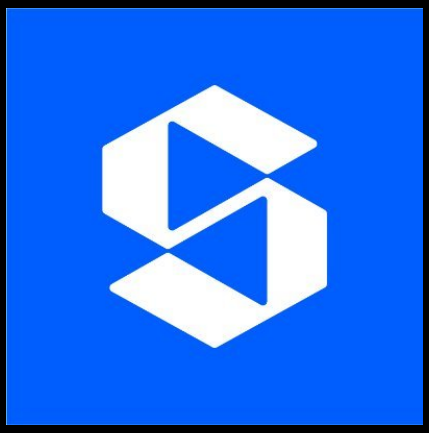
IMMUNEBYTES

Security Assessment

Smart Contract Audit

Standard Money



08th Jan 2026

Standard Money
v1.0

Introduction

Name	Standard Money
Website	https://www.standardmoney.com/
Repository/Source - Initial	0xa25Ae8d48A887dEe2D49DD7B8d92cd43Bd4F8caa
Repository/Source - Final	0x5Ff80C9acC207243183f5B03c49A05DE5d34B494
Platform	L1
Network	BSC and EVM compatible chains
Language	Solidity
Commit Hash	NA
Timeline	07th Jan 2026 - 08th Jan 2026

Table of Content

About Standard Money

Standard Money is a decentralized financial protocol that provides an on-chain stablecoin ecosystem designed to support capital efficiency and yield generation. The protocol enables users to mint, hold, and utilize a synthetic USD-denominated stablecoin backed by collateral and managed through algorithmic and market-based mechanisms. Standard Money integrates smart contracts, liquidity management, and yield strategies to maintain price stability, facilitate value transfer, and offer earning opportunities for participants while operating within a non-custodial, blockchain-based environment.

Executive Summary

This audit examines the StandardMoney contract, an ERC20 token implementation that extends OpenZeppelin's ERC20, ERC20Burnable, and Ownable contracts. The contract implements a standard token with a maximum supply that is minted entirely to a vault address upon deployment, along with an administrative function to withdraw accidentally sent ERC20 tokens.

The contract leverages well-audited OpenZeppelin libraries, which provides a solid foundation. However, several issues were identified that range from Medium to informational in severity.

Trust Assumptions

The following assumptions were made during the audit:

- **Standard ERC20 Behavior:** The audit assumes standard ERC20 token implementations that follow the EIP-20 specification. Non-standard tokens (e.g., USDT on Ethereum mainnet that returns false instead of reverting on failed transfers) were considered but not exhaustively tested.
- **OpenZeppelin Contracts:** Assume that OpenZeppelin contracts (ERC20, ERC20Burnable, Ownable) function as documented and have been properly audited.
- **Trusted Owner:** Assumed the contract owner is trusted and will not act maliciously. The audit focuses on technical vulnerabilities rather than social engineering or key management issues.
- **Standard Deployment:** Assumed standard deployment scenarios with valid parameters. Edge cases with zero values or invalid addresses were considered but may not be exhaustively tested in all scenarios.
- **No Malicious Tokens:** The audit did not test against malicious or reentrant ERC20 token implementations that could exploit the adminTokenWithdraw function.
- **No Front-Running:** Front-running attacks and MEV (Maximal Extractable Value) scenarios were not considered in this audit.
- **Gas Limitations:** Gas griefing attacks and scenarios where transactions might fail due to gas limits were not exhaustively tested.
- **Network Assumptions:** The audit assumes standard EVM behavior and does not account for network-specific quirks or hard forks.
- **Compiler Version:** Assumed Solidity 0.8.20 compiler behavior and built-in overflow protection.
- **Test Coverage:** The existing test suite provides good coverage for standard scenarios, but does not cover all edge cases or malicious token interactions.

