PUBLIC

Code Assessment

of the ETH B2C Staking Smart Contracts

August 17, 2023

Produced for

everstake

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Everstake team,

Thank you for trusting us to help Everstake with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of ETH B2C Staking according to Scope to support you in forming an opinion on their security risks.

Everstake implements a pooled staking service for Ethereum, where the rewards are reinvested in the pool and the validators are managed by Everstake.

The most critical subjects covered in our audit are the correctness of the accounting, asset solvency, access control and functional correctness. During the audit, the most important reported issues were:

- Replacing a Validator Eventually Blocks the System
- Usage of address(this).balance in restake Can Block the System that requires from Everstake to inject liquidity to correct the accounting in case of necessity.

The issues have been fixed during the second week of the audit.

Security regarding all the aforementioned subjects is satisfactory. Even though the probability of one of the validators getting slashed is low, slashing could occur. That would require manual, trust-based intervention, see Slashing is not taken into account and Trust Model.

The general subjects covered are documentation, unit testing, code complexity, and gas efficiency. Documentation has been greatly improved during the last iteration. Unit testing and testing in general is basic, a good test suite will help ensure corner cases are considered.

In summary, we find that the codebase provides a satisfactory level of security, provided the Trust Model.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	6
• Code Corrected	5
• Specification Changed	1
Low-Severity Findings	25
• Code Corrected	22
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the ETH B2C Staking repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	27 June 2023	ddde802c194eecbd1a8670eb3be9629a5e700722	Initial Version
2	05 July 2023	7b10ca452199f5a0e5fbd7596d72334582a48b32	Version 2
3	02 August 2023	8d3c73007da91da2d20dfb66524cedb74d10f80b	Version 3
4	08 August 2023	35f9b56b038be82a31946bd6b02533ec16ddd228	Version 4

For the solidity smart contracts, the compiler version `0.8.19` was chosen.

