



Securitize On-Off Ramp and Bridge Audit Report

Prepared by [Cyfrin](#)

Version 2.1

Lead Auditors

Hans

ChainDefenders ([1337web3](#) & [PeterSRWeb3](#))

July 23, 2025

Contents

1 About Cyfrin	2
2 Disclaimer	2
3 Risk Classification	2
4 Protocol Summary	2
4.1 Key Features	2
4.1.1 On-Ramp Operations	2
4.1.2 Off-Ramp Operations	2
4.1.3 Bridge Operations	3
4.2 Value Flows	3
4.2.1 On-Ramp Value Flow	3
4.2.2 Off-Ramp Value Flow	3
4.2.3 Fee Structure	3
4.2.4 Main Actors and Roles	3
4.3 Security Considerations	3
4.3.1 NAV Rate Calculation	3
4.3.2 Privileged Roles	3
4.3.3 External Dependencies	4
4.3.4 UUPS Upgradeability	4
5 Audit Scope	4
6 Executive Summary	4
6.1 Summary of Findings	4
6.2 Assessment of Test Coverage	5
7 Findings	7
7.1 Critical Risk	7
7.1.1 Missing nonce validation in signature verification allows transaction replay attacks	7
7.2 Medium Risk	8
7.2.1 Incorrect usage of minOutputAmount in executeTwoStepRedemption can cause unnecessary reverts	8
7.2.2 Incorrect address handling for account abstraction wallets in SecuritizeBridge::bridgeDSTokens	8
7.2.3 Pause modifier in bridge receiver functions causes receiver failures for in-flight messages	9
7.2.4 Calculation of available liquidity in CollateralLiquidityProvider::availableLiquidity assumes 1:1 ratio between collateral asset and liquidity tokens	9
7.3 Low Risk	11
7.3.1 Bridge gas limit parameter lacks lower bound validation leading to potential failed deliveries	11
7.3.2 Missing validation check for liquidityToken	11
7.3.3 liquidityProviderWallet is not set during initialization	11
7.3.4 Missing storage gap on upgradeable base contracts	12
7.3.5 Single-step ownership transfer pattern is not recommended	12
7.3.6 Use of msg.sender instead of _msgSender() prevents meta-transaction support	12
7.3.7 whChainId should not be stored as immutable constant	13
7.3.8 Refund address in bridgeDSTokens function should be configurable	13
7.3.9 Usage of unofficial wormhole-solidity-sdk npm package poses security and maintenance risks	13
7.4 Informational	15
7.4.1 Misleading comments and documentation inconsistencies in on-ramp contracts	15
7.4.2 Unnecessary override keywords on interface implementation functions	15
7.4.3 The swap function emits incorrect event type for existing investor purchases	16
7.4.4 Upgradeable contracts missing _disableInitializers() in constructors	16
7.4.5 Check for country code is not sufficient	16

7.4.6	Confusing variable naming in fee manager contracts	17
7.4.7	Unused parameter in address validation modifier SecuritizeOffRamp::addressNonZero	17
7.4.8	Unnecessary complexity in calculateLiquidityTokenAmountWithoutFee	18
7.4.9	Unused library and struct definitions increase deployment costs and reduce code clarity	18
7.4.10	Missing slippage protection in external collateral redemption call	18

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

The Securitize protocol is a comprehensive blockchain infrastructure designed to facilitate the tokenization and trading of real-world assets (RWA). The review scope consists of two main components:

- On-Off Ramp System: Enables investors to purchase Digital Securities (DS) tokens using stablecoins (primarily USDC) and allows investors to redeem their DS tokens back to stablecoins
- Bridge System: Facilitates cross-chain transfer of DS tokens and USDC using Wormhole protocol

4.1 Key Features

4.1.1 On-Ramp Operations

1. Subscribe Method: Permissioned registration and token purchase for new investors using EIP-712 signed transactions
2. Swap Method: Direct token purchases for registered investors
3. Two-Step Transfer: Optional mode for enhanced security where tokens first go to the contract before final transfer
4. Bridge Integration: Optional cross-chain USDC transfers via Circle's CCTP

4.1.2 Off-Ramp Operations

1. Redemption: Convert DS tokens back to stablecoins at current NAV rates
2. Liquidity Provision: Multiple provider strategies (Allowance-based, Collateral-based)
3. Country Restrictions: Geo-blocking functionality for compliance
4. Two-Step Redemption: Enhanced security mode for large transactions

4.1.3 Bridge Operations

1. DS Token Bridging: Cross-chain token transfers with burn-and-mint mechanism
2. USDC Bridging: Cross-chain stablecoin transfers via Circle's CCTP
3. Investor Data Migration: Synchronized KYC/AML attributes across chains

4.2 Value Flows

4.2.1 On-Ramp Value Flow

```
Investor USDC → [Fee Deduction] → [Bridge Transfer (optional)] → Custodian Wallet  
DS Tokens ← [Asset Provider] ← [NAV Rate Calculation] ← On-Ramp Contract
```

4.2.2 Off-Ramp Value Flow

```
Investor DS Tokens → [Burn/Transfer] → Off-Ramp Contract  
Stablecoins ← [Liquidity Provider] ← [NAV Rate Calculation] ← Off-Ramp Contract
```

4.2.3 Fee Structure

- MBPS: 1000 MBPS = 1%
- On-Ramp Fee: Deducted from liquidity amount before NAV calculation
- Off-Ramp Fee: Deducted from final liquidity amount after NAV calculation
- Bridge Fee: Native token payment for cross-chain gas costs

4.2.4 Main Actors and Roles

1. Investors: End users who purchase/redeem DS tokens. They are required to be registered in the DS Registry.
2. Contract Owner: Administrative control over system parameters
3. Custodian Wallet: Receives stablecoins from on-ramp operations
4. Asset Provider: Supplies DS tokens (via allowance or minting)
5. Liquidity Provider: Supplies stablecoins for redemptions
6. Fee Collector: Receives fee payments
7. EIP-712 Signers: Authorized signers for permissioned operations (EXCHANGE or ISSUER roles)
8. NAV Provider: External oracle providing asset pricing
9. Bridge Relayer: Wormhole relayer for cross-chain operations

4.3 Security Considerations

4.3.1 NAV Rate Calculation

In the current implementation, the NAV rate of asset token is assumed to reflect the current price of liquidity token (stable coin). Based on this assumption, the conversion from DS token to stablecoin does not include the liquidity token price.

