimals();
    uint8 assetDecimals = asset.decimals();
    int256 rsRate = priceFeed.latestAnswer();
    require(rsRate > 0, 'Rate must be positive');

    normalizedRate = Helper.normalizeRate(uint256(rsRate), oracleDecimals, assetDecimals);
    require(normalizedRate > 0, 'Normalized rate is zero');
}
```

Securitize: Fixed in commit f4bed90.

Cyfrin: Verified.

7.2.8 Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision

Description: The protocol has upgradeable contracts which other contracts inherit from. These contracts should either use:

- ERC7201 namespaced storage layouts - example
- storage gaps (though this is an older and no longer preferred method)

The ideal mitigation is that all upgradeable contracts use ERC7201 namespaced storage layouts; without using one of the above two techniques storage collision can occur during upgrades. The affected contracts are:

- `1-onoff-ramp/contracts/on-ramp/BaseOnRamp.sol` inherited by `PublicStockOnRamp.sol` which has its own state

Securitize: Fixed in commit 1656d74.

Cyfrin: Verified.

7.2.9 `SecuritizeOnRamp` doesn't provide a mechanism for investors to invalidate their nonces

Description: `SecuritizeOnRamp` has no function allowing investors to invalidate or increment their own nonce. The nonce only increments when `executePreApprovedTransaction` successfully executes.

Impact: If an investor signs an authorization and later changes their mind (market conditions changed, signed wrong parameters, etc.), they cannot cancel the pending authorization; the operator can still execute it.

Recommended Mitigation: Nonce implementations commonly provide a function for users to invalidate their nonces; for example OZ NoncesUpgradeable has a _useNonce function which can be exposed to allow users to invalidate their own nonces.

Securitize: Acknowledged; at this time we don't want to expose user nonce validation for this particular functionality.

7.2.10 Incorrect rounding direction in `SecuritizeAmmNavProvider::executeBuyBase` when scaling down `execPrice`

Description: In `SecuritizeAmmNavProvider::executeBuyBase`, the execution price is scaled down from WAD precision (18 decimals) to the asset's native decimal precision using integer division. The current implementation uses floor division, which systematically rounds DOWN the execution price:

```
uint256 scaleDown = 10 ** (18 - d);

execPrice = rawExecPriceWad / scaleDown;
```

When users buy base assets, rounding the price DOWN means they pay less than the true calculated price. For example, with a 6-decimal token where `scaleDown` = 10^12:

- True price: $100.0000007 → `rawExecPriceWad` = 100,000,000,700,000,000,000
- Rounded price: $100.000000 → `execPrice` = 100,000,000
- Loss: $0.0000007 per share (always favors the buyer)

This issue is especially significant in `CLOSED_MARKET` mode where AMM-based curve pricing is applied. In `CLOSED_MARKET`, the execution price calculation involves:

```
function _pricingFromCurveBuy(
        uint256 amountInQuote,
        uint256 curvePriceWad,
        uint256 anchorPriceWad
    ) internal view returns (uint256 baseOut, uint256 execPriceWad) {
        require(anchorPriceWad > 0, "anchor=0");
        require(priceScaleFactor > 0, "scaleFactor = 0");

        uint256 rOWad = (quoteBaseline * WAD) / baseBaseline;
        uint256 mWad = (curvePriceWad * WAD) / rOWad;

        uint256 baseExecPriceWad = (anchorPriceWad * mWad) / WAD;

        // Smooth towards anchor:
        // execPriceWad = anchor + (baseExec - anchor) / scaleFactor
        if (baseExecPriceWad >= anchorPriceWad) {
            uint256 diff = baseExecPriceWad - anchorPriceWad;
            execPriceWad = anchorPriceWad + (diff / priceScaleFactor);
        } else {
            uint256 diff = anchorPriceWad - baseExecPriceWad;
            execPriceWad = anchorPriceWad - (diff / priceScaleFactor);
        }
```

```
        baseOut = (amountInQuote * WAD) / execPriceWad;
    }
```

These operations produce prices with deep fractional precision that almost never align with the `scaleDown` boundaries. This means every `CLOSED_MARKET` trade experiences truncation loss, unlike `OPEN_MARKET` mode where prices are set directly to clean anchor values.

Impact: On buy operations the protocol systematically loses a fractional amount on every trade. While the per-trade loss is small (typically sub-cent), it accumulates over time and volume.

Recommended Mitigation: Round up the final `execPrice` in `executeBuyBase`. Consider:

- using OZ Math::mulDiv with explicit rounding for all multiplication followed by division
- documenting with comments why the rounding direction is logically correct at every place where rounding can occur

Securitize: Fixed in commit 0b268e8.

Cyfrin: Verified.

7.2.11 Incorrect rounding of rate in favor of users when they buy DS Tokens

Description: Consider this line in `Helper::normalizeRate` which always rounds down:

```
26:             return value / (10**(fromDecimals - toDecimals));
```

`Helper::normalizeRate` is called by `RedStoneNavProvider::rate`, which is called by both `SecuritizeOnRamp.sol` and `SecuritizeOffRamp.sol`.

`SecuritizeOnRamp::calculateDsTokenAmount` is used to buy DSTokens and does this:

```
rate = navProvider.rate(); // assumed to be in `assetDecimals`

dsTokenAmount = (liquidityAmountExcludingFee * (10 ** (2 * assetDecimals))) / (rate * (10 **
↪  liquidityTokenDecimals));
```

Since `rate` is used in the denominator, a smaller value due to rounding down results in the user getting slightly more DSTokens.

Impact: Incorrect rounding of rate in favor of users gives users slightly more DSTokens than they should receive.

Recommended Mitigation: * `ISecuritizeNavProvider::rate` should have an input parameter which specifies rounding direction so that callers can specify the appropriate rounding direction based on their needs

- the rounding direction should be passed as input to `Helper::normalizeRate`
- `Helper::normalizeRate` should use OZ Math::mulDiv with explicit rounding at L26 where it performs division

Securitize: Acknowledged for now as we don't want to change the interface at this time. In a real life scenario there shouldn't be any actual impact because for Redstone Push model they update the price querying our API, and our price never exceeds token decimals. So normally Redstone price has 8 decimals and our tokens 6, so it leads in a price with two zeros at the end.

7.2.12 Block number deadline is chain dependent and unreliable across L2s

Description: The `SecuritizeOnRamp::subscribe` uses `block.number` as a deadline mechanism to prevent stale transactions:

```
function subscribe(
        string memory _investorId,
        address _investorWallet,
        string memory _investorCountry,
```

```
        uint8[] memory _investorAttributeIds,
        uint256[] memory _investorAttributeValues,
        uint256[] memory _investorAttributeExpirations,
        uint256 _minOutAmount,
        uint256 _liquidityAmount,
        uint256 _blockLimit,
        bytes32 _agreementHash
    )
        public
        whenNotPaused
        onlySecuritizeOnRamp
        nonZeroNavRate
        validateMinSubscriptionAmount(_liquidityAmount)
    {
        if (_blockLimit < block.number) {
            revert TransactionTooOldError();
        }//@audit where this is going to be deployed?
        ...
```

The protocol is intended to be deployed in this chains as evidenced by the 'hardhat.config.ts:

```
networks: {
        sepolia: {
            chainId: 11155111,
            url: process.env.SEPOLIA_RPC_URL ?? '',
            accounts: [process.env.DEPLOYER_PRIV_KEY!].filter((x) => x),
        },
        arbitrum: {
            chainId: 421614,
            // ...
        },
        optimism: {
            chainId: 11155420,
            // ...
        },
        avaxtest: {
            // ...
        },
    },
```

Block production rates differ drastically across this chains:

- Ethereum mainnet: ~12 seconds per block
- Optimism: ~2 seconds per block
- Arbitrum: ~0.25-0.5 seconds per block

Impact: A `_blockLimit` value that provides a reasonable time window on Ethereum becomes nearly useless on L2s. Setting `_blockLimit = currentBlock + 300` provides:

- Ethereum: ~1 hour window
- Optimism: ~10 minutes window
- Arbitrum: ~75-150 seconds window

This could lead valid transactions unexpectedly reverting with `TransactionTooOldError()` on L2s if users/backend calculate `_blockLimit` based on Ethereum assumptions

Recommended Mitigation: Replace `block.number` with `block.timestamp` for chain-agnostic deadline enforcement.

Securitize: Acknowledged; in practice there is no risk as block span is set by our services and we configure that number per blockchain. We do not want to change this for now because it's attached to our backend.

7.2.13 `SecuritizeAmmNavProvider` violates core AMM invariant that `k` should never decrease

Description: Both `SecuritizeAmmNavProvider::_curveBuy` and `_curveSell` round down when calculating new reserves:

```
// `_curveBuy`
newQuote = Y + amountInQuote;
newBase = kLocal / newQuote; // @audit rounds down, k decreases

