Al-Qa'qa’

Venice

Auditing Report

Auditor: Al-Qa'qa’

12 January 2025

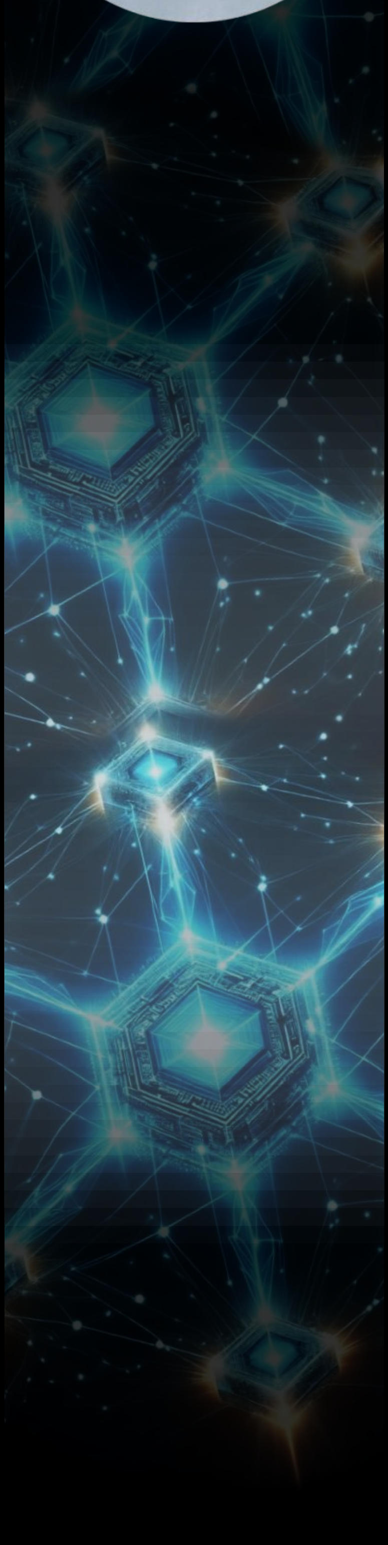


Table of Contents

1 Introduction

1.1 About Al-Qa'qa'

Al-Qa'qa' is an independent Web3 security researcher specializing in smart contract audits. Success in placing top 5 in multiple contests on code4rena and sherlock. In addition to smart contract audits, he has moderate experience in core EVM architecture, geth.

For security consulting, reach out to him on Twitter - @Al_Qa_qa

1.2 About Venice

Venice is Staking Protocol, which allows users to stake there Venice ERC20 token. They get a constant rate of there deposited amount. Staking rewards are destriputed instantaneously, where users can claim rewards for a short period of staking even if less than one hour.

1.3 Disclaimer

Security review cannot guarantee `100%` the protocol's safety. In the Auditing process, we try to identify all possible issues, and we can not be sure if we missed something.

Al-Qa'qa' is not responsible for any misbehavior, bugs, or exploits affecting the audited
code or any part of the deployment phase.

And change to the code after the mitigation process, puts the protocol at risk, and should be audited again.

1.4 Risk Classification

Severity	Impact:High	Impact:Medium	Impact:Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.4.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.4.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align or little-to-no incentive

2 Executive Summary

2.1 Overview

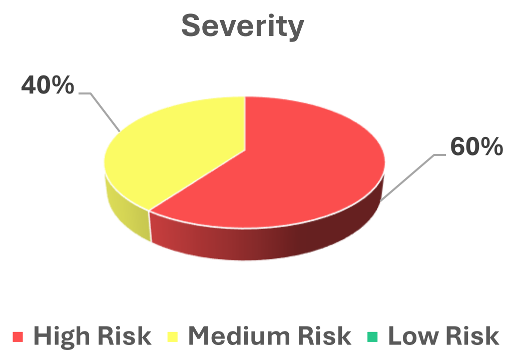
Project	**Venice**
Repository	Venice (Private)
Commit Hash	cbb7b6fcd8da67630c5830640f9579acfdd71e83
Mitigation Hash	d81860697c3c42d686bb34546cccc85dea4180f4
Audit Timeline	04 Jan 2025 and 05 Jan 2025

