QuillAudits

AUDIT REPORT

August 2025

For

Turing M

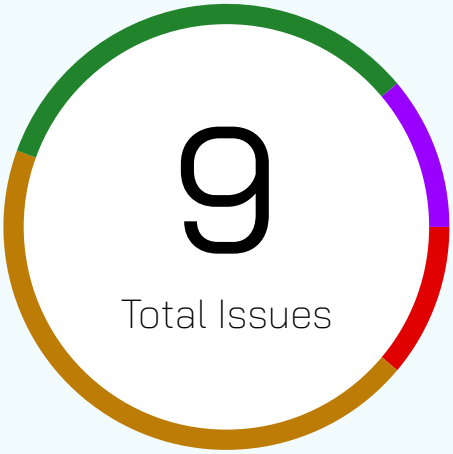
Table of Content

Executive Summary

Project name	TuringM
Project URL	https://TuringM.IO
Overview	The TuringM-EIP2535 is an EIP-2535 base smart contract framework that currently includes TuringMarketApp and TokenUnlockerApp contracts. The TuringMarketApp is an exchange protocol that facilitates atomic swaps between Conditional ERC1155 NFT Token assets and an ERC20 collateral asset. It is intended to be used in a hybrid-decentralized exchange model wherein there is an operator that provides offchain matching services while settlement happens on-chain, non-custodially. The TokenUnlockerApp is a contract that provide users to invest/refund, stake/unstake, vote for proposal for our TuringM DAO.
Audit Scope	The scope of this Audit was to analyze the Turing Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/TuringM-Labs/TuringM-EIP2535/tree/23bb1fb1cc7838781e312a4deaa9c127ff82fcc1
Branch	release/1.0.0
Commit Hash	23bb1fb1cc7838781e312a4deaa9c127ff82fcc1
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing

Review 1	20 June 2025 - 2 July 2025
Updated Code Received	15 July 2025
Review 2	15th July 2025 - 17th July 2025
Review 3	5th August 2025
Fixed In	https://github.com/TuringM-Labs/TuringM-EIP2535/pull/5/commits/02ec0d0e90e2a688f735e57ea992b739bbcb63ca Commit hash : 02ec0d0e90e2a688f735e57ea992b739bbcb63ca

Number of Issues per Severity



High	1 (11.11%)
Medium	4 (44.44%)
Low	3 (33.33%)
Informational	1 (11.11%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**1**	**2**	0	**1**
Acknowledged	0	**2**	**3**	0
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Users Can Be Permanently Locked Out of Staked Funds Due to Transfer Quota Enforcement

Resolved

Path

contracts/apps/TokenUnlocker/Staking/Facet.sol

Function

unstake

Description

A critical denial-of-service vulnerability exists in the unstake function where users can become permanently unable to withdraw their staked tokens. The function enforces a **`maxTokenTransferOutQuote`** limit that can be set lower than existing stake amounts, causing all unstake attempts to revert with **`ExceedsMaxTokenTransferOutQuote`**.

This creates a scenario where users who staked large amounts before quota limits were imposed cannot retrieve their funds, potentially resulting in permanent loss of access to legitimately staked assets.

The vulnerability affects any user whose individual stake amount exceeds the current transfer quota limit, making it a systemic risk for the protocol's largest stakeholders.

TuringM team's comment:

We remove the
`_isPayoutDisabled();`

And

```
_tokenTransferOutQuoteCheck("unstake", tokenAddress, amount);
```

Check for this method

Medium Severity Issues

Unchecked Token Transfer Return Value Can Lead to Silent Payment Failures and User Fund Loss

Resolved

Path

contracts/apps/TuringMarket/OrderMatcher/Base.sol

Function

_performOrderNormal

Description

A vulnerability exists in the order matching logic where a token transfer operation does not verify its return value, potentially causing silent failures. In `_performOrderNormal()`, when a taker sells NFTs, the final payment transfer to the taker uses `IERC20.transfer()` without checking the boolean return value.

This creates a scenario where the transfer could fail silently (particularly with non-standard ERC20 tokens like USDT that return void or tokens that return false on failure), while the contract continues execution as if the transfer succeeded. As a result, users could lose their NFTs through burning or transfer while never receiving their expected payment, leading to direct financial loss.

The vulnerability is particularly dangerous because all other token transfers in the codebase properly validate return values, making this an isolated but critical oversight in a high-value transaction path.

TuringM team's comment:

We have fixed it by replacing the code with::

```
bool paymentResult =
IERC20(takerOrder.paymentTokenAddress).transfer(takerOrder.maker,
takerOrder.paymentTokenAmount);
    if (!paymentResult)
      revert PaymentTransferFailed(
      takerOrder.salt,
      address(this),
      takerOrder.maker,
      takerOrder.paymentTokenAddress,
      takerOrder.paymentTokenAmount
 );
```

ERC1155 Batch Transfer Validation Uses Wrong Function Selector, Breaking Standard Compliance

Resolved

Path

contracts/facets/ERC1155/Base.sol

Function

`_checkOnERC1155BactchReceived`

Description

An implementation error exists in the **ERC1155** batch transfer validation logic where the **_checkOnERC1155BactchReceived** function incorrectly validates the return value from on**ERC1155BatchRe-ceived** calls.

When a contract receives batch transfers, the validation compares the returned selector against **IERC1155Receiver.onERC1155Received.selector** instead of the correct **IERC1155Receiver.on-ERC1155BatchReceived.selector**

This breaks **ERC1155** standard compliance and creates two potential issues: legitimate receiving con-tracts that properly implement the batch transfer handler may have their transfers rejected, and ma-licious contracts could potentially bypass proper validation by returning the wrong but accepted selector.

