CASHIVA

Cashiva Audit Report For WMT Prime Corp

Revision Date: 2025-06-09

Table of Contents

1. Executive Summary

1.1. Project Introduction

This report details the findings of a smart contract security audit performed by Cashiva Community LLC on the Cashiva Standard Tokens (CST) contracts, provided by WMT Prime Corp. Cashiva Standard Tokens are asset-backed cryptocurrencies built on audited ERC20 smart contracts, featuring upgradeability and a unique mechanism that writes event in a real-time exchange rate at the moment of transfer through a trusted oracle - ensuring each token is transparently backed by crypto and denominated in fiat.

1.2. Audit Objectives & Scope

The primary objective of this audit was to identify potential security vulnerabilities, design flaws, and deviations from best practices within the provided Solidity smart contract code, designed to manage the token on-chain. The scope was limited to the smart contracts associated with the commit hash `*f44786df418b66f702ee1fdbdc45b3c3805533a3*`. The audit does not cover verification of the off-chain asset backing or custodian legal structures.

1.3. Overall Security Assessment

The security audit of the CashivaMintableToken and CashivaNativeWrappedToken contracts reveals a critically concerning security posture. The sheer volume and severity of the identified vulnerabilities, particularly the six distinct "Critical" findings (CST-C-001 through CST-C-003), one "High" finding CST-H-001 and one "Medium" severity finding CST-M-001, indicate fundamental flaws in the current design and implementation.

These critical issues span multiple categories, including several instances of reentrancy vulnerabilities across different contract functionalities (CST-C-001, CST-C-002), which could lead directly to fund theft. Storage location shadowing (CST-C-003) presents a significant risk to data integrity and contract stability, especially during potential upgrades or maintenance. The presence of an arbitrary ETH transfer mechanism (CST-H-001) further expose the contracts to severe financial exploitation and manipulation.

A core characteristic exacerbating these issues is the extreme concentration of power in privileged roles (CST-M-001). The initial granting of `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE`, and `BURNER_ROLE` to a single `_msgSender()` during initialization, coupled with `onlyOwner` administrative functions, means the entire system's integrity and user assets are highly dependent on the security and trustworthiness of one or very few entities. This introduces significant centralization risks.

While the contracts may utilize elements from established libraries (as suggested by naming conventions like `ERC20Upgradeable` and initialization patterns), the specific implementation within the Cashiva Standard Tokens contracts has introduced these severe flaws. The conclusion of the report is unequivocal: the contracts should not be deployed to mainnet until all critical and high severity issues are comprehensively addressed.

Prioritized remediation of all identified vulnerabilities, especially the critical ones related to reentrancy, storage integrity, fund control, oracle safety, and role management, is essential. Following remediation, a thorough follow-up audit is mandatory to ensure the fixes are correctly implemented and do not introduce new vulnerabilities, thereby aiming to establish the long-term security, reliability, and operational safety of the Cashiva Standard Tokens system.

1.4. Summary of Findings

ID	Title	Severity	Status
CST-C-001	Reentrancy in Withdrawal Flow	Critical	Fixed
CST-C-002	Reentrancy in Unwrap Function	Critical	Fixed
CST-C-003	Storage Location Shadowing	Critical	Fixed
CST-H-001	Arbitrary ETH Transfer	High	Acknowledged
CST-M-001	Centralization Risks in Administrative Functions	Medium	Acknowledged

2. Disclaimer

This audit report is provided for informational purposes only and is based on the code provided to Cashiva Community LLC at a specific point in time (commit hash `*f44786df418b66f702ee1fdbdc45b3c3805533a3*`). Smart contract security is a complex and evolving field; an audit does not guarantee the absence of all vulnerabilities.

Crucially, the scope of this report is limited to the smart contract code; it does not constitute a financial audit or verification of any off-chain assets, processes, or legal claims referenced in project documentation.

Cashiva Community LLC makes no warranties, express or implied, regarding the complete security of the audited code or its fitness for any particular purpose. The client is solely responsible for the deployment, maintenance, and operation of the smart contracts, and for the veracity and execution of any off-chain procedures. This report should not be considered investment advice.

3. About Cashiva Community LLC

Cashiva Community LLC is a company that specializes in blockchain technologies, DeFi development, and smart contract auditing. We are dedicated to fostering a more secure and reliable Web3 ecosystem by providing comprehensive security assessments and development expertise.