4.3.2 Privileged Roles

1. System Configuration:
 - Update NAV providers (price manipulation risk)
 - Update fee managers (economic attack vector)
 - Configure bridge addresses (cross-chain attack vector)
2. Emergency Controls:
 - Pause/unpause operations
 - Update country restrictions
 - Modify minimum subscription amounts
3. Upgrade Authority:
 - UUPS upgradeable pattern
 - Owner-controlled upgrade authorization
 - Potential for logic manipulation

4.3.3 External Dependencies

1. Securitize DS Token: Core asset contract
2. DS Registry Service: Investor validation
3. DS Trust Service: Role-based authorization
4. NAV Provider: Asset pricing oracle
5. Wormhole Protocol: Cross-chain messaging
6. Circle CCTP: USDC cross-chain transfers

4.3.4 UUPS Upgradeability

- Advantages: Gas-efficient, owner-controlled upgrades
- Risks: Centralized upgrade authority, potential for malicious upgrades
- Mitigation: Multi-signature wallets, timelock contracts recommended

5 Audit Scope

The following contracts were reviewed:

```
\bc-on-off-contracts\contracts\common\BaseContract.sol
\bc-on-off-contracts\contracts\common\Errors.sol
\bc-on-off-contracts\contracts\fee\MbpsFeeManager.sol
\bc-on-off-contracts\contracts\off-ramp\CountryValidator.sol
\bc-on-off-contracts\contracts\off-ramp\provider\AllowanceLiquidityProvider.sol
\bc-on-off-contracts\contracts\off-ramp\provider\CollateralLiquidityProvider.sol
\bc-on-off-contracts\contracts\off-ramp\RedemptionManager.sol
\bc-on-off-contracts\contracts\off-ramp\RedemptionValidator.sol
\bc-on-off-contracts\contracts\off-ramp\SecuritizeOffRamp.sol
\bc-on-off-contracts\contracts\off-ramp\TokenCalculator.sol
\bc-on-off-contracts\contracts\on-ramp\ISecuritizeOnRamp.sol
\bc-on-off-contracts\contracts\on-ramp\provider\AllowanceAssetProvider.sol
\bc-on-off-contracts\contracts\on-ramp\provider\MintingAssetProvider.sol
\bc-on-off-contracts\contracts\on-ramp\SecuritizeOnRamp.sol
\bridge-contracts\contracts\bridge\SecuritizeBridge.sol
```

```
\bridge-contracts\contracts\bridge\USDCBridge.sol  
\bridge-contracts\contracts\utils\BaseContract.sol  
\bridge-contracts\contracts\utils\BaseRBACContract.sol  
\bridge-contracts\contracts\utils\Proxies.sol  
\bridge-contracts\contracts\wormhole-upgradeable\WormholeCCTPUpgradeable.sol
```

6 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the [Securitize On-Off Ramp and Bridge](#) smart contracts provided by [Securitize](#). In this period, a total of 24 issues were found.

6.1 Summary of Findings

A critical vulnerability was identified in the signature verification mechanism of the SecuritizeOnRamp contract that enables transaction replay attacks. The `executePreApprovedTransaction` function fails to validate that the provided nonce matches the expected nonce for the investor, allowing attackers to replay previously valid EIP-712 signed transactions and execute duplicate subscription operations, potentially leading to unintended token swaps and accounting failures.

The medium severity findings include slippage protection inconsistencies in the redemption process where `minOutputAmount` parameters don't account for fee deductions, causing unnecessary transaction reverts even when liquidity providers meet minimum requirements. Additionally, bridge receiver functions protected by pause modifiers create operational risks where funds associated with in-flight cross-chain messages become permanently stuck when contracts are paused, as Wormhole's architecture doesn't provide automatic retry mechanisms for receiver failures.

Low severity issues encompass initialization and validation problems across multiple contracts, including missing storage gaps in upgradeable base contracts that could lead to storage collisions during upgrades, uninitialized `liquidityProviderWallet` variables causing failed redemptions, insufficient gas limit validation in bridge parameters that could cause cross-chain delivery failures, and the use of single-step ownership transfer patterns that risk permanent administrative lockouts.

Informational findings include extensive documentation inconsistencies and misleading comments that misrepresent actual functionality, unnecessary code complexity in calculation functions, confusing variable naming conventions in fee managers, unused parameters in validation modifiers, and missing support for meta-transactions due to direct `msg.sender` usage instead of `_msgSender()`.

6.2 Assessment of Test Coverage

The audit findings reveal gaps in test coverage, particularly around edge cases involving signature replay attacks and cross-chain message handling during paused states. The detailed proof-of-concept provided for the critical nonce validation vulnerability demonstrates attack vectors that were not covered in the existing test suite. The absence of comprehensive testing suggests that the current test coverage may be insufficient to catch complex interaction vulnerabilities.

Summary

Project Name	Securitize On-Off Ramp and Bridge
Repository	bc-on-off-ramp-sc
Commit	a944bb11b106...
Fix Commit	4a426e689586...
Repository 2	bc-securitize-bridge-sc
Commit	60b6ef00e8a8...
Fix Commit	1da35cde31a5...
Audit Timeline	July 7 - July 14th
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	0
Medium Risk	4
Low Risk	9
Informational	10
Gas Optimizations	0
Total Issues	24

Summary of Findings

[C-1] Missing nonce validation in signature verification allows transaction replay attacks	Resolved
[M-1] Incorrect usage of minOutputAmount in executeTwoStepRedemption can cause unnecessary reverts	Resolved
[M-2] Incorrect address handling for account abstraction wallets in SecuritizeBridge::bridgeDSTokens	Acknowledged
[M-3] Pause modifier in bridge receiver functions causes receiver failures for in-flight messages	Resolved
[M-4] Calculation of available liquidity in CollateralLiquidityProvider::availableLiquidity assumes 1:1 ratio between collateral asset and liquidity tokens	Resolved
[L-1] Bridge gas limit parameter lacks lower bound validation leading to potential failed deliveries	Acknowledged
[L-2] Missing validation check for liquidityToken	Acknowledged
[L-3] liquidityProviderWallet is not set during initialization	Resolved
[L-4] Missing storage gap on upgradeable base contracts	Resolved

[L-5] Single-step ownership transfer pattern is not recommended	Acknowledged
[L-6] Use of <code>msg.sender</code> instead of <code>_msgSender()</code> prevents meta-transaction support	Resolved
[L-7] <code>whChainId</code> should not be stored as immutable constant	Closed
[L-8] Refund address in <code>bridgeDSTokens</code> function should be configurable	Acknowledged
[L-9] Usage of unofficial wormhole-solidity-sdk npm package poses security and maintenance risks	Resolved
[I-1] Misleading comments and documentation inconsistencies in on-ramp contracts	Resolved
[I-2] Unnecessary override keywords on interface implementation functions	Resolved
[I-3] The <code>swap</code> function emits incorrect event type for existing investor purchases	Resolved
[I-4] Upgradeable contracts missing <code>disableInitializers()</code> in constructors	Resolved
[I-5] Check for country code is not sufficient	Acknowledged
[I-6] Confusing variable naming in fee manager contracts	Resolved
[I-7] Unused parameter in address validation modifier <code>SecuritizeOffRamp::addressNonZero</code>	Resolved
[I-8] Unnecessary complexity in <code>calculateLiquidityTokenAmountWithoutFee</code>	Resolved
[I-9] Unused library and struct definitions increase deployment costs and reduce code clarity	Resolved
[I-10] Missing slippage protection in external collateral redemption call	Acknowledged

7 Findings

7.1 Critical Risk

7.1.1 Missing nonce validation in signature verification allows transaction replay attacks

Description: The SecuritizeOnRamp::executePreApprovedTransaction function fails to validate that the nonce provided in the transaction data matches the expected nonce for the investor before executing the transaction. While the function verifies the EIP-712 signature includes the nonce as part of the signed message, it does not check if txData.nonce equals the current noncePerInvestor[txData.senderInvestor] value. The function only increments the stored nonce after signature verification, allowing old valid signatures to be replayed with their original nonce values.

