Smart Contract Audit Report

AdvancedERC20Token

February 4th, 2025

(Amended on April 22nd, 2025)

Index

Index 2
Summary 3
Audit Process & Delivery 3
Audited Files in scope 3
Client Documents/Resources 3
Intended Behavior 4
Role-Based Access Control (RBAC): 4
Token Features: 4
Gas Optimization: 5
Security and Flexibility: 5
Issues Found 6
Critical 6
Major 6
Minor 6
1. Inconsistency Between `totalSupply` Comment and Mint Functionality 6
2. Reentrancy Consideration in `GenericFactoryV1.clone` 7
Notes 8
1. Version Constraints with Known Issues: 8
2. Visibility Optimization for Gas Savings: 8
3. Test Coverage: 9
4. Use of Block Timestamps in Authorization Functions: 9
5. Gas Optimization through Smaller Data Types 10
6. Simplification of add and sub Functions: 10
Closing Summary 11
Disclaimer 12

Summary

Audit report was prepared by an independent auditor for the AdvancedERC20Token smart contracts. The audit aimed to identify vulnerabilities, optimize gas efficiency, and ensure compliance with Solidity best practices.

Audit Process & Delivery

Two (2) independent experts performed an unbiased and isolated audit of the code below. Client discussions and debriefs occurred between 6–10 January 2025, followed by the audit process from 10 January – 4 February 2025, with the final results presented here.

Audited Files in scope

- **GenericFactoryV1.sol**
- **AdvancedERC20.sol**

Client Documents/Resources

The client provided a GitHub repository as the primary resource for this audit. The audit was conducted on commit `c81f2da81df287cb328f8a82e730f30d2c62c351`

Intended Behavior

The following points outline the intended behavior of the AdvancedERC20 Token, derived from the project requirements. These behaviors reflect the principles and implementation details, ensuring the token achieves its goals of flexibility, security, and advanced functionality.

Role-Based Access Control (RBAC):

Modular Security Architecture: Provides a secure framework for managing token operations, ensuring only authorized roles can execute critical functions.
Role Management: Utilizes a role-based system to manage permissions for minting, burning, and transferring tokens, supporting multiple roles to maintain strict access control.

Token Features:

Mintable and Burnable: Allows for the creation and destruction of tokens, with the ability to revoke these features if necessary, providing flexibility in token supply management.
Voting Delegation: Facilitates decentralized governance by enabling token holders to delegate their voting power, promoting community-driven decision-making.
ERC-1363 Payable Token: Implements callback execution mechanisms for transfers and approvals, allowing smart contracts to automatically respond to token transactions, enhancing interoperability.

Gas Optimization:

Efficient Transactions: Implements strategies to minimize gas usage, enhancing transaction efficiency and reducing costs for users.
Transfer Methods: Offers both safe and unsafe transfer methods, allowing users to choose between security and gas optimization based on their needs.

Security and Flexibility:

Advanced Transaction Methods: Supports secure transaction methods with EIP-2612 and EIP-3009, enabling gasless approvals and transfers for improved user experience.
Access Control: Ensures restricted access to sensitive operations using role-based permissions, maintaining high security standards.
Adaptable Interactions: Allows for flexible token interactions while ensuring robust security, supporting a wide range of use cases from simple transfers to complex decentralized applications.

This design aims to provide a robust and versatile token solution, supporting advanced use cases such as decentralized governance, efficient token interactions, and enhanced user experience through gas optimization and security.

Issues Found

Critical

No Critical issues were found.

Major

No Major issues were found.

Minor

1. **Inconsistency Between `totalSupply` Comment and Mint Functionality**

   Issue: The comment above `totalSupply` suggests that the supply can only decrease over time: "initially 10,000,000,000, with the potential to decline over time as some tokens may get burnt but not minted." However, the presence of a `mint` function contradicts this statement, allowing for potential increases in supply.

   Impact: This inconsistency can lead to confusion about the token's intended behavior and supply dynamics, potentially affecting stakeholder trust and contract interactions.

   Recommendation: Clarify the intended behavior in the documentation. If minting is allowed, update the comment to reflect this. If not, consider removing or restricting the `mint` function to align with the stated supply behavior.

Amended (April 22nd, 2025): The comment above totalSupply has been updated to reflect its dynamic nature accurately. Instead of implying that supply can only decrease, the revised comment clarifies that totalSupply is initially set in the constructor or postConstruct initializer and can change over time based on minting and burning functionalities. This update removes the inconsistency and aligns the documentation with the contract's actual behavior. The issue is no longer present as of commit 38a51f3e1e23573a46805f361e0cfedc12cbd15a.

2. Reentrancy Consideration in `GenericFactoryV1.clone`

Issue: The **`clone`** function involves external calls for proxy initialization, which could potentially be a vector for reentrancy if not handled carefully.

Impact: Reentrancy could lead to unintended state changes, where an attacker might exploit the function to manipulate the contract state unexpectedly.

Recommendation: Consider using reentrancy guards, such as the `nonReentrant` modifier, to ensure that the function cannot be called again until the first call is completed. Review the `clone` function in `GenericFactoryV1` to ensure state changes are secure.

Amended (April 22nd, 2025): The client clarified that GenericFactoryV1.clone() is intentionally designed to allow reentrancy, as it solely handles deployment and initialization without adding additional logic. Security considerations, including reentrancy protection, are the responsibility of the implementation contract. Furthermore, AdvancedERC20 does not interact with GenericFactoryV1, making reentrancy concerns irrelevant in this context. This approach was deemed valid, and the justification was accepted.