Our team is composed of seasoned blockchain professionals, security researchers, and smart contract engineers with deep expertise across a range of domains, including:

- **Smart Contract Auditing:** In-depth analysis of Solidity and other smart contract languages to identify vulnerabilities, logic flaws, and gas optimization opportunities.
- **DeFi Protocol Development & Security:** Extensive experience in designing, building, and securing complex decentralized finance applications, including lending platforms, DEXs, yield farming protocols, and more.
- **Blockchain Technology & Architecture:** Profound understanding of core blockchain principles, consensus mechanisms, cryptographic primitives, and various Layer 1 and Layer 2 solutions.
- **Exploitation Techniques & Mitigation Strategies:** Up-to-date knowledge of common and novel attack vectors targeting smart contracts and blockchain systems, and best practices for their prevention.

- **Security Best Practices & Standards:** Adherence to industry-leading security guidelines and development standards.

We employ a meticulous audit methodology that combines manual line-by-line code review, automated static and dynamic analysis tools, and conceptual logic evaluation to deliver thorough and actionable security reports. Our goal is to empower our clients with the insights and recommendations needed to build and deploy secure, robust, and innovative blockchain solutions.

Our Commitment:

- **Technical Excellence:** We pride ourselves on the depth and rigor of our technical analysis.
- **Actionable Insights:** Our reports are designed to be clear, concise, and provide practical recommendations.
- **Collaborative Partnership:** We work closely with our clients, fostering open communication throughout the audit and development lifecycle.
- **Upholding Security Standards:** We are committed to contributing to the overall security and integrity of the blockchain space.

For more information about Cashiva Community LLC, our services, and our contributions to the DeFi and blockchain ecosystem, please visit www.cashiva.com or contact us at crypto@cashiva.com.

4. Project Overview

4.1. Stated Purpose

According to project documentation provided by WMT Prime Corp, the Cashiva Standart Tokens are asset-backed cryptocurrency tokens operating on the Ethereum blockchain, with a unique feature of dynamic rate fixation.

Cashiva Standard Token (CST) are cryptocurrency-backed tokens operating on the Ethereum blockchain, featuring a unique dynamic exchange rate fixation. Each token is issued according to a specific currency pair, where the token value is denominated in fiat currency and collateralized by cryptocurrency (Bitcoin, Ethereum, etc.). With every transaction, the exchange rate is fixed via a trusted oracle, ensuring transparent and up-to-date determination of the asset's real value. The main advantage of CST tokens is the combination of fiat currency stability with cryptocurrency liquidity and potential, making them ideal for decentralized financial platforms (DeFi), lending services, derivatives trading, and flexible cross-chain transfers. Thanks to the rate-fixation mechanism, users can confidently rely on accurate value determination during transfers, while the automated token issuance and redemption system ensures a high level of security and transaction efficiency.

IMPORTANT NOTE ON AUDIT SCOPE:

This audit focuses **exclusively** on the technical implementation and security of the Solidity smart contract code, which manages the token on the blockchain.

The audit **DOES NOT** include verification of, and provides **NO** opinion on:

- The existence, quantity, quality, or audits of the cryptocurrency collateral backing the CST tokens.
- The security, insurance, or operational procedures of any wallets, vaults, or custodial infrastructure holding the collateral assets (e.g., Bitcoin, Ethereum).
- The validity, enforceability, or legal structure of any claims to fiat-denominated value or the mechanism of dynamic exchange rate fixation.

- Any off-chain token issuance or redemption processes, oracle reliability, or collateral reserve management procedures.

The following sections describe the technical architecture of the smart contract system reviewed.

4.2. Technical System Architecture

The Cashiva token system consists of two main token types, each designed for specific use cases:

- **`CashivaNativeWrappedToken`**: A token designed for wrapping native blockchain assets (like ETH) with the following inheritance structure:
    - ➢ `` `CashivaStandardToken` ``: Base token implementation with price feed integration
    - ➢ `` `NativeWrappable` ``: Handles wrapping/unwrapping of native blockchain assets
    - ➢ `` `UUPSUpgradeable` ``: Implements the Universal Upgradeable Proxy Standard
- **`CashivaMintableToken`**: A token designed for wrapped off-chain assets (like BTC) with the following inheritance structure:
    - ➢ `` `CashivaStandardToken` ``: Base token implementation with price feed integration
    - ➢ `` `Mintable` ``: Provides controlled minting capabilities with role-based access
    - ➢ `` `PausableUpgradeable` ``: Allows pausing all token operations in emergencies
    - ➢ `` `Freezable` ``: Enables freezing individual accounts
    - ➢ `` `UUPSUpgradeable` ``: Implements the Universal Upgradeable Proxy Standard

