Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Ankr

Ankr Token Staking

Veridise Inc.
February 21, 2023

► **Prepared For:**

ANKR
https://www.ankr.com/

► **Prepared By:**

Jon Stephens
Xiangan He
► **Contact Us:** contact@veridise.com

► **Version History:**

Feb 21, 2023	V1

Contents

Executive Summary 1

From Feb. 13 to Feb. 20, ANKR engaged Veridise to review the security of their Ankr Token Staking protocol. The review covered the on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 weeks on commit `57a718c`. 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 Ankr Token Staking protocol is an upgradable contract that is already deployed on mainnet. The new version of the protocol, which is the focus of this audit, streamlines the previous version of the contract. To do so, it removes validator punishment mechanisms and places the validator maintenance in the hands of governance so that they are the only ones that may add or remove validators. Similar to the previous version, delegators can then stake their funds with validators to receive a share of the rewards given to validators. Those funds are locked by the contract for a certain staking period of time, after which a user can undelegate to receive their funds back along with any unclaimed rewards. Since the contract will be upgraded to this new version, the developers also include migration code to migrate state from the old contract to the new one.

ANKR provided the source code for the Ankr Token Staking protocol for review. In addition, they provided a set of tests based on the Truffle testing framework.

Summary of issues detected. The audit uncovered 13 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-ATS-VUL-001 identifies a logic error in the migration logic that could prevent users from withdrawing funds and V-ATS-VUL-002 identifies a logic error that could allow users to claim rewards multiple times. In addition, the auditors identified a moderate-severity issue where user funds could be improperly tracked due to an overflow while downcasting (V-ATS-VUL-003). Finally, the auditors identified several other security concerns, including some potentially unsafe functions (V-ATS-VUL-008, V-ATS-VUL-010, V-ATS-VUL-011) and missing validation (V-ATS-VUL-006, V-ATS-VUL-007, V-ATS-VUL-009).

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
Ankr Token Staking	`57a718c`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Feb. 13 - Feb. 20, 2023	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	1	0
Low-Severity Issues	3	3
Warning-Severity Issues	7	5
Informational-Severity Issues	0	0
TOTAL	13	10

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Data Validation	2
Logic Error	2
Overflow	1
Validator Punishment	1
Denial of Service	1
Locked Funds	1
Reentrancy	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Ankr Token Staking protocol defined in the following scope. In our audit, we sought to answer the following questions:

- Can users steal funds from the pool?
- Will users be paid upon unstaking?
- Can rewards be claimed multiple times?
- Is there a method to punish malicious validators?
- Can a delegator unstake before the lock period has elapsed?
- Can funds be locked within the contract?

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, as well as the open-source tool Slither. 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 audit reviewed the on-chain behaviors of the Ankr Token Staking protocol, including delegator and validator migration, validator management and delegator staking. The Veridise engineers first inspected the provided documentation to understand the high-level design of the protocol. They then inspected the provided test-cases to better understand the specific contract behavior. Finally, the auditors performed a week long audit of the code assisted both by static analyzers and automated testing. In terms of the audit, the following files were in-scope:

- contracts/protocol/AnkrTokenStaking.sol
- contracts/staking/BaseStaking.sol
- contracts/staking/Staking.sol
- contracts/staking/StakingConfig.sol
- contracts/staking/ValidatorRegistry.sol
- contracts/staking/ValidatorStorage.sol
- contracts/staking/extension/TokenStaking.sol

- contracts/libs/SnapshotUtil.sol
- contracts/libs/ValidatorUtil.sol
- contracts/libs/DelegationUtil.sol
- contracts/libs/Multicall.sol

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:

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:

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., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ATS-VUL-001	Incorrect Credit on Migrate	High	Fixed
V-ATS-VUL-002	Rewards can be claimed multiple times	High	Fixed
V-ATS-VUL-003	Potential Overflow on Downcast	Medium	Open
V-ATS-VUL-004	No Validator Punishment Mechanisms	Low	Intended Behavior
V-ATS-VUL-005	Potential gas DoS	Low	Acknowledged
V-ATS-VUL-006	No Validation for Setters in StakingConfig	Low	Fixed
V-ATS-VUL-007	Potentially Stuck Validator due to Unset Status	Warning	Fixed
V-ATS-VUL-008	DelegateCall Dangerous in Upgradable Contracts	Warning	Open
V-ATS-VUL-009	No status check in Migrate	Warning	Fixed
V-ATS-VUL-010	Potential Validator State Inconsistencies	Warning	Open
V-ATS-VUL-011	TokenStaking does not override _safeTransferTo	Warning	Fixed
V-ATS-VUL-012	Locked Native Tokens	Warning	Fixed
V-ATS-VUL-013	Possible Reentrancy	Warning	Fixed