// `_curveSell`
newBase = X + amountInBase;
newQuote = kLocal / newBase; // @audit rounds down, k decreases
```

This causes users to receive slightly more output than mathematically correct and also `k` to effectively decrease over time.

Best practice is to have an invariant that `k` never decreases (as can be seen in `UniswapV2Pair::swap`). I.e. round in favor of the protocol.

Impact: Dust-level value leakage per trade; minimal practical impact due to virtual AMM design and periodic resets.

Recommended Mitigation: Consider explicitly rounding up using OZ Math::ceilDiv.

Securitize: Fixed in commit 04d2392.

Cyfrin: Verified.

7.2.14 `SecuritizeAmmNavProvider` trades when pool price and anchor price differ can leak value

Description: Due to the asymmetric smoothing formula and integer truncation, a BUY followed by an immediate SELL at the same `anchorPriceWad` and `marketStatus = CLOSED_MARKET` can return more quote than initially spent (profitable round-trip), leaking value from the virtual pool.

Impact: This can slowly leak value out of the pool if not monitored.

Proof of Concept: First add Foundry to the existing Hardhat repo, then create a new file `RoundTripTest.t.sol` inside the `test` folder containing:

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

import "forge-std/Test.sol";
import {SecuritizeAmmNavProvider} from "../contracts/nav/SecuritizeAmmNavProvider.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract AssetMock {
    uint8 public decimals = 6;
}

contract RoundTripTest is Test {
    SecuritizeAmmNavProvider public navProvider;
    uint256 constant WAD = 1e18;

    function setUp() public {
        AssetMock asset = new AssetMock();
        SecuritizeAmmNavProvider impl = new SecuritizeAmmNavProvider();

        ERC1967Proxy proxy = new ERC1967Proxy(
            address(impl),
            abi.encodeWithSelector(
                SecuritizeAmmNavProvider.initialize.selector,
                100_000e6,  // baseReserves
                100_000e6,  // quoteReserves
```

```
                address(asset)
            )
        );

        navProvider = SecuritizeAmmNavProvider(address(proxy));
        navProvider.grantRole(navProvider.EXECUTOR_ROLE(), address(this));
    }

    function test_roundTrip() public {
        uint256 quoteIn = 10e6;
        uint256 anchorPrice = 1.01e18;
        uint8 closedMarket = navProvider.CLOSED_MARKET();

        (uint256 baseOut,) = navProvider.executeBuyBase(quoteIn, anchorPrice, closedMarket);
        (uint256 quoteBack,) = navProvider.executeSellBase(baseOut, anchorPrice, closedMarket);

        assertLt(quoteBack, quoteIn, "Round-trip should result in loss");
    }
}
```

Then run using: `forge test --match-contract RoundTripTest`.

Recommended Mitigation: Consider redesigning the smoothing formula to be symmetric to prevent a slight leak of value.

Securitize: Formulas changed in commits `0b268e8` and `7d7a4cf` which minimizes the round trip issue.

Cyfrin: Partly verified. The round trip can still make a profit, however the cost of the round trip (gas and protocol fees) is higher than any profit hence not a viable attack.

7.3 Informational

7.3.1 Use named mapping parameters to explicitly denote the purpose of keys and values

Description: Use named mapping parameters to explicitly denote the purpose of keys and values:

```
off-ramp/BaseOffRamp.sol
43:     mapping(string => bool) public restrictedCountries;

off-ramp/CountryValidator.sol
26:         mapping(string => bool) storage _restrictedCountries

on-ramp/SecuritizeOnRamp.sol
41:     mapping(string => uint256) internal noncePerInvestor;
```

Securitize: Fixed in commit ef5367e.

Cyfrin: Verified.

7.3.2 Consider enforcing minimum redemption amounts in `PublicStockOffRamp`, `SecuritizeOffRamp::redeem`

Description: The `PublicStockOnRamp` contract enforces a minimum subscription amount check via the `validateMinSubscriptionAmount` modifier to prevent dust trades:

```
modifier validateMinSubscriptionAmount(uint256 _amount) {
        if (_amount < minSubscriptionAmount) {
            revert MinSubscriptionAmountError();
        }
        _;
    }
```

However, the `PublicStockOffRamp::redeem` and `SecuritizeOffRamp::redeem` lacks an equivalent validation on the `_assetAmount` input parameter. This allows users to submit redemption requests with arbitrarily small amounts (e.g., _assetAmount = 1).

Impact: * Dust redemptions that pass signature validation but fail at slippage check waste gas for the operator.

- possible problem due rounding issues in the amm

Recommended Mitigation: Add a minimum redemption amount validation to the offramp contracts, consistent with the OnRamp pattern:

Securitize: Acknowledged.

7.3.3 Missing `liquidityToken` validation in `CollateralLiquidityProvider::initialize`

Description: The `CollateralLiquidityProvider::setExternalCollateralRedemption` function includes a validation check to ensure the liquidity token of the new external collateral redemption contract matches the existing `liquidityToken`:

```
function setExternalCollateralRedemption(address _externalCollateralRedemption) external
↪  onlyRole(DEFAULT_ADMIN_ROLE) {
        if (_externalCollateralRedemption == address(0)) {
            revert NonZeroAddressError();
        }

        if (
            address(
                ILiquidityProvider(address(ISecuritizeOffRamp(_externalCollateralRedemption).liquidityP
                ↪  rovider()))
                    .liquidityToken()
            ) != address(liquidityToken)//@audit-ok (low) why this i not checked in intializatuion
```

```
        ) {
            revert LiquidityTokenMismatch();
        }
        address oldExternalCollateral = address(externalCollateralRedemption);
        externalCollateralRedemption = ISecuritizeOffRamp(_externalCollateralRedemption);
        emit ExternalCollateralRedemptionUpdated(oldExternalCollateral,
        ↪  address(externalCollateralRedemption));
    }
```

