Everstake

Ethereum Staking Protocol

by Ackee Blockchain

5.9.2023

Contents

ackee | blockchain security

1. Document Revisions

0.1	Draft report	26.7. 2023
1.0	Final report	27.7. 2023
2.0	Draft report	29.8. 2023
2.0	Final report	31.8. 2023
2.1	Final report	5.9. 2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Miroslav Škrabal	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

ackee | blockchain security

3. Executive Summary

Everstake Staking is a protocol that allows users to deposit amounts that can be less than 32 ETH. Once users' deposits exceed 32 ETH, a new validator is created and the users can start earning rewards. The staking rewards are restaked automatically and the users' pool shares are increased.

Revision 1.0

Everstake engaged Ackee Blockchain to perform a security review of the Everstake protocol with a total time donation of 14 engineering days in a period between July 3 and July 26, 2023 and the lead auditor was Miroslav Škrabal.

The audit has been performed on the commit `60688fc` [1] and the scope was the entire `contracts` folder:

```
contracts/
├── Accounting.sol
├── AutocompoundAccounting.sol
├── Governor.sol
├── Pool.sol
├── RewardsTreasury.sol
├── TreasuryBase.sol
├── WithdrawTreasury.sol
├── Withdrawer.sol
├── 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
```

We began our review by using static analysis tools, namely Woke. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Woke testing framework. We include the fuzz tests written during the review in the Woke appendix.

During the review, we paid special attention to:

- ensuring the accounting arithmetic of the system is correct,
- testing whether no unstaking path reverts,
- testing whether users can't withdraw more than they have deposited (+ rewards),
- analyzing the management of the validators,
- detecting possible ETH call reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- testing if rewards are distributed according to users' shares,
- analyzing that the contracts use proper data structures for storing deposits, withdrawals, etc,
- analyzing the upgradeability pattern (storage collission, access control, etc),
- analyzing whether the withdrawal credentials are created correctly,
- looking for common issues such as data validation.

Our review resulted in 26 findings, ranging from Info to High severity. The highest severity issues were related to denial of service and the inability to view the state of the protocol due to underflow reverts.

Overall, we do not recommend deploying the current version of the protocol. During the audit, we discovered a couple of issues that caused the protocol to revert, although the state was achieved only through normal, non-malicious transactions. This implies that the protocol is not sufficiently tested. Additionally, we found the documentation to be lacking, we dedicated a separate informational issue to this. Multiple issues highlighted the lack of following the best practices, see data validation, upgradeability or dead code issues.

At the same time, we would like to acknowledge that the development team was able to find some of the issues independently of our review (i.e. both the teams found it independently of each other), most notably it was H3. Additionally, during the audit, we observed multiple clever design decisions in the Pool and Accounting contracts. However, due to the high number of issues, including high-severity ones, a lot of work needs to be done to make the protocol production ready.

Due to the high number of issues, lack of good documentation, and the fact that the protocol reverted in certain scenarios, our auditing processes were slowed down. As such, we do not feel certain that the protocol will be fully secure after the fixes are applied. We strongly recommend pursuing another audit round, though a shorter one, to ensure that the protocol is secure.

Based on the observations made during the review, we recommend focusing on the following high-level objectives:

- extend the test suite to cover all reported reverting scenarions,
- consider employing more advanced testing techniques, such as fuzzing (see Foundry, Echidna or Woke),

- ensure that the documentation is complete and up-to-date,
- address all the reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 2.0

Everstake engaged Ackee Blockchain to perform a second audit revision of the Everstake ETH staking protocol with a total time donation of 6 engineering days in a period between August 21 and August 29, 2023 and the lead auditor was Miroslav Škrabal.

The review was done on the commit `35f9b56` [2] and the files in scope were the same as in the previous audit. Since the last audit 45 new commits were made, a large number of those were fix commits and refactoring commits. The most notable changes were:

- adding upgradeability to treasuries,
- making interchange optional,
- modifying the ordering logic of the validators,
- adding gas optimizations,
- improving naming of the variables and adding comments,
- fixing the issues found in the previous audit.

We followed the methodology established in the last revision; we focused on manually reviewing all the changes and wrote additional tests in Woke. We wrote a new simple differential fuzz test for the quick sort function and made it available in the Woke appendix. We also utilized Woke for static analysis which was mainly useful for analysis of reentrancies. During the review we had similar objectives as in the previous revision, additionally, we focused on:

- verifying that all the fixes were applied correctly,
- withdrawing edge-case amounts (like very small values, values equalling all shares, etc),
- minting shares and subsequent rewards distribution,
- reviewing the integer-division-based loss of precision introduced in the amount-share conversions,
- reviewing the view functions (mainly _simulateAutocompound),
- reviewing the new ordering logic of the validators,
- analyzing transaction ordering and front-running opportunities,
- reviewing the new upgradeability pattern.

Our second review resulted in 12 findings, ranging from info to high severity. The highest severity issue related to integer-division-based error which in certain protocol states resulted in withdrawal reverts and thus caused temporal lock of users' funds.

The code quality in the second revision was significantly improved. The code was more readable (mainly due to the use of better names for variables) and the documentation was better. Almost all issues from the previous revision were fixed.

Based on the observations made during the review, we recommend focusing on the following high-level objectives:

- the documentation is still lacking and could be improved,
- due to the occurance of another rounding-based issue, we recommend fuzzing protocol to ensure that no other subtle off-by-one errors are present,
- another bug was uncovered in the `_simulateAutocompound` function, it is

recommended to rethink the approach of writing the simulations and use more organized and structured approach,

- avoid overly complicated and overengineered solutions like in the case of the reorderings and replacements of validators, such optimizations are not usually worth it in the long run,
- fix all the reported issues.