The following contracts are in the scope of the review:

```
common:
    Errors.sol

interfaces:
    IAccounting.sol
    IDepositContract.sol
    IPool.sol
    IRewardsTreasury.sol
    ITreasuryBase.sol

lib:
    UnstructuredRefStorage.sol
    UnstructuredStorage.sol

structs:
    ValidatorList.sol
    WithdrawRequests.sol

utils:
    Math.sol
    OwnableWithSuperAdmin.sol

Accounting.sol
AutocompoundAccounting.sol
Governor.sol
Pool.sol
RewardsTreasury.sol
TreasuryBase.sol
Withdrawer.sol
WithdrawTreasury.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly listed above and third-party libraries are out of the scope of this review.

3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Everstake offers a staking service for Ethereum that allows stakers to deposit amounts that can be less than `32ETH`, the beacon amount, in a pool. The system also restakes the staking rewards.

ETH B2C Staking uses a smart way of exchanging incoming liquidity for outgoing liquidity, called interchange. Upon a withdrawal request from an activated validator that is less than `32ETH`, the requested amount is added to an exchange buffer. Now any incoming liquidity will first be exchanged in this buffer, allowing the exiting party to withdraw without closing the validator, and the entering party to have part of its stake instantly active in a validator.

The system works in rounds, the active round indicates the round that is currently being filled, and the activated round describes the next round for which a validator is expected to be activated. The active round is filled until the pending amount reaches `32ETH`, then a new active round starts. Once a validator has been activated on the beacon chain, Everstake will increment the activated round in the system.

The amounts that are due to the users are represented by shares. The shares are minted to users only when their stake becomes active in the beacon chain, either by interchange or by the activation of a new validator on the beacon chain.

The stakes provided by users can be in three states:

1. Pending: the stake is waiting in the `Pool`.
2. Deposited: the stake has been sent to a validator, but the validator is not yet active on the beacon chain.
3. Active: the stake is in a validator and the validator is active on the beacon chain.

The system is divided into three parts:

1. Pool: the main entry point to the system that holds the pending assets.
2. Accounting: takes care of the accounting for the stakers in a pool.
3. Treasuries: the reward treasury receives the staking rewards and the returned stake of closed validators, the withdraw treasury holds the amount that will be withdrawn.

3.1 Pool

The `Pool` holds a list of validators that are managed by Everstake and can be used when the pending amount in the `Pool` reaches `32ETH`. When it does, a validator is consumed from the list and the beacon deposit contract is called with the validator's public key, signature, and deposit data root along with the address of the rewards treasury encoded in the withdrawal credentials.

The `Pool` contract is the main entry point of the system, it offers the following functions for the users:

- `stake`: called along with ETH, the `msg.value` must be at least the minimum stake amount. The function will first reinvest any rewards (`Accounting._autocompound`) and deposit the caller's stake, or part of it, either in the withdraw treasury if some amount can be interchanged, in the pending buffer if the total pending amount is not sufficient to launch a new validator, or in one or more yet inactive validators if the sum of the user's value and the pending amount can cover the beacon amount or more. If some of the stake can be interchanged, the associated shares are instantly minted to the new staker. Can only be called when staking is not paused.
- `unstakePending`: a staker whose stake is still pending (not sent to a validator) can use this function to reduce its stake. This function will first update the internal accounting for the rewards treasury (`Accounting._update`), then reduce the pending stake of the caller and send back the requested value. Can only be called when withdrawals are not paused.
- `unstake`: a staker who has shares can use this function to initiate a withdrawal of their active stake. The function will first reinvest any rewards (`Accounting._autocompound`), then if some pending amount is available it will be interchanged as much as possible and the remaining value, if any, will be added to a withdrawal request. The withdrawal request will request the closing of one or more validators if the requested amount exceeds one or more beacon amounts and Everstake will be notified. The remaining amount will be made available for interchange, and the request will be added to a withdrawal request queue. The user can claim the withdrawal as soon as the requested amount is available in the withdraw treasury. If a user initiates the withdrawal of at least `32ETH`, a validator will be closed even if it could be that interchange could be covered to avoid closing one or several validators.

The privileged roles `owner`, `governor`, and `superAdmin` can call the following functions:

- `setPendingValidators`: add a new validator in the validators list. Everstake is trusted to regularly add new validators to the list to keep the system running.
- `replacePendingValidator`: replace a pending validator in the validators list.
- `markValidatorsAsExited`: marks one or several validators as exited so their slots in the validators list can be reused.
- `pauseStaking`: pause/unpause the staking.
- `pauseWithdraw`: pause/unpause the withdrawals.
- `setMinStakeAmount`: set a minimum staking amount.

3.2 Accounting

The `Accounting` contract holds all the logic related to internal accounting, rewards reinvestment, rounds management, deposits, interchange and withdrawals operations.

3.2.1 Exit stakes and rewards auto-compounding

The functions responsible for closed validators' stake management and rewards reinvestment are:

- `update() / _update()`: sends the stake returned by closed validators from the reward treasury to the withdraw treasury and computes the rewards and associated fees if any amount is left, then updates the internal accounting of the cached rewards treasury balance, as well as the fees and rewards balances held by the rewards treasury.
- `autocompound() / _autocompound()`: calls `_update()` first, if the rewards balance of the rewards treasury is above the minimum restake amount, the rewards are accounted for, and interchanged if possible. The rewards are added to the total deposited amount but no shares are minted. If the total pending amount is sufficient to launch one or more validators, the `Accounting` will trigger a `restake` which will deposit to new validators.

3.2.2 Internal accounting

The internal accounting tracks for each staker the amount they deposited in every current and past active rounds that are not yet activated, as well as the active stake in the form of shares. At the end of each active round, a snapshot of the user's part of the deposited beacon amount is taken, so that shares can be minted when the round becomes activated (when the validator they have their stake in is activated in the beacon chain). When the validator for a round is activated, Everstake will call `activateValidators`, which will take a snapshot of the current total deposited amount and the current supply of shares, mint the shares related to that round, and update the total deposited amount, as well as the total supply of shares. Users will claim their already minted shares for an activated round whenever they stake again or unstake, this is done within `_autocompoundAccount`.

Upon withdrawal of the active stake, the associated shares will be burned. The exiting amount will first be interchanged as much as possible with the pending stakers and the pending rewards to be restaked, if the pending amount cannot cover the withdrawal amount, the difference will be added in a withdrawal request that can be claimed when enough liquidity has been accrued in the withdraw treasury.

Note that the accounting functions `deposit`, `withdraw`, and `withdrawPending` are expected to be called for `msg.sender` by the `superAdmin` of the `Accounting`, which is the `Pool`, but they can also be called by the `owner` for arbitrary addresses.

3.3 Treasuries

The treasuries are contracts used by the `Accounting` to store the funds obtained from the validators (rewards or returned stake of closed validators) and the funds to be withdrawn by users.

Both treasuries have a function `sendEth`, this function sends Eth to some arbitrary address given as a parameter and can only be called by the `rewarder` of the contract.

3.3.1 Reward Treasury

In addition to `sendEth`, the `RewardTreasury` defines `reStake`, a function used to call the pool's `restake` function together with sending some amount of ETH.

The address of the `RewardTreasury` is given as part of the `withdrawal_credentials` parameter to the call to the `deposit` function of the beacon deposit contract done by the pool. This means that rewards and returned stakes from every validator will be sent to the treasury. The balance of the treasury is then used in the following ways by the accounting contract using `sendEth` and `reStake`:

- Funds from a closing validator are sent to the withdraw treasury.
- Funds from rewards are split in two by the accounting:
  - Rewards fees stay in the `RewardsTreasury`, but are accounted for in the `Accounting` so that the `owner` of the `Accounting` contract can claim them.
  - The rest of the rewards are sent to the pool using `restake` or to the withdraw treasury depending on if they get interchanged or not during the auto-compounding.

3.3.2 Withdraw Treasury

The `withdrawTreasury` can receive ETH in the following ways:

- From the `RewardTreasury` as mentioned above (either a closing of validator or by interchanging the rewards).
- From the `Pool` when a user is staking and part of his stake is interchanged.

The funds in the treasury are then used by `_claimWithdrawRequest` to send the ETH to a user claiming a withdrawal request.

3.4 Changes in Version 3

- The fee can be up to 100%.
- The two treasuries (`RewardsTreasury, WithdrawTreasury`), are now behind an upgradeable proxy.
- The `owner` of the `RewardsTreasury` contract can call the `sendEth` function as well, not only the `rewarder`.
- The `owner` of the `RewardsTreasury` contract can call the `reStake` function as well, not only the `rewarder`.
- When calling the function `Pool.unstake`, users can choose to use the interchange feature or not.
- A new feature has been added for users. They can now use the function `Pool.activateStake` to forcibly interchange their pending stake with the interchangeable amount.
- The `governor`, `owner`, and `superAdmin` can reorder the list of validators (`_values`) following the index of their public key in `_validatorsPubKeys`, starting from `_activePendingElementIndex`.
- The `owner` of the `Accounting` contract can pause the rewards update. This feature is meant to be used in the case of a validator getting slashed, so the incoming remaining stake is not consumed as a reward.
- When the shares are minted to the user, the activated amount is readjusted by being recomputed based on the newly minted shares in the user's deposited amount, this is done in the favor of the system.
- In `Accounting.withdraw`, the interchange is now done with the pending restake rewards first, and then with the pending stakers.
- In addition to `markValidatorsAsExited`, `Pool.markValidatorAsExited` can now be used to mark one validator as exited given its index in `_validatorsPubKeys`.

3.5 Trust Model

Users of the system are generally untrusted and expected to behave unpredictably.

The following roles are fully trusted and expected to behave honestly and correctly.

- The `owner`, the `superAdmin` and the `governor` of the `Pool`.
- The `owner` and the `governor` of the `Accounting`.
- The `owner` of the `RewardTreasury`.
- The `owner` of the `WithdrawTreasury`.

More specifically, the `owner` of the `RewardTreasury` and `Withdraw` are allowed to send ETH from those contracts and are trusted to use this functionality only in case of emergency. The `owner` of `Accounting` can make calls inside the `Accounting` contract in the name of arbitrary addresses and is trusted to use this functionality only in case of emergency.

The `superAdmin` of the `Accounting` is assumed to be the address of the `Pool`. The `rewarder` of both the `RewardTreasury` and the `WithdrawTreasury` are assumed to be the address of the `Accounting`.

Everstake is trusted in providing and managing the validators correctly by avoiding slashing, claiming the rewards when it is needed and closing the validators when the contract requires it.

Everstake is trusted in providing liquidity to facilitate stake and unstake demand.

The exact procedure in case of slashing is not known. Everstake provided us with the following flow in case of slashing:

1. Wait the closest time until the validator exit
2. Stop rewards update
3. Update Pool balances
4. Activate a new validator

The exact manner of how point `3.` will be executed is not fixed, see Slashing is not taken into account. We expect Everstake to document it well for potential users and users to be aware of this. Moreover, Everstake added that `3.` is:

