Chiliz

Security Assessment

February 10th, 2021

For :
Chiliz

By :
Camden Smallwood @ CertiK
camden.smallwood@certik.org

Angelos Apostolidis @ CertiK
angelos.apostolidis@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	Chiliz
Description	An upgradeable ERC20 implementation via the proxy upgrade pattern.
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	1. ed186b51bfbe8a28b3376bfcc82b5cc93806cc94 2. de64f33f9448c485d4cec0948daf2e25e6493b0d

Audit Summary

Delivery Date	February 10th, 2021
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	January 6th, 2021 - February 10th, 2021

Vulnerability Summary

Total Issues	10
Total Critical	0
Total Major	0
Total Medium	0
Total Minor	1
Total Informational	9

Executive Summary

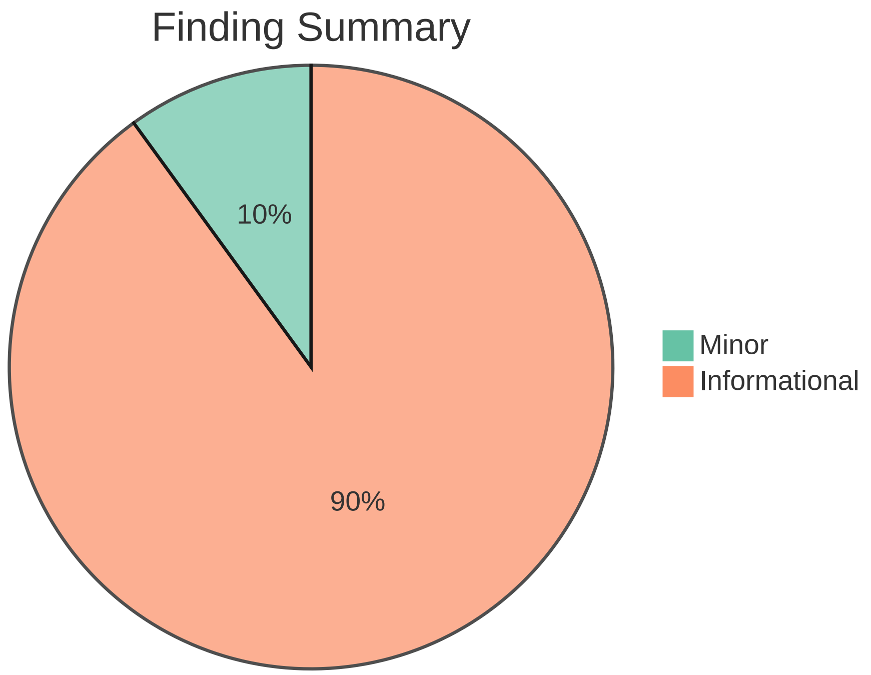
This report represents the results of CertiK's engagement with Chiliz on their implementation of the Chiliz token smart contract.

Our findings mainly refer to optimizations and Solidity coding standards, hence the issues identified pose no threat to the contract deployment's safety.

Files In Scope

ID	Contract	Location
VTP	VoteTokenProxy.sol	VoteTokenProxy.sol
VTI	VoteTokenImplementation.sol	VoteTokenImplementation.sol

Findings



ID	Title	Type	Severity	Resolved
VTI-01	Unlocked Compiler Version	Language Specific	Informational	✓
VTI-02	Array Size Alteration via `length`	Volatile Code	Minor	✓
VTI-03	Inefficient `address` Storage	Volatile Code	Informational	✓
VTI-04	Visibility Specifiers Missing	Language Specific	Informational	✓
VTI-05	Redundant `require` Statements	Dead Code	Informational	✓
VTI-06	Conditional Optimization	Gas Optimization	Informational	✓
VTI-07	Redundant Statement	Dead Code	Informational	✓
VTI-08	Multiple Instances of the `initialize()` Function	Volatile Code	Informational	✓
VTP-01	Unlocked Compiler Version	Language Specific	Informational	✓
VTP-02	Multiple Instances of the `initialize()` Function	Volatile Code	Informational	✓