4.1 Detailed Description of Bugs

4.1.1 V-ATS-VUL-001: Incorrect Credit on Migrate

Severity	High	**Commit**	57a718c
Type	Logic Error	**Status**	Fixed
Files	Staking.sol		
Functions	migrateDelegator		

The following code is used to migrate delegators from the old version of the staking contract to the new version. To do so, it must update added storage state such as `_stakerAmounts` and `_stakerShares`. During the course of this calculation, the `delegated` value is will either be `delegations.delegateQueue[j].amount` or `2 * delegations.delegateQueue[j].amount` depending on the path taken (where `j = delegations.delegateQueue.length - 1`). We believe this value is not computed correctly as it appears that the `_stakerAmounts` is intended to track the sum of the active delegated funds.

```
function migrateDelegator(address delegator) public {
    ...

    uint112 delegated;
    for (uint256 j = delegations.delegateGap; j < delegations.delegateQueue.length;
  j++) {
        if (delegations.delegateQueue[j].amount < delegated) {
            delegated -= delegated - delegations.delegateQueue[j].amount;
            continue;
        } else if (delegations.delegateQueue[j].amount == delegated) {
            continue;
        }
        uint112 realAmount = delegations.delegateQueue[j].amount - delegated;
        delegated += realAmount;
        newDelegations.push(Delegation(delegations.delegateQueue[j].epoch,
 realAmount, uint256(realAmount) * BALANCE_COMPACT_PRECISION, 0));
        delegated += delegations.delegateQueue[j].amount;
    }
    _stakerAmounts[validatorAddress][delegator] += delegated;
    _stakerShares[validatorAddress][delegator] += uint256(delegated) *
 BALANCE_COMPACT_PRECISION;

    ...
}
```

Snippet 4.1: The migration function that over-credits the `_stakerAmounts` and `_stakerShares`

Impact Since the `_stakerAmounts` is intended to track the sum of active delegated funds across the user's history, the value stored in `_stakerAmounts` is likely to be inconsistent with the true value that should be stored. From what we can tell if `_stakerAmounts` is too high, users may be able to withdraw too many funds from the pool if the delegation computation is also incorrect. Alternatively, if `_stakerAmounts` is too small this could prevent users from undelegating all of their funds. As `delegated` is likely to be too large, we believe that this code will likely cause user funds to become locked.

Recommendation Update the logic so that user funds will not be locked.

4.1.2 V-ATS-VUL-002: Rewards can be claimed multiple times

Severity	High	**Commit**	57a718c
Type	Logic Error	**Status**	Fixed
Files	Staking.sol		
Functions	_stashUnlocked		

When a user undelegates, the protocol must determine if sufficient funds have passed their lock period and what rewards have been claimed so far. This information is then used to credit the user with funds that they can withdraw and send the user any pending rewards. However, some paths through the function do not report any rewards have been claimed as shown below.

```
function _stashUnlocked(
    ValidatorPool memory validatorPool,
    address delegator,
    uint112 expectedAmount,
    uint64 beforeEpoch
) internal returns (uint96 claimed, uint256 spentShares) {
    ...

    while(history.delegationGap < delegations.length && delegations[history.
delegationGap].epoch + lockPeriod < beforeEpoch && expectedAmount > 0) {
        if (delegations[history.delegationGap].amount > expectedAmount) {
            // deduct undelegated amount
            uint256 shares = _toShares(validatorPool, uint256(expectedAmount) *
BALANCE_COMPACT_PRECISION);
            delegations[history.delegationGap].amount -= expectedAmount;
            require(delegations[history.delegationGap].shares >=  shares, "
overflow");
            delegations[history.delegationGap].shares -= shares;
            spentShares += shares;
            // expected amount is filled
            expectedAmount = 0;
            // save changes to storage
            storageDelegations[history.delegationGap] = delegations[history.
delegationGap];
            break;
        }
        expectedAmount -= delegations[history.delegationGap].amount;
        claimed += delegations[history.delegationGap].claimed;
        ...
    }

    ...
}
```

Snippet 4.2: Location where claimed rewards are not being calculated

Impact Since the return value of this function is used to determine how many rewards have yet to be paid to the user, an individual can make repeated requests to `undelegate` to receive rewards multiple times.

