WAGA Coffee Traceability System - Smart Contract Audit Report

Executive Summary

The WAGA Coffee Traceability System consists of four main contracts implementing an ERC1155-based coffee token system with Chainlink integration for inventory management and proof of reserves. The audit identified several critical vulnerabilities, multiple high-severity issues, and various optimization opportunities.

---

1. Security Vulnerabilities

CRITICAL Issues

C1: Missing Access Control in WAGAInventoryManager.requestInventoryVerification()

Severity: Critical
Location: WAGAInventoryManager.sol, line ~93
Description: The function uses `this.requestInventoryVerification()` in `performUpkeep()`, which makes an external call that bypasses the `onlyRole(INVENTORY_MANAGER_ROLE)` modifier.

```
// Line 93 - Vulnerable code
this.requestInventoryVerification(batchIds[i], 0, 0, "", "", "");
```

Impact: Anyone can manipulate inventory verification by calling `performUpkeep()` with crafted data.

Recommendation: Use internal function call or implement proper access control in `performUpkeep()`.

C2: Reentrancy in `WAGACoffeeRedemption.requestRedemption()`

Severity: Critical
Location: WAGACoffeeRedemption.sol, lines 74-104
Description: The function transfers tokens before updating state, creating a reentrancy vulnerability despite the ReentrancyGuard.

```
// Vulnerable pattern
```

```
coffeeToken.safeTransferFrom(msg.sender, address(this), batchId, quantity,
"");
// State updates happen after
```

Impact: Potential double-spending of redemptions.
Recommendation: Follow checks-effects-interactions pattern strictly.

HIGH Severity Issues

H1: Unchecked Assembly in `_parseResponse()`

Severity: High
Location: WAGAChainlinkFunctionsBase.sol, lines 44-53
Description: Assembly block doesn't validate response length properly before loading data.

```
assembly {
    result := mload(add(response, 32))
}
```

Impact: Potential memory corruption or incorrect data parsing.
Recommendation: Add proper bounds checking:

```
function _parseResponse(bytes memory response) internal pure returns
(uint256) {
    require(response.length >= 32, "Invalid response length");
    uint256 result;
    assembly {
        result := mload(add(response, 32))
    }
    return result;
}
```

H2: Front-running in Batch Creation

Severity: High
Location: WAGACoffeeToken.sol, `createBatch()`
Description: Batch IDs are predictable (`_nextBatchId++`), allowing front-running attacks.
Impact: Attackers can front-run batch creation to manipulate metadata or pricing.
Recommendation: Use commit-reveal scheme or add nonce-based randomness.

H3: Missing Validation in `updateRedemptionStatus()`

Severity: High
Location: WAGACoffeeRedemption.sol
Description: No validation that the caller has the authority to fulfill specific redemptions.
Impact: Any fulfiller can update any redemption status.
Recommendation: Add mapping of fulfillers to specific redemptions or regions.

MEDIUM Severity Issues

M1: Timestamp Dependence

Severity: Medium
Locations: Multiple contracts
Description: Heavy reliance on `block.timestamp` for critical business logic.
Impact: Miners can manipulate timestamps within ~15 seconds.
Recommendation: Use block numbers for critical timing or add tolerance margins.

M2: Missing Event Emission in Role Changes

Severity: Medium
Location: WAGACoffeeToken.sol, setInventoryManager() and setRedemptionContract()
Description: Role changes don't emit events.
Impact: Difficult to track permission changes off-chain.
Recommendation: Add events for all role modifications.

M3: Storage Collision Risk in Upgradeable Pattern

Severity: Medium
Location: All contracts using AccessControl
Description: No gap storage slots for future upgrades.
Impact: Storage collision if contracts are made upgradeable.
Recommendation: Add `uint256[50] private __gap;` in base contracts.

---

2. Business Logic Issues

B1: Incomplete Batch Lifecycle Management

Issue: No mechanism to handle partially fulfilled redemptions or batch recalls.
Impact: Cannot handle real-world scenarios like quality issues or logistics problems.
Recommendation: Implement batch recall and partial fulfillment mechanisms.

B2: Price Oracle Missing

Issue: `pricePerUnit` is set manually without oracle validation.
Impact: Price manipulation and lack of market price tracking.
Recommendation: Integrate Chainlink Price Feeds for coffee commodity prices.

B3: Metadata Verification Race Condition

Issue: Metadata can be verified before inventory verification completes.
Impact: Tokens might be minted for unverified inventory.
Recommendation: Enforce verification order: inventory → metadata → minting.

B4: Missing Slippage Protection

Issue: No maximum price protection during redemption.
Impact: Users might pay more than expected if prices change.
Recommendation: Add maximum price parameter to redemption requests.