The current implementation at lines 184-199 in SecuritizeOnRamp.sol:

```
function executePreApprovedTransaction(
    bytes memory signature,
    ExecutePreApprovedTransaction calldata txData
) public override whenNotPaused {
    bytes32 digest = hashTx(txData);
    address signer = ECDSA.recover(digest, signature);

    // Check recovered address role
    IDSTrustService trustService = IDSTrustService(dsToken.getDSService(dsToken.TRUST_SERVICE()));
    uint256 signerRole = trustService.getRole(signer);
    if (signerRole != trustService.EXCHANGE() && signerRole != trustService.ISSUER()) {
        revert InvalidEIP712SignatureError();
    }
    noncePerInvestor[txData.senderInvestor] = noncePerInvestor[txData.senderInvestor] + 1;
    Address.functionCall(txData.destination, txData.data);
}
```

This vulnerability allows an attacker to replay any previously valid signed transaction, potentially executing multiple subscriptions or other operations with the same signature.

Impact: Attackers can replay previously valid EIP-712 signed transactions to execute duplicate subscription operations, leading to unintended token swaps and accounting failure.

Proof Of Concept: Add the PoC below to on-ramp.test.ts.

```
it('Should allow replay attack - nonce not validated properly', async function () {
    // Setup initial balances and approvals for multiple transactions
    await usdcMock.mint(unknownWallet, 2e6); // Mint 2M USDC for two transactions
    await dsTokenMock.issueTokens(assetProviderWallet, 2e6); // Issue 2e6 DS tokens

    const liquidityFromInvestor = usdcMock.connect(unknownWallet) as Contract;
    await liquidityFromInvestor.approve(onRamp, 2e6); // Approve 2 USDC

    const dsTokenFromAssetProviderWallet = dsTokenMock.connect(assetProviderWallet) as Contract;
    await dsTokenFromAssetProviderWallet.approve(assetProvider, 2e6); // Approve 2e6 DS tokens

    const calculatedDSTokenAmount = await onRamp.calculateDsTokenAmount(1e6);

    // Create first transaction with nonce 0
    const subscribeParams = [
        '1',
        await unknownWallet.getAddress(),
        'US',
        [],
        [],
        [],
        980000,
        1e6,
```

```

        blockNumber + 10,
        HASH,
    ];
    const txData = await buildTypedData(onRamp, subscribeParams);
    const signature = await eip712OnRamp(eip712Signer, await onRamp.getAddress(), txData);

    // Verify initial nonce is 0
    expect(await onRamp.nonceByInvestor('1')).to.equal(0);

    // Execute first transaction successfully
    await expect(onRamp.executePreApprovedTransaction(signature, txData))
        .emit(onRamp, 'Swap')
        .withArgs(onRamp, calculatedDSTokenAmount, 1e6, unknownWallet);

    // Verify nonce is now 1 after first transaction
    expect(await onRamp.nonceByInvestor('1')).to.equal(1);

    // Verify first transaction effects
    expect(await dsTokenMock.balanceOf(unknownWallet)).to.equal(calculatedDSTokenAmount);
    expect(await usdcMock.balanceOf(unknownWallet)).to.equal(1e6); // 1e6 remaining

    // VULNERABILITY: Replay the same transaction with the same signature and nonce 0
    // This should fail but doesn't because nonce validation is missing
    await expect(onRamp.executePreApprovedTransaction(signature, txData))
        .emit(onRamp, 'Swap')
        .withArgs(onRamp, calculatedDSTokenAmount, 1e6, unknownWallet);

    // Verify the replay attack succeeded - investor got double the tokens
    expect(await dsTokenMock.balanceOf(unknownWallet)).to.equal(calculatedDSTokenAmount * 2n);
    expect(await usdcMock.balanceOf(unknownWallet)).to.equal(0); // All USDC spent

    // Verify nonce was incremented again (now 2) even though we replayed nonce 0
    expect(await onRamp.nonceByInvestor('1')).to.equal(2);
};


```

Recommended Mitigation: Add nonce validation before executing the transaction to ensure the provided nonce matches the expected nonce for the investor:

```

function executePreApprovedTransaction(
    bytes memory signature,
    ExecutePreApprovedTransaction calldata txData
) public override whenNotPaused {
+   // Validate nonce matches expected value
+   if (txData.nonce != noncePerInvestor[txData.senderInvestor]) {
+       revert InvalidEIP712SignatureError();
+   }
+
    bytes32 digest = hashTx(txData);
    address signer = ECDSA.recover(digest, signature);

    // Check recovered address role
    IDSTrustService trustService = IDSTrustService(dsToken.getDSService(dsToken.TRUST_SERVICE()));
    uint256 signerRole = trustService.getRole(signer);
    if (signerRole != trustService.EXCHANGE() && signerRole != trustService.ISSUER()) {
        revert InvalidEIP712SignatureError();
    }
    noncePerInvestor[txData.senderInvestor] = noncePerInvestor[txData.senderInvestor] + 1;
    Address.functionCall(txData.destination, txData.data);
}

```

Securitize: Fixed in commit [65179b](#).

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Incorrect usage of minOutputAmount in executeTwoStepRedemption can cause unnecessary reverts

Description: In the `RedemptionManager::executeTwoStepRedemption` function, the following call is made:

```
params.liquidityProvider.supplyTo(contractAddress, params.liquidityTokenAmount, params.minOutputAmount);
```

Here, `params.minOutputAmount` is used as the minimum expected return from the liquidity provider. However, this value does not account for any fee deductions that are applied later in the function.

Immediately after the `supplyTo` call, the contract performs a slippage protection check:

```
uint256 offRampBalance = params.liquidityProvider.liquidityToken().balanceOf(contractAddress);
uint256 fee = _getFee(params.feeManager, offRampBalance);

if (offRampBalance - fee < params.minOutputAmount) {
    revert Errors.SlippageControlError();
}
```

If the liquidity provider returns exactly `minOutputAmount`, then the deduction of the fee from that amount will cause `offRampBalance - fee` to fall below `minOutputAmount`, resulting in a slippage error—even though the liquidity provider met the minimum requirement.

The issue is not with the slippage check itself, which is correctly accounting for the fee. The problem is that the `minOutputAmount` passed to `supplyTo` should also include the fee, to ensure consistency with the later slippage check.

Impact: Unexpected transaction reverts may occur due to slippage errors, even when the liquidity provider meets the `minOutputAmount` requirement.