```
still under consideration due to business model finalization:

 - Slashing coverage within pool with higher fee rate in collaboration of Insurance services.
 - Pool without slashing coverage with lower fee rate, that will require in case of emergency
   to deploy changes that could help to update pool balances.
```

In order to mitigate users' loss in case of slashing, Everstake plans to deploy a special emergency treasury fund, where part of the fee can be used to refund the users, see also Users May Not Be Fully Refunded in Case of Slashing.

4 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

6.1 Pausing Auto-Compounding Can Be Unfair to Users

Design Low Version 3 Acknowledged

CS-EVERSTKB2C-001

The flag `PAUSE_REWARDS_POSITION` introduced in the `Accounting` allows for pausing rewards auto-compounding. This led to the following:

1. During the period rewards auto-compounding is disabled, all users joining the protocol and having their stake activated will collect the same amount of rewards independently of the time their stake became active.
2. Provided that enough interchanging is available, a user who wants to begin staking can monitor the mempool and frontrun the Everstake's transaction that reenables reward auto-compounding to keep its ETH liquid for the largest amount of time possible while maximizing its rewards.

Acknowledged:

Everstake is aware of this issue and responded that, depending on their SOP for each edge case, staking may be manually paused along with the pausing of the rewards.

6.2 Interchanging Is Not Performed in Order When Withdrawing

Design Low Version 1 Acknowledged

CS-EVERSTKB2C-004

Because of the implementation of `AddressSet`, when interchanging with pending deposits during a withdrawal, the interchanges are not done in the order the stakers deposited. This means that some users might have to wait for their deposit to become active longer than another user who deposited after them, hence missing the rewards that were distributed during this time.

This issue can be taken advantage of by a user who wants to stake `x` ETH but does not wish to wait for their deposit to become active. By looking at the mempool, the user can spot transactions withdrawing from the contract. If he finds a withdrawal greater than or equal to `x + y` where `y` is the amount that could be interchanged with `_slotPendingStakers()[activeRound][0]`, frontrunning the transaction with his call to `Pool.stake` will guarantee him to be interchanged with the withdrawal, overtaking all the users in `_slotPendingStakers()[activeRound]`.

Acknowledged:

Everstake responded that this behavior is known and kept as it is to save gas since using the proper order would be costly given OpenZeppelin's *AddressSet* implementation.

6.3 Minimum Stake Consistency

Design | Low | Version 1 | Acknowledged

CS-EVERSTKB2C-005

The `Pool` forces stakers to stake at least `MIN_STAKE_AMOUNT` per deposit, but it is possible to circumvent this by calling `stake()` and then either `unstakePending()` or `unstake()` to have a final deposited amount smaller than `MIN_STAKE_AMOUNT`.

Acknowledged:

Everstake acknowledged this behavior and explained that this check is more about excluding cases when the fees of the transaction would be relatively large compared to the actual deposited amount.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	2

- Replacing a Validator Eventually Blocks the System Code Corrected
- Usage of address(this).balance in restake Can Block the System Code Corrected

Medium-Severity Findings	6

- renounceOwnerShip Leaves the _pendingOwner Pending Code Corrected
- Missing Input Sanitization Code Corrected
- Pausing Withdrawing Is Ineffective Code Corrected
- Slashing Is Not Taken Into Account Specification Changed
- _simulateAutocompound's Computation of pendingRestaked Is Incorrect Code Corrected
- _simulateAutocompound's Computation of totalShare Is Incorrect Code Corrected

Low-Severity Findings	22

- Wrong Address in Event upon acceptOwnership Code Corrected
- RewardsTreasury Overrides SendETH With Identical Implementation Code Corrected
- ValidatorList.get Might Not Represent the Reality Code Corrected
- _simulateAutocompound Ignores Paused Rewards Code Corrected
- Batch Deposit in First Round Skips Shortcut Code Corrected
- Events Missing Code Corrected
- Inconsistent Event Emission Order Code Corrected
- Interchanged Part of a Deposit Is Not Added to depositBalance Code Corrected
- Interfaces Not Implemented Code Corrected
- Missing Documentation Code Corrected
- Missing Indexing of Events Code Corrected
- Status of Replaced Validators Is Not Reset Code Corrected
- Unnecessary Function Parameter Code Corrected
- Variables and Functions Names Are Not Representative Code Corrected
- View Functions Are Incorrect for Round 0 Code Corrected
- Withdrawing May Fail Due to Underflow Code Corrected
- Wrong Restake Condition Code Corrected
- InterchangeDeposit Emitted When No Interchange Code Corrected
- activatedRound Cached Value Not Updated Code Corrected
- activeRound==0 Shortcut Breaks Semantics Code Corrected

- onlyGovernor Not Used Code Corrected
- unstakePending Does Not Update _slotPendingStakers Code Corrected

7.1 Replacing a Validator Eventually Blocks the System

Correctness High Version 1 Code Corrected

CS-EVERSTKB2C-034

The function `ValidatorList.replace` does not set the status of the new validator to `Pending`. This will make `ValidatorList.shift()` revert when the next pending validator will be the new validator as its status will be `Unknown`. If this happens, staking and withdrawing will be blocked. The funds can only be unlocked if the validators are closed and the `rewarder` on the `RewardsTreasury` can be changed.

Code corrected:

The function has been updated and the new validator' status is now set to `Pending`.

7.2 Usage of `address(this).balance` in `restake` Can Block the System

Design High Version 1 Code Corrected

CS-EVERSTKB2C-025

The function `restake` defined in the pool contract is called when auto-compounding is performed and tries to deposit to the beacon deposit contract with fresh validators as much as possible given the balance of the contract.

As this function relies on the balance of the contract and not the balance computed by the accounting the following is possible:

- If ETH is forced into the pool contract (using `selfdestruct` for example), one can increase the balance of the contract such that any subsequent function call triggers a deposit to the beacon deposit contract via `_autocompound` when it would not without the extra ETH. As accounting's pending amount will now be much greater than the actual balance of the pool, any call to `deposit` or `withdraw` will eventually revert as it will try to send to the beacon deposit contract ETH that is no longer in the pool.