VTI-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	VoteTokenImplementation.sol L1077

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.2` the contract should contain the following line:

```
pragma solidity 0.6.2;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.5.1`.

VTI-02: Array Size Alteration via `length`

Type	Severity	Location
Volatile Code	Minor	VoteTokenImplementation.sol L1061

Description:

In general, it is a bad practice to alter the length of an array by directly increasing/decreasing the `length` member of the array.

Recommendation:

We advise to use the `pop()` array member, which in turn implicitly calls `delete` on the removed element.

Alleviation:

The development team opted to consider our references, removed the direct array size alteration along with the `delete` statement and used the `pop()` function instead.

VTI-03: Inefficient `address` Storage

Type	Severity	Location
Volatile Code	Informational	VoteTokenImplementation.sol L1021-L1030

Description:

The linked code segment redundantly stores the new entries of `address` in `ListItem.item`, `StoredList.storageMap` and `StoredList.storageList` members.

Recommendation:

We advise to revise the linked code block and implement a more efficient functionality.

Alleviation:

The development team opted to consider our references, removed the `ListItem` struct along with the `storageMap` member of the `StoredList` struct and only used the `storageList` of the latter struct to store the `address` entries.

VTI-04: Visibility Specifiers Missing

Type	Severity	Location
Language Specific	Informational	VoteTokenImplementation.sol L1085

Description:

The linked variable declarations do not have a visibility specifier explicitly set.

Recommendation:

Inconsistencies in the default visibility the Solidity compilers impose can cause issues in the functionality of the codebase. We advise that visibility specifiers for the linked variables are explicitly set.

Alleviation:

The development team opted to consider our references and added the `private` visibility specifier for the linked variable.

VTI-05: Redundant `require` Statements

Type	Severity	Location
Dead Code	Informational	VoteTokenImplementation.sol L1119-L1120, L1140-L1141

Description:

The linked statements redundantly check the input values, as the parent function is already checking against the same conditional.

Recommendation:

We advise to remove redundant code.

Alleviation:

The development team opted to consider our references and removed the redundant code.

VTI-06: Conditional Optimization

Type	Severity	Location
Gas Optimization	Informational	VoteTokenImplementation.sol L1177

Description:

The linked `for` conditional redundantly checks the array length on every iteration.

Recommendation:

We advise to declare a local variable and assign it the value of the length of the array and use this local variable on the conditional to save gas.

Alleviation:

The development team opted to consider our references and changed the linked code segment as proposed.

VTI-07: Redundant Statement

Type	Severity	Location
Dead Code	Informational	VoteTokenImplementation.sol L1200

Description:

The linked statement does not affect the functionality of the codebase, as the linked variable is declared, and initialized to zero by default, in L1192.

Recommendation:

We advise to remove the linked statement.

Alleviation:

The development team opted to consider our references and removed the redundant code.

VTI-08: Multiple Instances of the `initialize()` Function

Type	Severity	Location
Volatile Code	Informational	VoteTokenImplementation.sol General

Description:

Instances of the `initialize()` function with different signatures are exposed upon contract deployment. In the case an incorrect `initialize()` function is called, the majority (if not all) of the intented functionality will be rendered useless.

Recommendation:

We advise to ensure that a parent initializer is not invoked.

Alleviation:

The development team acknowledged this exhibit and will take it into consideration upon deployment.

VTP-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	VoteTokenProxy.sol L413

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.2` the contract should contain the following line:

```
pragma solidity 0.6.2;
```

Alleviation:

The development team opted to consider our references and locked the compiler to version `0.5.1`.

VTP-02: Multiple Instances of the `initialize()` Function

Type	Severity	Location
Volatile Code	Informational	VoteTokenProxy.sol General

Description:

Instances of the `initialize()` function with different signatures are exposed upon contract deployment. In the case an incorrect `initialize()` function is called, the majority (if not all) of the intented functionality will be rendered useless.

Recommendation:

We advise to ensure that a parent initializer is not invoked.

Alleviation:

The development team acknowledged this exhibit and will take it into consideration upon deployment.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.