CERTIK

# Audit Report

## Produced by CertiK

for Mysterium

# Contents

## Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Mysterium (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

## About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that projects are checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and assessments to each project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor. CertiK customizes its engineering tool kits, while applying cutting-edge research on smart contracts, for each client on its project to offer a high quality deliverable.  For more information: https://certik.io.

# Executive Summary

This report has been prepared for **Mysterium** to discover issues and vulnerabilities in the source code of their **multi-EIP compliant MYST v2 Token and Upgrade Mechanism** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

# Testing Summary

SECURITY LEVEL


Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Mysterium.

This audit was conducted to discover issues and vulnerabilities in the source code of Mysterium's multi-EIP Compliant MYST v2 token and Upgrade Mechanism.

| | |
|---|---|
| TYPE | Smart Contract |
| SOURCE CODE | https://github.com/mysteriumnetwork/token-v2 |
| PLATFORM | EVM |
| LANGUAGE | Solidity |
| REQUEST DATE | July 18, 2020 |
| DELIVERY DATE | August 13, 2020 |
| METHODS | A comprehensive examination has been performed using Dynamic Analysis, Static Analysis, and Manual Review. |

# Review Notes

## Introduction

CertiK was contracted by the Mysterium to audit the design and implementation of their MYST v2 token smart contract, its compliance with the multiple EIPs it is meant to implement as well as its upgrade mechanism.

The audited source code link is:

- Token & Upgrade Mechanism Source Code: https://github.com/mysteriumnetwork/token-v2/tree/1639772dee499c0801685e51014207780e13af27

The goal of this audit was to review the Solidity implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production. We additionally discussed the potential change to ERC777 or ERC20 + ERC2620 tokens, recommending the ERC20 + ERC2620 implementation as the more safe option. From a security perspective, since it is fully compliant with the standards and is not a custom implementation it is more likely to be future-proof than hand-made implementations.

After relaying our findings to the Mysterium team, a second commit hash was provided to us whereby the codebase of the Mysterium token was re-evaluated and an alleviation paragraph was introduced to all previously written findings. The newly audited source code link is:

- Token & Upgrade Mechanism Source Code: https://github.com/mysteriumnetwork/token-v2/tree/d160e18c5ed0303d0b634b3898bac0d64e2c4efe

## Documentation

The sources of truth regarding the operation of the contracts in scope were comprehensive and **greatly aided our efforts to audit the project as well as generally increase the legibility of the codebase**. To help aid our understanding of each contract's functionality we referred to in-line comments and naming conventions.

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Mysterium team or reported an issue.

## Summary

The codebase of the project is a typical EIP20 implementation with additional support for EIP712 and ERC2612.

**Certain optimization steps** that we pinpointed in the source code mostly referred to coding standards and inefficiencies, however **1 minor severity vulnerability was identified during our audit that solely concerns the EIP20 specification and is a widely known issue**. The codebase of the project strictly adheres to the standards and interfaces imposed by the OpenZeppelin open-source libraries and **can be deemed to be of high security and quality**.

## Recommendations

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code **to achieve a high standard of code quality and security**.

# Findings

## Exhibit 1

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Variable Mutability Optimization | Optimization | Informational | MystToken.sol: L14 - L15, L17, L24 - L25, L28 |

**[INFORMATIONAL] Description:**

The aforementioned variables are declared as “private” variables of the contract that are assigned only once during the contract’s “constructor”.

**Recommendations:**

Version 0.6.11 of Solidity supports the “immutable” mutability specifier for contract variables that replaces all occurrences of the variable within the code itself with the value accessed during its construction. This significantly reduces the gas cost of all functions regarding these variables as they are directly read as literals rather than from storage.

Additionally, the “_originalToken”, “_originalSupply”, “_upgradeMaster” and “_totalUpgraded” variables could be declared as “public” and omit the prefixed underscore to make their manual “getter” functions (i.e. L194 - L196) redundant.

**Alleviation:**

The corresponding variables had their attributes properly adjusted to aid the compiler in generating the necessary optimizations as well as getters where applicable.

We would like to note that it is generally a good coding practice to explicitly set visibility specifiers for contract-level variables and as such, we would advise that “_originalToken” and “_originalSupply” had theirs done so.

## Exhibit 2

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Getter Functions to “constant” Variables | Coding Style | Informational | MystToken.sol: L81 - L83, L203 - L205 |

**[INFORMATIONAL] Description:**

The functions included above return explicit numeric literals and act as read-only functions.