However, `CollateralLiquidityProvider::initialize` sets the same `externalCollateralRedemption` without performing this validation:

```
function initialize(
    address _liquidityToken,
    address _recipient,
    address _securitizeOffRamp,
    address _externalCollateralRedemption,
    address _collateralProvider
) public onlyProxy initializer {
    // ... zero address checks only ...

    liquidityToken = IERC20Metadata(_liquidityToken);
    externalCollateralRedemption = ISecuritizeOffRamp(_externalCollateralRedemption);
    ...
    //@audit missing validation that _externalCollateralRedemption's liquidity token matches
    ↪  _liquidityToken
}
```

Impact: During contract deployment, an admin could mistakenly initialize the contract with an `_externalCollateralRedemption` that uses a different liquidity token than the configured `_liquidityToken`.

Recommended Mitigation: Add the liquidity token validation to the `CollateralLiquidityProvider::initialize`.

Securitize: Fixed in commit d8fd4fb.

Cyfrin: Verified.

7.3.4 Not way to change the recipient in the liquidity provider in the offramp logic

Description: Both `AllowanceLiquidityProvider` and `CollateralLiquidityProvider` contracts set the recipient address only during initialization, with no function to update it afterwards. In `RedemptionManager.sol`, when `assetBurn` is false, tokens are transferred to the recipient:

```
if (_params.assetBurn) {
            _params.asset.burn(_params.redeemer, _params.assetAmount, "Redemption burn");
        } else {
            _params.asset.transferFrom(_params.redeemer, _params.liquidityProvider.recipient(),
            ↪  _params.assetAmount);//@audit not way to update the recipient?
        }
```

However, neither liquidity provider implements a setter for the recipient address. While both contracts provide setters for other addresses (`setAllowanceProviderWallet`, `setCollateralProvider`, `setExternalCollateralRedemption`), the recipient address cannot be modified after deployment.

Impact: If the recipient wallet is compromised, needs to change for operational reasons, or the receiving entity updates their wallet address, there is no way to update it without performing a full contract upgrade.

Recommended Mitigation: Add a `setRecipient` function to both `AllowanceLiquidityProvider` and `CollateralLiquidityProvider`.

Securitize: Acknowledged.

7.3.5 Single step redemption and two step redemption not equivalent logic

Description: The `RedemptionManager::executeSingleStepRedemption` has multiple issues when used with `CollateralLiquidityProvider`

1. Fee calculated on requested amount, not actual amount:

```
        // Apply fee if it exists, transfer it to the fee collector
        fee = TokenCalculator.calculateFee(_params.feeManager, _params.liquidityTokenAmount);
```

The fee is calculated based on `_params.liquidityTokenAmount` (the requested amount), but `CollateralLiquidityProvider::supplyTo` returns a different (smaller) amount due to external redemption fees:

```
function supplyTo(
        address _redeemer,
        uint256 _liquidityAmount
    ) public whenNotPaused onlySecuritizeRedemption returns (uint256 amountToSupply) {
        ...
        // Discount the fee charged by the external collateral redemption
        amountToSupply = externalCollateralRedemption.calculateLiquidityTokenAmount(collateralAmount);

        // Supply redeemer
        liquidityToken.transfer(_redeemer, amountToSupply);
    }
```

2. The return value of fee `supplyTo` is ignored:

```
 if (fee > 0) {
            _params.liquidityProvider.supplyTo(IFeeManager(_params.feeManager).feeCollector(), fee);
        }//@audit why not supplied here?
```

The return value is not captured. The fee collector receives less than fee due to external redemption fees, but the function returns the original calculated fee, causing incorrect accounting.

3. Double external redemption fees:

```
 // Supply liquidity tokens to the fee collector
        if (fee > 0) {
            _params.liquidityProvider.supplyTo(IFeeManager(_params.feeManager).feeCollector(), fee);
        }//@audit why not supplied here?

        // Supply liquidity tokens to the redeemer
        userSuppliedAmount = _params.liquidityProvider.supplyTo(_params.redeemer,
        ↪  _params.liquidityTokenAmount - fee);
```

Each `CollateralLiquidityProvider ::supplyTo` call triggers a full external redemption flow compare to `executeTwoStepRedemption` which correctly uses a single `supplyTo` call and calculates fees on the actual supplied amount

Recommended Mitigation: Consider make single step redemption and two step redemption equivalent.

Securitize: Acknowledged.