Recommendation If a single delegation can cover the remaining funds, calculate the amount of claimed funds and reduce the claimed funds appropriately (as otherwise some rewards may be unclaimable)

4.1.3 V-ATS-VUL-003: Potential Overflow on Downcast

Severity	Medium	**Commit**	57a718c
Type	Overflow	**Status**	Open
Files	Staking.sol, ValidatorRegistry.sol		
Functions	Multiple		

To reduce the storage cost, the developers store delegated amounts as multiples of `BALANCE_COMPACT_PRECISION` so that values can fit into a `uint112`. Doing so requires frequent downcasting as values are typically passed in as a `uint256`. Since downcasting can overflow without causing a revert to occur and there may be some left over dust that isn't accounted for, we recommend that the developers validate the given compact version is equivalent to what the user passes in.

```
    function _delegateUnsafe(address validator, address delegator, uint256 amount,
    uint64 sinceEpoch) internal override {
        uint112 compactAmount = uint112(amount / BALANCE_COMPACT_PRECISION);
        // add delegated amount to validator snapshot, revert if validator not exist

        ...
    }
```

Snippet 4.3: Snippet in `_delegateUnsafe` that converts an amount to its compact form

Impact If a user passes in a large value, for amount it is possible that this could cause an overflow, resulting in inaccurate accounting.

Recommendation Add a requirement that the compact representation is equivalent to the original value.

4.1.4 V-ATS-VUL-004: No Validator Punishment Mechanisms

Severity	Low	**Commit**	57a718c
Type	Validator Punishment	**Status**	Intended Behavior
Files	ValidatorRegistry.sol		
Functions	N/A		

Currently, the ValidatorRegistry contract allows the governance to add and activate validators. To enforce the good behavior of Validators, typically there is a mechanism to punish misbehaving or malicious validators by slashing or jailing them. In this contract, all of the slashing and jailing behaviors have been removed.

Impact With no punishment mechanism, validators may misbehave for financial gain.

Recommendation Add a punishment mechanism to the ValidatorRegistry contract.

Developer Response We plan to start with a single trusted validator so we do not need the slashing logic. If we allow other validators to be added in the future we will add this functionality back in.

4.1.5 V-ATS-VUL-005: Potential gas DoS

Severity	Low	**Commit**	57a718c
Type	Denial of Service	**Status**	Acknowledged
Files	Staking.sol		
Functions	migrateDelegator		

The protocol makes frequent use of `for` loops (and in some cases nested loops) over arrays and integer ranges which can lead to expensive gas costs.

```
function migrateDelegator(address delegator) public {
        ...

        for (uint256 i; i < validators.length; i++) {
            ...
            for (uint256 j = delegations.delegateGap; j < delegations.delegateQueue.
    length; j++) {
                ...
            }

            ...

            for (uint256 j = delegations.undelegateGap; j < delegations.
    undelegateQueue.length; j++) {
                ...
            }
            ...
        }

    ...
}
```

Snippet 4.4: Function with nested loops

Impact Such loops can result in prohibitive gas costs rendering certain functions as in-executable by users.

Recommendation If possible, bound the execution of loops.

4.1.6 V-ATS-VUL-006: No Validation for Setters in StakingConfig

Severity	Low	**Commit**	57a718c
Type	Data Validation	**Status**	Fixed
Files	StakingConfig.sol		
Functions	N/A		

The StakingConfig contract stores important configuration information about the system and allows the governance to change these values via setters. However, many of these functions lack validation on the input values.

```
    function setGovernanceAddress(address newValue) external override
    onlyFromGovernance {
        address prevValue = _slot0.governanceAddress;
        _slot0.governanceAddress = newValue;
        emit GovernanceAddressChanged(prevValue, newValue);
    }
```

Snippet 4.5: A setter in the StakingConfig contract that doesn't validate `newValue`

Impact Without input validation, an admin could make a mistake such as setting the governance address to `address(0)`.

Recommendation Perform appropriate validation of input values in the setters. In addition, to prevent the governance address from being transferred to a malicious user or a location that is inaccessible, the developers should consider a propose/accept mechanism. In this case the governance address would propose a new governance address and after an appropriate delay (24 hours for example) the new governance address can accept the appointment. Within the waiting period, the governance address transfer may be canceled.