- When a user stakes, `_autocompound()` is called by `deposit()` before the deposit is accounted for in `_deposit()` and after the deposit of the user has been added to the contract's balance. If the balance of the contract, upon transferring the rewards from the treasury to the pool contract, is greater than 32 ETH, `restake` will deposit to the beacon deposit contract. As part of the user's deposit will be gone, the accounting performed by `_deposit()` will not match the actual balance of the contract and the call will revert as `_stake` cannot send `BEACON_AMOUNT` to the beacon deposit contract.

Code corrected:

A parameter `activatedSlots` to indicate how many validators must be deposited to has been added in the `restake` functions, and the `Pool` does not rely on its internal balance anymore.

7.3 `renounceOwnerShip` Leaves the `_pendingOwner` Pending

Design | Medium | Version 3 | Code Corrected

CS-EVERSTKB2C-033

The functions `TreasuryBase.renounceOwnership` and `OwnableWithSuperAdmin.renounceOwnership` delete the `_owner`, but not the `_pendingOwner`. When renouncing ownership, there may be still a pending owner. The goal of renouncing ownership is to leave the contract without an owner forever, but if the current owner does initiate ownership transfer and then renounces, the pending owner can still claim ownership of the contract.

Code corrected:

The two functions have been updated to delete the `_owner` and the `_pendingOwner`.

7.4 Missing Input Sanitization

Design | Medium | Version 1 | Code Corrected

CS-EVERSTKB2C-031

Some function inputs are not sanitized:

1. `Pool.initialize()`: the parameters `rewardsTreasury` and `poolGovernor` are not checked for `address(0)`.
2. `Accounting.initialize()`: the parameter `accountingGovernor` is not checked for `address(0)`, and `poolFee` is not checked to be smaller than `FEE_DENOMINATOR`.
3. `TreasuryBase.setRewarder()`: `rewarder` is not checked for `address(0)`.
4. `TreasuryBase.setOwner()`: `owner` is not checked for `address(0)`.
5. `Pool.setGovernor()`: `newGovernor` is not checked for `address(0)`.

Code corrected:

1. Zero address checks are done.
2. Zero address and fee sanitization checks are done.
3. Zero address check is done.
4. the function has been removed and replaced by a transfer-and-accept pattern.
5. Zero address check is done.

7.5 Pausing Withdrawing Is Ineffective

Design | Medium | Version 1 | Code Corrected

CS-EVERSTKB2C-039

The function `pauseWithdraw` can be called by a privileged role to (un)pause the withdrawals. While the function `unstakePending` has the modifier `whenWithdrawActive` to ensure that it can only be called

only when withdrawals are allowed, `unstake()` is not and can be called independently of the pausing of withdrawals.

Code corrected:

The function `Pool.unstake` has been updated with the `whenWithdrawActive` modifier.

7.6 Slashing Is Not Taken Into Account

Design | Medium | Version 1 | Specification Changed

CS-EVERSTKB2C-003

The protocol assumes that slashing will never happen for any of its validators. While the risk of slashing can be greatly reduced by good infrastructure maintenance and monitoring, it can never be zero, in this case, the slashing of a validator could result in putting the protocol in unexpected states and a potential loss of funds for users. All implementations of nodes can potentially have bugs that might lead to undesired slashing.

For example, any transfer of less than 32 ETH from the validator is considered as an incoming reward, while if slashed this can be an entire stake of the validator. Thus, if slashed just before unstaking, the wrong accounting can lead to unexpected results.

Code corrected:

Everstake added the feature to stop the update of rewards in case of emergency. When it comes to updating the user's balance and refunding the users, see Trust Model and Users may not be fully refunded in case of slashing.

7.7 `_simulateAutocompound`'s Computation of `pendingRestaked` Is Incorrect

Correctness | Medium | Version 1 | Code Corrected

CS-EVERSTKB2C-017

When simulating the activation of rounds in `_simulateAutocompound()`, for each round being activated, `pendingRestaked` is decremented by `BEACON_AMOUNT`. As it might be the case that `pendingAmount/BEACON_AMOUNT > pendingRestaked/BEACON_AMOUNT`, a call to the function could revert after trying to underflow the variable `pendingRestaked`.

Code corrected:

`_simulateAutocompound` no longer makes the assumption that `pendingAmount/BEACON_AMOUNT > pendingRestaked/BEACON_AMOUNT` and correctly decrement `pendingRestaked`.

7.8 `_simulateAutocompound`'s Computation of `totalShare` Is Incorrect

Correctness | Medium | Version 1 | Code Corrected

CS-EVERSTKB2C-018

When `_deposit` is called by `_autocompound()`, the total amount deposited (`AUTO_COMPOUND_TOTAL_SHARE_POSITION`) is incremented with the rewards (both the part that has been interchanged and the part that will be deposited). On the other side, `__simulateAutocompound()` increments `totalShare` by `unclaimedReward` which, at this point, only includes the part of the rewards that is not interchanged. The `totalShare` returned by `_simulateAutocompound()` will hence not always match the result of an autocompounding.

Code corrected:

`_simulateAutocompound` now matches `_autocompound()` behavior and takes into account the interchanged amounts when computing the total pool's balance.

7.9 Wrong Address in Event upon `acceptOwnership`

Design | Low | Version 3 | Code Corrected

CS-EVERSTKB2C-043

The functions `TreasuryBase._transferOwnership` and `OwnableWithSuperAdmin._transferOwnership` emit the event the addresses of `_owner` and `newOwner`, which are the same at that point.

Code corrected:

The two functions have been updated to emit the event with the previous owner and the new owner.

7.10 `RewardsTreasury` Overrides `SendETH` With Identical Implementation

Design | Low | Version 3 | Code Corrected

CS-EVERSTKB2C-035

`RewardsTreasury` overrides `SendETH` with the same implementation as `TreasuryBase`.

Code corrected:

Everstake revised the inheritance between the contracts and their interface to allow for `RewardsTreasury` not to have to override `sendETH`.

7.11 `ValidatorList.get` Might Not Represent the Reality

Design | Low | Version 3 | Code Corrected

CS-EVERSTKB2C-002

Given a set of Validators with the status `Deposited`, they may be not closed in the same order they appear in `_validatorsPubKeys`. In such case, calling `markAsExited` will mark the n first `Deposited` validator of the list as excited, without caring about which exact validator was closed. This may lead the function `ValidatorList.get` to return a status that is not representative of reality in the case Everstake did not close validators in the order they appear in `_validatorsPubKeys` (e.g. in case of slashing or leaked private key).

Code corrected:

Everstake added the function `markValidatorAsExited` and a corresponding internal function to the `Pool` and to `ValidatorList` to allow for marking as closed validators given its index in `_validatorsPubKeys`. Provided that Everstake always uses the right method between `markValidatorsAsExited` and `markValidatorAsExited` to close the validators that have effectively been closed, `ValidatorList.get` should return the correct status.

7.12 `_simulateAutocompound` Ignores Paused Rewards

Correctness | Low | Version 3 | Code Corrected

CS-EVERSTKB2C-036

`Accounting._simulateAutocompound` does not take the pausing of rewards into account, and thus does not mirror what autocompound would do when the rewards are paused.

Code corrected:

The function `_simulateAutocompound` has been updated to reflect the behavior of `_autocompound()` when the rewards are paused.

7.13 Batch Deposit in First Round Skips Shortcut

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-022

If the first round (`activeRound==0`) is closed within a batch deposit, the shortcut in `Accounting._activateRound()` will be skipped as `activeRound > 0`. In this case, round 0 will have to be activated as any other round by calling `activateValidators`.

Code corrected:

The special handling of the case `activeRound==0` has been removed from `_activateRound()`.

7.14 Events Missing

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-029

Even though many events are emitted by the protocol, several important state changes do not emit events:

1. `OwnableWithSuperAdmin.__OwnableWithSuperAdmin_init_unchained()` does not emit `SetSuperAdmin` after setting the super admin.
2. `Accounting.withdraw()` does not emit `InterchangeDeposit` when interchanging with the pending restaked amount.
3. No event is emitted by `Accounting.activateValidators()` when one or several validators are activated.
4. No event is emitted when the minimum amount to restake is set using `Accounting.setMinRestakeAmount()`.
5. In `Accounting`, `deposit()`, `withdrawPending()` and `withdraw()` could emit events as they are not necessarily respectively called by `Pool.stake()`, `Pool.unstakePending()` and `Pool.unstake()`.
6. `Pool.unstake()` emits no event when no amount is withdrawn from the pending value of the pool.
7. `Pool.restake()`, `Pool.setPendingValidators()`, `Pool.replacePendingValidator()`, `Pool.markValidatorsAsExited()` emit no event while they perform important state changes.
8. `GovernorChanged` is emitted when setting the governor in `Pool.initialize()` and `Accounting.initialize()`, similarly, `FeeUpdated` is not emitted when setting the pool fee in `initialize()`.

Version 3:

9. `OwnableWithSuperAdmin.renounceOwnership` emits an event but `TreasuryBase.renounceOwnership` does not.
10. `markValidatorsAsExited` is defined and emitted in the library `ValidatorList`, meaning that it won't be part of the `Pool`'s ABI as it is not redefined there.

Code corrected:

The points 1., 2., 3., 4., 6., 7. (all but `setPendingValidators`), 8., 9., 10. have been fixed.

For 7., Everstake states:

> Pool.setPendingValidators() - not important state changes. It's internal processing which can be indexed without events.

For point `5.`:

Everstake answered that the concerned functions of the accounting can be called either by the pool or by the owner. In the former case, the pool emits relevant events. For the latter case, Everstake states that the owner should call these functions only in an emergency and thus, Everstake claims that no events are needed in those cases as observers would anyway know that this is the owner's actions.

7.15 Inconsistent Event Emission Order

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-024

1. While most of the functions emit events after calling functions that might themselves emit an event, `Pool._deposit` emits `StakeDeposited` before calling `deposit` which will itself emit events.
2. Additionally, in the codebase, the rule seems to be doing storage change first and then emitting events, however, some functions do not follow this pattern:
    - `OwnableWithSuperAdmin.transferOwnership`
    - `Governor._updateGovernor`

Code corrected:

1. The event has been moved after the call to `deposit`.
2. The events have been moved after the state changes.

7.16 Interchanged Part of a Deposit Is Not Added to `depositBalance`

Correctness | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-026

When staking, the part of the deposit that is interchanged with withdrawals is not added to `sourceStaker.depositBalance`.

Code corrected:

The function `AutocompoundAccounting._depositAutocompound` has been updated to add the interchanged amount to the `sourceStaker.depositBalance`.

7.17 Interfaces Not Implemented

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-027

Some of the contracts do not implement their interfaces (`FooBar is IFooBar`). This would be a guarantee for integrators that the contracts carry the same functions signatures as the interfaces. Such contracts are listed below:

- `TreasuryBase`
- `RewardsTreasury`

Code corrected:

The contracts `TreasuryBase` and `RewardsTreasury` have been updated to implement their respective interfaces.

7.18 Missing Documentation

Design Low Version 1 Code Corrected

CS-EVERSTKB2C-028

Most of the functions are poorly documented or have no NatSpec description at all.

Code corrected:

Extensive documentation has been added for `external` and `public` functions.

7.19 Missing Indexing of Events

Design Low Version 1 Code Corrected

CS-EVERSTKB2C-030

All events defined in `Accounting`, `Governor`, `Pool`, `RewardTreasury`, `TreasuryBase` and `Withdrawer` contain no indexed fields. Indexing some relevant fields will help for searching events quicker.

Code corrected:

The relevant fields have been indexed.

7.20 Status of Replaced Validators Is Not Reset

Correctness Low Version 1 Code Corrected

CS-EVERSTKB2C-037

In the function `ValidatorList.replace`, the status of the replaced validator is not assigned (=), but compared (==) to `ValidatorStatus.Unknown`. This line of code will have no effect, and it will not be possible to add again the replaced validator at a later stage.

Code corrected:

The status of the replaced validator is correctly reset to `Unknown`.

7.21 Unnecessary Function Parameter

Design Low Version 1 Code Corrected

CS-EVERSTKB2C-038

The function `AutocompoundAccounting._activatePendingBalance()`, is always called with the parameter `minPresentedAmount` set to `true`, thus the parameter and logic related to it should be removed from the codebase.

Code corrected:

The unnecessary function parameter `minPresentedAmount` and its related logic have been removed from the codebase.

7.22 Variables and Functions Names Are Not Representative

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-040