**Recommendations:**

We advise that they instead be swapped by “constant” and “public” variables, the middle of which would also require the “override” trait, to aid in the legibility of the codebase and make the variables more verbose.

**Alleviation:**

Our recommendation was applied to the letter.

## Exhibit 3

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| ERC20 Approval Race Condition | Race Condition | Minor | MystToken.sol: L184 - L190 |

**[MINOR] Description:**

The approval function of the ERC20 standard possesses a well-known vulnerability whereby two consecutive approvals can be exploited to instead “reset” the approval by consuming the approval that is meant to be replaced with a new one.

**Recommendations:**

We advise that a mitigation measure, such as zeroing the approval amount before setting it to a different value, is put in place to partially alleviate the race condition vulnerability.

**Alleviation:**

A mitigation in the form of the “increaseAllowance” and “decreaseAllowance” functions was applied whereby approvals are increased or decreased by a specific amount instead of being overridden.

## Exhibit 4

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Magic Number “10000000000” | Mathematics | Informational | MystToken.sol: L211, L213 |

**[INFORMATIONAL] Description:**

As documented in the repository, the original MYST token and the new MYST token have a decimal difference of 10. As such, minted values as well as the total supply comparisons need to offset all values by 10 zeroes to properly compare them.

**Recommendations:**

The literal “10000000000”, better depicted as “1e10”, should instead be stored in a “private” and “constant” variable titled “DECIMAL_OFFSET” along with explanatory text to its purpose. This poses no gas overhead whatsoever as the compiler swaps instances of “constant” variables with their literal contents.

Additionally, it would also be possible to retrieve the decimal difference during the “constructor” of the contract and store the number 10 to the power of this difference to an “immutable” variable to enable re-usable logic between upgrade-able tokens.

**Alleviation:**

A “constant” variable labelled “DECIMAL_OFFSET” was set according to our recommendation and subsequently utilized in the necessary code blocks. Its value was explicitly set instead of being dynamically retrieved.

## Exhibit 5

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Pull Over Push Pattern | Logical | Informational | MystToken.sol: L253 - L259 |

**[INFORMATIONAL] Description:**

The function “setUpgradeMaster” overrides the existing “_upgradeMaster” with the new one passed as the function’s argument.

**Recommendations:**

In Solidity, it is a secure practise to favor the pull over push pattern whereby a new “upgradeMaster” is proposed in this case and has to accept the proposal before the actual variable within the contract is replaced.

This prevents misconfigurations whereby an incorrect address is set as the “_upgradeMaster”, locking all functions that necessitate that privilege. The “setUpgradeAgent” performs a lot of additional checks with regards to the new agent that the pull over push pattern is not necessary here.

**Alleviation:**

The Mysterium team considered the drawbacks of the current implementation and opted to stick with it as the sanity check against the zero address, which is the default empty value, was considered adequate.

## Exhibit 6

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Inefficient Greater-Than Comparison w/ Zero | Optimization | Informational | MystToken.sol: L238 |

**[INFORMATIONAL] Description:**

The aforementioned line conducts a greater-than (“>”) comparison between the unsigned integer value of “amount” and the literal “0”.

**Recommendations:**

As the variable “amount” is confined to the positive range inclusive of zero, the comparison can instead be converted to an inequality one which is more optimal gas wise.

**Alleviation:**

The comparison was properly changed to its optimized inequality counterpart.

## Exhibit 7

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Error Messages | Coding Style | Informational | MystToken.sol: L263, L292 |

**[INFORMATIONAL] Description:**

The aforementioned “require” statements are not accompanied by an error message.

**Recommendations:**

We advise that a proper descriptive error message is set for these two “require” statements.

**Alleviation:**

The missing error message has been properly added. Previously, proper error messages were provided for both “require” statements. A previously missed “require” statement existed in the previous change that did not possess an error message and resided on L305 of the codebase.

## Exhibit 8

| TITLE | TYPE | SEVERITY | LOCATION |
| --- | --- | --- | --- |
| Incorrect Comment | Coding Style | Informational | MystToken.sol: L305 |

**[INFORMATIONAL] Description:**

The comment that accompanies the “claimTokens” function states that the selected tokens will be sent to the “owner address” yet a dedicated address variable is utilized instead.

**Recommendations:**

We advise that the comment is properly adjusted to reflect the true functionality of the “claimTokens” function.

**Alleviation:**

The comment was adjusted to reflect the proper destination address of the function.