USDi

Date	April 2025
Auditors	George Kobakhidze, Vladislav Yaroshuk

1 Executive Summary

This report presents the results of our engagement with **USDi Coin** to review their smart contract `USDiCoin`.

The review was conducted from **April 1, 2025** to **April 8, 2025**, by **George Kobakhidze** and **Vladislav Yaroshuk**. A total of **10** person-days were spent.

The review was conducted to ensure the correctness, security, and adherence of the contract's implementation to its intended functionality. The contract implements several distinctive features, including inflation-adjusted minting and burning, collateral backing with an ERC20 token - USDC token on the Ethereum network, deposit and withdrawal fees based on tiered structures, whitelisting of addresses with time delays, and oracle-driven price peg checks to prevent exploits.

The stablecoin's key mechanism revolves around deposits and withdrawals, whereby users deposit backing tokens in exchange for USDi tokens adjusted by CPI. The contract maintains tiered fees and minimum fee thresholds, directing collected fees to a designated treasury address. It also incorporates extensive access control through distinct roles—**Admin** and **Manager**—which control sensitive operations such as minting, burning, collateral management, CPI updates, and emergency pauses.

The system is designed to be centralized, the contract relies on privileged roles (Admins and Managers) who possess considerable powers, including the unrestricted ability to mint/burn tokens, withdraw collateral, alter CPI values, pause the system, and manage the whitelist. Deposited collateral tokens can be fully withdrawn and managed off-chain by the USDi team, underscoring the necessity for complete trust by users in the team's operational integrity, fund management practices, and compliance with inflation objectives.

The reviewed implementation is generally well-structured and follows common Solidity conventions. However, crucial aspects such as event emission for administrative operations, adequate checks in critical functions, extensive test coverage, and detailed documentation are notably absent.

2 Scope

Our review focused on the commit hash `302c2849ccd850cdfec55b7bc3855d7c884711b3` from the USDiCoin repository. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **USDi Coin** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- **Default Admin (`DEFAULT_ADMIN_ROLE`)**
  - Complete control over contract parameters, including fee management and collateral withdrawal.
  - Ability to mint or burn USDi arbitrarily without CPI adjustments.
  - Manage and update fee tiers, treasury address, and collateral backing tokens.
  - Grant or revoke roles, including the Manager role and additional Admin roles.
  - Pause and unpause the contract.
- **Manager (`MANAGER_ROLE`)**
  - Schedule new addresses for whitelisting and remove existing addresses from the whitelist.
  - Pause and unpause deposit and withdrawal operations in emergencies.
  - Update monthly CPI values used for inflation adjustments.
- **Regular User (Whitelisted)**
  - Deposit backing tokens to mint USDi tokens adjusted by CPI.
  - Withdraw collateral by burning USDi tokens, receiving backing tokens adjusted by CPI.
  - No administrative privileges or access control abilities.

3.2 Trust Model

In any system, it's important to identify what trust is expected or required between various actors. For this audit, we established the following trust model:

- **Token economics & backing increase**:
  - The big feature of the `USDi` token is its ability to track CPI, as defined by the administrators of the contract. This ability is concentrated in the contract's logic for deposits and withdrawals where with increasing CPI each `USDi` token requires more of the backing token (`USDC`) to be deposited, but also provides more for each withdrawal. However, there is nothing in the code that autonomously provides more of the backing token to the contract. In other words, the `USDi` users and holders are entirely dependent on the managers and administrators of the contracts to provide and safe keep an appropriate amount of `USDC` to correspond with the changing CPI values, and, thus, the conversion rates of `USDi` to `USDC`.
- **Access control**:
  - Users must fully trust Admins with their deposited collateral tokens, as Admins have unrestricted power to move and manage collateral on-chain.
  - Users trust Admins to responsibly manage fees, minting, burning, and contract parameters (e.g., CPI data, fee tiers).
  - Managers are trusted to responsibly manage whitelisting, CPI updates, and pausing mechanisms.
  - The auditing team assumes that all protocol roles are correctly assigned and trusted, not compromised. Actors with these roles must perform their duties responsibly and according to protocol rules, as they have significant control over key functionalities.

- **Reliance on Oracles**:
  - The system relies on external price oracles (e.g., Chainlink) for the accuracy and freshness of the backing token's USD price.
  - Users trust the oracle's reliability to prevent exploitation via manipulation of deposit and withdrawal eligibility.
- **Reliance on USDC**:
  - The protocol expects that the backing token is `USDC` and that the token is deployed on the Ethereum chain. Deflationary or non-compliant ERC-20 tokens will cause issues and are considered to be out-of-scope.
- **Correct Initialization**: All contracts must be initialized correctly.
- **Role-Based Access Control**: All protocol roles are correctly assigned and trusted, not compromised. Actors with these roles must perform their duties responsibly and according to protocol rules, as they have significant control over key functionalities.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Incorrect Fee Handling During Withdrawals Major ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed by applying the fee logic after `amount` is converted to `backingAmount` in `backingToken` units.

Description

As a way to generate fees, the contract deducts a portion of the user's deposit or withdrawal during processing. This behavior is evident in the use of the `getFee()` function and subsequent `backingToken` transfers, such as `backingToken.safeTransfer(treasury, fee)`. The fee is intended to be collected in the form of the `backingToken`.

In the `deposit()` function, the fee is correctly calculated as a fraction of the user-supplied `amount` (in `backingToken`) and transferred to the treasury. However, in the `withdraw()` function, the same logic is incorrectly applied. Specifically, the fee is calculated based on the `amount` of `USDi` being burned—before conversion to `backingToken`.

This results in the fee being denominated in `USDi`, while the actual fee transfer is performed using `backingToken`. Since the `amount` has not yet been converted to `backingToken`, the fee calculation occurs in the wrong unit. The fee logic should instead be applied after the `USDi` amount is converted to `backingToken` to ensure the correct token denomination and fee value.