4.1.7 V-ATS-VUL-007: Potentially Stuck Validator due to Unset Status

Severity	Warning	**Commit**	57a718c
Type	Maintainability	**Status**	Fixed
Files	ValidatorStorage.sol		
Functions	create		

`create` allows the pool to create validators. There is, however, no enforcement that the Validator's new created status is not `NotFound`. If the Validator's new status is set to `NotFound` (either by missing param input to status, or by mistake).

```
    function create(
        address validatorAddress,
        address validatorOwner,
        ValidatorStatus status,
        uint64 epoch
    ) external override onlyFromPool {
        Validator storage self = _validatorsMap[validatorAddress];
        require(self.status == ValidatorStatus.NotFound, "Validator: already exist");
        self.validatorAddress = validatorAddress;
        self.ownerAddress = validatorOwner;
        self.status = status;
        self.changedAt = epoch;

        // save validator owner
        require(validatorOwners[validatorOwner] == address(0x00), "owner in use");
        validatorOwners[validatorOwner] = validatorAddress;

        // add new validator to array
        if (status == ValidatorStatus.Active) {
            activeValidatorsList.push(validatorAddress);
        }
    }
```

Snippet 4.6: The function with no input validation on the status

Impact If `create` was called with the status `NotFound`, then the pool to call `create`, which will create a validator with bad status. Since `validatorOwners[validatorOwner] = validatorAddress`, any subsequent attempts to call `create` again on the validator address with the same owner will fail. In addition, `activate` requires `self.status == ValidatorStatus.Pending` so the validator will never be activated

Recommendation Check that the user passes in a valid status.

4.1.8 V-ATS-VUL-008: DelegateCall Dangerous in Upgradable Contracts

Severity	Warning	**Commit**	57a718c
Type	Data Validation	**Status**	Open
Files	Multicall.sol		
Functions	_fastDelegateCall		

The staking contract is both upgradable and it extends the MultiCall contract. The MultiCall contract allows a user to batch calls to the contract by repeatedly delegated calls to itself with calldata provided by the user. However, OpenZeppelin advises that users avoid the use of delegate call in their documentation on Upgradable contracts as they could potentially be used to destroy the implementation contract.

Impact If a self-destruct could be invoked using the delegate call on the implementation contract, it could prevent users from interacting with the staking contract.

Recommendation To avoid the case were a user could interact with the implementation contract, OpenZeppelin suggests "breaking" the implementation contract in the constructor so that it cannot be used. Their recommendation on how to do so is provided here.

4.1.9 V-ATS-VUL-009: No status check in Migrate

Severity	Warning	**Commit**	57a718c
Type	Maintainability	**Status**	Fixed
Files	ValidatorStorage.sol		
Functions	migrate		

The `migrate` function is used to migrate validators from the old contract version to the new contract version. When doing so the function automatically pushes the validator onto the active validators list but does not check the validator's status to ensure it is active.

```
function migrate(Validator calldata validator) external override onlyFromPool {
    _validatorsMap[validator.validatorAddress] = validator;
    validatorOwners[validator.ownerAddress] = validator.validatorAddress;
    activeValidatorsList.push(validator.validatorAddress);
}
```

Snippet 4.7: Function to migrate validators to new contract version

Impact This could allow a validator with a non-active status to be added to the active validator list, potentially allowing a validator to accidentally be marked as active.

Recommendation Check the validator's status to make sure it is active.

4.1.10 V-ATS-VUL-010: Potential Validator State Inconsistencies

Severity	Warning	**Commit**	57a718c
Type	Maintainability	**Status**	Open
Files	ValidatorRegistry		
Functions	_touchValidatorSnapshot		

The protocol stores snapshots of the validator state over epochs of time. It will then read and update information in these epochs as the validator state changes. Certain values, however, are intended to be maintained across epochs such as the `totalDelegated` and `commissionRate`. The API below, however, allows developers to request and modify in previous epochs without changing the ones that come after.

```
function _touchValidatorSnapshot(Validator memory validator, uint64 epoch)
internal returns (ValidatorSnapshot storage) {
    ValidatorSnapshot storage snapshot = _validatorSnapshots[validator.
validatorAddress][epoch];
    // if snapshot is already initialized then just return it
    if (snapshot.totalDelegated > 0) {
        return snapshot;
    }
    // find previous snapshot to copy parameters from it
    ValidatorSnapshot memory lastModifiedSnapshot = _validatorSnapshots[validator
.validatorAddress][validator.changedAt];
    // last modified snapshot might store zero value, for first delegation it
might happen and its not critical
    snapshot.totalDelegated = lastModifiedSnapshot.totalDelegated;
    snapshot.commissionRate = lastModifiedSnapshot.commissionRate;
    // we must save last affected epoch for this validator to be able to restore
total delegated
    // amount in the future (check condition upper)
    if (epoch > validator.changedAt) {
        _validatorStorage.change(validator.validatorAddress, epoch);
    }
    return snapshot;
}
```

Snippet 4.8: The function used to find validator state at a given epoch for modification.

