Venice Staking Security Review

Audit performed by: **zanderbyte**

Completed on: January 05, 2025

Table of Contents

1 About zanderbyte

I am an independent smart contract security researcher specializing in the security of decentralized applications. My focus is on identifying vulnerabilities and providing actionable recommendations to ensure the safe deployment of your project. I've achieved top placements in multiple audit competitions and aim to deliver reliable, high-quality security reviews. Feel free to reach out to discuss how I can help secure your project. Contact me on Telegram @zanderbytexyz or X @zanderbyte

2 Disclaimer

A smart contract security review is a time-limited process focused on identifying vulnerabilities within the given timeframe. While we work to uncover as many issues as possible, we cannot guarantee 100% security after the audit. Audits can reveal the presence of vulnerabilities, but they cannot guarantee their absence. Often, a single audit isn't sufficient, as even small changes in the code can introduce critical vulnerabilities. Mitigation reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

3 About Venice Staking

Venice Staking is a decentralized protocol that enables users to stake Venice tokens and earn rewards. A total of 14,000,000 tokens are allocated for staking rewards on a yearly basis. The protocol is designed to ensure efficient staking with transparent reward distribution.

4 Risk Classification

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

Impact - The potential technical, economic, and reputational damage from a successful exploit.

Likelihood - The probability that a vulnerability could be discovered and exploited.

Severity - The overall criticality of the risk based on impact and likelihood.

4.1 Impact

- **High** - Severe consequences for the project, including loss of funds, data breaches, or reputation damage.
- **Medium** - Significant but not catastrophic consequences. May affect functionality or cause financial loss.
- **Low** - Minor issues that do not affect core functionality or could be easily mitigated without major impact.

4.2 Likelihood

- **High** - The vulnerability is likely to be discovered and exploited in the near future.
- **Medium** - The vulnerability is somewhat likely to be discovered and exploited, but it's less probable.
- **Low** - The vulnerability is unlikely to be discovered or exploited, requires many assumptions or low incentive for the attacker.

4.3 Actions required by severity level

- **High** - **Must** fix the issue(before deployment if not already deployed)
- **Medium** - **Should** fix the issue
- **Low** - **Could** fix the issue

5 Security Assessment Summary

This audit evaluated the smart contract code for Venice Staking. During the assessment, all identified security and functionality issues were highlighted, along with recommendations to address vulnerabilities based on their severity.

5.1 Scope

Review commit hash: 83a29ec687625dd6df9ef532c19d3755c6a35859

The following smart contracts were in the scope of the audit:

- Oracle.sol
- Staking.sol
- Venice.sol

6 Findings

6.1 High Risk

6.1.1 [H-1] Staking contract can be drained due to the usage of `balanceOf`

- **Severity: High**
- **Location: Staking.sol-#L156 Staking.sol-#L85**
- **Description:** The `_harvest` function calculates user rewards based on `balanceOf(user)`, which only reflects the current balance of the user's staked tokens at the time of calculation. This design flaw allows users to manipulate rewards by transferring staked tokens between multiple addresses, enabling them to claim rewards repeatedly for the same tokens.
- **Exploit scenario:**

1. A user stakes tokens (e.g. 10,000 tokens)
2. The user waits until staking contract accumulates rewards.
3. The user calls `claimRewards()` and succesfully claims the accumulated rewards.
4. The user then transfers the staked tokens to a new address which has a `rewardDebt` of 0 (indicating no rewards claimed yet).
5. The user calls `claimRewards()` from the new address and claims the same accumulated rewards.
6. This process can be repeated continuously, draining the rewards pool by allowing the user to claim rewards for the same tokens multiple times.

- **Recommendation** Instead of relying on `balanceOf(user)` for reward calculations, introduce an internal record of each user's staked balance in the contract. This ensures that the contract can track the actual amount of tokens staked by each user and prevent the manipulation described in the exploit scenario.

```
    struct StakeInfo {
       uint rewardDebt;
       uint cooldownStart;
       uint cooldownAmount;
+       uint amount;
    }
```

Modify all logic that currently uses `balanceOf(user)` to use `stakes[user].amount`.