Security Checklist

During our comprehensive audit of the project, we have assessed the following potential vulnerabilities inspired by common issues encountered in Ethereum-based systems:

Access Control Issue	✅
Initialization Vulnerabilities	✅
Reentrancy Attacks	✅
Logic and Calculation Errors	✅
Unintended State Manipulation	✅
ERC20 Compliance	✅
Token Swapping Risks	✅
Denial of Service (DoS)	✅
Ownership Validation	✅
Integer Overflow and Underflow	✅
Error Handling	✅
Frontrunning Risks	✅
Audit-Specific Issues	✅
Merkle Tree and Leaf Nodes	✅

Methodology

ImmuneBytes team has performed thorough testing of the project, starting with analyzing the code design patterns where the smart contracts and provided technical documentation were reviewed to ensure the architecture of the codebase aligns with the business specifications from a bird's eye view. During this phase, the invariants were identified along with execution of the pre-existing test suite to ensure smooth working of the implemented functionality and set up for the auditor's edge case testing in upcoming phases

Our team then performed a formal line-by-line inspection of the Smart Contract to find potential issues like Signature Replay Attacks, Unchecked External Calls, Merkle Tree Exploits, Reentrancy, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the next phase, the team ran curated test cases in order to verify the findings and issues identified during previous phases so as to provide a proof of concept. The team also runs automated tests over the provided code base using a combination of external and in-house-developed tools to identify vulnerabilities and security flaws of static nature.

The code was audited by a team of independent auditors, which included -

- Testing the functionality of the Smart Contracts to determine proper logic has been followed throughout.
- Thorough, line-by-line review of the code, done by a team of security researchers with experience across Blockchain security, cybersecurity, software engineering, game theory, economics and finance.
- Deploying the code on localnet using open source testing frameworks and clients to run live tests.
- Analyzing failing transactions to check whether the Smart Contracts implementation handles bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest secure version.
- Analyzing the security of the on-chain data and components.

Security Review Report

Severity	Open	Acknowledged	Partially Resolved	Resolved	TOTAL
Critical	-	-	-	-	-
High	-	-	-	-	-
Medium	-	-	-	1	1
Low	-	3	-	-	3
Informational	-	2	-	-	2
TOTAL	-	5	-	1	6

Severity Reference

1. Critical
   Issues causing protocol insolvency, unauthorized theft, or irreversible loss or alteration of assets, or governance outcomes.

2. High
   Issues involving theft or freezing of unclaimed yield, royalties, or temporary locking of assets.

3. Medium
   Operational failures, inefficiencies, or exploitations causing delays, excessive gas usage, or disruption without direct loss.