Recommended Mitigation: Update the call to `supplyTo` to include the expected fee in the `minOutputAmount` parameter. For example:

```
uint256 expectedFee = _getFee(params.feeManager, params.minOutputAmount);
params.liquidityProvider.supplyTo(contractAddress, params.liquidityTokenAmount, params.minOutputAmount
    + expectedFee);
```

This ensures that the post-fee amount meets the expected minimum and aligns with the logic in the slippage protection check.

Securitize: Fixed in commit [54243f](#).

Cyfrin: Verified.

7.2.2 Incorrect address handling for account abstraction wallets in SecuritizeBridge::bridgeDSTokens

Description: The `SecuritizeBridge.bridgeDSTokens` function assumes that a user's wallet address (`msg.sender`) on the source chain is identical to their desired recipient address on the destination chain. This assumption is incorrect for Account Abstraction (AA) wallets.

Account Abstraction wallets are smart contract-based wallets (e.g., Safe, Argent, ERC-4337 wallets) that, unlike Externally Owned Accounts (EOAs), can have different addresses across different chains for the same logical user.

This occurs because:

- AA wallets are deployed using factory contracts with chain-specific parameters
- Deployment salts, nonces, or factory addresses may differ between chains
- The same user's wallet logic results in different contract addresses on different chains

This creates a security risk where bridged DSTokens may be minted to an uncontrolled address on the destination chain, potentially resulting in permanent loss of funds.

Impact: Tokens can be minted to an address that the user cannot control on the destination chain, and investor registration can be tied to the wrong addresses, breaking KYC/AML assumptions.

Proof of Concept:

1. User Alice uses a Safe wallet with address 0xAaa...111 on Ethereum
2. Alice calls `bridgeDSTokens` to bridge 1000 DSTokens to Polygon
3. The contract encodes `msg.sender` (0xAaa...111) as the destination address
4. On Polygon, Alice's Safe wallet has address 0xBbb...222 (different factory deployment)
5. DSTokens are minted to 0xAaa...111 on Polygon, which Alice cannot access
6. Alice's funds are permanently lost

Recommended Mitigation: Add an explicit `destinationWallet` parameter to allow users to specify their destination address:

```
function bridgeDSTokens(
    uint16 targetChain,
    uint256 value,
    address destinationWallet
) external override payable whenNotPaused {
    require(destinationWallet != address(0), "Invalid destination wallet");

    // ... existing validation code ...

    wormholeRelayer.sendPayloadToEvm{value: msg.value} (
        targetChain,
        targetAddress,
        abi.encode(
            investorDetail.investorId,
            value,
            destinationWallet, // ← User-specified destination
            investorDetail.country,
            investorDetail.attributeValues,
            investorDetail.attributeExpirations
        ),
        0,
        gasLimit,
        whChainId,
        msg.sender
    );
}
```

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.2.3 Pause modifier in bridge receiver functions causes receiver failures for in-flight messages

Description: The `USDCBridge::receivePayloadAndUSDC` and `SecuritizeBridge::receiveWormholeMessages` functions are protected by the `whenNotPaused` modifier, which causes these functions to revert when the respective bridge contracts are paused. According to the [Wormhole documentation](#), when receiver functions revert, the message status becomes "Receiver Failure" and there is no automatic retry mechanism available. The only way to recover from receiver failures is to restart the entire process from the source chain.

```
// USDCBridge.sol
function receivePayloadAndUSDC(
    bytes memory payload,
    uint256 amountUSDCReceived,
    bytes32 sourceAddress,
    uint16 sourceChain,
```

```

    bytes32 deliveryHash
) internal override onlyWormholeRelayer whenNotPaused {
    // Function will revert if contract is paused
    // ...
}

// SecuritizeBridge.sol
function receiveWormholeMessages(
    bytes memory payload,
    bytes[] memory additionalVaas,
    bytes32 sourceBridge,
    uint16 sourceChain,
    bytes32 deliveryHash
) public override payable whenNotPaused {
    // Function will revert if contract is paused
    // ...
}

```

This creates an operational issue where funds associated with in-flight messages become stuck without any built-in recovery mechanism provided by the bridge contracts.

The problematic scenario:

1. User initiates a cross-chain transfer from Chain A to Chain B
2. Bridge contract on Chain B gets paused due to an emergency or maintenance
3. Wormhole relayer attempts to deliver the message to Chain B
4. The receiver function reverts due to the `whenNotPaused` modifier
5. Message status becomes "Receiver Failure" permanently
6. Funds are stuck with no automatic recovery mechanism

Impact: Funds associated with in-flight cross-chain messages become stuck when bridge contracts are paused, requiring manual intervention to recover assets.

Recommended Mitigation: Remove the `whenNotPaused` modifier from receiver functions to prevent receiver failures. Furthermore, consider tracking received messages with the receive process success flag and allow the admin to retry the failed messages.

Securitize: Partially fixed in commit [97e37b](#), `whenNotPaused` has been removed for the `USDCBridge.receive` function.

Cyfrin: Verified.

7.2.4 Calculation of available liquidity in `CollateralLiquidityProvider::availableLiquidity` **assumes 1:1 ratio between collateral asset and liquidity tokens**

Description: The `CollateralLiquidityProvider::availableLiquidity` function incorrectly returns the balance of the collateral asset held by the collateral provider, assuming a 1:1 ratio between the collateral asset and the liquidity tokens that will actually be provided to redeemers. This assumption is flawed because the actual liquidity supplied to redeemers goes through the `externalCollateralRedemption.redeem()` function, which may apply fees, exchange rates, or other conversion mechanisms that break the 1:1 assumption.

```

function availableLiquidity() external view returns (uint256) {
    return IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider);
}

function _availableLiquidity() private view returns (uint256) {
    return IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider);
}

function supplyTo(

```

```

    address redeemer,
    uint256 amount,
    uint256 minOutputAmount
) public whenNotPaused onlySecuritizeRedemption {
    if (amount > _availableLiquidity()) {
        revert InsufficientLiquidity(amount, _availableLiquidity());
    }

    // ... collateral transfer and redemption logic ...

    // The actual liquidity provided is calculated here, not the raw collateral amount
    uint256 assetsAfterExternalCollateralRedemptionFee =
        externalCollateralRedemption.calculateLiquidityTokenAmount(
            amount
        );

    liquidityToken.transfer(redeemer, assetsAfterExternalCollateralRedemptionFee);
}

```

When CollateralLiquidityProvider::supplyTo is called, the flow involves: transferring collateral assets from the collateral provider, calling externalCollateralRedemption.redeem() to convert collateral to liquidity tokens, calculating the actual liquidity amount using externalCollateralRedemption.calculateLiquidityTokenAmount(), and finally transferring the calculated liquidity tokens to the redeemer. The availableLiquidity() function should query the external redemption contract to determine the actual liquidity that can be provided, rather than using the raw collateral asset balance. (e.g. externalCollateralRedemption.calculateLiquidityTokenAmount(IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider))

Additionally, the external availableLiquidity() function duplicates the logic of the internal _availableLiquidity() function instead of calling it, which goes against the intended design pattern and creates unnecessary code duplication.