2.2 Scope

- src/Oracle.sol
- src/Staking.sol
- src/Venice.sol
- script/Deploy.s.sol

2.3 Issues Found

Severity	**Count**
High Risk	3
Medium Risk	2
Low Risk	0



3 Findings Summary

ID	Title	Status
H-01	Rewards Distribution is Implemented totally incorrect	Fixed
H-02	Staking Contract Can't be upgraded in the Future	Fixed
H-03	Preventing Transferring also prevents minting and burning	Fixed
M-01	Accumilate Reward Rate is not synced with utilization rate	Acknowledge
M-02	Deployment Script uses Dummy values for Ownership, Treasury and Relayer	Acknowledge
I-01	Gap array is not needed	Acknowledge
I-02	naming `_update()` for updating accumulative rate is conflicting with ERC20 `_update()` overridden	Fixed
I-03	Unchecking transfer returned value when transferring ERC20 token	Acknowledge

4 Findings

4.1 High Risk

4.1.1 Rewards Distribution is Implemented totally incorrect

Description
In the `Stake.sol` each user's rewards are getting distributed using accumulated rewards claimed.

Stake::_harvest()

```
    function _harvest(address user) internal {
        ...

        // ------------------------
        // 2. Harvest user’s rewards
        // ------------------------
        uint userAccumulated    = (balanceOf(user) * accRewardPerShare) / 1e18;
>>      uint pending            = userAccumulated - stakes[user].rewardDebt;
        stakes[user].rewardDebt = userAccumulated;

        if (pending > 0) {
            venice.transfer(user, pending);
            emit Claimed(user, pending);
        }
    }
```

The problem is that each user starts with balance `0`, so `userAccumulated` will end up being `zero`.

- userA Staked `100` tokens (userAccumulated = zero)
- 7 days has been passed
- userB Staked `100` tokens (userAccumulated = zero)
- 7 days has been passed
- rewards will get distributed equally for both userA and userB equally

This is not the right way to implement `time-weighted`, it should take into consideration the duration the user has been staking in the contract, not the prev rewards he accumulates.

Doing this, will deal with all new users as they are stakers from the beginning of the launch of Staking contract.

This will make an unfair distribution of reward tokens as the one who staked for `1 year` will be the same as the one who just entered staking now.

The issue is more critical than this, as in case of withdrawing funds from some stakers, the Stakeing contract will reach a point, to overpay users, resulting in Contract getting drained with having stakers already do not unstake there rewards.

Proof of Concept

1. Add the following test in `test/Reward.t.sol`

```
    modifier _logTime(string memory user) {
        console2.log(user, "starts staking at:", block.timestamp);
        _;
    }

    function test_auditor_multipleStakes_differentPeriods() public
        _logTime("alice")
        _startPrank(alice)
        _deal      (alice, 100e18)
        _stake     (alice, 100e18)
        _warp      (block.timestamp + staking.COOLDOWN_DURATION())

        _logTime("bob")
        _startPrank(bob)
        _deal      (bob, 100e18)
        _stake     (bob, 100e18)
        _warp      (block.timestamp + staking.COOLDOWN_DURATION())

    {
        console2.log("Current timestamp:", block.timestamp);
        console2.log("Alice Rewards:", staking.pendingRewards(alice));
        console2.log("Bob Rewards:",staking.pendingRewards(bob)  );
    }
```

2. Import `console2` lib from forge

```
import {console2} from "forge-std/console2.sol";
```

3. Run `make` then `forge test --mt test_auditor_multipleStakes_differentPeriods -vv`

```
make
forge test --mt test_auditor_multipleStakes_differentPeriods -vv
```

Output

```
 alice starts staking at: 1
  bob starts staking at: 604801
  Current timestamp: 1209601
  Alice Rewards: 399413484591149771230200
  Bob Rewards: 399413484591149771230200
```

Recommendations:
Instead of depending on user Deposits, each user should have a checkpoint represents `accRewardPerShare` value at the time of his entrance, and the rewards are calculated by the difference between accumulated rewards checkpoints.

4.1.2 Staking Contract Can't be Upgraded in the Future

Description
Staking Contracts are intended to be upgradable contracts. But The logic for Upgradability is not existed either in Proxy or implementation contract.

There are two types of Upgradable proxy, Transparent Upgradable Proxies, and Universal Upgradable proxies (UUPS).

Transparent Upgradable Proxies implements the upgrading logic in The Proxy contract, while UUPS implements the upgrading logic in the implementation itself. Since our contract is intended to Be used as UUPS, upgrading logic should exist in the implementation contract.