Both token types inherit from **`CashivaStandardToken`** which provides:

- `` `ERC20Upgradeable` ``: Standard ERC20 token functionality
- `` `TransferPriceEmitter` ``: Price feed integration using Pyth-like oracles
- `` `OwnableUpgradeable` ``: Basic access control
- `` `ERC20PermitUpgradeable` ``: Gas-less approval mechanism (EIP-2612)

4.3. Key Components

- `CashivaNativeWrappedToken` Components:
    - ➢ Native asset wrapping/unwrapping functionality
    - ➢ Configurable wrap fees with min limit
    - ➢ Price feed integration for transfer price tracking
    - ➢ Fee collection mechanism for wrapped assets
- `CashivaMintableToken` Components:
    - ➢ Role-based minting control (`MINTER_ROLE`, `BURNER_ROLE`)
    - ➢ Account freezing capability for compliance
    - ➢ Pausable operations for emergency response

  - Configurable decimals for different asset types
  - Price feed integration for transfer price tracking
- Shared Infrastructure:
  - `TransferPriceEmitter`: Emits price data during transfers using Pyth-like oracle
  - `WrapFee`: Implements configurable fee mechanisms
  - `Freezable`: Provides account-level transfer restrictions
  - Storage pattern using ERC-7201 namespaced storage slot

The system uses a modular design with clear separation of concerns, allowing for flexible token implementations while maintaining security and upgradeability. Each component is designed to be independently upgradeable through the UUPS proxy pattern, with careful consideration for storage layouts to prevent collisions.

5. Audit Scope & Methodology

Commit Hash: *`f44786df418b66f702ee1fdbdc45b3c3805533a3`*

This audit covered the security and functionality of two primary smart contracts, **`CashivaMintableToken`** and **`CashivaNativeWrappedToken`**, along with their inherited contracts (`CashivaStandardToken`, `Mintable`, `Freezable`, `NativeWrappable`, `TransferPriceEmitter`).

Key focus areas included:

- **Smart Contract Security:** Identifying common vulnerabilities, ensuring secure native token wrapping/unwrapping, minting controls, access management, and the security of freezing/pause mechanisms.
- **Token Standards & Upgrades:** Verifying ERC20 adherence, correctness of UUPS upgradeability, and safe storage layout.
- **Custom Features:** Reviewing Pyth-like Oracle price feed integration, fee mechanisms for wrapping/unwrapping, native token handling, and controls for minting, burning, and freezing.
- **Implementation Quality:** Assessing gas optimization, Solidity best practices, event emission, and access controls.
- **Integrations:** Checking the security of the Pyth-like Oracle integration, native token handling safety, and fee calculation accuracy.

Out of scope items were:

- Off-chain components (e.g., client applications, external oracle data quality).
- Business logic (e.g., economic model, tokenomics, business strategy).
- Internal workings of external systems like the Pyth-like Oracle (beyond direct interactions).

5.1. Methodology

The audit was conducted using a combination of manual code review and conceptual analysis:

- **Understanding the Codebase:** Initial review to understand the architecture, intended on-chain functionality, and control flow.
- **Systematic Manual Review:** Line-by-line examination of the smart contracts to identify potential vulnerabilities based on known attack vectors and best practices.
- **Vulnerability Analysis:** Cross-referencing potential issues with common vulnerability checklists (e.g., SWC Registry).
- **Business Logic Review:** Ensuring the implemented on-chain logic aligns with the inferred intentions of the custom modules.
- **Access Control Analysis:** Verifying that sensitive functions are appropriately protected.
- **Upgradeability Review:** Assessing the UUPS implementation and custom storage slot management.
- **Reporting:** Documenting findings with severity, impact, and recommendations.

5.2. Severity Level Definitions

Critical: Vulnerabilities that could lead to a loss of funds, data manipulation, contract unavailability, or a takeover of contract ownership. Also includes flaws that make the contract highly unstable or prone to severe malfunction during routine maintenance, such as dependency upgrades.