See the related code:

contracts/USDiCoin.sol:L472-L484

```
function withdraw(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
    _requireBackingTokenPegInRange();

    require(amount > 0, "Amount must be greater than 0");
    require(balanceOf(msg.sender) >= amount, "Insufficient balance");

    // Compute fee, ensure min
    uint256 fee = getFee(amount, false);
    if (fee < minimumFee) {
        fee = minimumFee;
    }
    uint256 netAmount = amount - fee;
```

Recommendation

We recommend moving the fee calculation in the `withdraw()` function to occur **after** the `USDi` amount has been converted into its equivalent `backingToken` amount. This ensures that the fee is properly calculated and deducted in the correct units.

4.2 Risk of Backing Value Loss for Non-Whitelisted Token Holders Medium Acknowledged

Resolution

USDi team has acknowledged this finding and noted that whitelisted clients that perform deposits and withdrawals are aware of such potential market mechanics.

Description

The `withdraw` function restricts withdrawals to whitelisted addresses via `_requireWhitelisted(msg.sender)`, ensuring that only tokens held by whitelisted users can be redeemed for their backing value. However, if tokens are transferred to non-whitelisted addresses, those tokens lose their redemption guarantee and rely solely on external market (e.g., DEX) liquidity. This can result in significant price slippage or de-pegging of the token when non-whitelisted holders attempt to sell their tokens.

Such a design introduces potential economic imbalances and creates a two-tier market. Non-whitelisted holders may unknowingly receive or hold USDi that cannot be redeemed, resulting in diminished liquidity and increased susceptibility to price manipulation. This undermines the stablecoin's perceived value and could erode user trust.

Examples

contracts/USDiCoin.sol:L28

```
contract USDiCoin is ERC20, AccessControl, Pausable, ReentrancyGuard {
```

Recommendation

We recommend implementing safeguards to prevent or warn against transferring USDi tokens to non-whitelisted addresses. This could include transfer restrictions, redemption fallback mechanisms, or clearer documentation and interface signals to ensure users understand the implications of holding USDi outside the whitelist. Such measures would help preserve liquidity, protect users, and maintain the protocol's stability.

4.3 Unrestricted Withdrawal of Backing Tokens by Admin Medium Acknowledged

Resolution

The USDi team acknowledged this with the following note signifying offchain mechanisms to audit and verify appropriate economic activity and value store.

> We're backed by an audited real world asset fund, the actual money will always be withdrawn into the fund and regular financial audits will be published.

Description

The function `adminWithdrawBacking` allows an admin to withdraw tokens from the contract without any restrictions. The tokens injected through `adminInjectBacking` are not tracked separately, nor is there any limit or condition on the withdrawal amount. This could potentially allow an admin to withdraw all the tokens from the contract at once, resulting in the loss of user funds or unintended draining of the protocol's balance. This creates a single point of failure—if the admin address is compromised, all funds in the protocol could be stolen.

Examples

contracts/USDiCoin.sol:L264-L280

```
/// @notice Returns true if `account` is whitelisted and the delay has passed
function isWhitelisted(address account) public view returns (bool) {
    uint256 start = whitelistStartTime[account];
    if (start == 0) {
        // never scheduled
        return false;
    }
    // If the block time is after their readyTime, they're good
    return (block.timestamp >= start);
}

/// @notice Admin injects backing tokens into the contract (admin only)
function adminInjectBacking(uint256 amount) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(amount > 0, "Amount must be > 0");
    backingToken.safeTransferFrom(msg.sender, address(this), amount);
    emit BackingInjected(msg.sender, amount);
}
```

Recommendation

We recommend introducing clear tracking for the tokens injected via `adminInjectBacking` and restricting the `adminWithdrawBacking` function to allow only the withdrawal of the tracked amount. Alternatively, consider enforcing time-based limits that restrict withdrawals to a small percentage of the total balance at a time, such as a maximum of 3%.

4.4 Validation for BPS Values in Fee Tiers Medium ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed.

Description

The `addFeeTier` and `updateFeeTier` functions allow setting `mintFee` and `burnFee` without enforcing upper bounds. Since these fees are expressed in basis points (BPS), omitting such validation may lead to accidental or malicious misconfiguration. For example, an admin could set the fee to 100% (10,000 BPS), confiscating the entire user deposit or withdrawal. Worse, if a fee is set above 100%, all deposits and withdrawals would revert due to underflow, effectively halting user interactions with the protocol.

Examples

contracts/USDiCoin.sol:L213-L228

```
/// @notice Adds a new fee tier to the schedule (admin only)
function addFeeTier(uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    feeTiers.push(FeeTier(min, max, mintFee, burnFee));
}

/// @notice Updates an existing fee tier by index (admin only)
function updateFeeTier(uint256 index, uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    require(index < feeTiers.length, "Invalid index");
    feeTiers[index] = FeeTier(min, max, mintFee, burnFee);
}
```

Recommendation

We recommend implementing validation checks to ensure that `mintFee` and `burnFee` remain within a safe and reasonable range. For example, to cap fees at 5%, enforce the following:

```
require(mintFee <= 500 && burnFee <= 500, "Fee exceeds maximum allowed");
```

This ensures fee logic remains consistent and protects users from configuration errors or misuse.

4.5 Managers Can Overwrite Existing `monthlyCPI` Medium ✓ Fixed

Resolution

Addressed in commit `1ae396b8da0f15367703764e3071e8cdffd926a5` by checking if there is already a CPI value set for the month before setting it. If there is, the code now checks that `msg.sender` has the administrator role.