The Staking Contract does not inherit `UUPSUpgradeable`, which means that we can't call `upgadeToAndCall()` or `upgradeTo()` to upgrade our staking contract.

Staking.sol

```
contract Staking is IStaking, ERC20Upgradeable, OwnableUpgradeable {
  ...
}
```

Upgrading The Proxy implementation script will revert this case.

```
    function upgradeProxyTo(address proxy, address newImpl, bytes memory data)
internal {
        Vm vm = Vm(Utils.CHEATCODE_ADDRESS);

>>      bytes32 adminSlot = vm.load(proxy, ADMIN_SLOT);
        if (adminSlot == bytes32(0)) {
            string memory upgradeInterfaceVersion =
getUpgradeInterfaceVersion(proxy);
            if (upgradeInterfaceVersion.toSlice().equals("5.0.0".toSlice()) ||
data.length > 0) {
>>              IUpgradeableProxy(proxy).upgradeToAndCall(newImpl, data);
            } else {
                IUpgradeableProxy(proxy).upgradeTo(newImpl);
            }
        } else {
            address admin = address(uint160(uint256(adminSlot)));
            string memory upgradeInterfaceVersion =
getUpgradeInterfaceVersion(admin);
            if (upgradeInterfaceVersion.toSlice().equals("5.0.0".toSlice()) ||
data.length > 0) {
                IProxyAdmin(admin).upgradeAndCall(proxy, newImpl, data);
            } else {
                IProxyAdmin(admin).upgrade(proxy, newImpl);
            }
        }
    }
```

Since our Proxy is UUPS `ERC1967` it do not have an Admin Slot, so this will result in calling `proxy::upgradeTo*()` which will delegate call it to the old implementation which do not implement the function resulting in revert.

This will prevent upgrading Staking Contract.

Recommendations

Make `Stake` contract inherit OpenZeppelin `UUPSUpgradeable` contracts, with initializing it, and don't forge overriding `_authorizeUpgrade()` function to be restricted with Owner.

4.1.3 Preventing Transfereing also prevents minting and burning

Description
Because of transfering Shares problem, which could result in Incorrect reward calculations. The developers choice was to prevent transferring of assets, this is done by overriding `_upgrade()` function from OpenZeppelin ERC20 contract.

```
    function _update(address from, address to, uint value) internal override
virtual {
        if (from != address(0) && to != address(0)) {
            revert(Errors.NOT_TRANSFERRABLE);
        }
    }
```

The problem here is that `_update()` function is used for changing balances, it is using in transferring as well as minting and burning.

In minting `from` is `address(0)` and in burning `to` is `address(0)` so it seems the check will pass for minting and burning but in transferring it will revert. But the problem is that we override the function without implementing its old logic, the new `_update()` function now only checks for `from` and `to` without doing anything. This will make all users balance as well as total supply is `0` as there is no updating occur to the ERC20 (Shares), making the protocol do nothing.

Recommendations
Call the prev `_update()` function after checking `from` and `to` addresses.

```
    function _update(address from, address to, uint value) internal override
virtual {
        if (from != address(0) && to != address(0)) {
            revert(Errors.NOT_TRANSFERRABLE);
        }
+       super._update(from, to, value);
    }
```

4.2 Medium Risk

4.2.1 Accumulate Reward Rate is not synced with utilization rate

Description
Staking Contract distributes rewards into two parties, treasury and stakers. The percentage going for each of them is determined by Oracle utilization rate.

Stake::_harvest()

```
    uint minted = timeElapsed * EMISSION_RATE_PER_SEC;
    if (minted > 0) {
>>      uint venicePortion = (minted * oracle.utilizationRate()) / 1e18;
>>      uint stakerPortion = minted - venicePortion;

>>      if (venicePortion > 0) { venice.mint(treasury, venicePortion); }

        if (stakerPortion > 0 && totalSupply() > 0) {
>>          venice.mint(address(this), stakerPortion);
            accRewardPerShare += (stakerPortion * 1e18) / totalSupply();
        }
    }
```

Utilization rate is changed by Oracle contract and it can be changed every 24 hours.

The problem is that the `venice` to be minted is not minted unless one staker either stake or initialize an unstake operation. and in case of one did that operation the `minted` value, the number of tokens to mint will be calculated for that hole period which can have different utilization rates.