Revision 2.1

Everstake engaged Ackee Blockchain to perform a fix-review of the second audit revision on the commit `38970a6` [3].

The new changes included fixes to the issues reported in the second audit revision and a few gas optimizations. We analyzed all the fixes and attached a summary of each fix. The statuses of all issues were updated.

We consider all the issues to be fixed correctly. We believe that by fixing the H5 rounding issue no other rounding issues that would cause reverts in main user flows are present. Yet still we recommend fuzzing the protocol to analyze the protocol's behavior in random scenarios and protocol states.

[1] full commit hash: 60688fce62538138cfe43e9185c06d8d9093b187

[2] full commit hash: 35f9b56b038be82a31946bd6b02533ec16ddd228

[3] full commit hash: 38970a6bf94a05bb3c6a49c254cbd667c7ef8f78

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
H1: _simulateAutocompound can revert	High	1.0	Fixed
H2: DoS due to 0 pending deposits	High	1.0	Fixed
H3: Partial DoS due to interchange	High	1.0	Fixed
H4: DoS due to underflow	High	1.0	Fixed
M1: Missing whenWithdrawActive modifier	Medium	1.0	Fixed
M2: Call to depositedBalanceOf reverts	Medium	1.0	Fixed

	Severity	Reported	Status
L1: Withdraw request array monotonically grows	Low	1.0	Fixed
L2: Lack of 2-phase role transfers	Low	1.0	Fixed
L3: Exiting validators can revert	Low	1.0	Fixed
L4: Replacing validators lacks validation	Low	1.0	Fixed
L5: Validation of owner in treasuries	Low	1.0	Fixed
L6: Data validation in initialize functions	Low	1.0	Fixed
L7: Incorrect return value of _simulateAutocompound	Low	1.0	Fixed
L8: Upgradeable contract constructor without intializer	Low	1.0	Fixed
L9: Insuffitient data validation when composing contracts	Low	1.0	Acknowledged
W1: Usage of `solc` optimizer	Warning	1.0	Acknowledged
W2: Dead code in _autoCompoundUserBalance	Warning	1.0	Fixed
W3: Unchecked return of _update	Warning	1.0	Fixed

	Severity	Reported	Status
W4: Lack of contract prefix in storage position	Warning	1.0	Fixed
W5: Pool fee can be set extremely high	Warning	1.0	Acknowledged
I1: Used library	Info	1.0	Fixed
I2: Comparison with role outside modifier	Info	1.0	Acknowledged
I3: Function always returns true	Info	1.0	Fixed
I4: Lack of logging in setters	Info	1.0	Fixed
I5: Code and comment discrepancy	Info	1.0	Fixed
I6: Lack of documentation	Info	1.0	Partially fixed
H5: Withdrawal of autocompoundBalanceOf amount reverts	High	2.0	Fixed
M3: simulateAutocompound checks only balance diff	Medium	2.0	Fixed
L10: Pending deposited can't be withdrawn	Low	2.0	Acknowledged
L11: Lack of call to disableInitializers()	Low	2.0	Fixed
L12: Lack of 0 shares check in simulateAutocompound	Low	2.0	Fixed
L13: Lack of 0 shares check in feeBalance	Low	2.0	Fixed

	Severity	Reported	Status
W6: Withdraw can return by 1 wei more than requested	Warning	2.0	Acknowledged
W7: Withdrawal revert due to rounding	Warning	2.0	Acknowledged
W8: unstakePending and activateBalance can revert due to bad timing	Warning	2.0	Acknowledged
I7: Code duplication for ownership	Info	2.0	Acknowledged
I8: Typos in code and comments	Info	2.0	Fixed
I9: Array length validation	Info	2.0	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

Pool.sol

A core contract that represents the staking pool. Users stake and unstake ETH there. It also allows for the management of the validator set - validators can be added, replaced and exited. Additionally, it facilitates the deposits to the Beacon Deposit Contract.

Accounting.sol

A core contract that manages the accounting of the user balances. Apart from keeping track of the deposited balances, it manages which deposits are active (i.e. already in the Deposit Contract), and which are pending. It also tracks the shares and rewards of the users which stem from the autocompounding and restaking mechanism.

Additionally, it implements the logic of managing user withdrawal requests. It tracks who and when requested a withdrawal, and how much.

UnstructuredStorage.sol

The core contracts are upgradeable and use the unstructured storage pattern. The storage addresses of the storage variables are defined through keccak256 hashes of the qualified variable names. The UnstructuredStorage.sol library then manages accesses to the storage variables.

WithdrawTreasury.sol

A simple treasury contract that holds the pending withdrawal amounts. Once enough funds are filled, the user can claim the withdrawal through the Accounting contract and the funds are transferred from the treasury to the user.

RewardsTreasury.sol

A simple treasury contract that holds the funds from staking rewards. These rewards are continuously restaked to the staking pool. The funds can also be used to cover interchange amounts.

DepositContract.sol

The Beacon Deposit Contract is a contract that allows for the deposit of ETH to activate the validators. It is not managed by the Everstake team but deployed and used by the whole ecosystem.

The Pool contract sends to this contract deposits of the users (once the 32 ETH amount is accumulated).

Actors

This part describes actors of the system, their roles, and permissions.

Proxy Owner

The contracts are upgradeable and the owner of the proxy can upgrade the contracts.

Owner

The protocol contracts are ownable - Pool, Accounting and Treasuries all have an owner.

SuperAdmin

The Pool and Accounting contracts introduce a SuperAdmin role. This role can manage important parameters of the protocol.

Governor

The Pool and Accounting contracts introduce a Governor role. This role can manage validators, pause the protocol and set some protocol parameters.

Validator

Validator is an entity that participates in the consensus of the Ethereum protocol. It can be activated by depositing 32 ETH and can earn staking rewards.