Description

As part of the contract's management, managers can call `setCPI` to set the Consumer Price Index (CPI) for a given month. This functionality is essential, as the contract relies on timely CPI data to enable deposits and withdrawals.

contracts/USDiCoin.sol:L532-L537

```
/// @notice Updates or sets the CPI data for a particular YYYYMM
/// Now accessible by Manager or Admin.
function setCPI(uint256 yearMonth, uint256 cpiValue) external onlyManagerOrAdmin {
    monthlyCPI[yearMonth] = cpiValue;
    emit CPIUpdated(yearMonth, cpiValue);
}
```

However, the current implementation allows managers to update the CPI for a month even if a value has already been set. This introduces a potential attack vector: a compromised manager could inflate a previously set CPI value, which would retroactively impact the redemption ratio and allow the attacker to withdraw a disproportionate amount of backing tokens after depositing under a lower CPI. While there may be rare cases where correction of CPI values is justified, such changes should be limited and tightly controlled.

Recommendation

We recommend tightening permissions in the `setCPI()` function so that only admins can update the CPI for a month **if** that month already has a CPI value set. This ensures that CPI corrections are deliberate and subject to stricter access control, while still allowing managers to perform their intended role in regularly submitting new CPI data.

4.6 Insufficient Whitelist Management Flexibility Minor Acknowledged

Resolution

USDi team has acknowledged this finding with the following note:

> We don't anticipate many small clients, rather few large clients (for the whitelist).

Description

The `deposit` function relies on `_requireWhitelisted(msg.sender)` to restrict access to approved addresses. However, the contract lacks any logic to remove or disable the whitelist functionality entirely, which is particularly problematic in a non-upgradable contract. As the protocol scales, maintaining a growing list of whitelisted users may become impractical, leading to operational limitations and reduced accessibility.

Examples

contracts/USDiCoin.sol:L438-L439

```
function deposit(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
```

contracts/USDiCoin.sol:L472-L473

```
function withdraw(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
```

Recommendation

We recommend adding functionality to manage the whitelist more flexibly, including the ability to disable whitelist enforcement entirely if needed. This will help ensure the system remains adaptable and scalable over time.

4.7 Potential Underflow Risk and Suboptimal Call Ordering Minor Acknowledged

Resolution

The USDi team acknowledged this and plans to address in future versions.

Description

The `deposit` and `withdraw` functions calculate a `fee` based on the deposit or withdrawal `amount` and then subtract this `fee` to determine the net amount used for minting or burning. However, if the `amount` is less than the `minimumFee`, the subtraction may underflow and revert with a panic error before any validations are reached. Additionally, the token transfer is executed before fee calculation, which can lead to suboptimal error handling and unnecessary gas consumption in failing transactions.

The code structure would be improved by removing the early `require(amount > 0)` validation and instead performing the fee calculation first, followed by a check such as `require(netAmount > 0)`. This would preserve the original intention while improving gas efficiency and clarity. In any case, the `amount` must still be greater than `0` for minting or burning to succeed.

Moreover, the contract does not verify whether sufficient backing funds have been injected by the admin before allowing deposits. This implicitly relies on user trust that the protocol is adequately funded. Failing to verify this could result in users depositing funds into an under-collateralized system, potentially affecting their ability to redeem in the future.

Examples

contracts/USDiCoin.sol:L436-L502

```
/// @notice Allows a user to deposit backing tokens and mint USDi adjusted by CPI
/// Fees are deducted if and only if the deposit succeeds. If anything fails, no fees are collected.
function deposit(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
    // Check if the backing token remains in the accepted peg range around $1
    _requireBackingTokenPegInRange();

    require(amount > 0, "Amount must be greater than 0");

    // Transfer entire deposit in one call
    backingToken.safeTransferFrom(msg.sender, address(this), amount);

    // Compute fee, ensure min
    uint256 fee = getFee(amount, true);
    if (fee < minimumFee) {
        fee = minimumFee;
    }
    // The net portion used for minting
    uint256 netAmount = amount - fee;
    require(netAmount > 0, "Fee exceeds deposit");

    // Calculate minted amount from net deposit
    uint256 currentCPI = getProratedCPI();
    uint256 adjustedAmount = (netAmount * startingCPI) / currentCPI;

    // Mint the tokens to the user
    _mint(msg.sender, adjustedAmount);

    // Transfer fee to the treasury
    backingToken.safeTransfer(treasury, fee);

    emit Deposit(msg.sender, amount, fee);
}

/// @notice Allows a user to burn USDi and withdraw backing tokens adjusted by CPI
/// Fees are deducted if and only if the withdrawal succeeds. If anything fails, no fees are collected.
function withdraw(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
    _requireBackingTokenPegInRange();

    require(amount > 0, "Amount must be greater than 0");
    require(balanceOf(msg.sender) >= amount, "Insufficient balance");

    // Compute fee, ensure min
    uint256 fee = getFee(amount, false);
    if (fee < minimumFee) {
        fee = minimumFee;
    }
    uint256 netAmount = amount - fee;
    require(netAmount > 0, "Fee exceeds withdrawal");

    // Calculate how many backing tokens the user receives
    uint256 currentCPI = getProratedCPI();
    uint256 backingAmount = (netAmount * currentCPI) / startingCPI;

    // Burn user's USDi
    _burn(msg.sender, amount);

    // Transfer net backing to user
    backingToken.safeTransfer(msg.sender, backingAmount);

    // Transfer fee to treasury
    backingToken.safeTransfer(treasury, fee);

    lastWithdrawTime[msg.sender] = block.timestamp;
    emit Withdrawal(msg.sender, backingAmount, fee);
}
```

Recommendation

We recommend reordering the logic so that fee calculations and validations occur before any token transfers are executed. Additionally, we recommend adding a check to verify that sufficient backing funds have been injected into the contract before allowing deposits. This ensures a more robust and user-safe experience while maintaining proper economic guarantees.