Proof of Concept
Lets say each day we distribute `100` tokens.

- utilization rate is `0.5`
- 1 day passed (treasury +50 tokens, stakers + 50 tokens)
- utilization rate is `0.75
- 1 day passed (treasury +75 tokens, stakers + 25 tokens)
- userA calls `stake()`
- minted = 200 tokens (100 token per day)
- venicePortion = 200 * 0.75 = 150 tokens (Should be 75 + 50 = 125)
- stakerPortion = 200 * 0.25 = 50 tokens (Should be 50 + 75 = 75)

The potions will be distributed using only the last value of the utilization rate for all periods. Even if the utilization rate changes several times during this period, only the last value will be used to calculate the total portion reserved for Venice and staker.

Recommendations
The time of changing utilization rate, should change accumilateRewardRate. All the logic of updating the global state should be made in a separate function. When changing the utilization rate from Oracle, Oracle should call this function to update the global rewards distributions for both treasury and stakers.

NOTE: anyone can call the function, as it will just update the global accumulate reward rate and distribute rewards. This will make gas costs increase when calling `setUtilizationRate()` which is callable each 24 hour

4.2.2 Deployment Script uses Dummy values for Ownership, Treasury and Relayer

Description
The Deployment Script deploys Oracle, Venice, and Staking contracts. These contracts have an Owner and treasury contract, as well as the Oracle Relayer, which should have real values.

Currently, the values used for deploying are dummy values.

lob/Params.sol

```
library Params {
>>  address constant OWNER     = address(99); // TODO: set to real owner
>>  address constant TREASURY  = address(77); // TODO: set to real treasury
>>  address constant RELAYER   = 0x96564db7Ac639129fdAe30028E714029e4331484;
// TODO: set to real relay
```

```
    address constant BASE_ORACLE = 0x95ba99aa87627E51623ca3e0125B538AcFc93ADD;
}
```
This will make the Deploying process go for these incorrect values, resulting in an incorrect deploying process.

Recommendations
Use real value before deploying the Protocol Contracts

4.3 Informational Findings

4.3.1 Gap array is not needed

Description
The staking contract is upgradable, having variables. Since the contract can be upgraded in the future, having a gap for the variables to be added is used in case the contract inherits from contracts implementing variables.

If `contractA` is upgradeable and inherits from `contractB`, and `contractB` and `contractA` has some variables, then having a gap for `contractB` is crucial, so in case of adding a new variable in `contractB`, it is not getting collision with the first variable in `contractA`.

Since Staking contract uses the OpenZeppelin upgradable packages and is not inherited from any contract, having the gap is not needed.

Recommendations
Remove gap array

4.3.2 naming `_update()` for updating accumilative rate is conflicting with ERC20 `_update()` overridden

Description
When updating the global `accRewardPerShare`, the function name for doing this is `_update()`.

This nameing is not suitable and not represent the exact task for the function, in addition of having another `_update()` function which is the function for ERC20 tokens.

```
    function _update(address from, address to, uint value) internal override
virtual {
        // Internal OZ::ERC20 update function
    }
// ----------------
    function _update() internal {
        // Function for updating `accRewardPerShare` and distribute rewards
    }
```

This is not the way of using Polymorphism as the functions with the same name with different parameters should do similar task or the same task with different inputs, but the two functions are totally different.

Recommendations
rename the function to `_updateGlobalReward()` is better, or any name that represents its functionality

4.3.3 Unchecking transfer returned value when transferring ERC20 token

Description
The Staking contract uses Venice staking token, The operations for transferring tokens is not checking the returned boolean value
for `transfer()` and `transferFrom()` operations.

```
    function finalizeUnstake() external {
        ...
>>      venice.transfer(msg.sender, amount);
        ...
    }
// ----------------
    function _harvestUser(address user) internal {
        uint pending = pendingRewards(user);
        if (pending > 0) {
>>          venice.transfer(user, pending);
            emit Claimed(user, pending);
        }
    }
// ----------------
    function stake(address recipient, uint amount) external {
        ...
>>      venice.transferFrom(msg.sender, address(this), amount);
        ...
    }
```

Checking the returned value when transferring ERC20 is important to ensure the transferring process is made correctly.

Recommendations
Check the returned value, and revert if it returns `false`