Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RIBBON

aevo-governance

Veridise Inc.

January 20, 2024

► **Prepared For:**

Ribbon
https://www.ribbon.finance

► **Prepared By:**

Benjamin Sepanski
Sorawee Porncharoenwase

► **Contact Us:** contact@veridise.com

► **Version History:**

Jan. 20, 2024 V1

Contents

1 Executive Summary

From Jan. 16, 2024 to Jan. 18, 2024, Ribbon engaged Veridise to review the security of their aevo-governance smart contracts. The review covered their `AevoToken`, and a staking contract which facilitated both migration from RBN to AEVO, and staking either RBN or AEVO into the contract. Veridise conducted the assessment over 6 person-days, with 2 engineers reviewing code over 3 days from commits `c53149e3-125d53e0`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The aevo-governance developers provided the source code of the aevo-governance contracts for review. The source code is original code written by the aevo-governance developers, containing updates to prior contracts audited by Veridise which implemented migration from RBN to AEVO.

The logic of the application consists of a OpenZeppelin-based ERC20 AEVO token, and a staking contract. Once a certain date is reached, migration from RBN to AEVO may be permissionlessly performed on a 1-for-1 basis. RBN or AEVO may also be staked into the contract. Staked RBN or AEVO may be withdrawn as AEVO (at a 1-1 exchange rate) once the staking period has ended.

It contains some documentation in the form of comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the aevo-governance developers shared a document detailing the intended behavior and usage of the contracts. The source code contained a test suite, which the Veridise auditors noted included thorough tests of all main behaviors, as well as ensuring safety in several scenarios (e.g. checking that access control protects certain key methods).

During the audit, the Ribbon developers made one minor change to the contracts to add a view function. The added function was nearly identical to another function which was already in scope.

Summary of issues detected. The audit uncovered 5 issues, none of which represented active threats to user funds. Two were designated as low-severity; V-RBN-VUL-001 describes a steady increase in gas costs when unstaking which could eventually become prohibitive, and V-RBN-VUL-002 describes the centralization risks present in the protocol. Veridise auditors also flagged 1 warning and 2 informational findings.

Recommendations. After auditing the protocol, the auditors found aevo-governance to be a very high-quality project. The one remaining concern for users would be that, if they misuse the contract by simply transferring funds to the `AevoStaking` contract, rather than using the contract functions, their funds will be locked. This is an intentional design choice, as the contract is intended to unalterably migrate from RBN to AEVO. The Veridise auditors recommend the Ribbon team document very clearly to users both the intended way of staking/migrating tokens, and emphasize that simply transferring funds to the contract will lead to loss of funds.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
aevo-governance	`c53149e3-125d53e0`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 16 - Jan. 18, 2024	Manual & Tools	2	6 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Fixed	Acknowledged
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	2	1	2
Warning-Severity Issues	1	1	1
Informational-Severity Issues	2	2	2
TOTAL	5	4	5

Table 2.4: Category Breakdown.

Name	Number
Data Validation	2
Denial of Service	1
Access Control	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of aevo-governance's smart contracts. In our audit, we sought to answer questions such as:

- Can users withdraw staked funds after the appropriate staking period?
- Are users prevented from withdrawing staked funds before the staking period ends?
- Are staked user funds protected from privileged contract accounts?
- Are common Solidity vulnerabilities (such as reentrancy, front-running risks, or possible flash loan attack vectors) present in the codebase?
- Is the exchange rate during migration or staking correct?
- Can any of the contract behaviors lead to unbounded gas usage?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the `src/` folder of the source code provided by the aevo-governance developers, which contains the smart contract implementation of the aevo-governance.

Methodology. Veridise auditors reviewed the reports of previous audits for aevo-governance, inspected the provided tests, and read the aevo-governance documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly communicated with the aevo-governance developers via Telegram to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RBN-VUL-001	Cost of _getUnlocked() increases with time	Low	Fixed
V-RBN-VUL-002	Centralization Risk	Low	Acknowledged
V-RBN-VUL-003	Discrepancies between the constructor and setSt...	Warning	Fixed
V-RBN-VUL-004	Variables must be within appropriate ranges	Info	Fixed
V-RBN-VUL-005	Unused event	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-RBN-VUL-001: Cost of _getUnlocked() increases with time

Severity	Low	**Commit**	125d53e
Type	Denial of Service	**Status**	Fixed
File(s)	`src/AevoStaking.sol`		
Location(s)	_getUnlocked()		
Confirmed Fix At	35bffd7		

The `_getUnlocked()` function computes the amount owed to a user by summing over all epochs, from the first valid one to the most recent "unlocked" epoch.

```
function _getUnlocked(address _recipient, uint16 _epoch) internal returns (uint256
    unstakeTotal) {
    uint16 shift = isLocked ? uint16(stakeTime / epochTime) : 0;
    uint16 end = _epoch > shift ? _epoch - shift : 0;
    uint16 cEpoch = 1;

    while (cEpoch <= end) {
        unstakeTotal += balance[_recipient][cEpoch];
        balance[_recipient][cEpoch] = 0;
        cEpoch++;
    }
}
```

Snippet 4.1: Definition of `_getUnlocked()`

The number of iterations in the `while(cEpoch <= end) {...}` loop will increase by one each epoch (around 1 week) in perpetuity, making unstaking more and more expensive with time.

Impact Unstaking funds may eventually become prohibitively expensive. At the current rate of 1 epoch per week, this is likely not going to lead to denial of service, but will still cost users additional funds.

Recommendation Allow users to supply a `startEpoch` and `endEpoch` to iterate over so that they can reduce the cost of claiming staked funds.