4.8 Missing Batch Processing for Whitelist Scheduling Minor Acknowledged

Resolution

USDi team has acknowledged this finding with the following note:

> We don’t anticipate many small clients, rather few large clients (for the whitelist).

Description

The `scheduleWhitelist` function schedules a single address for whitelisting by setting its ready time and emitting a `WhitelistRequested` event. However, the function does not support batch processing, which can lead to higher gas costs when adding multiple users. This becomes a scalability limitation, as each user must be added in a separate transaction, each incurring a base cost of 21,000 gas, in addition to computation and storage costs.

Examples

contracts/USDiCoin.sol:L248-L255

```
/// @notice Initiates the whitelist schedule for an address; after `whitelistChangeDelay` has elapsed, address can transact
/// Now only a MANAGER or ADMIN can call this.
function scheduleWhitelist(address account) external onlyManagerOrAdmin {
    require(account != address(0), "Invalid address");
    uint256 readyTime = block.timestamp + whitelistChangeDelay;
    whitelistStartTime[account] = readyTime;
    emit WhitelistRequested(account, readyTime);
}
```

Recommendation

We recommend implementing a batch version of the `scheduleWhitelist` function to efficiently process multiple addresses in a single transaction. This will significantly reduce the overall gas cost and improve the scalability of the whitelisting mechanism.

4.9 `minimumFee` Is Not Validated for Acceptable Bounds Minor Acknowledged

Resolution

USDi team acknowledged this with the following note:

> We don't plan on changing fees, and deposits/withdrawals are expected to be in large sums of money.

Description

The `setMinimumFee` function updates the `minimumFee` without validating that the new value is within acceptable bounds. This could allow an admin to set the fee to an unreasonably high value, potentially disrupting the protocol's fee structure and negatively impacting user experience.

Examples

contracts/USDiCoin.sol:L237-L240

```
/// @notice Updates the minimum mint/burn fee charged (admin only)
function setMinimumFee(uint256 newMinimumFee) external onlyRole(DEFAULT_ADMIN_ROLE) {
    minimumFee = newMinimumFee;
}
```

Recommendation

We recommend adding a validation check to ensure that `newMinimumFee` does not exceed a predefined maximum value, thereby maintaining consistency and protecting against misconfiguration.

4.10 Potentially Unnecessary Usage of `_toString()` Minor ✓ Fixed

Resolution

Addressed in the `1ae396b8da0f15367703764e3071e8cdffd926a5` and `3c30ce7d4172d04bba9e4c8f4b7da5d1058c1ef9` commits for fix review. The `_toString()` function is no longer used for mints and burns, and instead a `timestamp` argument has been added to the events. The `_toString ()` function is now also removed.

Description

The contract uses the `_toString()` function to convert `block.timestamp` into a `string`, which is then concatenated with a message string for event emission:

contracts/USDiCoin.sol:L507-L509

```
string memory baseReason = bytes(reason).length > 0 ? reason : "manual mint";
string memory fullReason = string(abi.encodePacked("[", _toString(block.timestamp), "] ", baseReason));
emit ManualMint(to, amount, fullReason);
```

However, if the goal is simply to include timestamp information in the event, it would be more efficient and cleaner to emit `block.timestamp` directly as a `uint256` field in the event. This would eliminate unnecessary computation from converting a `uint256` to a `string` and reduce gas usage from concatenating strings.

Recommendation

We recommend removing the `_toString()` usage and instead modifying the `ManualMint` and `ManualBurn` events to include a `uint256` timestamp field directly. This improves efficiency and simplifies event formatting.

4.11 Default Fee Fallback to Zero When No Fee Tier Matches Minor ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit the issue has been addressed by introducing a revert if amounts don't apply to any fee tier.

Description

The `getFee` function calculates the applicable fee by iterating through the `feeTiers` array and returns `0` if no tier matches the transaction amount. This default behavior can unintentionally exempt certain transactions from fees if the fee tiers are not exhaustively defined, potentially reducing protocol revenue and disrupting fee consistency.

Additionally, the configuration of the fee tiers contains a logical inconsistency. The tier for transaction amounts from `10_000_000e6` to `199_999_999e6` applies a `0` bps deposit fee and `2` bps withdrawal fee, while the tier from `200_000_000e6` to `type(uint256).max` applies `2` bps and `5` bps respectively. Since the fee is calculated based on the transaction amount—not on the user's total balance—users would be incentivized to split large transactions into multiple smaller ones to remain within the lower-fee tier. This undermines the protocol's intended fee structure and could result in higher on-chain activity and inefficiency.

Examples

contracts/USDiCoin.sol:L133-L134

```
feeTiers.push(FeeTier(10_000_000e6, 199_999_999e6, 0, 2));
feeTiers.push(FeeTier(200_000_000e6, type(uint256).max, 2, 5));
```

contracts/USDiCoin.sol:L200-L211

```
/// @notice Computes the applicable fee for a given transaction amount
/// @param amount The size of the transaction
/// @param isMint True for deposits, false for withdrawals
function getFee(uint256 amount, bool isMint) public view returns (uint256) {
    for (uint256 i = 0; i < feeTiers.length; i++) {
        if (amount >= feeTiers[i].min && amount <= feeTiers[i].max) {
            uint256 bps = isMint ? feeTiers[i].mintFee : feeTiers[i].burnFee;
            return (amount * bps) / 10000;
        }
    }
    return 0;
}
```

Recommendation

We recommend ensuring that the fee tiers comprehensively cover all possible transaction amounts and that a non-zero default fee is returned if no tier matches. Additionally, we advise reviewing and correcting the configuration of the highest fee tier to maintain logical and consistent fee progression, preventing users from bypassing fees through transaction splitting.

4.12 Overlapping Fee Tiers Not Checked Minor Acknowledged

Resolution

USDi team acknowledged this with the following note:

> We don't plan on changing fee tiers and don't expect them to overlap.