7.3.6 Use `SafeERC20` approval and transfer functions instead of standard IERC20 functions for `liquidityToken`

Description: The on-ramping and off-ramping processes are linked to external liquidity tokens such as stablecoins whose code is not controlled by the protocol; hence use `SafeERC20::forceApprove, transfer, safeTransfer` when dealing with a range of potential tokens:

```
on-ramp/provider/AllowanceAssetProvider.sol
98:         asset.transferFrom(assetProviderWallet, _buyer, _amount);
```

```
on-ramp/BaseOnRamp.sol
122:        liquidityToken.transferFrom(from, address(this), amount);
125:            liquidityToken.transfer(feeManager.feeCollector(), fee);
131:            liquidityToken.approve(address(USDCBridge), amountExcludingFee);
134:            liquidityToken.transfer(custodianWallet, amountExcludingFee);

off-ramp/provider/AllowanceLiquidityProvider.sol
141:        liquidityToken.transferFrom(liquidityProviderWallet, _redeemer, _liquidityAmount);

off-ramp/provider/CollateralLiquidityProvider.sol
186:        collateralToken.transferFrom(collateralProvider, address(this), collateralAmount);
189:        collateralToken.approve(address(externalCollateralRedemption), collateralAmount);
198:        liquidityToken.transfer(_redeemer, amountToSupply);

off-ramp/RedemptionManager.sol
43:            _params.asset.transferFrom(_params.redeemer, _params.liquidityProvider.recipient(),
↪  _params.assetAmount);
74:        _params.asset.transferFrom(_params.redeemer, _contractAddress, _params.assetAmount);
80:            _params.asset.transfer(_params.liquidityProvider.recipient(), _params.assetAmount);
96:        _params.liquidityProvider.liquidityToken().transfer(_params.redeemer, userSuppliedAmount);
100:
↪  _params.liquidityProvider.liquidityToken().transfer(IFeeManager(_params.feeManager).feeCollector(),
↪  fee);
```

Securitize: Fixed in commit a694dc3.

Cyfrin: Verified.

7.3.7 Unused nonce Field in `ExecutePreApprovedTransaction` struct

Description: The `ExecutePreApprovedTransaction` struct contains a nonce field that is never used or validated:

```
  /**
    * @dev Tx type - EIP712
    */
   struct ExecutePreApprovedTransaction {
       string senderInvestor;
       address destination;
       bytes data;
       uint256 nonce;
   }
```

In `SecuritizeOnRamp::executePreApprovedTransaction`, the `txData.nonce` value passed by the caller is completely ignored. The function uses the internal `noncePerInvestor` mapping instead:

```
function hashTx(ExecutePreApprovedTransaction calldata txData) private view returns (bytes32) {
       bytes32 structHash = keccak256(
           abi.encode(
               TXTYPE_HASH,
               keccak256(bytes(txData.senderInvestor)),
               txData.destination,
               keccak256(txData.data),
               noncePerInvestor[txData.senderInvestor]
           )
       );

       return _hashTypedDataV4(structHash);
   }
```

This creates dead code and a confusing API where callers must include a nonce value in their transaction data that has no effect on execution.

Recommended Mitigation: Remove the unused nonce field from the struct.

Securitize: Acknowledged for now to avoid breaking backend changes.

7.3.8 Redundant Access Control Check in `SecuritizeInternalNavProvider::addRateUpdater, removeRateUpdater`

Description: `SecuritizeInternalNavProvider::addRateUpdater, removeRateUpdater` perform redundant access control checks. These functions have an `onlyRole(DEFAULT_ADMIN_ROLE)` modifier, but then call `grantRole/revokeRole` which internally perform the same check:

```
  function addRateUpdater(address _account) external onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(RATE_UPDATER, _account);
        emit RateUpdaterAdded(_account);
    }
```

Since `RATE_UPDATER` admin role defaults to `DEFAULT_ADMIN_ROLE`, both checks require the same role, resulting in duplicate authorization verification and unnecessary gas consumption.

Recommended Mitigation: Use the internal `_grantRole` and `_revokeRole` functions directly since access control is already enforced by the function modifiers:

Securitize: Fixed in commit 77a9a52.

Cyfrin: Verified.

7.3.9 Refactor identical code from `SecuritizeAmmNavProvider::_pricingFromCurveBuy, _pricingFromCurveSell` into internal function

Description: `SecuritizeAmmNavProvider::_pricingFromCurveBuy, _pricingFromCurveSell` are virtually identical except for the very last line; refactor the identical code into an internal function to avoid unnecessary code duplication:

```
function _computeExecPrice(
    uint256 curvePriceWad,
    uint256 anchorPriceWad
) internal view returns (uint256 execPriceWad) {
    require(anchorPriceWad > 0, "anchor=0");
    require(priceScaleFactor > 0, "scaleFactor = 0");

    uint256 mWad = (curvePriceWad * baseBaseline) / quoteBaseline;
    uint256 baseExecPriceWad = (anchorPriceWad * mWad) / WAD;

    if (baseExecPriceWad >= anchorPriceWad) {
        uint256 diff = baseExecPriceWad - anchorPriceWad;
        execPriceWad = anchorPriceWad + (diff / priceScaleFactor);
    } else {
        uint256 diff = anchorPriceWad - baseExecPriceWad;
        execPriceWad = anchorPriceWad - (diff / priceScaleFactor);
    }
}

function _pricingFromCurveBuy(...) internal view returns (uint256 baseOut, uint256 execPriceWad) {
    execPriceWad = _computeExecPrice(curvePriceWad, anchorPriceWad);
    baseOut = (amountInQuote * WAD) / execPriceWad;
}

function _pricingFromCurveSell(...) internal view returns (uint256 quoteOut, uint256 execPriceWad) {
    execPriceWad = _computeExecPrice(curvePriceWad, anchorPriceWad);
    quoteOut = (amountInBase * execPriceWad) / WAD;
}
```