Impact: Users and integrating systems may receive incorrect information about available liquidity, potentially leading to failed transactions when the actual convertible liquidity is less than the reported collateral asset balance.

Recommended Mitigation: Update the availableLiquidity() function to calculate the actual liquidity that can be provided by querying the external redemption contract, and fix the function to call the internal _availableLiquidity() function as intended:

```

function availableLiquidity() external view returns (uint256) {
-    return IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider);
+    return _availableLiquidity();
}

function _availableLiquidity() private view returns (uint256) {
-    return IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider);
+    uint256 collateralBalance =
    IERC20(externalCollateralRedemption.asset()).balanceOf(collateralProvider);
+    return externalCollateralRedemption.calculateLiquidityTokenAmount(collateralBalance);
}

```

Securitize: Fixed in commit [1da35c](#) and [4a426e](#).

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Bridge gas limit parameter lacks lower bound validation leading to potential failed deliveries

Description: The `updateGasLimit` function in the `USDCBridge` contract allows an admin to set the `gasLimit` parameter to any arbitrary value. This parameter is used when quoting and sending cross-chain USDC transfers, specifically as the gas limit for the execution of the `receivePayloadAndUSDC` function on the target chain. If the admin sets this value too low, the cross-chain message delivery will not have enough gas to complete execution on the destination chain.

Impact: If the `gasLimit` is set below the required threshold, all cross-chain deliveries initiated by the bridge will consistently fail due to out-of-gas errors during the `receivePayloadAndUSDC` execution. This would prevent users from successfully transferring USDC across chains using the bridge, effectively halting the bridge's core functionality. Funds may become stuck or require manual intervention to resolve failed deliveries.

Recommended Mitigation: Implement a minimum gas limit check in the `updateGasLimit` function to prevent the admin from setting the `gasLimit` below a safe operational threshold. This threshold should be determined based on the maximum expected gas usage of the `receivePayloadAndUSDC` function, with an additional safety margin. Optionally, emit an event or revert the transaction if an attempt is made to set the gas limit below this minimum value.

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.3.2 Missing validation check for liquidityToken

Description: The `CollateralLiquidityProvider::initialize()` function does **not verify** that the `liquidityToken` of the passed `securitizeOffRamp`'s `liquidityProvider()` matches the expected token. This check is present in `setExternalCollateralRedemption()` but missing here.

Impact: A mismatched `liquidityToken` could lead to misconfiguration, loss of funds, or unintended asset interactions.

Recommended Mitigation: Add a check in the `initialize()` function to verify that the `liquidityToken` of the `securitizeOffRamp`'s liquidity provider matches the `_liquidityToken` parameter:

```
function initialize(
    address _liquidityToken,
    address _recipient,
    address _securitizeOffRamp
) public onlyProxy initializer {
    if (_recipient == address(0) || _liquidityToken == address(0) || _securitizeOffRamp == address(0)) {
        revert NonZeroAddressError();
    }

    address expectedToken = ILiquidityProvider(
        ISecuritizeOffRamp(_securitizeOffRamp).liquidityProvider()
    ).liquidityToken();

    if (expectedToken != _liquidityToken) {
        revert LiquidityTokenMismatch();
    }

    __BaseContract_init();
    recipient = _recipient;
    liquidityToken = IERC20(_liquidityToken);
    securitizeOffRamp = ISecuritizeOffRamp(_securitizeOffRamp);
}
```

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.3.3 liquidityProviderWallet is not set during initialization

Description: In AllowanceLiquidityProvider::initialize, one of the key properties, liquidityProviderWallet, is not initialized. This property is declared as a public variable but never set during contract initialization. Since the contract does not enforce or set this value at any point in initialize, any functionality that depends on liquidityProviderWallet may behave incorrectly.

```
address public liquidityProviderWallet;
```

Impact: Until liquidityProviderWallet is set, functions like _availableLiquidity() and supplyTo() will rely on the default address(0) value. This could result in:

- Returning an incorrect liquidity value (typically zero).
- Causing failed or unexpected behavior during redemptions, since transferFrom(address(0), ...) will fail.

Recommended Mitigation: Update the initialize function to accept a _liquidityProviderWallet parameter and ensure it is validated and assigned:

```
function initialize(
    address _liquidityToken,
    address _recipient,
    address _securitizeOffRamp,
    address _liquidityProviderWallet
) public onlyProxy initializer {
    if (_recipient == address(0)) revert NonZeroAddressError();
    if (_liquidityToken == address(0)) revert NonZeroAddressError();
    if (_securitizeOffRamp == address(0)) revert NonZeroAddressError();
    if (_liquidityProviderWallet == address(0)) revert NonZeroAddressError();

    __BaseContract_init();
    recipient = _recipient;
    liquidityToken = IERC20(_liquidityToken);
    securitizeOffRamp = ISecuritizeOffRamp(_securitizeOffRamp);
    liquidityProviderWallet = _liquidityProviderWallet;
}
```

This ensures that liquidityProviderWallet is set once during contract initialization and cannot be accidentally or maliciously left uninitialized.

Securitize: Fixed in commit [ab08ae](#).

Cyfrin: Verified.

7.3.4 Missing storage gap on upgradeable base contracts

Description: The contract /contracts/utils/BaseContract of bc-securitize-bridge-sc repository is upgradeable (inherits from UUPSUpgradeable, OwnableUpgradeable, and PausableUpgradeable) but does not include a storage gap.

Storage gaps are essential for ensuring that new state variables can be added to the base contracts in future upgrades without affecting the storage layout of inheriting child contracts.

Impact: Any addition of new state variables in future versions of BaseContract can lead to storage collisions in the children contracts.

Recommendation: Add a storage gap to the BaseContract.

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

Securitize: Fixed in commit [1da35c](#).

Cyfrin: Verified.

7.3.5 Single-step ownership transfer pattern is not recommended

Description: The two BaseContract implementations currently inherit from OpenZeppelin's OwnableUpgradeable, which uses a **single-step ownership transfer** pattern. This approach is risky: if an incorrect address is set as the new owner, the contract may become permanently inaccessible to administrative functions (`onlyOwner` methods).

Impact: A misconfigured ownership transfer could lock critical administrative functionality, potentially disrupting operations or requiring emergency upgrades.

Recommended Mitigation: Adopt the **Ownable2StepUpgradeable** contract from OpenZeppelin. This two-step ownership transfer pattern ensures the new owner must explicitly accept the role, reducing the risk of accidental lockouts.

```
-abstract contract BaseContract is UUPSUpgradeable, PausableUpgradeable, OwnableUpgradeable {  
+abstract contract BaseContract is UUPSUpgradeable, PausableUpgradeable, Ownable2StepUpgradeable {
```

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.3.6 Use of `msg.sender` instead of `_msgSender()` prevents meta-transaction support

Description: Several contracts in the codebase use `msg.sender` directly instead of `_msgSender()`, which prevents proper meta-transaction support. This inconsistency affects both initialization and core functionality across the system.