Description

The `addFeeTier` and `updateFeeTier` functions currently allow adding overlapping fee tiers without validation. This can lead to unintended fee calculations, as the logic to determine the applicable fee relies on non-overlapping tier ranges.

Examples

contracts/USDiCoin.sol:L200-L211

```
/// @notice Computes the applicable fee for a given transaction amount
/// @param amount The size of the transaction
/// @param isMint True for deposits, false for withdrawals
function getFee(uint256 amount, bool isMint) public view returns (uint256) {
    for (uint256 i = 0; i < feeTiers.length; i++) {
        if (amount >= feeTiers[i].min && amount <= feeTiers[i].max) {
            uint256 bps = isMint ? feeTiers[i].mintFee : feeTiers[i].burnFee;
            return (amount * bps) / 10000;
        }
    }
    return 0;
}
```

contracts/USDiCoin.sol:L213-L228

```
/// @notice Adds a new fee tier to the schedule (admin only)
function addFeeTier(uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    feeTiers.push(FeeTier(min, max, mintFee, burnFee));
}

/// @notice Updates an existing fee tier by index (admin only)
function updateFeeTier(uint256 index, uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    require(index < feeTiers.length, "Invalid index");
    feeTiers[index] = FeeTier(min, max, mintFee, burnFee);
}
```

The impact of overlapping fee tiers is amplified in this case because the tiers are not sorted, and the first tier that matches the amount may not necessarily be the most optimal or intended one.

Recommendation

We recommend implementing validation logic within the `addFeeTier` and `updateFeeTier` functions to ensure newly added or modified fee tiers do not overlap with existing ones. This validation can mirror the logic used in the `getFee` function by iterating over all tiers. Additionally, we recommend using the `EnumerableSet` library to store and manage the `feeTiers` variable for better structure and data integrity.

4.13 Chainlink's `latestRoundData` Might Return Stale Results Minor ✓ Fixed

Resolution

Addressed in commit `45181fbc99153658f32fd6202bd0604e3e80d588` by removing the price oracle altogether as well as the associated functions for checking that the `backingToken` price is within acceptable range of the peg. The USDi team acknowledges that the token can be off peg at specific times and accepts that risk as the `backingToken` conversion will happen offchain after they receive mint and withdrawal requests from their clients, and the price oracle in the smart contract does not define any price or conversion logic for `USDi`. Specifically, since the intended `backingToken` is `USDC`, the USDi team feels comfortable with accepting the off-peg risk.

Description

The `_getBackingTokenPriceInUsd` function fetches the latest price from the oracle without verifying the freshness of the data. Without a freshness check, the contract could rely on outdated price data, potentially exposing the protocol to price manipulation attacks or significant economic inaccuracies.

Examples

contracts/USDiCoin.sol:L168-L198

```
/// @notice Internal helper to get the backing token's price in USD from the oracle
/// Typically scaled by 1e8 if we're using a standard Chainlink feed.
function _getBackingTokenPriceInUsd() internal view returns (uint256) {
    (
        /* uint80 roundID */,
        int256 answer,
        /* uint256 startedAt */,
        /* uint256 updatedAt */,
        /* uint80 answeredInRound */
    ) = backingTokenPriceOracle.latestRoundData();
    require(answer > 0, "Invalid price from oracle");
    return uint256(answer); // e.g. 100000000 => $1.00
}

/// @notice Ensures the backing token's price is within an acceptable range around $1.00
function _requireBackingTokenPegInRange() internal view {
    // 1 USD in 1e8 is 100000000
    uint256 oneUsd = 100000000;

    // offset = (oneUsd * maxDeviationBps) / 10000
    // e.g. if maxDeviationBps=200 => offset=2% of oneUsd => 2000000 => $0.02
    uint256 offset = (oneUsd * maxDeviationBps) / 10000;
    uint256 lowerBound = oneUsd - offset;
    uint256 upperBound = oneUsd + offset;

    uint256 currentPrice = _getBackingTokenPriceInUsd();
    require(
        currentPrice >= lowerBound && currentPrice <= upperBound,
        "Backing token price out of range"
    );
}
```

Recommendation

We recommend implementing a freshness check by validating the `updatedAt` timestamp returned by the oracle. Ensure the data retrieved is within a reasonable timeframe (e.g., not older than several hours, depending on protocol needs) before using it within the contract logic.

4.14 Missing Event for Critical Parameter Changes ✓ Fixed

Resolution
In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed.

Description

Setter functions for key protocol parameters (e.g., `maxDeviationBps`, `backingTokenPriceOracle`, fee tiers, `minimumFee`, and `treasury`) do not emit events when changes are made. This omission limits visibility into parameter updates and hinders off-chain monitoring tools. Events for such updates should include both the old and new values to ensure transparency and auditability.

contracts/USDiCoin.sol:L213-L246

```
/// @notice Adds a new fee tier to the schedule (admin only)
function addFeeTier(uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    feeTiers.push(FeeTier(min, max, mintFee, burnFee));
}

/// @notice Updates an existing fee tier by index (admin only)
function updateFeeTier(uint256 index, uint256 min, uint256 max, uint256 mintFee, uint256 burnFee)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    require(index < feeTiers.length, "Invalid index");
    feeTiers[index] = FeeTier(min, max, mintFee, burnFee);
}

/// @notice Removes a fee tier by index (admin only)
function removeFeeTier(uint256 index) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(index < feeTiers.length, "Invalid index");
    feeTiers[index] = feeTiers[feeTiers.length - 1];
    feeTiers.pop();
}

/// @notice Updates the minimum mint/burn fee charged (admin only)
function setMinimumFee(uint256 newMinimumFee) external onlyRole(DEFAULT_ADMIN_ROLE) {
    minimumFee = newMinimumFee;
}

/// @notice Updates the treasury address (admin only)
function setTreasury(address newTreasury) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(newTreasury != address(0), "Invalid treasury address");
    treasury = newTreasury;
}
```

contracts/USDiCoin.sol:L157-L166

```
function setMaxDeviationBps(uint256 newDeviationBps) external onlyRole(DEFAULT_ADMIN_ROLE) {
    maxDeviationBps = newDeviationBps;
}

/// @notice Lets admin update the oracle address if needed
/// (still admin only)
function setBackingTokenPriceOracle(address newOracle) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(newOracle != address(0), "Invalid oracle");
    backingTokenPriceOracle = IPriceOracle(newOracle);
}
```

Recommendation

Implement more events as appropriate.