4. Low
   Minor deviations in promised functionality without impact on underlying value.

5. Informational
   Recommendations for code readability, maintainability, or best practices.

Status Reference

1. Open
   The issue has been identified and is pending resolution.

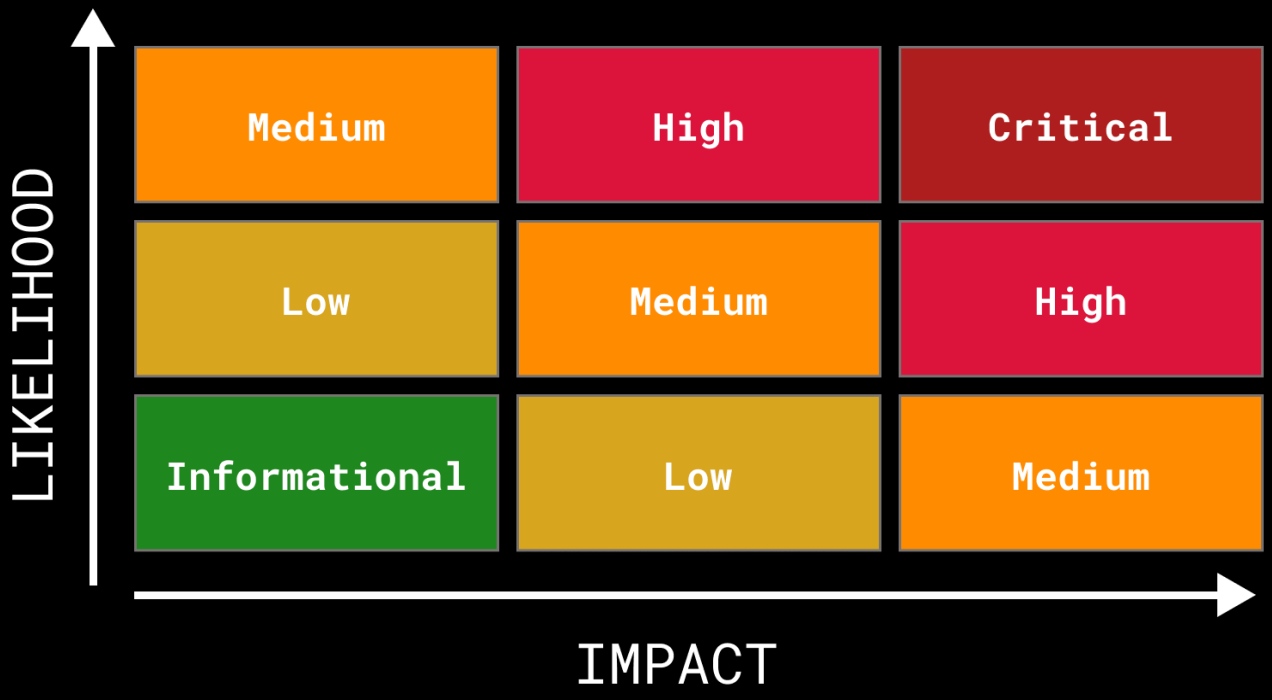
2. Acknowledged
   The issue has been reviewed and accepted by the team but remains unresolved.

3. Partially Resolved
   The issue has been mitigated to some extent, but residual risks or concerns persist.

4. Resolved
   The issue has been fully addressed, with no further action required.

5. Closed
   The issue is no longer relevant or actionable, regardless of resolution.

Risk Matrix


LIKELIHOOD
Medium
High
Critical
Low
Medium
High
Informational
Low
Medium
IMPACT

Summary of Findings

CODE	ISSUE	SEVERITY	STATUS
SM01	Unchecked ERC20 Transfer Return Value	Medium	Resolved
SM02	Missing Zero Address Validation for Token Address	Low	Acknowledged
SM03	Missing Validation for Zero Amount	Low	Acknowledged
SM04	Missing Validation for Zero Max Supply	Low	Acknowledged
SM05	Function Naming Convention	Info	Acknowledged
SM06	Missing Events	Info	Acknowledged

Findings Overview

Severity: Medium	
SM01	**Unchecked ERC20 Transfer Return Value**
Status	Resolved

Location: src/StandardMoney.sol:26

Description

The adminTokenWithdraw function calls token.transfer() without checking the return value. While most modern ERC20 implementations revert on failure, some tokens (notably USDT on Ethereum mainnet) return false instead of reverting. If such a token is used and the transfer fails (e.g., insufficient balance, paused token, or other conditions), the function will not revert and will appear to succeed, potentially leading to incorrect state assumptions and loss of funds.

Impact

- Failed transfers may go unnoticed
- Owner may believe tokens were withdrawn when they weren't
- Could lead to loss of funds if tokens remain stuck in the contract

Code Block Affected

```
function adminTokenWithdraw(address _tokenAddress, uint256 _amount) public onlyOwner {
    IERC20 token = IERC20(_tokenAddress);
    token.transfer(owner(), _amount);  // ❌ Return value not checked
}
```

Recommendation

Use OpenZeppelin's SafeERC20 library to handle transfers safely:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
using SafeERC20 for IERC20;
function adminTokenWithdraw(address _tokenAddress, uint256 _amount) public onlyOwner {
    require(_tokenAddress != address(0), "Token address cannot be zero");
    IERC20 token = IERC20(_tokenAddress);
    token.safeTransfer(owner(), _amount);
}
```

Severity: Low	
SM02	**Missing Zero Address Validation for Token Address**
Status	Acknowledged

Location: src/StandardMoney.sol:24

Description

The adminTokenWithdraw function does not validate that _tokenAddress is not the zero address. While calling transfer() on the zero address would revert (preventing actual harm), explicitly checking for this condition provides better error messages and follows defensive programming practices.

Impact

- Less clear error messages if zero address is accidentally passed
- Does not follow best practices for input validation
- No actual security risk - transaction will revert on low-level call failure

Code Block Affected