The affected contracts and functions include:

- `BaseContract::__BaseContract_init()`
- `SecuritizeOffRamp::redeem()`
- `SecuritizeBridge::bridgeDSTokens()`
- `SecuritizeBridge::validateLockedTokens()`

The protocol properly uses `_msgSender()` in their other functions and access control patterns, indicating awareness of meta-transaction support, but this was not consistently applied across all functions.

When a user performs a meta-transaction through a trusted forwarder:

1. The forwarder calls the contract on behalf of the user
2. Functions using `msg.sender` receive the forwarder's address instead of the user's address
3. Validation, authorization, and business logic fail or operate incorrectly

Impact: Meta-transaction functionality is broken as the forwarder contract becomes the transaction sender instead of the intended user, potentially causing authorization failures, incorrect event emissions, and improper validation logic.

Recommended Mitigation: Replace all instances of `msg.sender` with `_msgSender()` to support meta-transactions:

```
// BaseContract.sol  
function __BaseContract_init() internal onlyInitializing {  
    __UUPSUpgradeable_init();  
    __Pausable_init();  
-    __Ownable_init(msg.sender);  
+    __Ownable_init(_msgSender());  
}  
  
// SecuritizeOffRamp.sol  
function redeem(uint256 assetAmount, uint256 minOutputAmount) external whenNotPaused nonZeroNavRate  
↳    nonZeroLiquidityProvider {  
        uint256 rate = navProvider.rate();
```

```

- RedemptionValidator.validateRedemption(msg.sender, assetAmount, asset);
+ RedemptionValidator.validateRedemption(_msgSender(), assetAmount, asset);

- CountryValidator.validateCountryRestriction(msg.sender, dsServiceConsumer, restrictedCountries);
+ CountryValidator.validateCountryRestriction(_msgSender(), dsServiceConsumer, restrictedCountries);

// ... calculations ...

RedemptionManager.RedemptionParams memory params = RedemptionManager.RedemptionParams({
    asset: asset,
    liquidityProvider: liquidityProvider,
    feeManager: feeManager,
    assetAmount: assetAmount,
    liquidityTokenAmount: liquidityTokenAmount,
    minOutputAmount: minOutputAmount,
-    redeemer: msg.sender,
+    redeemer: _msgSender(),
    assetBurn: assetBurn
});

// ... execution logic ...

emit RedemptionCompleted(
-    msg.sender,
+    _msgSender(),
    assetAmount,
    liquidityTokenAmount,
    rate,
    fee,
    address(liquidityProvider.liquidityToken())
);
}

// SecuritizeBridge.sol
function bridgeDSTokens(uint16 targetChain, uint256 value) external override payable whenNotPaused {
    uint256 cost = quoteBridge(targetChain);
    require(msg.value >= cost, "Transaction value should be equal or greater than quoteBridge
    ↪ response");
-    require(dsToken.balanceOf(msg.sender) >= value, "Not enough balance in source chain to bridge");
+    require(dsToken.balanceOf(_msgSender()) >= value, "Not enough balance in source chain to bridge");

    // ... validation logic ...

-    require(registryService.isWallet(msg.sender), "Investor not registered");
+    require(registryService.isWallet(_msgSender()), "Investor not registered");

-    string memory investorId = registryService.getInvestor(msg.sender);
+    string memory investorId = registryService.getInvestor(_msgSender());

    // ... other logic ...

-    dsToken.burn(msg.sender, value, BRIDGE_REASON);
+    dsToken.burn(_msgSender(), value, BRIDGE_REASON);

    wormholeRelayer.sendPayloadToEvm{value: msg.value}(
        targetChain,
        targetAddress,
        abi.encode(
            investorDetail.investorId,
            value,
-            msg.sender,
+            _msgSender(),
            investorDetail.country,

```

```

        investorDetail.attributeValues,
        investorDetail.attributeExpirations
    ),
    0,
    gasLimit,
    whChainId,
-    msg.sender
+    _msgSender()
);

- emit DSTokenBridgeSend(targetChain, address(dsToken), msg.sender, value);
+ emit DSTokenBridgeSend(targetChain, address(dsToken), _msgSender(), value);
}

function validateLockedTokens(string memory investorId, uint256 value, IDSRegistryService
→ registryService) private view {
    // ... compliance service logic ...

- uint256 availableBalanceForTransfer = complianceService.getComplianceTransferableTokens(msg.sender,
→ block.timestamp, uint64(lockPeriod));
+ uint256 availableBalanceForTransfer =
→ complianceService.getComplianceTransferableTokens(_msgSender(), block.timestamp,
→ uint64(lockPeriod));
    require(availableBalanceForTransfer >= value, "Not enough unlocked balance in source chain to
    → bridge");
}

```

Securitize: Fixed in commit [045925](#) and commit [1da35c](#).

Cyfrin: Verified.

7.3.7 whChainId should not be stored as immutable constant

Description: The SecuritizeBridge contract's use of an immutable whChainId constant, set during deployment, creates a critical vulnerability during chain forks. If the stored whChainId no longer matches the actual chain ID(block.chainid), cross-chain operations via sendPayloadToEvm may use an incorrect source chain ID.

Impact: This mismatch can cause cross-chain message deliveries to fail or refunds to process incorrectly, disrupting the contract's functionality. As a result, funds may become locked in the refund flow, leading to financial loss for users and undermining trust in the system's reliability.

Recommended Mitigation: Replace the immutable whChainId with dynamic chain ID retrieval:

```

// Remove: uint16 public immutable whChainId;

function getCurrentChainId() public view returns (uint16) {
    return uint16(block.chainid);
}

// In bridgeDSTokens function:
wormholeRelayer.sendPayloadToEvm{value: msg.value} (
    targetChain,
    targetAddress,
    abi.encode(/* payload data */),
    0,
    gasLimit,
    getCurrentChainId(),
    msg.sender
);

```

Securitize: Rejected. whChainId is not the EVM chain id, it's the Wormhole-specific chain id that is defined [here](#).

Cyfrin: Acknowledged.

7.3.8 Refund address in bridgeDSTokens function should be configurable

Description: In the SecuritizeBridge::bridgeDSTokens function, msg.sender is assigned as the refund address.

```
// Send Relayer message
wormholeRelayer.sendPayloadToEvm{value: msg.value} (
    targetChain,
    targetAddress,
    abi.encode(
        investorDetail.investorId,
        value,
        msg.sender,
        investorDetail.country,
        investorDetail.attributeValues,
        investorDetail.attributeExpirations
    ), // payload
0, // no receiver value needed since we're just passing a message
gasLimit,
whChainId,
msg.sender //audit refund address, any leftover gas will be sent to this address
);
```

Wormhole will refund the leftover gas to refundAddress. However, if the msg.sender is a smart contract that does not implement a function to withdraw the native gas token, any potential refund sent to it will become permanently inaccessible. This could result in locked funds that cannot be recovered by the original user.