4.15 Use a `modifier` Instead of a `require` / `if` Statement for a Special `msg.sender` Actor Acknowledged

Resolution

Acknowledged by the USDi team as the intended style.

Description

The contract uses `_requireWhitelisted(msg.sender)` checks inline to restrict access to whitelisted users. Replacing these repeated checks with a dedicated modifier (e.g., `onlyWhitelisted`) would improve readability and maintain consistency across the codebase.

contracts/USDiCoin.sol:L438-L439

```
function deposit(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
```

contracts/USDiCoin.sol:L472-L473

```
function withdraw(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
```

Recommendation

Utilize modifiers instead of passing caller's address directly to a function.

4.16 `public` Functions Not Called by the Contract Should Be Declared `external` Instead ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed.

Description

The function `isWhitelisted` is declared as `public` but is not used internally. Declaring it as `external` would save gas and clearly signal its intended use.

contracts/USDiCoin.sol:L265

```
function isWhitelisted(address account) public view returns (bool) {
```

Recommendation

Change functions' visibility as appropriate.

4.17 `immutable startingCPI` Could Be `constant` ✓ Fixed

Resolution

In the `3c30ce7d4172d04bba9e4c8f4b7da5d1058c1ef9` commit for fix review the finding has been fixed.

Description

The variable `startingCPI` is marked as `immutable` but is assigned a hard-coded value (`315605`) in the constructor. Since it does not depend on a deployer-provided value, it would be more efficient to declare it as `constant`. Furthermore, the `StartingCPISet` event emission becomes redundant, as the value is always the same.

contracts/USDiCoin.sol:L71

```
uint256 public immutable startingCPI; // CPI baseline for adjustment
```

contracts/USDiCoin.sol:L118

```
startingCPI = 315605;
```

Recommendation

We recommend either allowing deployer-provided values or declaring the variable as `constant` with an inline value.

4.18 Constant Declaration for Whitelist Change Delay ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed.

Description

The `whitelistChangeDelay` variable is declared as a public mutable variable instead of a constant. Since this value never changes after deployment, declaring it as a constant would reduce storage costs and improve code readability.

Examples

contracts/USDiCoin.sol:L78

```
uint256 public whitelistChangeDelay = 12 hours; // delay before whitelisting becomes active
```

Recommendation

We recommend declaring `whitelistChangeDelay` as a constant to optimize gas usage and enhance clarity in the code.

4.19 `if` Condition Can Be Optimized in `_daysInMonth` Function ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review `_daysInMonth` function has been removed.

Description

The `_daysInMonth` function currently checks explicitly for each month with 31 days, listing them individually. While functionally correct, this approach reduces clarity and maintainability. The same logic could be expressed more cleanly by using a simplified conditional structure. For instance, replacing the multiple checks with `else if (month != 2)` would preserve the intended behavior while improving readability and minimizing redundancy.

Examples

contracts/USDiCoin.sol:L305-L329

```
/// @notice Returns the number of days in a given month of a particular year
function _daysInMonth(uint256 year, uint256 month) internal pure returns (uint256) {
    require(month >= 1 && month <= 12, "Month out of range");
    if (
        month == 1 || // Jan
        month == 3 || // Mar
        month == 5 || // May
        month == 7 || // Jul
        month == 8 || // Aug
        month == 10 || // Oct
        month == 12    // Dec
    ) {
        return 31;
    } else if (
        month == 4 || // Apr
        month == 6 || // Jun
        month == 9 || // Sep
        month == 11   // Nov
    ) {
        return 30;
    } else {
        // February
        return _isLeapYear(year) ? 29 : 28;
    }
}
```

Recommendation

We recommend refactoring the conditional logic in `_daysInMonth` to avoid explicitly listing each 31-day month. A simplified structure using `else if (month != 2)` would enhance readability and maintainability without changing the underlying functionality.

4.20 The Interface `IPriceOracle` Should Be in Its Own File ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the finding has been fixed. Later, in commit `45181fbc99153658f32fd6202bd0604e3e80d588`, the `IPriceOracle` interface file has been removed altogether after removing calls to the price oracle contract.

Description

Interfaces, especially those not inherited by the main contract in the same file, should typically be imported from separate files. This improves maintainability, especially in the event of future updates (e.g., changes to the oracle interface).

contracts/USDiCoin.sol:L14-L25

```
interface IPriceOracle {
    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        );
}
```

Recommendation

Separate out the interface into its own file.

4.21 Lack of Roles Segregation Acknowledged

Resolution

Acknowledged by the USDi team as a feature for future versions.

Description

The contract uses a utility modifier `onlyManagerOrAdmin` that grants access if `msg.sender` holds either `MANAGER_ROLE` or `DEFAULT_ADMIN_ROLE`. This approach does not adequately separate responsibilities, as an address with `DEFAULT_ADMIN_ROLE` can grant itself `MANAGER_ROLE` and perform both administrative and operational tasks without restriction. This blurs the boundaries between roles and undermines the granularity offered by `AccessControl`. As a result, role management in this implementation more closely resembles an `Ownable` pattern rather than a fully segregated access control system.