Note that a design change to using a Fenwick Tree could also potentially save gas for these operations.

Developer Response We added start and stop epochs.

4.1.2 V-RBN-VUL-002: Centralization Risk

Severity	Low	**Commit**	125d53e
Type	Access Control	**Status**	Acknowledged
File(s)	`src/AevoToken.sol, src/AevoStaking.sol`		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Similar to many projects, Ribbon's `AevoToken` and `AevoStaking` declare administrator roles that are given special permissions. In particular

- Privileged `AevoToken`'s users may alter the privileges of other users, or mint tokens.
- The `AevoStaking`'s owner may pause the contract, and unlock staked funds.
- The `AevoStaking`'s `bootstrappers` may perform early migration

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious minter could mint a large number of tokens for themselves.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Developer Response We are using multi-sigs to manage permissioned roles.

4.1.3 V-RBN-VUL-003: Discrepancies between the constructor and setStakeTime

Severity	Warning	**Commit**	125d53e
Type	Data Validation	**Status**	Fixed
File(s)	`src/AevoStaking.sol`		
Location(s)	constructor(), setStakeTime()		
Confirmed Fix At	d352c64		

`setStakeTime()` ensures that `stakeTime` is a multiple of `epochTime` and non-zero. Meanwhile, the constructor checks that `stakeTime` should be strictly greater than `epochTime`, which must also be non-zero.

Each of these preconditions exists in one location but not the other.

```
require(_epochTime > 0, "!_epochTime");
require(_stakeTime > _epochTime, "!_stakeTime");
```

Snippet 4.2: Preconditions in the `constructor()`

```
require(_stakeTime > 0 && _stakeTime % epochTime == 0 && _stakeTime < stakeTime, "!
    _stakeTime");
```

Snippet 4.3: Preconditions in `setStakeTime()`

Impact

- It is possible to call the constructor with values that violate the multiplicity constraint (e.g., `epochTime` = 7 days, `stakeTime` = 8 days)
- It is possible to call `setStakeTime` to lower `stakeTime` to be exactly equal `epochTime`.

Recommendation Add the missing checks to both locations.

Developer Response We now consistently check that `_stakeTime` is any non-zero multiple of `epochTime` in both functions.

4.1.4 V-RBN-VUL-004: Variables must be within appropriate ranges

Severity	Info	**Commit**	125d53e
Type	Data Validation	**Status**	Fixed
File(s)	`src/AevoStaking.sol`		
Location(s)	AevoStaking		
Confirmed Fix At	666a75f		

Certain variables in the `AevoStaking` contract must be within appropriate ranges for proper operation of the contract.

1. `epochTime` needs to be large enough that `uint16` can easily store the desired number of total epochs. As can be seen in the below snippet, `epoch()` is stored in a `uint16`.

```
function epoch() public view returns (uint16) {
    return block.timestamp >= start ? uint16((block.timestamp - start) / epochTime +
    1) : 0;
}
```

Snippet 4.4: Definition of `epoch()`.

   If the inner computation `(block.timestamp - start) / epochTime + 1` is large enough, the cast value will silently overflow. The `epochTime` is only validated to be larger than zero.
2. `startTime / epochTime` needs to be small enough that a loop over `startTime / epochTime` is cheap enough to be usable. Loops of length `startTime / epochTime` occur, for instance, in the `staked()` and `staked(address)` functions. See the below snippet for an example.

```
function staked() external view returns (uint256 totalStaked) {
    uint16 cEpoch = epoch();
    uint16 window = 0;

    while (cEpoch - window > 0 && window < stakeTime / epochTime) {
        totalStaked += epochBalance[cEpoch - window];
        window++;
    }
}
```

Snippet 4.5: Definition of `staked()`.

Impact

1. For small enough `epochTimes`, the epoch count may eventually decrease back to `1`. For the current planned setting (`7 days`), this will not be an issue for thousands of years. Users of this contract who wish for smaller epochs, however, should be wary. For example, an epoch length of an hour will only last for 3 years before epochs begin to repeat.
2. The current setting for `startTime / epochTime` is around 12. This is small enough to not be prohibitively expensive. Users of this contract who wish for longer staking periods may find the costs to be much higher than expected.

Recommendation

1. Either note this in the documentation, or add a higher minimum value for `epochTime`.
2. Either note this in the documentation, or add a maximum value for `startTime / epochTime`.

Note that, as mentioned in V-RBN-VUL-001, a design change to using a Fenwick Tree could potentially provide gas savings.

Developer response We added documentation mentioning these facts.

4.1.5 V-RBN-VUL-005: Unused event

Severity	Info	**Commit**	125d53e
Type	Maintainability	**Status**	Fixed
File(s)	`src/AevoStaking.sol`		
Location(s)	event Reward		
Confirmed Fix At	7b5c42a		

In the `AevoStaking` contract, the `Reward` event is unused.

Impact Off-chain listeners may listen on the wrong event.

Recommendation Remove the unused event.

Developer Response We removed the event.

Glossary

flash loan A loan which must be repaid in the same transaction, typically offered at a much more affordable rate than traditional loans . 5

front-running A vulnerability in which a malicious user takes advantage of information about a transaction while it is in the mempool. 5

OpenZeppelin A security company which provides many standard implementations of common contract specifications. See `https://www.openzeppelin.com`. 1

reentrancy A vulnerability in which a smart contract hands off control flow to an unknown party while in an intermediate state, allowing the external party to take advantage of the situation. 5

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure.. 1, 15

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See `https://docs.soliditylang.org/en/v0.8.19/` to learn more. 5