The vulnerability undermines the safety guarantees of the **ERC1155** receiver pattern and could lead to unexpected failures in batch transfer operations with compliant receiving contracts.

TuringM team's comment:

We update to **onERC1155BatchReceived**

Low Severity Issues

EIP712 Domain Separator Permanently Cached Without Chain Fork Detection, Breaking Standard Compliance

Acknowledged

Path

contracts/facets/ERC712/Base.sol

Function

`_getDigest`

Description

A vulnerability exists in the **`EIP712`** implementation where the domain separator is calculated once during initialization and permanently cached in storage, without any mechanism to detect or handle blockchain forks that change the chain ID.

The **`_setEIP712Config()`** function calculates the domain separator using the current **`block.chainid`** and stores it as **`cachedDomainSeparator`**, while **`_getDigest()`** always uses this cached value without checking if the chain ID has changed.

This violates **`EIP712`** standard requirements for dynamic chain ID handling and creates two critical issues:

1. Legitimate user signatures become invalid after network forks/upgrades that change the chain ID

2. The contract loses proper replay protection across different chain versions.

The vulnerability affects all signature-dependent operations and can lead to user lockouts during network upgrades or potential signature replay attacks in fork scenarios.

TuringM team's comment:

We add **`setEIP712Config`** function in **`EIP712Facet`** now

Ownership is not implemented in TokenUnlocker

Acknowledged

Path

contracts/apps/TokenUnlocker/App.sol

Description

no ownership is implemented

TuringM team's comment:

The ownership is implemented by `OwnableFacet`, and the facet cut into the app while deploying the application

Ownership is not implemented in TuringMarket

Acknowledged

Path

contracts/apps/TuringMarket/App.sol

Description

no ownership is implemented

TuringM team's comment:

The ownership is implemented by `OwnableFacet`, and the facet cut into the app while deploying the application

Informational Severity Issues

Redundant Non-Negativity Check for Unsigned Vote Parameters Wastes Gas and Reduces Code Quality

Resolved

Path

contracts/apps/TokenUnlocker/Vote/Facet.sol

Function

`vote`

Description

A code quality issue exists in the vote function where an unnecessary validation checks if `yesVotes >= 0 && noVotes >= 0` for parameters declared as uint256. Since `uint256` is an unsigned integer type in Solidity that can only hold values from `0 to 2^256-1`, both conditions `yesVotes >= 0 and noVotes >= 0` are mathematically always true and serve no purpose.

This redundant check wastes gas on every voting transaction by performing unnecessary comparison operations and reduces code readability by including logically impossible conditions. The validation should be removed entirely as `uint256` parameters cannot be negative by definition.

While this issue does not pose any security risk, it represents poor coding practices and unnecessary computational overhead that accumulates across multiple voting operations in the governance system.

TuringM team's comment:

Ok, we delete it, also delete the `0 <= proposalId` check too.

Fee Token Info Getter Function Can Revert Unexpectedly Due to Unchecked External Contract Calls

Acknowledged

Path

contracts/apps/TuringMarket/Admin/Facet.sol

Function

`getFeeTokenInfo`

Description

An issue exists in the `getFeeTokenInfo` function where it unconditionally calls `IERC20(val).decimals()` on the provided address without validating if the address is a valid ERC20 contract or even a fee token.

This creates multiple failure scenarios:

1. The function will revert when called with invalid addresses (such as EOAs, non-contract addresses,

2. Contracts that don't implement the ERC20 interface), making it unreliable for frontend integrations and off-chain queries.

Additionally, the function returns fee token information for any address, regardless of whether it's actually configured as a fee token, potentially misleading callers who expect validation. The function should either validate that the provided address is a registered fee token before proceeding, or implement graceful error handling to return a boolean success flag along with the data.

This affects the usability and reliability of the getter function, potentially causing integration issues and unexpected reverts in applications that depend on this interface.

TuringM team's comment:

We think this is all right, as we have the "`isEnabled: s.feeTokenAddressMap[val]`" field that can inde-cated the token status(if it is enabled, it will be true).

Unnecessary Storage Pointer Creation in Paused State Getter Wastes Gas on Every Access

Acknowledged

Path

contracts/utils/Pausable.sol

Function

`_paused`

Description

A gas optimization issue exists in the `_paused` function where it unnecessarily creates a storage pointer to read a single boolean value. The function declares `PausableStorage` storage `$ = _get-PausableStorage()` only to access one field `$._paused`, which creates unnecessary overhead on every pause state check.

Since this function is called frequently through the `whenNotPaused` modifier on most protected func-tions throughout the contract system, this inefficiency compounds across all user interactions. The storage pointer creation involves additional gas costs for pointer setup when a direct access pattern `_getPausableStorage()._paused` would be more efficient. This represents a minor but pervasive gas waste that affects every transaction in the system, as pause state checking is ubiquitous.

While not a security vulnerability, this optimization opportunity could result in measurable gas savings across the entire application, especially given the high frequency of pause state checks in the Diamond Standard architecture.

TuringM team's comment:

OK, we update the func to be:

```
function _paused() internal view returns (bool) {
        return _getPausableStorage()._paused;
    }
```

Closing Summary

In this report, we have considered the security of TuringM. We performed our audit according to the procedure described above.

1 issue of High severity, 2 issues of medium severity, 3 issues of low severity & 3 issues of informational severity were found. TuringM Team resolved few of the issues and acknowledged remaining ones.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2025

For

Turing M

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com