Examples

contracts/USDiCoin.sol:L142-L153

```
////////////////////////////////////////////////////////////
/// Access modifiers
////////////////////////////////////////////////////////////

// A utility modifier that checks if msg.sender has MANAGER_ROLE or DEFAULT_ADMIN_ROLE
modifier onlyManagerOrAdmin() {
    require(
        hasRole(MANAGER_ROLE, msg.sender) || hasRole(DEFAULT_ADMIN_ROLE, msg.sender),
        "Not manager or admin"
    );
    _;
}
```

Recommendation

We recommend segregating responsibilities by introducing distinct roles for specific functionalities. For example:

- Use `DEFAULT_ADMIN_ROLE` strictly for high-privilege operations and role management.
- Use `MANAGER_ROLE` for whitelist management and CPI updates.
- Use `PAUSER_ROLE` for pause/unpause operations.
- Use `FEE_MANAGER_ROLE` for managing fee tiers.

This separation enhances clarity, enforces least privilege, and reduces the risk of misuse or accidental overreach by a single role.

4.22 `monthlyCPI` Not Initialized in Constructor ✓ Fixed

Resolution

In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the problem has been fixed.

Description

The `monthlyCPI` variable is not initialized in the `constructor`, even though the contract sets the `startingCPI`. As a result, the contract will not be operational immediately after deployment, since this variable is used in the `getProratedCPI` function, which is invoked by both the `deposit` and `withdraw` functions.

Examples

contracts/USDiCoin.sol:L106-L135

```
/// @notice Constructor to initialize the contract with backing token, oracle, and CPI baseline
/// @param _backingToken The ERC20 used as backing asset
/// @param _backingTokenOracle The oracle returning the backing token price in USD
constructor(address _backingToken, address _backingTokenOracle)
    ERC20("USDi Coin", "USDi")
{
    require(_backingToken != address(0), "Invalid backing token address");
    require(_backingTokenOracle != address(0), "Invalid oracle address");

    backingToken = IERC20(_backingToken);
    backingTokenPriceOracle = IPriceOracle(_backingTokenOracle);

    startingCPI = 315605;
    treasury = msg.sender;

    // The deployer is granted DEFAULT_ADMIN_ROLE.
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    // The deployer is also granted MANAGER_ROLE.
    _grantRole(MANAGER_ROLE, msg.sender);

    emit StartingCPISet(startingCPI);

    // Initialize tiered fee structure
    feeTiers.push(FeeTier(0, 100_000e6, 50, 100));
    feeTiers.push(FeeTier(100_000e6 + 1, 999_999e6, 20, 50));
    feeTiers.push(FeeTier(1_000_000e6, 4_999_999e6, 5, 10));
    feeTiers.push(FeeTier(5_000_000e6, 9_999_999e6, 0, 5));
    feeTiers.push(FeeTier(10_000_000e6, 199_999_999e6, 0, 2));
    feeTiers.push(FeeTier(200_000_000e6, type(uint256).max, 2, 5));
}
```

contracts/USDiCoin.sol:L401-L426

```
/// @notice Calculates a CPI value prorated based on how far into the *actual* current month we are
/// monthlyCPI from 3 months ago as previousCPI, from 2 months ago as currentCPI.
function getProratedCPI() public view returns (uint256) {
    uint256 realNowYM = _getRealCurrentYearMonth();
    uint256 cpiCurrentYM = _shiftYearMonth(realNowYM, -2);
    uint256 cpiPreviousYM = _shiftYearMonth(realNowYM, -3);

    uint256 cpiCurrent = monthlyCPI[cpiCurrentYM];
    uint256 cpiPrevious = monthlyCPI[cpiPreviousYM];
    require(cpiPrevious > 0 && cpiCurrent > 0, "No CPI data available");

    uint256 startOfCurrent = _startOfMonth(realNowYM);
    uint256 startOfNext = _startOfNextMonth(realNowYM);
    if (block.timestamp < startOfCurrent) {
        return cpiPrevious;
    }
    if (block.timestamp >= startOfNext) {
        return cpiCurrent;
    }

    uint256 timeElapsed = block.timestamp - startOfCurrent;
    uint256 monthLength = startOfNext - startOfCurrent;
    uint256 fraction = (timeElapsed * 1e18) / monthLength;

    return ((cpiPrevious * (1e18 - fraction)) + (cpiCurrent * fraction)) / 1e18;
}
```

contracts/USDiCoin.sol:L436-L502

```
/// @notice Allows a user to deposit backing tokens and mint USDi adjusted by CPI
/// Fees are deducted if and only if the deposit succeeds. If anything fails, no fees are collected.
function deposit(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
    // Check if the backing token remains in the accepted peg range around $1
    _requireBackingTokenPegInRange();

    require(amount > 0, "Amount must be greater than 0");

    // Transfer entire deposit in one call
    backingToken.safeTransferFrom(msg.sender, address(this), amount);

    // Compute fee, ensure min
    uint256 fee = getFee(amount, true);
    if (fee < minimumFee) {
        fee = minimumFee;
    }
    // The net portion used for minting
    uint256 netAmount = amount - fee;
    require(netAmount > 0, "Fee exceeds deposit");

    // Calculate minted amount from net deposit
    uint256 currentCPI = getProratedCPI();
    uint256 adjustedAmount = (netAmount * startingCPI) / currentCPI;

    // Mint the tokens to the user
    _mint(msg.sender, adjustedAmount);

    // Transfer fee to the treasury
    backingToken.safeTransfer(treasury, fee);

    emit Deposit(msg.sender, amount, fee);
}

/// @notice Allows a user to burn USDi and withdraw backing tokens adjusted by CPI
/// Fees are deducted if and only if the withdrawal succeeds. If anything fails, no fees are collected.
function withdraw(uint256 amount) external nonReentrant whenNotPaused {
    _requireWhitelisted(msg.sender);
    _requireBackingTokenPegInRange();

    require(amount > 0, "Amount must be greater than 0");
    require(balanceOf(msg.sender) >= amount, "Insufficient balance");

    // Compute fee, ensure min
    uint256 fee = getFee(amount, false);
    if (fee < minimumFee) {
        fee = minimumFee;
    }
    uint256 netAmount = amount - fee;
    require(netAmount > 0, "Fee exceeds withdrawal");

    // Calculate how many backing tokens the user receives
    uint256 currentCPI = getProratedCPI();
    uint256 backingAmount = (netAmount * currentCPI) / startingCPI;

    // Burn user's USDi
    _burn(msg.sender, amount);

    // Transfer net backing to user
    backingToken.safeTransfer(msg.sender, backingAmount);

    // Transfer fee to treasury
    backingToken.safeTransfer(treasury, fee);

    lastWithdrawTime[msg.sender] = block.timestamp;
    emit Withdrawal(msg.sender, backingAmount, fee);
}
```