High: Vulnerabilities that could lead to unexpected behavior, minor fund loss, or significantly hinder contract functionality, but are not as easily exploitable as Critical. (Note: No High severity findings were identified in this illustrative report, but the definition is retained for completeness).

Medium: Vulnerabilities that represent a deviation from best practices, could lead to inefficiencies, or have a security impact. The **likelihood** of exploitation might be lower for some medium issues, but the **potential impact** could still be significant to severe, warranting prioritized attention.

Low: Minor issues, such as gas optimizations or code style suggestions, that do not pose a direct security threat.

Informational: Observations, suggestions for code clarity, or comments that do not directly impact security but could improve maintainability or understanding.

Resolved: The finding has been addressed by the client.

Acknowledged: The client has acknowledged the finding but may not fix it due to specific reasons.

Unresolved: The finding has not yet been addressed.

6. Findings

6.1. Critical Severity Findings

6.1.1. CST-C-001: Reentrancy in Withdrawal Flow (CashivaMintableToken)

Status: Fixed

Justification for Prioritized Attention: This finding is classified as Critical because it could allow an attacker to re-enter the `` `requestWithdrawal` `` function before critical state updates are complete, potentially leading to multiple withdrawals for the same amount or draining funds from the contract. Immediate attention is required.

Description: The `requestWithdrawal` function in `CashivaMintableToken` performs an external call (`_transfer`) before all state variables related to the withdrawal request (like incrementing `_requestId`) are fully updated. This violates the checks-effects-interactions pattern.

```
function requestWithdrawal(string memory recipient, uint256 amount) public
virtual returns (WithdrawRequest memory) {
    require(amount > 0, "Amount must be greater than 0");
    require(balanceOf(_msgSender()) >= amount, "Insufficient balance");

    MintableStorage storage $ = _getMintableStorage();
    $._withdrawRequests[$._requestId].status = WithdrawRequestStatus.Pending;
    // ... state changes ...

    _transfer(_msgSender(), address(this), amount); // External call before state
update completion

    uint256 fee = _calcWrapFee(amount);
    emit WithdrawalRequestCreated(_msgSender(), $._requestId, recipient, amount,
fee);
    return $._withdrawRequests[$._requestId++];
}
```

Impact:

1. **Fund Theft:** An attacker could craft a malicious contract to re-enter the function after the `_transfer` but before `_requestId` is incremented, effectively requesting the same withdrawal multiple times or exploiting other state inconsistencies.

2. **State Corruption:** Reentrancy can lead to inconsistent state within the contract regarding withdrawal requests and balances.

Recommendations:

1. Implement the checks-effects-interactions pattern and use a reentrancy guard. Increment `_requestId` and update request details before the external `_transfer` call.

```
function requestWithdrawal(string memory recipient, uint256 amount) public
virtual nonReentrant returns (WithdrawRequest memory) { // Added nonReentrant
    require(amount > 0, "Amount must be greater than 0");
    require(balanceOf(_msgSender()) >= amount, "Insufficient balance");

    MintableStorage storage $ = _getMintableStorage();
    uint256 currentRequestId = $._requestId; // Read before increment
    $._requestId++; // Effect: Increment first

    // Effects: Update request details
    $._withdrawRequests[currentRequestId].status = WithdrawRequestStatus.Pending;
    $._withdrawRequests[currentRequestId].account = _msgSender();
    $._withdrawRequests[currentRequestId].recipient = recipient;
    $._withdrawRequests[currentRequestId].amount = amount;
    // Interaction: External call last after all state changes for this specific
request creation
    _transfer(_msgSender(), address(this), amount);

    uint256 fee = _calcWrapFee(amount); // Can be calculated earlier if not
dependent on post-transfer state
    emit WithdrawalRequestCreated(_msgSender(), currentRequestId, recipient,
amount, fee);
    return $._withdrawRequests[currentRequestId];
}
```

6.1.2. CST-C-002: Reentrancy in Unwrap Function (CashivaNativeWrappedToken)

Status: Fixed

Justification for Prioritized Attention: This finding is classified as Critical because the external call `payable(recipient).transfer(amount)` occurs before the token burning (`_burn`) operation. A malicious recipient contract could re-enter and potentially trigger multiple ETH transfers for a single unwrap operation, leading to fund loss. Immediate attention is required.