Securitize: Acknowledged.

7.3.10 `MbpsFeeManager::setFeePercentageMBPS` allows setting fee greater than max

Description: `MbpsFeeManager::setFeePercentageMBPS` allows setting fee greater than max; recommend enforcing a maximum fee percentage eg:

```
contract MbpsFeeManager is IFeeManager, BaseContract {
    uint256 public constant FEE_DENOMINATOR = 100_000;
+   uint256 public constant MAX_FEE = 10_000; // 10%

    function setFeePercentageMBPS(uint256 _fee) external onlyRole(DEFAULT_ADMIN_ROLE) {
+       require(_fee <= MAX_FEE);
        uint256 oldFee = feePercentageMBPS;
        feePercentageMBPS = _fee;
        emit FeeUpdated(oldFee, _fee);
    }
```

Securitize: Acknowledged.

7.4 Gas Optimization

7.4.1 Better storage packing by changing declaration order

Description: Better storage packing by changing declaration order:

- `2-nav-provider/contracts/nav/SecuritizeAmmNavProvider.sol` - declare `lastMarketStatus` immediately after `asset`:

```
IERC20Metadata public asset;
uint8 public lastMarketStatus;
```

Securitize: Fixed in commit 41815fa.

Cyfrin: Verified.

7.4.2 Emit events first to refactor away local variables storing previous values

Description: When values are being changed, emit events first to refactor away local variables storing previous values. For example in `SecuritizeAmmNavProvider::setPriceScaleFactor`:

```
    function setPriceScaleFactor(uint256 newScaleFactor) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(newScaleFactor > 0, "scaleFactor = 0");

-       uint256 oldScaleFactor = priceScaleFactor;
+       emit PriceScaleFactorUpdated(priceScaleFactor, newScaleFactor);
        priceScaleFactor = newScaleFactor;

-       emit PriceScaleFactorUpdated(oldScaleFactor, newScaleFactor);
    }
```

Similar optimizations can be made in:

- `SecuritizeInternalNavProvider::setRate`
- `MbpsFeeManager::setFeePercentageMBPS, setFeeCollector`
- `AllowanceLiquidityProvider::setAllowanceProviderWallet`
- `CollateralLiquidityProvider::setExternalCollateralRedemption, setCollateralProvider`
- `BaseOffRamp::updateLiquidityProvider`
- `PublicStockOffRamp::updateNavProvider`
- `SecuritizeOffRamp::updateNavProvider`
- `AllowanceAssetProvider::setAllowanceProviderWallet`
- `SecuritizeOnRamp::updateNavProvider`
- `BaseOnRamp::updateAssetProvider, updateMinSubscriptionAmount`
- `PublicStockOnRamp::updateNavProvider`

Securitize: Fixed in commits 41538fa, 7f500c1.

Cyfrin: Verified.

7.4.3 Cache storage to prevent identical storage reads

Description: Reading from storage is expensive, cache storage to prevent identical storage reads:

- `SecuritizeAmmNavProvider::_pricingFromCurveBuy, _pricingFromCurveSell` - cache `priceScaleFactor`

- `SecuritizeAmmNavProvider::_checkAndResetBaseline` - cache `lastAnchorPriceWad, lastMarketStatus` prior to first `if` statement
- `AllowanceLiquidityProvider::_availableLiquidity` - cache `liquidityToken, liquidityProviderWallet`
- `BaseOffRamp::_redeem` - cache `asset`
- `BaseOnRamp::_executeLiquidityTransfer` - cache `liquidityToken, feeManager, bridgeChainId, USDCBridge`

Securitize: Fixed in commits e631361, ea883b9, d32d6a0.

Cyfrin: Verified.