5.2. Trust Model

First and foremost, the protocol is upgradeable. That means that the owner of the proxy can change the logic of the protocol. That means that if the owner is compromised, the attacker can steal user funds or brick the protocol.

The implementation of the protocol is currently not open-source therefore the users cannot verify the logic themselves.

Apart from being upgradeable, the protocol has several additional trust assumptions:

- the protocol can be paused - both staking and unstaking,

- fee can be changed to an arbitrary value,
- the treasuries are ownable.

All of the 3 roles (SuperAdmin, Governor, Owner) have considerable privileges and must be trusted. In the case of Accounting, the SuperAdmin should be the Pool contract, which reduces the risk but the SuperAdmin can be changed by the Owner.

The compromise of any of these roles can have severe consequences.

> It is planned to manage the proxy contract with a multisig operated by multiple independent parties such that a single entity wouldn’t be able to compromise the protocol. It is also considered to make the protocol non-upgradeable after the initial stage after it is ensured that the protocol behaves correctly.
>
> Additionally, it is planned to open-source the repository in the future to allow the users to verify the logic themselves. And lastly, any of the special privileges are to be used only in the case of emergency.
>
> — Everstake team

H1: _simulateAutocompound can revert

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Accounting.sol	Type:	Invalid logic

Description

The `_simulateAutocompound` function is one of the core `view` functions for retrieving data about the protocol state. It is used to calculate autocompound balances, pending balances, restaked balances and others.

However, in certain protocol states the function `reverts`. The revert is caused by the following:

1. In the function, `pendingAmount` and `pendingRestaked` amounts are tracked:

```
function _simulateAutocompound() private view returns (uint256
totalShare, uint256 pendingRestaked, uint256 pendingAmount, uint256
activeRound, uint256 unclaimedReward, WithdrawRequestQueue memory queue)
{
    totalShare =
AUTO_COMPOUND_TOTAL_SHARE_POSITION.getStorageUint256();
    pendingRestaked =
PENDING_RESTAKED_VALUE_POSITION.getStorageUint256();
    pendingAmount =
AUTO_COMPOUND_PENDING_SHARE_POSITION.getStorageUint256();
```

2. `pendingAmount` can be higher than `pendingRestaked` when read from the storage in the beginning of the function. This can happen in normal protocol usage. See the following simplified PoC on how to achieve such state.

```
init_deposit_data(rewards_treasury)
```

```
pool.setPendingValidators(deposit_data[:2], from_=a.governor)
pool.stake(SOURCE, value=16 * BN_ETH, from_= a.alice)
pool.stake(SOURCE, value=BN_BEACON from_= a.bob)
rewards_treasury.transact(value=BN_ETH * 20, from_=a.owner)
```

3. Later, unclaimed rewards are added to both of them:

```
pendingAmount += unclaimedReward;
pendingRestaked += unclaimedReward;
totalShare += unclaimedReward;
```

4. Then the following while can be triggered:

```
 while (pendingAmount >= BEACON_AMOUNT){
     activeRound++;
     pendingAmount -= BEACON_AMOUNT;
     pendingRestaked -= BEACON_AMOUNT;
}
```

Because of 3) the `pendingAmount` can be greater than the BEACON_AMOUNT. But because of 2) the `pendingRestaked` can be less than BEACON_AMOUNT. This can result in underflow and revert here: `pendingRestaked -= BEACON_AMOUNT;`.

Vulnerability scenario

Users interact with the protocol in a normal way and it reaches the state as described previously. Then, users want to interact with the protocol, however, for that, they first want to check the state of their balances. Because of the revert, they are not able to do so.

Recommendation

Rewrite the logic of the function such that the assumptions about the relation between `pendingAmount` and `pendingRestaked` are corrected. `pendingRestaked` amount can be lower when entering the function and this fact should be taken into account during the execution of the function.

Fix 2.0

The `pendingRestaked` amount is now handled separately. The activation loop now only decreases the `pendingBalance` and the function assigns the `pendingRestaked` amount after the loop finishes.

Go back to Findings Summary

ackee | blockchain security

H2: DoS due to 0 pending deposits

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Pool.sol, Accounting.sol	Type:	DoS

Description

The protocol allows covering withdrawals through pending deposits. If a staker stakes his ETH whilst not supplying enough ETH to activate a round, the deposit goes to pending and the staker is added to a set of pending stakers for the given round:

```
_slotPendingStakers()[activeRound].add(account);
```

The protocol additionally allows the users to withdraw their pending balance through the `withdrawPending` function. The users can withdraw an arbitrary amount of their pending balance, this includes even the whole share.

If the user decides to withdraw his full pending balance, then the user should be removed from the pending set. However, the `withdrawPending` function lacks the logic to do so.

This has severe implications for the `withdraw` function, which is the main function for managing withdrawals. When a user withdraws, his shares can be interchanged with the pending stakers:

```
while (interchangeWithPendingDeposits > 0 && index < lenght) {
    pendingStaker = pendingStakers.at(index);
    (activatedAmount, isFullyDeposited) =
_activatePendingBalance(pendingStaker, interchangeWithPendingDeposits,
activeRound, activatedRound, true);
    emit InterchangeDeposit(pendingStaker, activatedAmount);
```

```
        interchangeWithPendingDeposits -= activatedAmount;
        if (isFullyDeposited) {
            pendingStakers.remove(pendingStaker);
            lenght--;
        }else{
            index++;
        }
    }
```

It can be seen that the loop runs as long as: `while (interchangeWithPendingDeposits > 0 && index < lenght)`. This is problematic for the following reasons:

1. If the pending staker has a 0 pending balance (it was shown earlier how this can happen), then the `interchangeWithPendingDeposits` variable will not decrease.
2. Additionally, the `length` of the set is basically unbounded. So if normal users are withdrawing from pending deposits, or if an attacker intentionally stakes small amounts and unstakes from pending, then the set can arbitrarily grow.