Description: The _unwrap function in `CashivaNativeWrappedToken` sends ETH to the recipient before burning the corresponding wrapped tokens. This is a classic reentrancy pattern.

```
function _unwrap(address recipient, uint256 value) internal virtual {
    uint256 fee = _calcWrapFee(value);
    uint256 amount = value - fee;
    payable(recipient).transfer(amount);  // External call before state update
    emit Unwrap(recipient, value);
    _burn(address(this), value);
}
```

Impact:

1. **Fund Theft:** A malicious recipient contract could re-enter the unwrap process (or another interacting function) after receiving ETH but before its wrapped tokens are burned, allowing it to claim ETH multiple times for the same tokens.

2. **State Inconsistency:** The total supply of wrapped tokens might not accurately reflect the amount of underlying native currency held by the contract if reentrancy occurs.

Recommendations:

2. Apply the checks-effects-interactions pattern: burn tokens (_burn - effect) *before* transferring ETH (`payable(recipient).transfer` - interaction). Add a `nonReentrant` modifier.

```
function _unwrap(address recipient, uint256 value) internal virtual nonReentrant
{ // Added nonReentrant
    uint256 fee = _calcWrapFee(value);
    uint256 amountToTransfer = value - fee;

    // Effect: Burn tokens first
    _burn(address(this), value); // Assuming value is the amount to burn (gross)
                                // If only amountToTransfer should be burned from
recipient, adjust logic
```

```
                                // and ensure contract balance is sufficient for
this.
                                // The original report implies
_burn(address(this), value) which burns from contract's own wrapped token
balance.

    emit Unwrap(recipient, value); // Event before interaction is acceptable if
it reflects intent prior to external call

    // Interaction: Transfer ETH last
    payable(recipient).transfer(amountToTransfer);
}
```

6.1.3. CST-C-003: Storage Location Shadowing (CashivaMintableToken & Dependencies)

Status: Fixed

Justification for Prioritized Attention: This finding is classified as Critical because using identical constant names (`` `STORAGE_LOCATION` ``) for different storage slot pointers across an inheritance hierarchy can lead to severe storage collisions. This can result in one contract's storage overwriting another's, leading to unpredictable behavior, data corruption, and potential loss of funds. Immediate attention is required for contract stability.

Description: Multiple contracts in the inheritance chain (`` `CashivaMintableToken` ``, `` `Freezable` ``, `` `Mintable` ``, `` `WrapFee` ``, `` `TransferPriceEmitter` ``) define a `bytes32 private constant` `` `STORAGE_LOCATION` `` with different values. This is highly dangerous when these contracts are inherited, as the diamond problem or simple linear inheritance can cause these to point to unintended storage slots if not managed with extreme care, especially with upgradeable contracts. While `private` limits direct collision, it's a very risky pattern that often indicates deeper issues with storage layout management in upgradeable contracts. The primary risk is in how storage pointers are derived or used based on these constants.

```
// In CashivaMintableToken.sol
bytes32 private constant STORAGE_LOCATION =
0x9b9e7ab05886f036ccbbe5f70f770ff36db154b0409040f4610d6927942ad500;

// In Freezable.sol
bytes32 private constant STORAGE_LOCATION =
0x98f5cbd3380b8191db24ff05e05a319c5f63cab76da3ae1bc25d634271302700;

// In Mintable.sol
bytes32 private constant STORAGE_LOCATION =
0xddb9e61613b3299de1a8214e91c696a267968494eb8f384023aadbd92496b700;

// In WrapFee.sol
bytes32 private constant STORAGE_LOCATION = /* value */;

// In TransferPriceEmitter.sol
bytes32 private constant STORAGE_LOCATION =
0xf116ee31fa11d5f3e9f2ed675718b59844fe1729415bbc6ccfe55c1ab01e2c00;
```

Impact:

1. **State Corruption:** Contract state variables can be overwritten by others, leading to incorrect balances, allowances, roles, or other critical data.

2. **Unpredictable Behavior:** Functions may read or write to the wrong storage slots, causing severe malfunctions.

3. **Upgrade Incompatibility:** This makes future upgrades extremely risky and prone to errors.

6.2. High Severity Findings

6.2.1. CST-H-001: Arbitrary ETH Transfer (CashivaNativeWrappedToken)

Status: Acknowledged

Justification for Prioritized Attention: This finding is classified as High because the `_withdrawFee` function transfers the contract's withdrawable fee balance to `_msgSender()`. If this internal function can be called by an unauthorized party or by an authorized party at an inappropriate time (e.g., via a reentrancy or logic flaw elsewhere), it could lead to theft of accumulated fees.