Having self-explanatory names for variables and functions greatly help the understanding of the code. The names of some of the variables and functions in the codebase are misleading. Here is a non-exhaustive list:

- all the functions named with `autocompound`, except `autocompound()` and `_autocompound()` have nothing to do with autocompounding.
- `AUTO_COMPOUND_TOTAL_SHARE_POSITION` represents the total amount of `ETH` currently deposited in the validators, not a share. Moreover, the amount is not only from auto-compounded rewards.
- `ShareState.totalShare` represent the total deposited amount at some period, not a share.
- `ShareState.shareIndex` represent the total minted shares at some period, not an index.
- `STAKER_AUTOCOMPOUND_BALANCES_POSITION` and the struct `AutocompoundStakerMining` have nothing to do with autocompounding.

Code corrected:

The functions and variables names have been changed.

7.23 View Functions Are Incorrect for Round 0

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-041

The special case for `activeRound==0` in `_activateRound()` sets `activatedRound` to 1 although the validator is not necessarily active yet.

This means that the following functions might return incorrect results relative to the semantics of `pendingDeposited` and `active`:

- `pendingDepositedBalance()`
- `pendingDepositedBalanceOf()`
- `depositedBalanceOf()`
- `autocompoundBalanceOf()`

Code corrected:

The special handling of the case `activeRound==0` has been removed from `_activateRound()`.

7.24 Withdrawing May Fail Due to Underflow

Correctness | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-042

When computing `_shareToAmount(totalShare, autoCompoundShareIndex, autoCompoundTotalShare) - originActiveDepositedBalance` in `_withdrawFromAutocompound`, amounts deposited by the user are compared with amounts obtained from shares using `_shareToAmount()`, as there might have been a rounding error in the computation of the latter, their comparison might result in an underflow, leading the call to revert.

An easy way to obtain this behavior is to have the user deposit a very low amount of ETH x (10 wei for example) by calling `stake()` with some large value before calling `unstakePending()` to leave in the pending deposit of the user x ETH. Supposing now that the price of a share is high at the moment of the activation of the validator, it is possible that `_shareToAmount(_amountToShare(x)) < x` as `_amountToShare()` might have done some rounding.

Code corrected:

When only a portion of the user's shares are burned, the accounting subtracts the deposited balance to whatever is larger between the deposited balance and the amount obtained from the shares to be burned to avoid underflow.

When all the shares are burned, no such comparison is done and the deposited amount is updated to be the user's pending amount.

7.25 Wrong Restake Condition

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-044

The condition for the `Pool` to deposit on a restake is `balance > BEACON_AMOUNT`. If `balance == BEACON_AMOUNT` the active round would have been incremented and the pending amount updated accordingly during `_autocompound()` because the system expects a new validator to be provisioned. But in that case, the validator will not be provisioned because of the strict inequality above. Moreover, the internal accounting of the pending amount will not be representative of the true pending value until the next deposit or reward auto-compounding.

Code corrected:

The restake condition has been modified and relies on `activatedSlots` instead of `balance`.

7.26 `InterchangeDeposit` Emitted When No Interchange

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-023

In the function `withdraw`, `InterchangeDeposit` is emitted even if no interchange happened for the given pending staker (`activatedAmount==0`).

Code corrected:

The `withdraw` function has been updated so the `InterchangeDeposit` event is emitted only when some amount is interchanged.

7.27 `activatedRound` Cached Value Not Updated

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-020

At the beginning of the function `Accounting._depositBalance`, `activatedRound` is cached in memory and later used when calling `_depositAccount()`. If `activeRound==0` and enough ETH is deposited so that `_activateRound()` is called, `activatedRound` is set to `1` in the storage but its cached value is not updated. Because of this, if `_depositBalance` was to call `_depositAccount()` after `_activateRound`'s call (the user deposited enough to activate two or more rounds), `_autocompoundAccount()` would cache the user's deposit for the round `0` in `pendingDepositedBalances` although the deposit should be active at this time and shares should be minted.

Code corrected:

The special handling of the case `activeRound==0` has been removed from `_activateRound()`.

7.28 `activeRound==0` Shortcut Breaks Semantics

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-021

The implemented shortcut in `_activateRound()`, the shortcut that marks the round `0` activated breaks the semantics of the `activatedRound`, which should only represent the number of validators that have been effectively activated.

Code corrected:

The special handling of the case `activeRound==0` has been removed from `_activateRound()`.

7.29 `onlyGovernor` Not Used

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-032

The modifier `onlyGovernor` defined in the `Governor` is never used in the code base and thus should be removed.

Moreover, as the check ensuring that the `msg.sender` is the governor is performed after the function's call the modifier is applied to, if the modifier was to be used on a function updating the governor, it could be that the function is not protected and could be called by anyone providing themselves as the new governor.

Code corrected:

The modifier `onlyGovernor` has been removed from the `Governor` contract.

7.30 `unstakePending` Does Not Update `_slotPendingStakers`

Design | Low | Version 1 | Code Corrected

CS-EVERSTKB2C-019

When a user withdraws his full pending stake using `unstakePending`, `_slotPendingStakers[activeRound]` is not updated to remove the staker.

Code corrected:

The `Accounting.withdrawPending` function has been updated such that the staker is removed from `_slotPendingStakers[activeRound]` if they remove all of their pending stakes.

7.31 Deleting a `struct` With a Mapping Has No Effect

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-008

In Solidity, if a `struct` contains a mapping and one deletes the `struct`, the mapping will not be deleted. In the codebase, `Accounting._activateRound()` deletes `_slotPendingStakers()[activeRound]`, an `AddressSet`, but only the `_values` field of the Set will be defaulted.

Code corrected:

The deletion of `_slotPendingStakers()[activeRound]` has been removed.

7.32 Event Reentrancy