Specifically, at least two months must be initialized with corresponding `monthlyCPI` values—those from two and three months before the current date—in order for the contract to function as intended.

Recommendation

We recommend explicitly initializing the `monthlyCPI` values for the required prior months in the `constructor` to ensure the contract is fully operational immediately after deployment.

4.23 Withdrawal Fees Can Be Manipulated Using Flash Loans Acknowledged

Resolution

USDi team has acknowledged this finding with the following note:

> Whitelist partners are vetted, trusted, and are under legal obligations to not act game the system. If they misbehave, we will remove them manually.

Description

The withdrawal fee mechanism relies on the withdrawal amount to determine the applicable tier and fee percentage. However, this system can be exploited using flash loans to temporarily inflate a user's balance into a higher tier. By crossing a threshold into a lower-fee tier, the user is able to withdraw the full amount at a reduced fee. After repaying the flash loan, the user effectively benefits from a significant fee reduction, even after accounting for borrowing costs.

For instance, a user with a balance of `995,000`—falling in the 0.5% fee tier—could take a flash loan of `5,000` to reach `1,000,000`, thereby entering the 0.1% tier. This allows the user to withdraw at a much lower cost, defeating the intended proportionality of the fee system.

Consider the following scenario:

```
User balance: 995,000
Current withdrawal tier (100,001 to 999,999): 0.5% withdrawal fee
Next tier (1,000,000 to 4,999,999): 0.1% withdrawal fee
Flash loan cost (total): 0.5% (0.25% borrow + 0.25% repay, flashloan is free and 0.25 % is Uniswap fee)

Without flash loan:
Total cost = 995,000 × 0.005 = 4,975

With flash loan:
Withdrawal fee = 1,000,000 × 0.001 = 1,000
Flash loan fee = 5,000 × 0.005 = 25
Total cost = 1,000 + 25 = 1,025
```

This results in a savings of `3,950` simply by temporarily inflating the balance.

Examples

contracts/USDiCoin.sol:L128-L134

```
// Initialize tiered fee structure
feeTiers.push(FeeTier(0, 100_000e6, 50, 100));
feeTiers.push(FeeTier(100_000e6 + 1, 999_999e6, 20, 50));
feeTiers.push(FeeTier(1_000_000e6, 4_999_999e6, 5, 10));
feeTiers.push(FeeTier(5_000_000e6, 9_999_999e6, 0, 5));
feeTiers.push(FeeTier(10_000_000e6, 199_999_999e6, 0, 2));
feeTiers.push(FeeTier(200_000_000e6, type(uint256).max, 2, 5));
```

Recommendation

We recommend reviewing the fee tier structure and introducing additional tiers with more gradual fee reductions. This reduces the financial incentive for flash loan exploits and improves the fairness of the fee mechanism across transaction sizes.

4.24 Unused Functions ✓ Fixed

Resolution
Addressed in commits `1ae396b8da0f15367703764e3071e8cdffd926a5` and `3c30ce7d4172d04bba9e4c8f4b7da5d1058c1ef9` by removing `_daysInMonth()` and `_isLeapYear()` functions.

Description

The contract includes a function `_daysInMonth()` to calculate the number of days in a given month, but this function is never used. Similarly, the `_isLeapYear()` function is only used once—within `_daysInMonth()`—which itself is unused, rendering both functions effectively redundant in the current implementation.

However, there is a part of the code where a leap year check is manually performed instead of using the existing `_isLeapYear()` helper function. Despite having a dedicated utility function for this purpose, the code duplicates the leap year logic instead of reusing the helper.

See the relevant section of code:

contracts/USDiCoin.sol:L352-L354

```
if (iMonth > 2 && ((iYear % 4 == 0 && (iYear % 100 != 0)) || (iYear % 400 == 0))) {
    _days += 1;
}
```

We recommend calling `_isLeapYear(iYear)` directly in that context to improve readability and maintainability.

Recommendation

We recommend removing the unused `_daysInMonth()` function and replacing manual leap year logic with a call to `_isLeapYear()` where applicable, promoting cleaner and more maintainable code.

4.25 Function Order Is Incorrect ✓ Fixed

Resolution
In the `1ae396b8da0f15367703764e3071e8cdffd926a5` commit for fix review the problem has been fixed.

Description

The contract utilizes an unconventional function ordering. For example, the internal `_toString()` function is placed before the contract's constructor. The rest of the functions are ordered inconsistently, mixing internal, public, and external functions. Additionally, internal functions are scattered throughout the contract rather than grouped together.

Recommendation

We recommend following the official Solidity Style Guide.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
contracts/USDiCoin.sol	`7b5f8c95b240bc4d6572c2f968753a851b6b8db0`

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.