CERTIK

Security Assessment

Manifold - Ash Solidity

May 12th, 2021

Summary

This report has been prepared for Manifold - Ash Solidity smart contracts, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing 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.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases given they are currently missing in the repository;
- Provide more comments per each function for readability, especially contracts are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Majority of the findings are of informational nature with one minor finding. The minor finding comprise the lack of input sanitization of the function parameter.

Overview

Project Summary

Project Name	Manifold - Ash Solidity
Description	The audited codebase comprise the `Burn` ERC20 and `ASHRateEngine` contracts. The `Burn` contract allows receiving of `ERC721` and `ERC1155` tokens, burns them and in return mints `Burn` tokens for the sender depending upon the rate stored for that `ERC721` or `ERC1155` contract in `ASHRateEngine` contract.
Platform	Ethereum
Language	Solidity
Codebase	https://github.com/manifoldxyz/ash-solidity/tree/dcbff2889f1f10df663d8dcc06d89f6de4b75a3b/contracts
Commits	1. https://github.com/manifoldxyz/ash-solidity/tree/dcbff2889f1f10df663d8dcc06d89f6de4b75a3b/contracts 2. https://github.com/manifoldxyz/ash-solidity/commit/462e2e919c2546a4c8ac691fa003bc39220ab499

Audit Summary

Delivery Date	May 12, 2021
Audit Methodology	Static Analysis, Manual Review
Key Components	

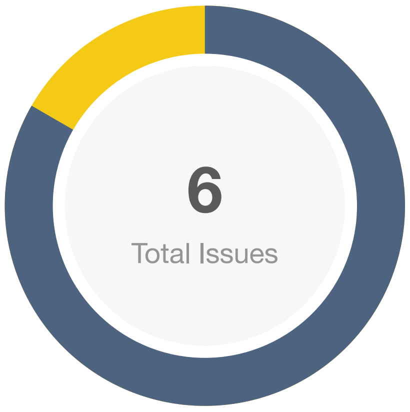
Vulnerability Summary

Total Issues	6
● Critical	0
● Major	0
● Medium	0
● Minor	1
● Informational	5
● Discussion	0

Audit Scope

ID	file	SHA256 Checksum
ASH	ASH.sol	5269466328a2d3fcfb93571332f0756d128338e8e6dfcf0a8e4b166c84e054c4
MIG	Migrations.sol	4fd6092bdfa8b42f19d535c5ac69c4323b0b894717c699e58d5552eeabd04cd4
RMZ	libraries/RealMath.sol	eb10affe00d89d10f8aea1211433ea1ae75c5d77d4015fbc10a680f4292894ca
ASR	rates/ASHRateEngine.sol	44ce9d88fc84047d090cb8f913f5d4b5b37ca198ec3942cb8c6d8a864ff4b3ff
IAS	rates/IASHRateEngine.sol	5383c462d4c94a2fe1509d8da6a00cf56071a8057aa418e6236af65ad75fcf96

Findings



🟥 **Critical**	**0** (0.00%)
🟧 **Major**	**0** (0.00%)
🟧 **Medium**	**0** (0.00%)
🟨 **Minor**	**1** (16.67%)
🟦 **Informational**	**5** (83.33%)
🟩 **Discussion**	**0** (0.00%)

ID	Title	Category	Severity	Status
ASH-01	Unlocked Compiler Version	Language Specific	● Informational	✓ Resolved
ASR-01	Unlocked Compiler Version	Language Specific	● Informational	✓ Resolved
ASR-02	Redundant Variable Initialization	Coding Style	● Informational	✓ Resolved
ASR-03	Inefficient storage read	Gas Optimization	● Informational	✓ Resolved
ASR-04	Lack of validation for the function parameter	Logical Issue	● Minor	ⓘ Acknowledged
IAS-01	Unlocked Compiler Version	Language Specific	● Informational	✓ Resolved

ASH-01 | Unlocked Compiler Version

Category	Severity	Location	Status
Language Specific	● Informational	ASH.sol: 3	✓ Resolved

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 `v0.7.0` the contract should contain the following line: `pragma solidity 0.8.2;`.

Alleviation

Alleviations were applies as of commit hash `462e2e919c2546a4c8ac691fa003bc39220ab499 `.

ASR-01 | Unlocked Compiler Version

Category	Severity	Location	Status
Language Specific	● Informational	rates/ASHRateEngine.sol: 3	✓ Resolved

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 `v0.7.0` the contract should contain the following line: `pragma solidity 0.8.2;`.

Alleviation

Alleviations were applied as of commit hash `462e2e919c2546a4c8ac691fa003bc39220ab499 `.

ASR-02 | Redundant Variable Initialization

Category	Severity	Location	Status
Coding Style	● Informational	rates/ASHRateEngine.sol: 26	✓ Resolved

Description

All variable types within Solidity are initialized to their default `empty` value, which is usually their zeroed out representation.

- `uint` / `int`: All `uint` and `int` variable types are initialized at `0`
- `address`: All `address` types are initialized to `address(0)`
- `byte`: All `byte` types are initialized to their `byte(0)` representation
- `bool`: All `bool` types are initialized to `false`
- `ContractType`: All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct`: All `struct` types are initialized with all their members zeroed out according to this table

Recommendation

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation

Alleviations were applied as of commit hash `462e2e919c2546a4c8ac691fa003bc39220ab499` .

ASR-03 | Inefficient storage read

Category	Severity	Location	Status
Gas Optimization	● Informational	rates/ASHRateEngine.sol: 86~91	✓ Resolved

Description

The code on aforementioned lines read `_contractTokenRateClass[tokenContract][args[0]]` from contract's storage twice which causes increased gas cost as the code can be rectified to limit the storage read to only once.

Recommendation

We advise to rectify the code on the aforementioned lines to limit the storage read of `_contractTokenRateClass[tokenContract][args[0]]` to only once to save gas cost associated with the extra storage read operation.

```
uint8 rateClass = _contractTokenRateClass[tokenContract][args[0]];
if (rateClass == 0) {
    rateClass = _contractRateClass[tokenContract];
}
```

Alleviation

Alleviations were applied as of commit hash `462e2e919c2546a4c8ac691fa003bc39220ab499`.

ASR-04 | Lack of validation for the function parameter

Category	Severity	Location	Status
Logical Issue	● Minor	rates/ASHRateEngine.sol: 48, 63	ⓘ Acknowledged

Description

The `require` checks on the aforementioned lines ensure that the `rateClass` value should be less than 3 yet it does not ensure that value should be greater than 0.

Recommendation

We advise to extend the `require` checks on the aforementioned lines to ensure that the `rateClass` value should be greater than 0.

Alleviation

Alleviations are not considered with the Manifold team stating "This is intentional in order to provide ability to unset a rate class by providing a 0 value. uint already dictates that it can't be negative."

IAS-01 | Unlocked Compiler Version

Category	Severity	Location	Status
Language Specific	● Informational	rates/IASHRateEngine.sol: 3	✓ Resolved

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 `v0.7.0` the contract should contain the following line: `pragma solidity 0.8.2;`.

Alleviation

Alleviations are applied as of commit hash `462e2e919c2546a4c8ac691fa003bc39220ab499`.

Appendix

Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings 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 findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings 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 but rather 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.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

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 Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with 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.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is 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 assessment. This report does 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.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing 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.

About

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

CERTIK