Because of that, the loop can run arbitrarily long, which can lead to denial of service. The DoS will be caused by not having enough gas to execute the loop.

Exploit scenario

The pending set is filled with stakers with 0 pending balance. This can happen through normal users staking and unstaking from pending (though this is unlikely, but possible), or through an attacker intentionally staking small amounts and unstaking from pending. Such an attack could be for example subsidized by a competing team, which wants to outcompete the protocol.

Then, a normal user tries to unstake his stake. However, because of extremely high gas fees, he is not able to do so.

Recommendation

Fix the `withdrawPending` function to remove the user from the pending set if he withdraws his full pending balance. Additionally, consider whether it makes sense to interchange 0 amount.

Fix 2.0

The function `_withdrawPending` (and generally all functions operating with pending withdrawals) now returns a bool indicating whether there is some pending balance left. If there isn't that means that the account is `fullyWithdrawn` and the account is removed from the pending set.

[Go back to Findings Summary]

H3: Partial DoS due to interchange

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Pool.sol, Accounting.sol	Type:	DoS

Description

This issue is similar to DoS through 0 pending deposits. It is based on the architecture of how the withdrawals are interchanged with pending deposits.

When a user makes a withdrawal, the pending deposits can be interchanged. This allows the pending stakers to immediately gain shares and additionally, it allows the protocol to withdraw only necessary ETH.

However, the problem with using too much gas can happen again. When the user withdraws he can go through the interchange loop:

```
while (interchangeWithPendingDeposits > 0 && index < lenght) {
    pendingStaker = pendingStakers.at(index);
    (activatedAmount, isFullyDeposited) =
_activatePendingBalance(pendingStaker, interchangeWithPendingDeposits,
activeRound, activatedRound, true);
    emit InterchangeDeposit(pendingStaker, activatedAmount);
    interchangeWithPendingDeposits -= activatedAmount;
    if (isFullyDeposited) {
        pendingStakers.remove(pendingStaker);
        lenght--;
    }else{
        index++;
    }
}
```

If there are a lot of pending stakers with low pending balances and the unstaked amount is high, the gas fees can be very high. This can result in the user being denied service.

This will happen because the `interchangeWithPendingDeposits` will decrease too slowly (at the rate of the height of the pending balances of the pending stakers).

Exploit scenario

1. Normal users stake a small amount of ETH, eg the minimum staking amount, so their pending balance is very low.
2. Alternatively, the users can stake higher amounts, but then do a partial withdrawal of the pending balance by calling `withdrawPending` function. Thus their pending balance can be even sub the minimal staking amount.
3. Now, another user wants to unstake his stake and goes through pending deposits interchange. This can results in thousands of interchanges (if the users did withdraw pending balance). As a result, the gas fees can be very high and he is again denied service.
4. Though, the user can be denied service only partially. He can decide to unstake only a small amount of his stake, which will result in only a few interchanges. However, this is not a good user experience.

Recommendation

It is hard to give a recommendation besides a generic one as it would involve redesigning part of the withdrawal logic. The issue lies in the architecture of the interchanges, which can be inherently gas costly. Because this issue can happen fairly often during normal execution, it is recommended to reconsider the architecture of the interchanges.

Fix 2.0

The withdrawal process now contains a new parameter indicating whether the user wishes to perform interchanges or not. It is a tradeoff between withdrawal speed and withdrawal efficiency and users can choose their preferred variant.

However, only adding this parameter doesn't solve the issue completely. Due to bad timing or front-running of the withdrawal transaction, multiple pending deposits can happen before the withdrawal transaction is mined. This can again lead to high gas fees if the interchanges are allowed. The gas usage increases after the user decides to withdraw (based on an estimate of the gas usage) and thus this behavior can be unexpected and unwanted for the user.

Fix 2.1

The withdrawal process was modified to allow the user to select the maximum amount of interchanges he wishes to perform. This allows for deterministic and predictable fees for the user and solves the issue described above.

Go back to Findings Summary

ackee | blockchain security

H4: DoS due to underflow

High severity issue

Impact:	High	Likelihood:	Medium
Target:	AutocompoundAccounting.sol	Type:	DoS

Description

In certain protocol states, the `_withdrawFromAutocompound` can revert. This function gets called from the `withdraw` function and is part of the unstaking process.

The underflow happens on the following line:

```
uint256 rewardsBalance = _shareToAmount(totalShare,
autoCompoundShareIndex, autoCompoundTotalShare) -
originActiveDepositedBalance;
```

It is caused by rounding when performing the integer arithmetic in the conversion functions. Here are the simplified conversion functions:

```
function _shareToAmount(uint256 share, uint256 autoCompoundShareIndex,
uint256 autoCompoundTotalShare) private pure returns (uint256 amount){
    return share * autoCompoundTotalShare / autoCompoundShareIndex;
}

function _amountToShare(uint256 amount, uint256 autoCompoundShareIndex,
uint256 autoCompoundTotalShare) private pure returns (uint256 share){
    share = amount;
    if ((autoCompoundShareIndex > 0) && (autoCompoundTotalShare > 0)) {
        share = share * autoCompoundShareIndex / autoCompoundTotalShare;
    }
    return share;
```

```
}
```

There is no guarantee that the numerator will be divisible by the denominator and thus rounding errors can occur. This can lead to subtle off-by-one errors.

