Cashier System Contract Audit Report

Updated: January 06, 2026

Table of Contents

Introduction

The purpose of this report is to document the security analysis and contract structure of the `Cashier.sol` contract. This report delves deeper into the contract's mechanisms, potential risk mitigations, and design choices to provide a more comprehensive evaluation of its robustness, adherence to best practices, and overall integrity.

Disclaimer

This report is based on the information provided at the time of the audit and does not guarantee the absence of future vulnerabilities. Subsequent security reviews and on-chain monitoring are strongly recommended.

Scope of Audit

The audit focused on the following aspects of `Cashier.sol`:

- Security mechanisms, including access control, token handling, and financial invariants
- Code correctness, logical flow, and edge case handling
- Adherence to best practices, such as upgradeability and event logging
- Gas efficiency, mathematical precision, and upgrade safety

Methodology

The audit process involved:

- Manual code review
- Automated analysis using Slither and Aderyn for static checks
- Scenario-based testing using Foundry, including simulations of cashouts, role assignments, and upgrade paths

---

Security Review Summary

Review commit hash:

- `41d8fe1676de777373b611bda0b161835cfec477`

The `Cashier.sol` contract was reviewed in depth for security, emphasizing access control, token interaction safety, and economic model integrity. The contract employs robust practices from OpenZeppelin libraries, ensuring resilience against common vectors like reentrancy, overflow, and unauthorized access. Key strengths include precise arithmetic handling (e.g., basis points scaling) and comprehensive event logging for auditability. The design effectively balances functionality with security, though ongoing monitoring is advised for runtime behaviors.

Security Analysis

The `Cashier` contract functions as a centralized vault for token-to-USDT exchanges, incorporating tax and burn mechanisms to enforce economic policies.

Its security architecture leverages OpenZeppelin's `AccessControlUpgradeable` for fine-grained role management:

- `DEFAULT_ADMIN_ROLE` controls high-level configurations like the tax/burn rates via `setTaxRate` and `setBurnRate`,
- `LISTER_ROLE` handles token listings and rates (e.g., `registerToken`, `updateTokenRate`),
- `VAULT_ROLE` manages fund flows (e.g., `depositUSDT`, `withdrawUSDT`). This segregation minimizes privilege escalation risks, as no single role can unilaterally compromise the vault— for instance, a compromised `LISTER_ROLE` cannot withdraw funds.

Token interactions are fortified with `SafeERC20`, which wraps `IERC20` calls to handle non-compliant tokens like USDT, preventing silent failures in `safeTransfer` and `safeTransferFrom`. In the `cashout` function, critical checks precede actions: verifying token activity (`config.isActive`), rate validity (`config.rate > 0`), verification status (`config.isVerified`), and vault sufficiency (`usdt.balanceOf(address(this)) >= usdtAmount`).

Mathematical operations use safe scaling (e.g., `/ 1e18` for rates, `/ BASIS_POINTS` for percentages), avoiding precision loss in large numbers. The `recoverERC20` function excludes USDT recovery, preserving vault integrity. Upgradeability is secured via `Initializable` (disabling direct constructors) and a 48-slot `__gap`, allowing storage expansions without conflicts. Events like `Cashout`, `TokenRateUpdated`, and `VaultWithdrawn` enable real-time monitoring and post-mortem analysis. Potential risks, such as oracle-like rate updates or admin key compromise, are mitigated through role revocation capabilities.

Contract Structure

The contract is architected for modularity, extensibility, and readability, inheriting from `Initializable` and `AccessControlUpgradeable` for proxy compatibility.

- **State Variables**: Core variables include `usdt` (IERC20 for payout token), `taxRate` and `burnRate` (uint16 scaled to basis points for efficiency), `supportedTokens` (mapping address to `TokenConfig` struct containing `rate` (uint256, 1e18-scaled), `token` (IBurnable), `isActive`, and `isVerified`), role keccaks (`LISTER_ROLE`, `VAULT_ROLE`), and `VMCC_GOLD_PRLZ_COLLECTOR` for transfer routing. Constants like `BASIS_POINTS = 10_000` enhance clarity.

- **Initialization**: The `initialize` function sets `usdt`, grants roles to `admin`, and defaults `taxRate` and `burnRate` to 5000 (50%), ensuring atomic setup and preventing front-running during deployment.

- **Core Functions**:

- **Token Management**: `registerToken` initializes `TokenConfig` with activity and verification flags; `updateTokenRate` and `toggleCashout` allow dynamic adjustments, with requires for non-zero rates and registered tokens.
- **Cashout Logic**: `cashout` computes taxed/net amounts, burns/transfers tokens via `IBurnable` and `SafeERC20`, and pays out USDT, with payout address flexibility for user convenience.
- **Vault Operations**: `depositUSDT` and `withdrawUSDT` use safe transfers with balance checks, restricting to `VAULT_ROLE`.
- **Configuration**: Admin functions like `setTaxRate` cap at `BASIS_POINTS` to prevent over-taxing; `setVMCCgPRLZCollector` ensures non-zero addresses.
- **Recovery**: `recoverERC20` safely transfers non-USDT tokens, limited to admins.