Notes

1. **Version Constraints with Known Issues:**

   Issue: The contracts currently use `>=0.8.4`, which might miss recent compiler bug fixes and improvements.
   Impact: Using an older version could expose contracts to previously resolved issues or inefficiencies.
   Recommendation: Consider upgrading all contracts to the latest stable Solidity version (`^0.8.26`) for enhanced security and functionality.

   Amended (April 22nd, 2025): The client updated the compiler version to 0.8.29 in hardhat.config.js instead of individual files. This decision was justified by the need to maximize compatibility for RBAC-based applications and serve as a Solidity library. The issue is mitigated and no longer present as of commit 4fbf7e233ec9a558f853b1a00aa196f96fd550be

2. **Visibility Optimization for Gas Savings:**

   Issue: Several functions are declared as `public`, though they could benefit from more specific visibility (`external` or `internal`). Some are not called within the contract or inherited contracts, meaning they could be marked as `external`. Similarly, functions that are not supposed to be called from outside can be marked internal.

   Impact: Keeping functions `public` when they can be `internal` or `external` increases gas usage unnecessarily due to argument memory copying or visibility checks.

   Recommendation: Consider declaring functions as external or internal as per the requirement.

   Amended (April 22nd, 2025): Functions were updated to external where applicable. This optimization reduces gas usage for external calls. The issue is no longer present as of commit 38a51f3e1e23573a46805f361e0cfedc12cbd15a.

3. **Test Coverage:**

   Issue: While test cases exist, branch coverage is not 100%.

   Impact: Incomplete test coverage leaves potential edge cases untested, increasing the risk of bugs going unnoticed in production.

   Recommendation: Consider Conducting a detailed review of uncovered branches and adding tests to ensure complete coverage, especially for critical functions.

   Amended (April 22nd, 2025): Additional test cases were added to cover previously uncovered branches, achieving 100% branch coverage. This ensures all functionalities, including edge cases, are rigorously tested. The issue is no longer present as of the commit 8501a5d119e02d7973e2c8862b633b56fc06d1d3

4. **Use of Block Timestamps in Authorization Functions:**

   Issue: Functions like `permit` and `transferWithAuthorization` rely on block timestamps for validation.

   Impact: While generally acceptable, block timestamps can be slightly manipulated by miners, though the risk is minimal for short time frames.

   Recommendation: Ensure that the time frames are reasonable and account for potential minor discrepancies. If precise timing is critical, consider using block numbers instead of timestamps.

   Amended (April 22nd, 2025): The client clarified that precise timing is not critical for signature verification, as the key requirement is that timestamps increase monotonically. This ensures that past signatures cannot be reused indefinitely, while the minimal risk of miner manipulation remains negligible. This approach was deemed valid, and the justification was accepted.

5. Gas Optimization through Smaller Data Types

Issue: The comment above totalSupply states: "initially 10,000,000,000, with the potential to decline over time as some tokens may get burnt but not minted." This gives the impression that totalSupply can only decrease and never exceed 10 billion so It can utilize smaller data types. Similarly, tokenBalances, v in KV, and nonces could potentially use smaller data types as they are not gonna exceed totalSupply.

Impact: Using larger data types than necessary can lead to increased gas costs. Optimizing these can reduce storage and transaction costs, improving overall efficiency.

Recommendation: Consider using smaller data types: Use uint128 or uint192 for totalSupply and tokenBalances since their maximum value fits within these smaller units. Use uint128 for v in KV to align with totalSupply. Use a smaller data type for nonces if the expected number of transactions allows it.

Amended (April 22nd, 2025): As clarified in Minor Issue 1, totalSupply can increase, so restricting it to a smaller data type is not applicable. Consequently, optimizing variables based on this is also not applicable. This approach was deemed valid, and the justification was accepted.

6. Simplification of add and sub Functions:

Issue: The `**add**` and `**sub**` functions are used for basic arithmetic operations within the contract.

Impact: These functions introduce unnecessary function calls, which can slightly increase complexity and reduce readability.

Recommendation: Consider replacing these function calls with direct arithmetic operations within **__updateHistory**. This could streamline the code, making it more straightforward to understand.

Amended (April 22nd, 2025): The **`add`** and **`sub`** functions were removed, and direct arithmetic operations are now used within __updateHistory. This change simplifies the code, reduces unnecessary function calls, and improves readability. The issue is no longer present as of commit 4fbf7e233ec9a558f853b1a00aa196f96fd550be.

Closing Summary

Upon audit of the AdvancedERC20Token smart contract, it was observed that the contracts contain minor issues, along with several areas of notes. The audit process involved a comprehensive manual review supported by automated analysis using the Slither tool. Each identified issue was carefully validated, and the code was found to be of high quality, featuring well-structured logic, proper commenting, and adherence to best practices. Comprehensive test cases were provided, achieving 100% coverage for functions and lines of code, although branch coverage was not fully achieved, leaving some edge cases untested.

We recommend that minor issues should be resolved. Resolving the areas of notes is up to the client's discretion, as the notes refer to potential improvements in the operations of the smart contract.

Disclaimer

This audit is not a security warranty, investment advice, or an endorsement of the AdvancedERC20Token contract. This audit does not guarantee the complete security or correctness of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The audit was conducted on commit `c81f2da81df287cb328f8a82e730f30d2c62c351`. All findings and recommendations in this report apply specifically to the codebase at this commit.

The individual audit reports are anonymized and combined during a debrief process to provide an unbiased delivery and protect the auditors from legal and financial liability.