6.1.2 [H-2] Reward logic is fundamentally flawed resulting in excessive rewards and DoS stakes

- **Severity: High**
- **Location: Staking.sol-#L69-#L76 Staking.sol-#L130-#L164**
- **Description:** The `_harvest` function is used for both calculating and distributing rewards in the staking contract. However, this implementation is incorrect and leads to several significant issues: *1. Immediate reward claiming:* The contract allows users to stake tokens and immediately

claim rewards, even if no rewards have been accumulated for them. This behavior allows users to game the system and claim rewards disproportionate to their stake or staking duration. The following PoC demonstrates this issue:

```
    function test_stakeAndGainReward() public
    _deal(alice, 1000e18)
    _startPrank(alice)
{
    venice.approve(address(staking), 1000e18);
    staking.stake(alice, 1000e18);
    vm.stopPrank();

    skip(1 hours);
    deal(address(venice), bob, 1000e18);

    vm.startPrank(bob);
    venice.approve(address(staking), 1000e18);
    staking.stake(bob, 1000e18);

    emit log_named_uint("Bob's reward before claim", venice.balanceOf(bob));
    staking.claimRewards();
    emit log_named_uint("Bob's reward after claim", venice.balanceOf(bob));
}
```

Logs:

```
    [PASS] test_stakeAndGainReward() (gas: 560288)
    Logs:
    Bob's reward before claim: 0
    Bob's reward after claim: 1598173515981735159000
```

This result indicates that Bob, who staked 1000 tokens, immediately received 1600 tokens as rewards in the same block. *2. Staking DoS:* The `_harvest` function is invoked during staking, but it incorrectly performs reward calculations and distributions. This can cause inefficiencies, misallocation of rewards and underflow issues, leading to a denial of service for users who attempt to stake multiple times. This behavior is incorrect, as staking events should only update reward variables (e.g. `accRewardPerShare`, `lastRewardTimestamp`, `rewardDebt`) without distributing rewards. This issue is demonstrated in the following test, which fails with an underflow error:

```
    function test_stakeMultipleTime() public
        _deal      (alice, 300e18)
        _startPrank(alice)
    {
        venice.approve(address(staking), 300e18);
        staking.stake(alice, 100e18);

        skip(1 days);
        staking.stake(alice, 100e18);

        skip(1 days);
        staking.stake(alice, 100e18);
    }
```

Failure:

```
Encountered 1 failing test in test/Stake.t.sol:Stake_Test
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11)]
    test_stakeMultipleTime() (gas: 431625)
```

- **Recommendation**

1. The `_harvest` function should be split into separate functions to handle distinct tasks (e.g., reward calculation, reward distribution, and updating reward variables). This will make the logic more efficient, modular, and avoid issues related to incorrect reward distributions during staking. Refer to well-known protocols like MasterChef and Synthetix for guidance. These protocols separate the logic of updating reward variables and distributing rewards.
2. Refactor the staking logic to only update reward related variables during staking and avoid distributing rewards. This can be done by updating only the accRewardPerShare, lastRewardTimestamp, and rewardDebt when a user stakes. The rewards should be distributed separately via the `_harvest` function, which can be called at a later time. Example fix:

```
    function stake(address recipient, uint amount) external {
        require(amount > 0, Errors.STAKE_ZERO);
        updatePoolRewards();

        stakes[recipient].rewardDebt = (stakes[recipient].amount * accRewardPerShare
            ) / 1e18;
        stakes[recipient].amount += amount;
        _mint(recipient, amount);
        totalStaked += amount;
        venice.transferFrom(msg.sender, address(this), amount);
    }

    function updatePoolRewards() internal {
        if (totalStaked == 0) {
            lastRewardTimestamp = block.timestamp;
            return;
        }

        uint timeElapsed = block.timestamp - lastRewardTimestamp;
        uint rewards = timeElapsed * EMISSION_RATE_PER_SEC;

        venice.mint(address(this), rewards);

        accRewardPerShare += (rewards * 1e18) / totalStaked;
        lastRewardTimestamp = block.timestamp;
    }
```

6.2 Medium Risk

6.2.1 [M-1] First stakers can highly inflate the `accRewardPerShare`

- **Severity: Medium**
- **Location: Staking.sol-#L148**
- **Description:** During the first staking events, users can stake very small amounts of tokens, which can disproportionately inflate the `accRewardPerShare` variable. This will lead to tremendeosly inlfated rewards and exceeding the max yearly token emission:

```
  uint public constant YEARLY_EMISSION = 14_000_000 * 10 ** 18; // 14M
```

This issue can be exploited by early stakers who stake small amounts and inflate the `accRewardPerShare`, thereby claiming more rewards than intended.