Description: The `_withdrawFee` function in `CashivaNativeWrappedToken` (likely inherited or part of its structure) directly transfers fees to `_msgSender()`. The risk depends on how and by whom this internal function can be invoked. If `_msgSender()` within the context of this call can be an arbitrary user or a less privileged role than intended for fee collection, it's a vulnerability.

```
function _withdrawFee() internal virtual {
    payable(_msgSender()).transfer(withdrawableFee());
}
```

Impact:

1. **Fee Theft:** Accumulated fees in the contract could be drained by an unauthorized address if it can trigger this function directly or indirectly.

Privilege Escalation: A lower-privileged role might be able to call this function if access control is improperly implemented on a public/external function that internally calls `_withdrawFee`.

6.3. Medium Severity Findings

6.3.1. CST-M-001: Centralization Risks in Owner-Controlled Privileged Functions (CashivaMintableToken)

Status: Acknowledged

Justification for Prioritized Attention: This finding is classified as Medium as it represents a deviation from best practices regarding decentralized governance. Several critical administrative and fund management functions, including `pause()`, `unpause()`, `setWrapFeeParams()` and `burnFrozenFunds()`, are controlled by a single onlyOwner. While some level of administrative control is often necessary, reliance on a single owner account for such potent functions (including the ability to burn user funds from frozen accounts) introduces a significant single point of failure and potential for abuse or error.
Description:
Administrative and potent fund management functions are restricted to `onlyOwner`. These include:
Pausing/unpausing the contract (`pause()`, `unpause()`).
Setting fee parameters (`setWrapFeeParams()`).

Burning all tokens from an account that has been marked as frozen (`burnFrozenFunds()`). The security of `burnFrozenFunds` also depends heavily on the legitimacy and control over the freezing mechanism (`onlyFrozen(account)`).

```
// In CashivaMintableToken
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}

function burnFrozenFunds(address account) external onlyOwner onlyFrozen(account){
    uint256 balance = balanceOf(account);
    _burn(account, balance);
    emit FrozenFundsBurned(account, balance);
}

// In Both Contracts (via inheritance)
function setWrapFeeParams(uint256 feeRate_, uint256 minFee_, uint256 maxFee_)
public onlyOwner {
    _setWrapFeeParams(feeRate_, minFee_, maxFee_);
}
```

Impact:

- Single Point of Failure: If the owner's private key is compromised, malicious actors could take devastating actions, such as pausing the contract indefinitely, setting exorbitant fees, or unilaterally burning funds from frozen accounts.
- Risk of Abuse or Error: A malicious or careless owner could misuse these privileges. This could involve:
- Implementing unfair fee structures.
- Unjustifiably pausing the contract, disrupting operations.
- Potentially abusing the freeze-and-burn mechanism: if the freezing mechanism itself is owner-controlled without sufficient safeguards, an owner could maliciously freeze an account and then burn its funds.
- Irreversible Loss for Users: In the case of `burnFrozenFunds`, if funds are burned mistakenly or maliciously, they are permanently lost to the user.
- Reduced Trust & Centralization Concerns: Users may be wary of systems where a single entity wields significant administrative and potentially destructive power over the contract's state and user assets. This concentration of control deviates from decentralized principles.

6.4. Low Severity Findings

No Low severity findings were identified during this audit.

7. Privileged Roles & Access Control Analysis

The Cashiva token contracts implement a multi-layered access control system combining OpenZeppelin's `` `AccessControlUpgradeable` `` and `` `OwnableUpgradeable` `` patterns. This creates a more granular but still centralized permission structure.

Role Structure

- **Owner Role** (`OwnableUpgradeable`):
  - ➢ Operational Control: Pause/Unpause token operations (`` `pause()` ``, `` `unpause()` ``)
  - ➢ Account Management: Freeze/Unfreeze accounts (`` `freeze()` ``, `` `unfreeze()` ``), burn frozen funds (`` `burnFrozenFunds()` ``)
  - ➢ Fee Management: Set wrap fee parameters (`` `setWrapFeeParams()` ``)
  - ➢ Contract Governance: Authorize upgrades (`` `_authorizeUpgrade()` ``)
- **Minter Role** (`MINTER_ROLE`):
  - ➢ Token Creation: Process deposits and mint new tokens (`` `deposit()` ``)
  - ➢ Restricted to authorized addresses through `` `AccessControl` ``
- **Burner Role** (BURNER_ROLE):
  - ➢ Withdrawal Management: Complete or cancel withdrawal requests (`` `completeWithdrawal()` ``, `` `cancelWithdrawal()` ``)
  - ➢ Restricted to authorized addresses through `` `AccessControl` ``