---

3. Gas Optimization Opportunities

G1: Redundant Storage Reads

Location: WAGACoffeeToken.sol, multiple functions
Issue: `batchInfo[batchId]` read multiple times.
Gas Savings: ~2,100 gas per transaction
Fix:

```
BatchInfo memory info = batchInfo[batchId];
// Use info instead of multiple storage reads
```

G2: Inefficient Array Management

Location: WAGACoffeeToken.sol, getActiveBatchIds()
Issue: Double iteration over allBatchIds.
Gas Savings: ~50% reduction in gas cost
Fix: Use single-pass algorithm with dynamic array.

G3: Unnecessary External Calls

Location: WAGAInventoryManager.sol, performUpkeep()
Issue: Uses `this.requestInventoryVerification()` external call.
Gas Savings: ~2,500 gas per call
Fix: Use internal function.

G4: String Comparison Inefficiency

Location: WAGACoffeeToken.sol, `createBatch()`
Issue: Using keccak256 for string comparison.
Gas Savings: ~200 gas
Fix: Use bytes comparison for fixed-size strings.

---

4. Upgradeability Concerns

U1: No Upgrade Pattern Implementation

Issue: Contracts are not upgradeable but might need updates.
Impact: Cannot fix bugs or add features post-deployment.
Recommendation: Consider UUPS or Transparent Proxy pattern.

U2: Immutable Contract Dependencies

Issue: Contract addresses are set once without upgrade path.
Impact: Cannot update dependent contracts.
Recommendation: Implement registry pattern for contract addresses.

---

5. Code Quality Issues

Q1: Inconsistent Error Messages

Issue: Some requires have messages, others don't.
Recommendation: Standardize all error messages using custom errors (Solidity 0.8.4+).

Q2: Missing NatSpec Documentation

Issue: Many functions lack complete NatSpec.
Recommendation: Add `@notice`, `@param`, `@return` for all public/external functions.

Q3: Magic Numbers

Issue: Hardcoded values (7 days, 30 days, etc.).
Recommendation: Use named constants.

Q4: Commented Audit Notes in Production

Location: WAGAProofOfReserve.sol
Issue: Contains `@audit` comments.
Recommendation: Remove audit comments before deployment.

---

6. Standards Compliance

S1: ERC1155 Compliance

Status: Compliant
Note: Properly implements ERC1155 with extensions.

S2: AccessControl Implementation

Status: Mostly Compliant
Issue: Missing `supportsInterface` in some contracts.

---

7. Protocol-Specific Issues

P1: Chainlink Integration Security

Issue: No validation of Chainlink response data format.
Impact: Malformed responses could cause reverts or incorrect state.
Recommendation: Add response validation and error handling.

P2: Inventory Sync Race Conditions

Issue: Multiple parallel verification requests can cause state inconsistencies.
Impact: Inventory counts might be incorrect.
Recommendation: Implement request queuing or mutex pattern.

P3: Missing Batch Provenance

Issue: No way to track batch origin or supply chain.
Impact: Limited traceability despite being main goal.
Recommendation: Add origin tracking and supply chain events.

Severity Summary

- **Critical:** 2 issues
- **High:** 3 issues
- **Medium:** 3 issues
- **Low:** Multiple optimization and quality issues

Recommendations Priority

1. **Immediate (Before Deployment):**
    - Fix reentrancy in redemption contract
    - Fix access control in inventory manager
    - Add proper assembly validation
    - Implement front-running protection
2. **Short-term (Within 2 weeks):**
    - Add comprehensive event logging
    - Implement price oracle integration
    - Fix gas optimization issues
    - Complete NatSpec documentation
3. **Long-term (Within 1 month):**
    - Consider upgradeability pattern
    - Implement batch provenance tracking
    - Add slippage protection
    - Enhance Chainlink integration security

Testing Recommendations

1. **Security Testing:**
    - Reentrancy attack scenarios
    - Front-running simulations
    - Access control edge cases
    - Chainlink failure scenarios
2. **Integration Testing:**
    - Multi-contract interaction flows
    - Chainlink Functions responses
    - Gas consumption analysis
    - Load testing with multiple batches
3. **Business Logic Testing:**
    - Full redemption lifecycle
    - Batch expiry scenarios
    - Inventory verification edge cases

- Price update mechanisms

Conclusion

The WAGA Coffee Traceability System shows promise but contains several critical vulnerabilities that must be addressed before mainnet deployment. The architecture is generally sound, but security hardening and gas optimizations are essential. Special attention should be paid to access control, reentrancy protection, and Chainlink integration security.