The following test demonstrates how the `accRewardPerShare` can be inflated:

```
    function test_inflateAccRewardPerShare() public
        _deal      (alice, 100)
        _startPrank(alice)
    {
        venice.approve(address(staking), 1);
        staking.stake(alice, 1);
        vm.stopPrank();
        skip(1 minutes);
        deal(address(venice), bob, 1);
        vm.startPrank(bob);
        venice.approve(address(staking), 1);
        staking.stake(bob, 1);

        console.log("accRewardPerShare: ", staking.accRewardPerShare());
    }
```

Output from running the test:

```
[PASS] test_inflateAccRewardPerShare() (gas: 522581)
Logs:
  accRewardPerShare:  266362252663622526600000000000000000000
```

- **Recommendation**

1. Deploy contract with an initial stake: To prevent the `accRewardPerShare` from being excessively inflated by early stakers, deploy the contract with a small initial stake (e.g., 1 token or a similar low amount). This ensures that the initial reward distribution is set appropriately before any stakers join.
2. Introduce a minimum stake amount: To prevent users from staking very small amounts implement a minimum stake requirement.

6.3 Low Risk

6.3.1 [L-1] Use two-step ownership transfer

- **Severity: Low**
- **Location: Staking.sol**
- **Description:** The `Staking.sol` contract currently uses `OwnableUpgradeable` to manage ownership. The `OwnableUpgradeable` contract provides functionality to change the ownership to a new address. However, this can lead to potential issues if the admin mistakenly uses an invalid address. If the new address is incorrect or not controlled by the admin, it could lock the system, as the admin would not be able to interact with the contract.
- **Recommendation** It is recommended to implement a two-step ownership transfer process, using `Ownable2StepUpgradeable`:

6.3.2 [L-2] Introduce a max amount for `utilizationRate` in Oracle contract

- **Severity: Low**
- **Location: Oracle.sol**
- **Description:** The `utilizationRate` in the `Oracle` contract is being used in reward calculations for the staking contract. If the `utilizationRate` exceeds 1e18, it could lead to underflow issues when calculating the `stakerPortion` during reward distribution:

```
contract Oracle is Owned(msg.sender) {
    uint public utilizationRate;

    event UtilizationRateUpdated(uint newRate);

    function setUtilizationRate(uint _utilizationRate) external onlyOwner {
        utilizationRate = _utilizationRate;
        emit UtilizationRateUpdated(_utilizationRate);
    }
}
```

```
    uint venicePortion = (minted * oracle.utilizationRate()) / 1e18;
    uint stakerPortion = minted - venicePortion;
```

- **Recommendation** Introduce a validation to ensure that the `utilizationRate` does not exceed 1e18:

```
    function setUtilizationRate(uint _utilizationRate) external onlyOwner {
+       require(_utilicationRate <= 1e18);
        utilizationRate = _utilizationRate;
        emit UtilizationRateUpdated(_utilizationRate);
    }
```

6.3.3 [L-3] Use OZ's `SafeTransferLib`

- **Severity: Low**
- **Location: Staking.sol**
- **Description:** In the contract, the following token transfer operations are performed without utilizing the `SafeTransferLib` from OpenZeppelin: In `stake` function:

```
    venice.transferFrom(msg.sender, address(this), amount);
```

In `finalizeUnstake` function:

```
    venice.transfer(msg.sender, amount);
```

In `_harvest` function:

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

Using the `SafeTransferLib` from `OpenZeppelin` would make these operations safer, helping to prevent issues like failed transfers or unexpected behavior.

- **Recommendation** Refactor the token transfer operations to use OpenZeppelin's `SafeTransferLib` for increased safety.