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-009

In several functions, an event is emitted after an external call to some address, in the case that the call would reenter the contract, it would be possible to have events emitted out of order.

The list of such patterns is shown below.

- `Pool.unstake()` with `_safeEthSend()` and the event `Unstake`.
- `Pool.unstakePending()` with `_safeEthSend()` and the event `StakeCanceled`.
- `Withdrawer._claimWithdrawRequest()` with `ITreasuryBase.sendEth()` and the event `ClaimWithdrawRequest`.
- `Accounting.claimPoolFee()` with `IRewardsTreasury.sendEth()` and the event `ClaimPoolFee`.

Code corrected:

All the patterns above have been updated to emit the event first, and then transfer ETH.

7.33 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-011

1. The type casting from `address` to `address` is not required in `Pool.initialize()`, removing it might save gas during initialization depending on the compiler's optimization setting.
2. `pendingValidatorPubKey` is read twice from storage in `Pool._deposit()`, the value could be cached in memory to avoid one SLOAD.
3. The checks of the form `a != b && a != c` can be modified following De Morgan's law (`!(a == b || a == c)`) to leverage the lazy evaluation of the condition and save gas on runtime.
4. Some function arguments on call can be replaced by constants. Some examples are:
    - `Accounting._activateRound()`: the variable `activeRound` can be replaced by `0` in the call `_makeAutocompoundRoundCheckpoint(activeRound)`.
    - `Accounting._depositBalance()`: in the call to `_activateRound()` of the branch `if (pendingAmount > 0)`, the parameter `pendingTotalShare + closeCurrentRoundAmount` can be replaced by `BEACON_AMOUNT`.
    - `Accounting._depositBalance()`: in the call to `_depositAccount()` of the branch `if (depositToPendingAmount > 0)`, the parameter `interchangedAmount` will always be `0`.
    - `Accounting._depositBalance()`: in the call `AUTO_COMPOUND_PENDING_SHARE_POSITION.setStorageUint256()` of the branch `if (depositToPendingAmount > 0)`, the parameter `pendingTotalShare` will always be `0`.
5. The `activatedSlots` in the branch `if (pendingTotalShare > 0)` of the function `Accounting._depositBalance()` can be set to `1` instead of incrementing the variable to save gas on runtime.
6. The `while` loop and multiple stack variables increments in the branch `if (depositToPendingAmount >= BEACON_AMOUNT)` of the function `Accounting._depositBalance` can be replaced by one update for each involved variable. If the `while` loop was to stay, a `do-while` construct could save gas. The same applies in `_simulateAutocompound()`.
7. Setting `pendingTotalShare` to `0` in the branch `if (depositToPendingAmount >= BEACON_AMOUNT)` of the function `Accounting._depositBalance` is redundant.
8. The `while` loop in the function `Accounting.withdraw()` can be simplified since in the case `isFullyDeposited==false`, then the remaining `interchangeWithPendingDeposits` is zero.
9. In the branch `if (withdrawFromPendingAmount > 0)` of the function `Accounting.withdraw`, `pendingRestakedValue - withdrawFromPendingAmount` is computed twice while it could be done only once.

10. In the function `Accounting.withdraw`, the `pendingTotalShare` is read from storage twice when it could be cached in the memory.
11. In the return statement of the branch `if (unclaimedReward < MIN_RESTAKE_POSITION.getStorageUint256())` of the function `Accounting._simulateAutocompound()`, the constant `0` can be used instead of `unclaimedReward`
12. When simulating the withdraw queue filling in `Accounting._simulateAutocompound()`, the if/else branches could be unified in the same way it is done in `Withdrawer._interchangeWithdraw()`.
13. The modifier `Governor.onlyGovernor()` does the address check after executing the code. Reverting early would save gas.
14. In the function `Pool._stake()`, `value` cannot be zero.
15. The increment `i++` can be in an `unchecked` block in multiple `for` loops.
16. The function `Withdrawer. _calculateValidatorClose` can return only one value, as the two values are linked by a constant factor, one can easily deduce a value from the other one.
17. In the function `Withdrawer._calculateWithdrawRequestAmount`, the condition `withdrawFromActiveDeposit > 0` will always be true if `withdrawFromActiveDeposit > pendingTotalShare` is true and is hence redundant.
18. In the function `WithdrawRequests.add`, the assignation `requests._values[i] = request` can be moved inside the `if (requests._values[i].value == 0)` block and the function can return right after.
19. In the functions `WithdrawRequests.claim` and `WithdrawRequests.info`, `requests.value[i].afterFilledAmount` is read twice from the storage while it could be cached to avoid one SLOAD.
20. In the functions `WithdrawRequests.claim` and `WithdrawRequests.info`, the condition `requests._values[i].afterFilledAmount > actualFilledAmount` can be relaxed to an unstrict comparison since if `requests._values[i].afterFilledAmount == actualFilledAmount` their difference is null.
21. In the function `WithdrawRequests.info`, `requests._values.length` is read from the storage at each iteration of the loop. Caching it in the memory would avoid several SLOAD.
22. In the function `ValidatorList.add`, `set._activeValidatorIndex` and `set._activePendingElementIndex` are both read three times from the storage when their value could be cached in the memory.
23. In the function `ValidatorList.shift`, `set._activePendingElementIndex` is read two times from the storage when its value could be cached in the memory.
24. In the functions, `_autocompoundAccount`, `_autoCompoundUserPendingDepositedBalance`, `_autoCompoundUserBalance` and `_withdrawFromAutocompound` of `AutocompoundAccounting`, the field `pendingDepositedBalances.length` of the staker is read from the storage at each iteration of the loop. Caching it in the memory would avoid several SLOAD.
25. In the first `for` loop of the function `AutocompoundAccounting._autocompoundAccount`, both `staker.pendingDepositedBalances[j].period` and `staker.activePendingDepositedElementIndex` are read twice from the storage and could be cached.
26. In `AutocompoundAccounting._autocompoundAccount()`, when updating the `pending` status to `pendingDeposited`, one execution path read three times `staker.activePendingDepositedElementIndex` from storage, it could be cached.

27. In `AutocompoundAccounting._autocompoundAccount()`, when updating the `pending` status to `pendingDeposited` or to `activated`, both `staker.pendingBalance.balance` and `staker.pendingBalance.period` are read twice from storage.
28. In `AutocompoundAccounting._autoCompoundUserPendingDepositedBalance()`, `staker.pendingBalance.period` is read twice from storage.