Recommended Mitigation: Add a parameter to explicitly specify a refund address instead of defaulting to msg.sender. This allows the user or calling contract to define a fallback or externally owned account (EOA) for receiving refunds.

```
function bridgeDSTokens(..., address refundAddress) external {
    require(refundAddress != address(0), "Invalid refund address");
    ...
}
```

Securitize: Acknowledged.

Cyfrin: Acknowledged.

7.3.9 Usage of unofficial wormhole-solidity-sdk npm package poses security and maintenance risks

Description: The bridge contracts in the codebase are using wormhole-solidity-sdk version 0.9.0 from npm, which has been confirmed by the Wormhole team to be an unofficial deployment. According to the Wormhole team, the npm package published by sullof <francesco@sullo.co> is not their official release, and the only approved version is v0.1.0 available on GitHub. The official recommended approach is to use forge install wormhole-foundation/wormhole-solidity-sdk@v0.1.0.

The following contracts are affected by this issue:

- SecuritizeBridge.sol - imports IWormholeReceiver and IWormholeRelayer
- WormholeCCTPUpgradeable.sol - imports IWormholeRelayer, IWormhole, and ITokenMessenger
- USDCBridge.sol - inherits from WormholeCCTPUpgradeable via CCTPSender and CCTPReceiver
- RelayerMock.sol - imports IWormholeRelayer

The unofficial package is declared as a dependency in package.json with "wormhole-solidity-sdk": "^0.9.0" and is used throughout the bridge implementation for cross-chain message passing and CCTP (Circle Cross-Chain Transfer Protocol) functionality.

Impact: Using an unofficial SDK introduces potential security vulnerabilities, compatibility issues, and maintenance challenges as the codebase depends on unverified third-party code.

Recommended Mitigation: Replace the unofficial npm package with the official GitHub release.

Securitize: Fixed in commit [1da35c](#).

Cyfrin: Verified. We posted a [tweet](#) to alert others using the package, and the author replied, confirming it was intended solely for personal use, not for protocols. To prevent further confusion, the author has taken down the [package](#).

7.4 Informational

7.4.1 Misleading comments and documentation inconsistencies in on-ramp contracts

Description: Multiple contracts in the on-ramp system contain misleading comments, incorrect documentation, and interface inconsistencies that misrepresent the actual functionality.

- The `ISecuritizeOnRamp` interface documents a `Buy` event that is never emitted anywhere in the codebase, and references a non-existent `swapFor` function in the `toggleInvestorSubscription` documentation.
- The `ISecuritizeOnRamp` interface incorrectly declares `nonceByInvestor` and `calculateDsTokenAmount` as state-changing functions when they are actually view functions in the implementation.
- `MintingAssetProvider` uses `@title IAssetProvider` instead of its actual contract name, creating confusion about which contract is being documented.
- `IAssetProvider::securitizeOnRamp` function documentation contains a typo referring to "on ramo contract" instead of "on ramp contract".
- `MpbsFeeManager::setRedemptionFee` function documentation mentions the fee percentage is in basis points while it is supposed to be MBPS.

Impact: These misleading comments can cause developers to incorrectly integrate with the contracts by expecting functionality that doesn't exist.

Securitize: Fixed in commit [2b6c3a](#).

Cyfrin: Verified.

7.4.2 Unnecessary override keywords on interface implementation functions

Description: Multiple functions in the `SecuritizeOnRamp` contract use the `override` keyword unnecessarily. In Solidity, the `override` keyword is only required when overriding functions from parent contracts, not when implementing interface functions.

The following functions unnecessarily use the `override` keyword:

- `SecuritizeOnRamp::nonceByInvestor`
- `SecuritizeOnRamp::subscribe`
- `SecuritizeOnRamp::swap`
- `SecuritizeOnRamp::executePreApprovedTransaction`
- `SecuritizeOnRamp::calculateDsTokenAmount`
- `SecuritizeOnRamp::updateAssetProvider`
- `SecuritizeOnRamp::updateNavProvider`
- `SecuritizeOnRamp::updateMinSubscriptionAmount`
- `SecuritizeOnRamp::updateBridgeParams`
- `SecuritizeOnRamp::toggleInvestorSubscription`

Impact: The unnecessary `override` keywords create confusion about the contract's inheritance structure.

Recommended Mitigation: Remove the `override` keyword.

Securitize: Fixed in commit [bf7b87](#).

Cyfrin: Verified.

7.4.3 The swap function emits incorrect event type for existing investor purchases

Description: The SecuritizeOnRamp::swap function emits a Swap event instead of the expected Buy event when existing investors purchase assets. According to the interface documentation, the Swap event is intended for "new subscription agreements" while the Buy event should be emitted "when an existing investor buy assets". The swap function is specifically designed for existing registered investors (enforced by the investorExists modifier) to purchase additional assets, making it semantically a "buy" operation rather than a "swap" operation.

Impact: Off-chain systems and event listeners may incorrectly categorize existing investor purchases as new subscription agreements, leading to inaccurate tracking and reporting of investor activities.

Recommended Mitigation: Replace the Swap event emission with the appropriate Buy event in the swap function:

```
- emit Swap(_msgSender(), dsTokenAmount, _liquidityAmount, _msgSender());
+ emit Buy(_msgSender(), _liquidityAmount, dsTokenAmount, navProvider.rate());
```

Securitize: Fixed in [2b6c3a](#). Buy event was deprecated and deleted.

Cyfrin: Verified.

7.4.4 Upgradeable contracts missing _disableInitializers() in constructors

Description: The following contracts are upgradeable but do not call _disableInitializers() in their constructors:

- MintingAssetProvider
- AllowanceAssetProvider
- SecuritizeOnRamp
- MbpsFeeManager
- SecuritizeOffRamp
- AllowanceLiquidityProvider
- CollateralLiquidityProvider

In upgradeable contract patterns (such as those using OpenZeppelin's UUPS or Transparent proxies), the implementation (logic) contract is deployed independently from the proxy. If the implementation contract does not call _disableInitializers() in its constructor, it can be initialized directly by anyone, which is not intended and can lead to security risks. ([reference](#))

Impact: If the implementation contract is initialized directly, an attacker could set themselves as the owner or assign other privileged roles, potentially interfering with the upgrade process or causing confusion. While this does not directly affect the proxy's state, it can break upgradeability, allow denial of service, or create unexpected behaviors in the system.

Recommended Mitigation: Add a constructor to each affected contract that calls _disableInitializers(). This ensures the implementation contract cannot be initialized or reinitialized, preventing any unauthorized or accidental initialization outside the proxy context.

```
constructor() {
    _disableInitializers();
}
```

Add this to each of the affected contracts.

Securitize: Fixed in commit [088048](#).

Cyfrin: Verified.

7.4.5 Check for country code is not sufficient

Description: In the CountryValidator class, the method validateCountryCode is intended to check the validity of a given country code. However, it currently does not correctly filter out invalid codes. As a result, inputs such as XX or YYX, which are not valid ISO 3166-1 (and alpha-2 or alpha-3) country codes, are incorrectly considered valid.

Recommended Mitigation: Consider using a map of allowed country codes.