7.4.4 When emitting events don't read known values from storage

Description: When emitting events don't read known values from storage; for example in `SecuritizeAmmNavProvider::_resetBaseline`:

```
    function _resetBaseline(uint256 newBase, uint256 newQuote) internal {
        require(newBase > 0, "newBase=0");
        require(newQuote > 0, "newQuote=0");

        baseReserves = newBase;
        quoteReserves = newQuote;

        baseBaseline = newBase;
        quoteBaseline = newQuote;

        k = newBase * newQuote;

-       emit BaselineReset(baseBaseline, quoteBaseline);
+       emit BaselineReset(newBase, newQuote);
    }
```

Similar optimizations can be made in:

- `AllowanceLiquidityProvider::setAllowanceProviderWallet`
- `CollateralLiquidityProvider::setExternalCollateralRedemption, setCollateralProvider`
- `BaseOffRamp::updateLiquidityProvider` at this line `uint256 _liquidityDecimals = IERC20Metadata(address(liquidityProvider.liquidityToken())).decimals();` - use `_liquidityProvider` instead of `liquidityProvider`
- `BaseOffRamp::toggleTwoStepTransfer`
- `PublicStockOffRamp::updateNavProvider`
- `SecuritizeOffRamp::updateNavProvider`
- `AllowanceAssetProvider::setAllowanceProviderWallet`
- `BaseOnRamp::updateMinSubscriptionAmount, toggleInvestorSubscription, toggleTwoStepTransfer`

Securitize: Fixed in commits fe9f910, 5d39cc1.

Cyfrin: Verified.

7.4.5 In Solidity don't initialize to default values

Description: In Solidity don't initialize to default values:

```
SecuritizeAmmNavProvider.sol
380:        bool shouldReset = false;
```

```
off-ramp/BaseOffRamp.sol
124:         for (uint256 i = 0; i < _countries.length; i++) {
```

Securitize: Fixed in commits 7594671, 6833173.

Cyfrin: Verified.

7.4.6 Cache required storage slots in top-level functions then pass cached values to child functions

Description: A common pattern of inefficiency is when parent functions read certain storage slots, then call child functions (which can themselves call other child functions), and the child functions re-read the same storage slots even though it isn't possible for their values to have changed.

In such cases it is much more efficient for the parent functions to cache the required storage slots once, then pass them as input to child functions:

- `AllowanceLiquidityProvider::supplyTo` - cache `liquidityToken, liquidityProviderWallet`, pass as inputs to `_availableLiquidity`
- `CollateralLiquidityProvider::supplyTo` cache `externalCollateralRedemption, collateralToken, collateralProvider`, pass as inputs to `_availableLiquidity, _liquidityTokenToExternalCollateralToken`
- `SecuritizeOnRamp::subscribe` cache `liquidityToken` pass as inputs to `calculateDsTokenAmount, _executeLiquidityTransfer`
- `SecuritizeOffRamp::redeem` cache `liquidityProvider` pass as input to `_redeem` then use to emit `RedemptionCompleted` event

Securitize: Acknowledged for now to avoid a big internal functions refactor.

7.4.7 Use named return variables where this can refactor away local variables

Description: Use named return variables where this can refactor away local variables:

- `CountryValidator::getCountry`
- `PublicStockOnRamp::calculateDsTokenAmount` - read result of `navProvider.quoteBuyBase` directly into `rate` and remove local variable `execPrice`

Securitize: Fixed in commit f6055a0.

Cyfrin: Verified.

7.4.8 Cache computation where it is performed multiple times

Description: Cache computation where it is performed multiple times:

- `CountryValidator::validateCountryCode` - cache `bytes(_country).length`

Securitize: Acknowledged.

7.4.9 Cache decimals of underlying asset at initialization in `SecuritizeAmmNavProvider`

Description: `SecuritizeAmmNavProvider` has no function to change the value of `asset` after initialization. Since ERC20 tokens typically never changed their decimals, the asset decimals can be cached at initialization to save external calls in `quoteBuyBase, quoteSellBase, executeBuyBase, executeSellBase`.

Since the decimals are only used to calculate `scaleDown`, can just cache this eg:

```
contract SecuritizeAmmNavProvider {
    /* snip : existing storage layout */
+   uint256 public SCALE_DOWN;
```

```
    function initialize(uint256 _baseReserves, uint256 _quoteReserves, address _asset) public onlyProxy
    ↪  initializer {
        /* snip : existing code */
        asset = IERC20Metadata(_asset);
+       uint8 d = asset.decimals();
+       require(d <= 18, "decimals > 18");
+       SCALE_DOWN = 10 ** (18 - d);
    }
```

Then the following code can be removed from `quoteBuyBase`, `quoteSellBase`, `executeBuyBase`, `executeSellBase`:

```
-       uint8 d = asset.decimals();
-       require(d <= 18, "decimals > 18");
-       uint256 scaleDown = 10 ** (18 - d);
```

And just use `SCALE_DOWN` where it is required; this is more efficient and also reduces duplicated code, increasing code quality.

If there is a need to change the asset or the `SCALE_DOWN` factor, just add a new privileged function which sets a new `asset` and recalculates `SCALE_DOWN` using the new asset's decimals.

Securitize: Acknowledged.