Exploit scenario

Here is a PoC that demonstrates the issue:

```
pool.stake(SOURCE, value=BN_ETH, from_= a.alice)
pool.stake(SOURCE, value=BN_BEACON, from_= a.bob)
pool.stake(SOURCE, value=BN_ETH, from_= a.alice)
pool.stake(SOURCE, value=BN_BEACON, from_= a.bob)

accounting.activateValidators(1, from_=a.governor)

rewards_treasury.transact(value=BN_ETH, from_=a.owner)

pool.stake(SOURCE, value=BN_ETH, from_= a.alice)

pool.unstakePending(BN_ETH, from_= a.alice)

pool.unstake(BN_ETH, from_= a.alice)
pool.unstake(BN_ETH, from_= a.alice)
```

The last unstake will revert due to underflow and thus, Alice will be denied access to her funds.

Recommendation

Relying on precise calculations which are based on integer arithmetic is inherently dangerous. One of the options to avoid rounding errors would be to use a different data type, such as fixed point numbers. However, this would require a major refactoring of the codebase.

Another option could be to check for the loss of precision and to fix it posthoc. That is, the function could be inverted and the result checked that it is the same as the original input.

Fix 2.0

The underlying issue with the loss of precision when performing the integer division is still present and can still lead to subtle off-by-one errors. However, the subtraction that could lead to underflow was fixed by adding an explicit check for the difference introduced by the underflow.

Go back to Findings Summary

M1: Missing whenWithdrawActive modifier

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Pool.sol	Type:	Access controls

Description

The `Pool` contract implements a modifier to stop withdrawals. The stoppage can be done by privileged roles and is then enforced by the `whenWithdrawActive` modifier:

```
modifier whenWithdrawActive() {
     if (PAUSE_WITHDRAW_POSITION.getStorageBool()) revert
Errors.Paused("withdraw");
     _;
 }
```

However, the `unstake` function does not use the modifier:

```
function unstake(uint256 value) external {
```

Exploit scenario

A vulnerability is found in the protocol. To protect the users' funds, the withdrawals are turned off by the privileged roles. However, because the `unstake` function is missing the `whenWithdrawActive` modifier, the withdraw restriction is not enforced and the vulnerability can be exploited.

Recommendation

Add the missing modifier to the `unstake` function.

Fix 2.0

The modifier was added.

[Go back to Findings Summary]

M2: Call to depositedBalanceOf reverts

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Accounting.sol, AutocompoundAccounting.sol	Type:	Contract logic

Description

The `Accounting` contract exposes the function `depositedBalanceOf`. This function calls `_depositedUserBalance` which in certain situations reverts.

The reverting function has the following body:

```
(, depositedBalance) = _autoCompoundUserBalance(account, totalShare,
activeRound, activatedRound);
return depositedBalance - _autoCompoundUserPendingBalance(account,
activeRound) - _autoCompoundUserPendingDepositedBalance(account,
activeRound, activatedRound);
```

The revert is caused by an underflow in the return statement:

The `_autoCompoundUserBalance` has the following if statement at the beginning of its body:

```
if (totalShare == 0){
    return (0, 0);
}
```

The second element in the tuple corresponds to `depositedBalance`. However, it can happen that the user's pending balance will be greater than 0. Thus combined with the default return value of 0, the function will revert due to underflow.

Here is a simplified sequence of steps to achieve this state:

```
pool.setPendingValidators(deposit_data[:2], from_=a.governor)
pool.stake(SOURCE, value=BN_ETH from_= a.alice)
acc.depositedBalanceOf(a.alice, from_=a.alice)
```

Because the if statement with the default return value of 0 is taken only if `totalShare == 0`, it has a low likelihood of happening.

Vulnerability scenario

Users interact with the protocol in a normal way and it reaches the state as described previously. Then, users want to interact with the protocol, however, for that, they first want to check the state of their balances. Because of the revert, they are not able to do so.

Recommendation

Change the default return value, which should be equal to the deposit balance of the user.

Fix 2.0

The function was rewritten. It doesn't now perform the subtraction, instead, it performs a simple comparison and returns autocompound balance if it is smaller than the origin deposited amount.

[Go back to Findings Summary]

L1: Withdraw request array monotonically grows

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	WithdrawRequests.sol	Type:	Gas consumption

Description

Withdrawing staked funds is handled through withdrawal requests. Such a request contains the corresponding value and can be claimed once a sufficient amount of funds accumulates in the withdrawal queue.

The requests are put into an array. When the request is claimed, it is only cleared, the array isn't popped. At the same time, however, if a request was claimed and cleared then future requests can be put into its former place.

However, in certain scenarios (like adding without claiming) the array only grows. As a result, the gas consumption becomes higher and higher the more the given user uses the protocol. This is because the `add` and `claim` functions traverse the whole request array.

Vulnerability scenario

Users use the protocol in unexpected patterns. As a result, they fill the withdraw request array to high values. As such the usage of the protocol becomes very gas expensive for them.

Recommendation

Measure the gas consumption in more unconventional scenarios. If the gas consumption is too high, consider using a different data structure for the requests. Such an approach could be replacing the deleted element with the last element in the array and then popping the last element. This would shrink the array with each claim.

Additionally, mention this in the documentation so the users can interact with the protocol in a way that doesn’t cause high gas consumption.

Fix 2.0

A length threshold was introduced. If is is surpassed than during the claiming of a request the array is shrunk.

Go back to Findings Summary

L2: Lack of 2-phase role transfers

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	OwnableWithSuperAdmin.sol, Governor.sol, TreasuryBase.sol	Type:	Data validation

Description

The protocol defines multiple important roles: `owner`, `SuperAdmin` and `Governor`. The roles can be transferred to new addresses. However, the transfers are done via the classical 1-step approach.

Suppose the following transfer in the `OwnableWithSuperAdmin` contract:

```
function transferOwnership(address newOwner) external virtual onlyOwner {
        if (newOwner == address(0)) revert Errors.ZeroValue("newOwner");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
```

If the `newOwner` was supplied incorrectly, this could lead to dire consequences.