Securitize: Acknowledged. Strict ISO country code validation isn't necessary on-chain, as invalid codes are edge cases and can be handled off-chain during onboarding or KYC. On-chain format checks are sufficient for our use case.

Cyfrin: Acknowledged.

7.4.6 Confusing variable naming in fee manager contracts

Description: The fee manager contracts use confusing variable names for fee percentage values. In MbpsFeeManager, the variable fee represents a fee percentage in MBPS (milli basis points), not an actual fee amount. The comment even clarifies "Fee expressed in mbps (1000 mbps = 1%)", and the calculation formula (amount * fee + FEE_DENOMINATOR - 1) / FEE_DENOMINATOR shows that fee is used as a percentage rate. However, the variable name fee typically implies an actual fee amount rather than a percentage rate. For example, the fee manager contract exposes a function getFee(uint256 amount).

This naming convention creates confusion for those who expect fee to represent an actual fee amount rather than a percentage rate used in calculations.

Impact: The confusing variable names could lead to integration errors, misunderstanding of fee calculations, and potential bugs in contracts that interact with the fee managers.

Recommended Mitigation: Rename the fee variables to clearly indicate they represent percentages:

```
- uint256 public fee;
+ uint256 public feeMBPS;
```

Consider using feePercentageMBPS for even better clarity.

Securitize: Fixed in commit [6a5d45](#).

Cyfrin: Verified.

7.4.7 Unused parameter in address validation modifier SecuritizeOffRamp::addressNonZero

Description: The addressNonZero modifier used in SecuritizeOffRamp::initialize(), SecuritizeOffRamp::updateLiquidityProvider(), and SecuritizeOffRamp::updateNavProvider() functions accepts a string memory parameter argument but never uses it within the modifier logic. This parameter appears to be intended for providing context about which address parameter is being validated, but it remains unused in the error handling.

```
modifier addressNonZero(address _address, string memory parameter) {
    if (_address == address(0)) {
        revert NonZeroAddressError();
    }
    _;
}
```

The modifier is called with descriptive strings like "asset", "navProvider", "feeManager", and "liquidityProvider" but this contextual information is not utilized in the error reporting or validation logic. For comparison, other contracts in the codebase use similar address validation modifiers without unused parameters, such as addressNotZero in USDCBridge.sol which correctly implements the validation without taking unnecessary parameters.

Impact: The unused parameter creates inconsistent code patterns and represents a missed opportunity to provide meaningful error context when address validation fails.

Recommended Mitigation: Remove the unused parameter from the addressNonZero modifier to maintain consistency with similar validation patterns in other contracts:

```
- modifier addressNonZero(address _address, string memory parameter) {
+ modifier addressNonZero(address _address) {
    if (_address == address(0)) {
        revert NonZeroAddressError();
    }
}
```

And update all usage sites to remove the string parameter:

```
- addressNonZero(_asset, "asset")
+ addressNonZero(_asset)
- addressNonZero(_navProvider, "navProvider")
+ addressNonZero(_navProvider)
- addressNonZero(_feeManager, "feeManager")
+ addressNonZero(_feeManager)
- addressNonZero(_liquidityProvider, "liquidityProvider")
+ addressNonZero(_liquidityProvider)
```

Securitize: Fixed in commit [fd5511](#).

Cyfrin: Verified.

7.4.8 Unnecessary complexity in calculateLiquidityTokenAmountWithoutFee

Description: The SecuritizeOffRamp::calculateLiquidityTokenAmountWithoutFee function contains unnecessary complexity through its three-branch conditional logic that handles decimal conversions between asset and liquidity tokens.

This redundant branching increases code complexity, gas consumption, and potential for inconsistent rounding behavior between branches. The function can be significantly simplified while maintaining identical functionality.

Recommended Mitigation: Since the rate parameter is expressed in liquidity decimals, the function can be simplified to a single calculation that handles all decimal scenarios:

```
function calculateLiquidityTokenAmountWithoutFee(
    uint256 assetAmount,
    uint256 rate,
    uint256 liquidityDecimals,
    uint256 assetDecimals
) internal pure returns (uint256) {
    return (assetAmount * rate) / (10 ** assetDecimals);
}
```

Securitize: Fixed in [2bb438](#) and [0deec2](#).

Cyfrin: Verified. There was a miscommunication around the decimals of NAV rate and the original recommendation was not accurate. The team confirmed that the rate is NOT in any other liquidity token decimals but is in the decimal of the asset token itself. While this is not common practice in other protocols and it introduces some weird computation, the mitigated formula itself is technically correct.

7.4.9 Unused library and struct definitions increase deployment costs and reduce code clarity

Description: The CCTPMessageLib library and CCTPMessage struct are defined but never used throughout the CCTP integration contracts. In WormholeCCTPUpgradeable.sol, the library defines a CCTPMessage struct containing message and signature fields, and the contract imports it with using CCTPMessageLib for *. However, the CCTPBase::redeemUSDC function manually decodes CCTP messages using abi.decode(cctpMessage, (bytes, bytes)) instead of utilizing the defined struct.

```

function redeemUSDC(bytes memory cctpMessage) internal returns (uint256 amount) {
    (bytes memory message, bytes memory signature) = abi.decode(cctpMessage, (bytes, bytes));
    uint256 beforeBalance = IERC20(USDC).balanceOf(address(this));
    circleMessageTransmitter.receiveMessage(message, signature);
    return IERC20(USDC).balanceOf(address(this)) - beforeBalance;
}

```

The same issue exists in the upstream wormhole SDK's CCTPBase.sol file, suggesting this may have been copied without proper cleanup.

Impact: The unused code increases deployment gas costs and reduces code maintainability without providing any functional benefit.

Recommended Mitigation: Remove the unused CCTPMessageLib library and the using statement from the contracts:

```

- library CCTPMessageLib {
-     struct CCTPMessage {
-         bytes message;
-         bytes signature;
-     }
- }

abstract contract CCTPSender is CCTPBase {
    uint8 internal constant CONSISTENCY_LEVEL_FINALIZED = 15;

-   using CCTPMessageLib for *;

    mapping(uint16 => uint32) public chainIdToCCTPDomain;

```

Securitize: Fixed in commit [97e37b](#).

Cyfrin: Verified.

7.4.10 Missing slippage protection in external collateral redemption call

Description: In CollateralLiquidityProvider::supplyTo(), the function calls externalCollateralRedemption.redeem(collateralAmount, 0) with 0 as the minOutputAmount parameter, providing no slippage protection for the external redemption transaction. While this function is protected by the onlySecuritizeRedemption modifier and is intended to be called only by the SecuritizeOffRamp contract where slippage control is implemented at a higher level, it can lead to a potential problem if any other integration uses this function in future implementation.

Recommended Mitigation: Calculate an appropriate minimum output amount based on the expected liquidity amount and apply a reasonable slippage tolerance. Or consider modifying the function to accept a minOutputAmount parameter or calculate it based on the expected output:

Securitize: Acknowledged.

Cyfrin: Acknowledged.