7.4.10 Use of modifier `nonZeroNavRate` in `SecuritizeOnRamp` and `SecuritizeOffRamp` results in duplicate external call with identical result

Description: `SecuritizeOnRamp` and `SecuritizeOffRamp` both have modifier `nonZeroNavRate` which makes an external call to enforce a positive rate:

```
modifier nonZeroNavRate() {
    if (navProvider.rate() <= 0) {
        revert NonZeroNavRateError();
    }
    _;
}
```

The problem is that the functions which use this modifier (such as `swap`) subsequently call a child function (such as `calculateDsTokenAmount`) which ends up making the same `navProvider.rate` external call again.

Impact: The same external call is made twice in each affected transaction, even though the answer between calls can't change.

Recommended Mitigation: Convert the modifier into an internal function which does the revert check and returns the rate, the pass the cached rate to any child functions that require it.

Also `ISecuritizeNavProvider::rate` returns `uint256` so the `< 0` comparison is non-sensical, just enforce that the rate is `!= 0`.

Securitize: Acknowledged.

7.4.11 Fee calculation occurs twice

Description: `SecuritizeOnRamp::subscribe, swap` calls `calculateDsTokenAmount` which performs an external call to calculate the fee:

```
function calculateDsTokenAmount(uint256 _liquidityAmount) public view returns (uint256 dsTokenAmount,
↪  uint256 rate, uint256 fee) {
    fee = feeManager.getFee(_liquidityAmount);
    uint256 liquidityAmountExcludingFee = _liquidityAmount - fee;
```

Subsequently `BaseOnRamp::_executeLiquidityTransfer` is called which does it again:

```
uint256 fee = feeManager.getFee(amount);
if (fee > 0) {
    liquidityToken.transfer(feeManager.feeCollector(), fee);
}
```

Impact: Duplicate storage reads of `feeManager` and duplicate external calls.

Recommended Mitigation: Calculate the fee once in top-level functions then pass it to child functions as an input parameter. In the `SecuritizeOnRamp::subscribe` example `fee` is already returned by `calculateDsTokenAmount` so it could be passed as input to `_executeLiquidityTransfer`, though this still results in multiple storage reads of `feeManager` which is not ideal - ideally `feeManager` would be read once from storage and passed to any child functions that require it as well.

Securitize: Acknowledged.

7.4.12 Refactor `PublicStockOnRamp::initializedNavProvider`, `BaseOffRamp::nonZeroLiquidityProvider` into internal functions to prevent identical storage reads

Description: `PublicStockOnRamp` has a modifier `initializedNavProvider` that reads the `navProvider` storage slot, but then later on in the functions which have this modifier (`swap, calculateDsTokenAmount`), the `navProvider` storage slot is read again even though its value has not changed.

Impact: Storage reads are expensive; we want to avoid reading the same storage slot multiple times when the value hasn't changed.

Recommended Mitigation: Refactor the modifier `initializedNavProvider` into an `internal` function:

```
    function _getNavProviderStrict() internal returns(address navProviderAddr) {
        navProviderAddr = address(navProvider);
        if (navProviderAddr == address(0)) revert NavProviderNotSetError();
    }
```

Then call this internal function in `PublicStockOnRamp::swap, calculateDsTokenAmount` like this:

```
ISecuritizeAmmNavProvider navProviderCache = ISecuritizeAmmNavProvider(_getNavProviderStrict());
```

Then if it didn't revert, use `navProviderCache` wherever the nav provider is required.

Do a similar optimization for `BaseOffRamp::nonZeroLiquidityProvider` used in `_redeem`.

Securitize: Acknowledged.

7.4.13 Refactor away unnecessary local variables in `SecuritizeAmmNavProvider::_curveBuy, _curveSell`

Description: In `SecuritizeAmmNavProvider::_curveBuy, _curveSell` the local variables `X, Y, kLocal` are only read once so there is no need to use them to cache storage, just read the storage slots directly when required:

```
function _curveBuy(uint256 amountInQuote) internal view initialized returns (uint256 curvePriceWad,
↪   uint256 newBase, uint256 newQuote) {
    require(amountInQuote > 0, "amountInQuote=0");

    newQuote = quoteReserves + amountInQuote;
    newBase = k / newQuote;

    uint256 deltaBase = baseReserves - newBase;
    require(deltaBase > 0, "deltaBase=0");

    curvePriceWad = (amountInQuote * WAD) / deltaBase;
}
```

```
function _curveSell(uint256 amountInBase) internal view initialized returns (uint256 curvePriceWad,
↪   uint256 newBase, uint256 newQuote) {
    require(amountInBase > 0, "amountInBase=0");

    newBase = baseReserves + amountInBase;
    newQuote = k / newBase;

    uint256 deltaQuote = quoteReserves - newQuote;
    require(deltaQuote > 0, "deltaQuote=0");

    curvePriceWad = (deltaQuote * WAD) / amountInBase;
}
```

Securitize: Fixed in commit 5ca8229.

Cyfrin: Verified.