```
function adminTokenWithdraw(address _tokenAddress, uint256 _amount) public onlyOwner {
    IERC20 token = IERC20(_tokenAddress);  // ❌ No zero address check
    token.transfer(owner(), _amount);
}
```

Recommendation

Add explicit zero address validation for better error messages:

```
function adminTokenWithdraw(address _tokenAddress, uint256 _amount) public onlyOwner {
    require(_tokenAddress != address(0), "Token address cannot be zero");
    // ... rest of function
}
```

Severity: Low	
SM03	**Missing Validation for Zero Amount**
Status	Acknowledged

Location: src/StandardMoney.sol:24

Description

The adminTokenWithdraw function does not validate that _amount is greater than zero. While transferring zero tokens is harmless, it wastes gas and could indicate an error in the calling code.

Impact

- Minor gas waste
- Potential indication of bugs in calling code

Recommendation

Add zero amount validation if desired:

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

Note: This is optional as zero transfers are harmless, but it can help catch errors early.

Severity: Low	
SM04	**Missing Validation for Zero Max Supply**
Status	Acknowledged

Location: src/StandardMoney.sol:20

Description

The constructor does not validate that _maxSupply is greater than zero. If zero is passed, the contract will deploy successfully but mint zero tokens, creating a token with no supply.

Impact

- Contract could be deployed with zero supply, rendering it unusable
- May indicate a deployment error
- **No runtime security risk** - issue occurs at deployment time

Recommendation

Add validation in constructor:

require(_maxSupply > 0, "Max supply must be greater than zero");

Severity: Informational	
SM05	**Function Naming Convention**
Status	Acknowledged

Location: src/StandardMoney.sol:24

Description

The function name adminTokenWithdraw could be more descriptive. Consider naming conventions that clearly indicate the function's purpose, such as recoverERC20 or withdrawERC20, which are common patterns in OpenZeppelin contracts.

Recommendation

Consider renaming to recoverERC20 or withdrawERC20 for better clarity and consistency with common patterns.

Severity: Informational	
SM06	**Missing Events**
Status	Acknowledged

Description

The adminTokenWithdraw function does not emit an event when tokens are withdrawn. Events are important for off-chain monitoring and transparency.

Impact

- Difficult to track token withdrawals off-chain
- Reduced transparency for administrative actions

Recommendation

Add an Event:

event TokenRecovered(address indexed token, address indexed to, uint256 amount);

function adminTokenWithdraw(address _tokenAddress, uint256 _amount) public onlyOwner {

// ... implementation

emit TokenRecovered(_tokenAddress, owner(), _amount);

}

.

Disclaimer

ImmuneBytes' security audit services are designed to help identify and mitigate potential security risks in smart contracts and related systems. However, it is essential to recognize that no security audit can guarantee absolute protection against all possible security threats. The audit findings and recommendations are based on the information provided during the assessment.

Furthermore, the security landscape is dynamic and ever-evolving, with new vulnerabilities and threats emerging regularly. As a result, it is strongly advised to conduct multiple audits and establish robust bug bounty programs as part of a comprehensive and continuous security strategy. A single audit alone cannot protect against all potential risks, making ongoing vigilance and proactive risk management essential.

To ensure maximum security, the project must implement the recommendations provided in the audit report and regularly monitor its smart contracts for vulnerabilities. It is imperative to adopt appropriate risk management strategies and remain proactive in addressing potential threats as they arise.

Please note that auditors cannot be held liable for any security breaches, losses, or damages that may occur after the audit has been completed despite using audit services. The responsibility for implementing the recommendations and maintaining the ongoing security of the systems rests solely with the project.