Vulnerability scenario

The ownership is to be transferred, however, an incorrect address is supplied. As a result, the `onlyOwner` functions can't be called anymore (although it could be fixed by an upgrade of the protocol).

Recommendation

One of the common and safer approaches to ownership transfer is to use a two-step transfer process.

Suppose Alice wants to transfer the ownership to Bob. The two-step process would have the following steps:

1. Alice proposes a new owner, namely Bob. This proposal is saved to a variable `candidate`.
2. Bob, the candidate, calls the `acceptOwnership` function. The function verifies that the caller is the new proposed candidate, and if the verification passes, the function sets the caller as the new owner.
3. If Alice proposes a wrong candidate, she can change it. However, it can happen, though with a very low probability that the wrong candidate is malicious (most often it would be a dead address).

Regarding the details of the implementation, one can review the relevant OpenZeppelin contract.

It is recommended to use the two-step approach for all the roles in the protocol.

Fix 2.0

Two-phase ownership transfers were added. Other roles were not changed, but they are under the direct control of the owner.

Go back to Findings Summary

L3: Exiting validators can revert

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Pool.sol	Type:	Contract logic

Description

The `Pool` contract exposes the `markValidatorsAsExited`` function which allows to mark validators' status exited.

The function takes an `uint256 num` argument which represents the number of validators to exit.

The function only requires that the validators are in the state deposited, i.e. this can be seen as the precondition for exitability.

The issue is that not all validators that are deposited can be exited. This is because the exit function requires the deposited validators to be sequentially ordered:

```
for (uint j = 0; j < num; j++) {
     // Deposited validator
     pubKey = set._validatorsPubKeys[set._activeValidatorIndex + j];
     validatorHash = sha256(abi.encodePacked(pubKey));
     // Check and update status
     if (set._validatorStatus[validatorHash] != ValidatorStatus.Deposited)
revert Errors.InvalidValue("status");
     set._validatorStatus[validatorHash] = ValidatorStatus.Exited;
 }
```

This doesn't have to be the case. In certain combinations of adding adding and shifting validators, it can happen that between deposited validators there will be a validator in the state pending and thus the function for exiting will revert although there are enough deposited validators.

Marking the validators exited isn't an essential part of the validator management and the protocol can work without it, thus this is a low-impact issue.

Vulnerability scenario

The protocol is in such a state that n validators could potentially be exited. Additionally, the pending and deposited validators are mixed. When the `markValidatorsAsExited` function is called with n as the argument, the function reverts. This can cause concerns about the validity of the validator management.

Recommendation

Consider ordering (or reordering) the validators in the main validator queue.

Fix 2.0

Pending validators can be reordered by index in the main queue. That allows shifting the validator from the head of main queue. Additionally, the exit function traverses the array and can skip the pending and deposited validators.

We would like to point out, that the current implementation is overly complicated and that we recommend simplification. We recommend properly calculating the gas savings of replacing the elements and comparing them with the additional complexity of quick-sorting the array traversing the array and checking the validator statuses. We consider this to be an overengineered solution.

[Go back to Findings Summary]

L4: Replacing validators lacks validation

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Pool.sol	Type:	Data validation

Description

The `Pool` contract exposes the `replacePendingValidator` function which allows to replace a pending validator with a new one.

However, the function lacks data validation of the data regarding the new validator.

At least, the function should validate the same properties as the function `setPendingValidators`:

```
if (pendingValidators[i].pubkey.length != 48) revert
Errors.InvalidParam("pubkey");
if (pendingValidators[i].signature.length != 96) revert
Errors.InvalidParam("signature");
```

Vulnerability scenario

A pending validator is replaced with a new one. However, the supplied data is incorrect. As a result, this new validator has to be replaced again. Or in the worse case, the mistake will go unnoticed until depositing where it will revert.

Recommendation

Implement as strong validation as possible. At least, the same validation as in the function `setPendingValidators`.

Fix 2.0

The recommended validation was added.

[Go back to Findings Summary]

L5: Validation of owner in treasuries

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	WithdrawTreasury.sol, RewardsTreasury.sol	Type:	Data validation

Description

Both the WithdrawTreasury, and RewardsTreasury receive the owner addresses as an argument in their constructor. However, no data validation is performed in the constructor.

Vulnerability scenario

Due to a bug in the deployment script, the address is not supplied. As a result, the default value (zero address) is used. As such, the contract is deployed without an owner.

Recommendation

For the sake of consistency and adherence to classical development standards, add a zero check to both the constructors.

Fix 2.0

The validations were added.

Go back to Findings Summary

L6: Data validation in initialize functions

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Pool.sol, Accounting.sol	Type:	Data validation

Description

The `Pool` and `Accounting` contracts are initializible. The `initialize` function lack proper data validation.

Pool

The following addresses are not validated: rewardsTreasury, poolGovernor.

Accounting

The following address is not validated: accountingGovernor.

Additionally, the `poolFee` variable also isn't validated.

Vulnerability scenario

Due to a bug in the deployment script, the addresses are not supplied to the initialize functions. As a result, the default value (zero address) is used. As such, the contract is initialized to an invalid state.

Alternatively, a too high fee may be supplied and due to improper validation, it is not caught during the intialization.

Recommendation

For the sake of consistency and adherence to classical development standards, add a zero check to both the constructors. Additionally, add a check for the `poolFee` variable in the `Accounting` contract (at least add the check that is present in the `setFee` function).

Fix 2.0

The validations were added.

[Go back to Findings Summary]

L7: Incorrect return value of _simulateAutocompound

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	Accounting.sol	Type:	Invalid logic

Description

The `_simulateAutocompound` function is one of the core `view` functions for retrieving data about the protocol state. It is used to calculate autocompound balances, pending balances, restaked balances and others.