Impact If a user changes values in an epoch before `validator.changedAt` those values will not be reflected in the most recent epoch. Since it appears that certain values, such as `totalDelegated`, are intended to be consistent across epochs such modifications could result in inconsistent states and locked funds (i.e. `totalDelegated` in some epoch X should be the same in X+1 unless a user undelegated those funds). Note that this could also be used to copy recent changes into a previous epoch but this only appears to impact the frontend.

Recommendation It appears that this API is currently used almost exclusively to get the next epoch. In the one case where this does not occur, the modified value does not appear to be tracked across epochs (`totalRewards`). To prevent future errors, we would recommend checking that only states at or more recent than `changedAt` are modified.

4.1.11 V-ATS-VUL-011: TokenStaking does not override _safeTransferTo

Severity	Warning	**Commit**	57a718c
Type	Maintainability	**Status**	Fixed
Files	TokenStaking.sol		
Functions	_safeTransferTo		

The `TokenStaking` contract overrides the behavior of `Staking` so that ERC20 tokens can be staked instead of native tokens. To do so, it overrides the payment methods used by `Staking` so that ERC20 payments are performed instead. The `TokenStaking` contract, however, only overrides two of the 3 native payment methods as the `_safeTransferTo` function still performs native payments.

Impact Since `_safeTransferTo` is never used, this doesn't impact the current version of the protocol. If future changes to the contract do call this function, however, users may be paid using the wrong currency.

Recommendation Override `_safeTransferTo` in `TokenStaking` so that it also pays users ERC20 tokens.

4.1.12 V-ATS-VUL-012: Locked Native Tokens

Severity	Warning	**Commit**	57a718c
Type	Locked Funds	**Status**	Fixed
Files	TokenStaking.sol		
Functions	receive		

The `TokenStaking` contract overrides the behavior of `Staking` so that ERC20 tokens can be staked instead of native tokens. This contract still accepts native tokens though due to the definition of `receive` shown below in the Staking contract.

```
    receive() external payable {
    }
```

Snippet 4.9: The `receive` function defined by `Staking` and not overriden by `TokenStaking`

Impact Since `TokenStaking` does not provide support for native tokens, any tokens sent to this contract will be locked in the contract.

Recommendation Override `receive` in TokenStaking so that funds are rejected or add in functionality to rescue such tokens.

4.1.13 V-ATS-VUL-013: Possible Reentrancy

Severity	Warning	**Commit**	57a718c
Type	Reentrancy	**Status**	Fixed
Files	Staking.sol		
Functions	migrateDelegator		

The Staking contract migrates users from the old version of the contract to their new version with the function `migrateDelegator`. This function processes the old state of the contract and updates delegations in the new contract state. While doing so, the migration process will transfer any unclaimed rewards to the user with the function shown below. While it is likely that the native version of the `_safeTransferWithGasLimit` would not succeed since it strictly limits gas and the `migradeDelegator` function is gas-intensive, it is possible that the ERC20 version of the API could reenter. This is because some ERC20 tokens, such as those that adhere to the ERC777 specification (which extends ERC20) introduce an unsafe callback in `transfer` and `transferFrom`.

```
function _transferDelegatorRewards(address validator, address delegator) internal
 {
    // next epoch to claim all rewards including pending
    uint64 beforeEpochExclude = _MIGRATION_EPOCH;
    // claim rewards and undelegates
    uint256 availableFunds = _processDelegateQueue(validator, delegator,
 beforeEpochExclude);
    // for transfer claim mode just all rewards to the user
    _safeTransferWithGasLimit(payable(delegator), availableFunds);
    // emit event
    emit Claimed(validator, delegator, availableFunds, beforeEpochExclude);
 }
```

Snippet 4.10: Function used to send unclaimed rewards to users during the migration process

Impact If a reentrancy were to occur here, someone could receive rewards multiple times.

Recommendation Since `migrateDelegator` must either execute fully or revert, move the statement `isMigratedDelegator[delegator] = true;` to right after the `isMigratedDelegator` check.