Key Differences Between Token Types:

- **CashivaMintableToken**:
  - ➢ Implements full role-based access control with `` `MINTER_ROLE` `` and `` `BURNER_ROLE` ``
  - ➢ Includes freezing and pausing capabilities
  - ➢ Supports manual minting and burning through controlled roles
- **CashivaNativeWrappedToken**:
  - ➢ Focuses on wrapping native tokens (ETH)
  - ➢ Simpler permission model primarily using `` `onlyOwner` ``
  - ➢ Includes fee collection and withdrawal mechanisms
  - ➢ Automatic minting/burning based on wrap/unwrap operations

Security Implications:

- **Centralization Risks**:
  - ➢ Both contracts concentrate significant power in the owner role
  - ➢ The owner can unilaterally freeze accounts and burn their funds
  - ➢ No time-delay or multi-signature requirements for critical operations
  - ➢ Owner can modify fee parameters without limits or delays
- **Role Management Vulnerabilities**:
  - ➢ Initial setup grants all roles to the deployer

   - ➢ No separation of duties between role administration and execution
   - ➢ No restrictions on role combinations (same address can have multiple roles)

- **Operational Risks**:
   - ➢ No emergency role recovery mechanism if owner key is compromised
   - ➢ Lack of time-locks for sensitive operations
   - ➢ No limits on consecutive operations or value thresholds

The `CashivaStandardToken` contract utilizes `OwnableUpgradeable`, granting extensive and critical capabilities to a single `owner` address. This `owner` role is the sole administrative entity within the smart contract system.

Owner Privileges (Summary):

- **Operational Control:** Pause/Unpause all key token operations (`pause()`, `unpause()`).
- **Account Management:** Freeze/Unfreeze individual accounts (`freeze()`, `unfreeze()`), and burn all tokens from a frozen account (`burnFrozenFunds()`).
- **Token Supply Control:** Create new tokens (`mint()`), and burn tokens from the owner's balance (`redeem()`).
- **Economic Policy:** Set transfer fee rates, maximum fees, and the fee recipient (`setFeeParams()`, `setFeeRecipient()`).
- **Contract Governance:** Authorize upgrades to a new implementation logic for the entire contract (`_authorizeUpgrade()`).
- **Ownership Transfer**: Manage the transfer of the `owner` role itself (`transferOwnership()`, `acceptOwnership()`).

Security Implications:

The concentration of such sweeping powers into a single `owner` role means that **the security and integrity of the entire Cashiva Standard Tokens system fundamentally depends on the security and trustworthiness of the entity controlling this single `owner` address**. If this address is compromised or acts maliciously, there are no on-chain mechanisms within the contract to prevent misuse of these powers. The subsequent "Centralization Risks & Owner Capabilities" section discusses the implications of these extensive powers in more detail.

Recommendations:

1. **Multisig Wallet for Owner Role (Strongly Recommended):** Replace single-signature ownership of the owner address with a multisignature (multisig) wallet (e.g., Gnosis Safe). This significantly reduces the risk of a single point of failure by requiring multiple trusted parties to approve critical actions. A multisig setup distributes control and enhances governance security, mitigating risks from key compromise, mistakes, or malicious insiders

2. **Timelock Mechanism:** Implement an additional timelock contract through which all `owner` actions must pass. A timelock introduces a mandatory delay between the proposal of an action and its execution. This provides transparency and a window for the community and users to review, discuss, and potentially react (e.g., by exiting positions if possible) to significant upcoming changes, providing a crucial check against immediate, unilateral actions by the `owner`.

8. Centralization Risks & Owner Capabilities

The Cashiva token contracts (`CashivaMintableToken` and `CashivaNativeWrappedToken`) implement a role-based access control system with significant powers vested in privileged roles. This section details the centralization risks and capabilities of these privileged roles.