However, in certain protocol states the function returns an incorrect value.

The function contains the following if statement:

```
    if (unclaimedReward < MIN_RESTAKE_POSITION.getStorageUint256()) {
            unclaimedReward = 0;
            return (totalShare, pendingRestaked, pendingAmount,
activeRound, unclaimedReward, queue);
        }
```

If the branch is taken. Then the `unclaimedReward` variable is set to 0. If the variable originally had a non-zero value, then the function incorrectly returns 0.

Because in the current implementation, the other `view` functions do not consume this particular return field, this issue is rated MEDIUM, as compared to the _simulateAutocompound revert issue where the users are directly impacted.

However, because the protocol is upgradeable, this could become problematic in future versions of the protocol.

Vulnerability scenario

The `_simulateAutocompound` function is called and the mentioned branch is taken. If the `unclaimedReward` variable originally had a non-zero value, then the function incorrectly returns 0. As a result, the consumer of the function can proceed to make invalid calculations.

Recommendation

Remove the erroneous `unclaimedReward = 0;` assignment.

Fix 2.0

The erroneous assignment was removed.

[Go back to Findings Summary]

L8: Upgradeable contract constructor without intializer

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Pool.sol, Accounting.sol	Type:	Upgradeability

Description

The core contracts of the protocol are upgradeable. The implementation contracts use the unstructured storage pattern and have `initialize` functions.

It is a common pattern to disable initialization of the implementation contracts (via a constructor with `initializer` or with a call to `disableInitializers()`). This pattern is not used in the protocol.

The pattern used here is not clean nor conventional, but it works. It works as follows:

1. The implementation contract constructors don't have `initializer` modifier nor do they call `disableInitializers()`.
2. However, the `initialize` function has the `initializer` modifier and calls the init on the parent like this: `OwnableWithSuperAdmin.__OwnableWithSuperAdmin_init();`
3. And the init function also has the `initializer` modifier. A double call to `initializer` will revert in the second call.

Because of 3) the implementation contracts can not be initialized by an attacker.

At the same time, the call to `initialize` will not revert when called from proxy, because the following condition in the `initializer` modifier will pass:

```
(!AddressUpgradeable.isContract(address(this)) && _initialized == 1)
```

The condition will pass because here the address(this) is the address of the proxy and the `initialize` function is called from the constructor of the proxy, so `return account.code.length > 0;` (no code yet) will return false.

So the contracts are initializable when called from the proxy, but not when called directly.

The implementation contracts should be safe even if they were hijacked by an attacker, as they don't contain a self-destruct or a delegatecall, however, it's best to follow best practices.

Recommendation

It is recommended to use the conventional pattern of disabling the initialization in the constructor of the implementation contracts via a call to `disableInitializers()`. Additionally, it is recommended to use the `onlyInitializing` modifier in the init functions.

Guides on upgradeability by OpenZeppelin such as this one can be used as a reference.

Fix 2.0

The fix was not performed correctly and introduced a new bug, which is described in issue L11.

Fix 2.1

The issue L11 was fixed by adding a call to `disableInitializers()` in the constructor of the logic contracts. That also fixed this issue.

Go back to Findings Summary

L9: Insuffitient data validation when composing contracts

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	`**/*`	Type:	Data validation

Description

The protocol consists of multiple contracts. The contracts are then composed together, eg. in the `initialize` function of the `Pool` contract the contract is composed with these addresses:

- address accountingContract,
- address withdrawTreasury,
- address rewardsTreasury.

However, no validation besides zero-address checking is done.

For this purpose, contract ids can be utilized:

1. Define an id for each contract, eg: `bytes32 public constant CONTRACT_ID = keccak256("everstake.accounting")`.
2. When composing contracts, check that the contract id matches:

```
require(
Accounting(implementationAddress).CONTRACT_ID() == keccak256
("everstake.accounting"),
"Not everstake.accounting"
);
```

A similar approach is utilized by OpenZeppelin in their upgradeable pattern.

Vulnerability scenario

A wrong contract address is passed to the `initialize` function of the `Pool` contract. The contract is composed with the wrong contract, which will lead to unintended behavior.

Recommendation

Contract ids are a very cheap and simple way to validate that the contract is composed with the correct contracts. It is recommended to utilize them.

Fix 2.0

Additional validations were not added.

[Go back to Findings Summary]

W1: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Fix 2.0

The optimizer is enabled.

Go back to Findings Summary

W2: Dead code in _autoCompoundUserBalance

Impact:	Warning	Likelihood:	N/A
Target:	AutocompoundAccounting.sol	Type:	Dead code

Description

The `_autoCompoundUserBalance` has the following 2 if statements that have the same guard expression:

```
if (totalShare == 0){
    return (0, 0);
}
```

and:

```
if (totalShare == 0){
    return (0, 0);
}
```

The following reasoning shows why the second statement is dead code:

1. If the `totalShare` argument is 0 at the beginning of the function, the first if statement will be taken and the function will return.
2. The `totalShare` is not written into throughout the execution of the function.
3. Because the variable is not written into, then if at the beginning of the function, it was non-zero, then it will be non-zero also at the place of the second if-statement.

As can be seen, the second statement is dead-code (and the if body is unreachable).

Recommendation

Ensure that the body of the function is well understood. If the developer expects that the `totalShare` can be zero at the place of the second if-statement, then his understanding of the function might be wrong.

If the statement is present as a safety check, then it could be replaced by an `assert`` to express the intentions more clearly.

Fix 2.0

The second said if guard was replaced with an expression checking userTotalShare, not totalShare.