Recommendation: Don't deploy to to mainnet until all critical and high-severity issues are resolved.

Audit conducted by: Antenhe Z. Tena
Date: Tuesday June 17th
Commit hash audited:
https://github.com/wagatoken/wagatoken/commit/66f21340616306aba0abc875a6f07021b1e54231
Solidity Version: 0.8.18
Framework: OpenZeppelin 4.x, Chainlink

Audited Contracts

Interfaces:

- IProofOfReserve.sol
- IRedemption.sol
- IWAGACoffeeToken.sol

Libraries:

- WAGAUpkeepLib.sol

Main Contracts:

- WAGAChainlinkFunctionsBase.sol
- WAGACoffeeRedemption.sol
- WAGACoffeeToken.sol
- WAGAInventoryManager.sol
- WAGAProofOfReserve.sol

Approach

This audit was conducted using a comprehensive multi-layered approach combining manual review, automated analysis, and security best practices. The methodology ensures thorough coverage of potential vulnerabilities, code quality issues, and business logic flaws.

1. Manual Code Review

- **Line-by-line analysis** of all smart contracts
- **Control flow analysis** to identify logic vulnerabilities
- **Access control verification** for all privileged functions
- **State machine validation** for complex workflows
- **Cross-contract interaction analysis** for integration risks

2. Security Pattern Analysis

- **Known vulnerability patterns** checked against SWC Registry
- **Common attack vectors** including reentrancy, overflow/underflow, front-running
- **Economic attack scenarios** including flash loans and price manipulation
- **Privilege escalation paths** and access control weaknesses
- **External dependency risks** particularly for Chainlink integrations

3. Business Logic Verification

- **Invariant analysis** to ensure protocol rules are maintained
- **Edge case identification** for boundary conditions
- **State transition validation** for all contract states
- **Economic model verification** for tokenomics integrity
- **Compliance verification** with ERC standards

4. Gas Optimization Analysis

- **Storage pattern optimization** opportunities
- **Computation efficiency** review
- **External call minimization** strategies
- **Event emission optimization** patterns

5. Best Practices Compliance

- **Solidity style guide** adherence
- **OpenZeppelin standards** compliance
- **Chainlink integration patterns** verification
- **Documentation completeness** assessment

Tools Used

Manual Review Tools

- **VSCode** with Solidity extensions
- **Remix IDE** for deployment simulation
- **Hardhat** for local testing environment

Reference Standards

- **SWC Registry** - Smart Contract Weakness Classification
- **OWASP Smart Contract Top 10**
- **ConsenSys Best Practices**
- **OpenZeppelin Security Guidelines**

Follow-up Recommendations

1. Static Analysis Tools

Immediate Implementation

Run the following static analysis tools before deployment:

Slither (by Trail of Bits)

```
slither . --print human-summary
slither . --print contract-summary
```

```
slither . --detect reentrancy-eth,reentrancy-no-eth,reentrancy-benign
```

Mythril (Security analysis)

```
myth analyze contracts/*.sol --execution-timeout 300
```

Solhint (Linter)

```
solhint 'contracts/**/*.sol' -f table
```

Configuration Files Needed

.solhint.json

json

```
{
  "extends": "solhint:recommended",
  "rules": {
    "compiler-version": ["error", "^0.8.18"],
    "func-visibility": ["error", {"ignoreConstructors": true}],
    "avoid-suicide": "error",
    "avoid-sha3": "warn",
    "avoid-tx-origin": "error",
    "check-send-result": "error"
  }
}
```

2. Automated Security Testing

Continuous Integration Pipeline

Implement automated security checks in CI/CD:

```yaml
# .github/workflows/security.yml
name: Security Analysis
on: [push, pull_request]
jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Slither
        uses: crytic/slither-action@v0.3.0
      - name: Run Mythril
        run: |
          docker run -v $(pwd):/tmp mythril/myth analyze /tmp/contracts/*.sol
```

Formal Verification

Consider formal verification for critical functions:

- Use **Certora Prover** for invariant checking
- Implement **SMTChecker** assertions in Solidity code
- Define formal specifications for core protocol properties

3. Testing Best Practices

Unit Testing Coverage

Achieve minimum 95% code coverage:

```
npx hardhat coverage
```

Fuzz Testing

Implement property-based testing:

(javascript)

```
// Example using Echidna
contract WAGACoffeeTokenEchidna is WAGACoffeeToken {
    function echidna_test_supply_invariant() public view returns (bool) {
        return totalSupply(0) <= maxSupply;
    }
}
```

Integration Testing

- Test all contract interactions
- Simulate Chainlink oracle responses
- Test upgrade scenarios if applicable