29. In `AutocompoundAccounting._autoCompoundUserBalance()`, at each iteration of the `for` loop, if the condition of the `if` statement is not met, both `stakerAutocompoundBalance.pendingDepositedBalances[j].balance` and `stakerAutocompoundBalance.pendingDepositedBalances[j].period` are read twice from storage.

Version 3:

30. The calls to `_userActiveBalance` to get only the `depositedBalance` could be replaced by a simple storage read to save gas.
31. At the end of `Accounting._simulateAutocompound()`, `pendingAmount == pendingRestaked` always holds as if `if (pendingAmount > 0)` is entered, then they are both set to 0. Otherwise `pendingAmount==0` and hopefully one should always have `pendingAmount >= pendingRestaked` meaning that there is no need to keep both var for the while loop.

Code corrected:

The gas optimizations have been applied.

7.34 Governor Is Immutable

Informational **Version 1** **Code Corrected**

CS-EVERSTKB2C-012

While the `Governor` role of the `Pool` can be transferred to another address by the `Owner`, the `SuperAdmin` or the governor himself at any time, the `Governor` of the `Accounting` contract can only be set when calling `Accounting.initialize`.

Code corrected:

The `Governor` of the `Accounting` can now be updated using the function `setGovernor`.

7.35 Unneeded `return` Statement

Informational **Version 1** **Code Corrected**

CS-EVERSTKB2C-015

When a function signature looks like `function foo() external returns(uint a, uint b)`, the statement `return (a, b)` is not necessary when the values to be returned have been assigned earlier to the returned variables.

Some examples:

- `AutocompoundAccounting._withdrawFromAutocompound()`
- `AutocompoundAccounting._autocompoundAccount()`

Code corrected:

Some of the unnecessary `return` statements have been removed.

7.36 Unused Imports

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-016

The following imports are not used:

1. "./interfaces/IPool.sol" and "./interfaces/ITreasuryBase.sol" in `Accounting`.
2. "./interfaces/IPool.sol" in `WithdrawTreasury`.
3. "./interfaces/IRewardsTreasury.sol" in `RewardsTreasury`.
4. "./ITreasuryBase.sol" in `IRewardsTreasury`.

Code corrected:

All unused imports have been removed.

7.37 Validator Cannot Be Marked as Exited Immediately

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-010

In `ValidatorList`, `markAsExited()` changes the status of `num` validators from `Deposited` to `Exited` provided that:

- Their status is `Deposited`.
- They are all stored consecutively in `set._validatorsPubKeys`.
- The first validator is stored at index `set._activeValidatorIndex` of `set._validatorsPubKeys`.

By the design of the `List` struct and the functions `add` and `shift`, validators that are `Deposited` are not always at the "front" of the slice of `set._validatorsPubKeys` starting at `set._activeValidatorIndex` and can be interleaved by validator with another status. A `Deposited` validator in such configuration cannot have his status changed to `Exited` until all previous validators have the state `Deposited` or `Exited`.

Code corrected:

`markAsExited()` has been updated in such ways that the `num` validators to be marked as `Exited` no longer need to be stored consecutively. Additionally, `Pool.reorderPending()` can be used to order pending validators to be deposited to in the order as they appear in `_validatorsPubKeys`. Depending on how `Pool.reorderPending()` is called this can be used to keep `_validatorsPubKeys`'s length from growing.

7.38 `initializer` Used Over `onlyInitializing`

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-014

The functions `__OwnableWithSuperAdmin_init` and `__OwnableWithSuperAdmin_init_unchained` have the `initializer` modifier while `onlyInitializing` would be more correct.

Code corrected:

The modifier `onlyInitializing` is used instead of `initializer`.

7.39 `minStake`'s Value Differs From the Documentation

Informational | Version 1 | Code Corrected

CS-EVERSTKB2C-013

In the `Pool.initialize`, the minimum stake is set to `0.01` ETH while the documentation states that the minimum users are allowed to stake is `0.1` ETH.

Code corrected:

The minimum stake is now set to `0.1` ETH in `Pool.initialize`.

8 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

8.1 Inconsistent Use of `override` and `virtual`

Informational **Version 1**

CS-EVERSTKB2C-006

In solidity, it is not mandatory to use the `override` keyword when implementing a function from a parent interface. For the sake of consistency, either none or all implementations should be annotated with `override`.

Additionally, `Pool.setGovernor` is set as `virtual` although no contract inherits from `Pool`.

8.2 The Sum of Shares Can Be Less Than the Total Shares Supply

Informational **Version 1**

CS-EVERSTKB2C-007

Due to some rounding errors, the shares distributed to individual stakers for a given round might not match the total number of shares minted for that round, i.e. `_amountToShare(X+Y+Z) >= _amountToShare(X) + _amountToShare(Y) + _amountToShare(Z)`. The difference in value cannot be claimed by anyone.

9 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

9.1 Deposited Amount Is Gifted if Less Than 1 Share

Note Version 1

Users should be aware that any amount resulting in less than a share will be a donation to the pool. Even though this should be avoided by the minimum stake constraint, it is possible to stake and unstake pending, leaving a small amount to be activated, which may result in `0` share.

9.2 Users May Miss Rewards on Closed Validator

Note Version 1

If some validator is expected to close, i.e. `EXPECTED_CLOSE_VALIDATORS > 0`, any rewards accumulated in the `RewardsTreasury` that is above `32ETH` will be considered as a stake returned by a closing validator instead of a reward. So any staker withdrawing when one or more validators are expected to close and when `32ETH` of staking rewards or more are available in the `RewardsTreasury` will miss that reward.

9.3 Users May Not Be Fully Refunded in Case of Slashing

Note Version 1

According to the Trust Model, Everstake plans to deploy an emergency treasury fund, but do not guarantee all the users to be fully refunded in case of slashing.

Everstake states:

```
We understand and don't neglect the risks related to slashing and we will have a special Emergency Treasury
Fund - an Ethereum wallet address for Emergency Cases. Emergency Treasury fund will have some amount of Ethereum
to cover at least partly possible unlikely slashing related issues. We also plan to send some defined share of
Ethereum service fee received from the Pool by Everstake, approximately 10%.

Example:
Pool Service fee is 10%
Emergency Treasury Fund share is 10%
If all Validators within the Pool generated 10 000 ETH
Then Everstake will receive 1 000 ETH as a Pool Service Fee
And 100 ETH from Pool Service fee will be send to Emergency Treasury Fund
```