8.1. Privileged Role Capabilities:

- **Owner**
  - Fee Management (Both contracts):
    - `setWrapFeeParams`: Sets wrap/unwrap fee parameters (rate capped at 100%).
    - `withdrawFee`: Withdraws accumulated fees.
  - Oracle Management (Both contracts):
    - `setOracle`: Updates the price oracle.
    - `setValidTimePeriod`: Sets valid price data time window.
  - Account Control (`CashivaMintableToken`):
    - `freeze` / `unfreeze`: Freezes or unfreezes user accounts.
    - `burnFrozenFunds`: Burns tokens of frozen accounts.
  - Upgrades (Both contracts):
    - `_authorizeUpgrade`: Authorizes contract upgrades (UUPS).
  - Operations (`CashivaMintableToken`):
    - `pause` / `unpause`: Pauses/resumes all token functions.
- **Minter (`CashivaMintableToken` - MINTER_ROLE)**
  - `deposit`: Mints tokens via deposit process.
- **Burner (`CashivaMintableToken` - BURNER_ROLE)**
  - Manages token burning by completing or canceling withdrawal requests.

8.2. Manifestation of Centralization Risks:

The existence of these unchecked (by the contract itself) powers leads to significant risks:

- **Single Point of Failure for Security:** Compromise of the single `owner` key grants the attacker complete and immediate control over all aspects of the token and its ecosystem.
- **Potential for Malicious Owner Action:** A rogue or compromised owner can:
  - Mint tokens to themselves and drain liquidity.
  - Set fees to 100% and redirect them.
  - Freeze legitimate user accounts and burn their funds without recourse.
  - Upgrade the contract to a malicious version.
- **Operational Error by Owner:** A mistake by the entity controlling the `owner` key when exercising these powerful functions can have widespread, irreversible negative consequences.

- **Target for Coercion/Regulatory Pressure:** The entity controlling the `owner` address becomes a central point of failure and a prime target for external pressure to misuse these powers against users or the system.
- **Undermining Trust and Decentralization:** For users who expect a high degree of decentralization and censorship resistance from a blockchain-based token, such extensive and unilateral owner powers are a fundamental concern, as the token's behavior and user assets are entirely subject to the owner's discretion and security.

9. Conclusion

The Cashiva contracts suite (`CashivaMintableToken` and `CashivaNativeWrappedToken`) demonstrates the use of OpenZeppelin's upgradeable contracts for its on-chain implementation. The audit identified three **Critical** severity findings, one **High** severity finding and one **Medium** severity finding that require immediate attention.

Notably, the multiple reentrancy vulnerabilities (CST-C-001) in `requestWithdrawal` and in _unwrap (CST-C-002) - require urgent attention due to their potential to enable direct fund theft and corrupt contract state. Another critical issue (CST-C-003) involves dangerous storage location shadowing across inherited contracts, which could result in unpredictable behavior and data corruption. The high-severity issue (CST-H-001) exposes a risk of arbitrary ETH transfers via `_withdrawFee`, potentially allowing fee theft. The medium-severity issue (CST-M-001) highlights significant centralization risks tied to owner-controlled privileged functions such as `burnFrozenFunds`.

A defining characteristic of this smart contract system is the absolute and unilateral power vested in the owner role, alongside significant capabilities granted to Minter and Burner roles. This owner role has complete control over critical aspects such as fee management (`setWrapFeeParams`, `withdrawFee`), oracle management (`setOracle`), account control (`freeze`, `unfreeze`, `burnFrozenFunds`), operational control (`pause`, `unpause`), and contract governance via upgrades (`_authorizeUpgrade`), as detailed in the "Privileged Roles & Access Control Analysis" and "Centralization Risks & Owner Capabilities" sections.

Consequently, the integrity of the entire system and the security of user assets are wholly dependent on the singular owner address (and other privileged role addresses) remaining secure and acting benevolently. This presents a critical centralization risk that users must be aware of.

Implementing external safeguards such as multisignature wallets for privileged roles (especially the owner) and timelock mechanisms for sensitive operations is strongly recommended as an immediate measure. For the long-term health and trustworthiness of the token, the Cashiva team should consider redesigning aspects of the contract in future iterations to incorporate on-chain constraints on owner powers, enforce stricter separation of duties, and potentially distribute administrative controls more broadly.

By addressing all identified technical findings, particularly the critical reentrancy and storage issues, and by transparently acknowledging and working to mitigate the profound centralization risks inherent in the current contract design, the Cashiva team can work towards enhancing the security, reliability, and trustworthiness of the Cashiva smart contracts.