Go back to Findings Summary

W3: Unchecked return of _update

Impact:	Warning	Likelihood:	N/A
Target:	Accounting.sol	Type:	Unchecked return

Description

The Accounting contract has the `_update` function which checks the treasury balance and updates the rewards storage slots.

It returns a bool indicating whether an actual update took place or not. However, these return values are never checked.

Recommendation

In this case, the return values, indeed don't need to be checked, however, it should be clearly indicated, why it is so. Ideally, this would involve a comment explaining why the value can be ignored. Alternatively, a linter directive about unchecked return value could be used.

This issue is mainly included to point out the best security practices - ignoring returns can lead to dire consequences and when a return value is ignored it should be clearly documented why. It will make the code more readable and will help to prevent bugs in the future.

Fix 2.0

The `_update` function now doesn't have a return value.

Go back to Findings Summary

W4: Lack of contract prefix in storage position

Impact:	Warning	Likelihood:	N/A
Target:	Governor.sol	Type:	Storage collision

Description

The protocol uses the unstructured storage pattern in the implementation contracts. Storage variables are stored at positions defined as `keccak` values of the corresponding string. For example, the fee would be stored on the position defined as: `keccak256("accounting.poolFee");`.

To avoid unwanted collisions caused by using the same keccak string, the variable names are prefixed with the contract name, where the variable is declared.

However, this rule is violated in the `Governor` contract, where the `GOVERNOR_POSITION` is defined as:

```
bytes32 internal constant GOVERNOR_POSITION = keccak256("governor");
```

Vulnerability scenario

A future upgrade adds a new contract to the inheritance chain which defines the same slot. As a result, a collision can happen and the storage is corrupted.

Recommendation

Prefix the `GOVERNOR_POSITION` with the contract name.

Fix 2.0

The governor slot is now created using `governor.governor` string.

Go back to Findings Summary

ackee | blockchain security

W5: Pool fee can be set extremely high

Impact:	Warning	Likelihood:	N/A
Target:	Accounting.sol	Type:	Data validation

Description

The pool fee in the `Accounting` contract can be set extremely high:

```
function setFee(uint256 feeValue) external ownerOrSuper {
    if (FEE_DENOMINATOR <= feeValue) revert Errors.InvalidValue("fee");
    _update();
    POOL_FEE_POSITION.setStorageUint256(feeValue);
    emit FeeUpdated(feeValue);
}
```

The fee is later used in calculations as:

```
return amount * poolFee / FEE_DENOMINATOR;
```

So it is possible to set the fee so high, that the protocol fee will almost equal the whole amount.

Vulnerability scenario

By mistake, the `setFee` function is called with a very high value. Because the only check is the check against the denominator, the call passes. As a result, the protocol charges extremely high fees.

Recommendation

It is recommended to add a more fine-grained check for the height of the fee. Calculate the maximal possible fee percentage and check against it. Currently, the fee can be set to 99% which is not realistic in normal protocol operation.

Fix 2.0

The client wants to retain the ability set the fee even to very high values.

[Go back to Findings Summary]

I1: Used library

Impact:	Info	Likelihood:	N/A
Target:	UnstructuredRefStorage.sol	Type:	Unused code

Description

The library `UnstructuredRefStorage` isn't used anywhere in the whole codebase.

Recommendation

To clean up the codebase and make it more readable, remove all unused code.

Fix 2.0

The library was removed.

Go back to Findings Summary

I2: Comparison with role outside modifier

Impact:	Info	Likelihood:	N/A
Target:	Pool.sol	Type:	Best practices

Description

The `Pool` contract has a public function `restake`. The function has access controls, it can only be called by the `WITHDRAW_AUTHORITY`:

```
if (msg.sender != address(uint160(
uint256(WITHDRAW_AUTHORITY_POSITION.getStorageBytes32())))) revert
Errors.InvalidParam("caller");
```

This lowers the readability and is inconsistent with other role checks.

Recommendation

Create a new modifier and perform the check there.

Fix 2.0

The client doesn't want to create a modifier that would be used just in this one place.

Go back to Findings Summary

I3: Function always returns true

Impact:	Info	Likelihood:	N/A
Target:	ValidatorList.sol	Type:	Contract logic

Description

The function `add` in `ValidatorList` has a bool return type. However, in all code paths, it returns true.

Thus, the return is redundant, because it always computes the same information.

Recommendation

Ensure that the function should not return false in any code path. If not, remove the returns entirely.

Fix 2.0

The function now doesn't have a return value.

Go back to Findings Summary

I4: Lack of logging in setters

Impact:	Info	Likelihood:	N/A
Target:	Accounting.sol	Type:	Logging

Description

The `Accounting` contract has various setters, one of them is `setMinRestakeAmount`. The mentioned setter does not emit any events.

In other setters like `setMinStakeAmount` or `setFee` the events are emitted.

Recommendation

Ensure that easy monitoring of the mentioned variable isn't necessary. If it is, add events to the setter function.

Fix 2.0

Logging was greatly improved, additionally, event-based reentracies were fixed.

[Go back to Findings Summary]

I5: Code and comment discrepancy

Impact:	Info	Likelihood:	N/A
Target:	ValidatorList.sol	Type:	Documentation

Description

The comment for the `shift` function in `ValidatorList` states the following:

```
* Returns true if the value is active, false when list hasn't values.
```

However, this is the corresponding function declaration:

```
function shift(List storage set) internal returns (ValidatorListElement
storage validator, bytes storage pendingValidatorPubKey) {
```

As can be seen, the function has different types of return values.

Recommendation

Update the comment to reflect the current implementation of the `shift` function.

Fix 2.0

The comment remained the same while keeping the function signature.

Fix 2.1

The comment was fixed and was updated to:

```
* Returns validator deposit data and pending validator pubkey.
```

[Go back to Findings Summary]

I6: Lack of documentation

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Documentation

Description

The project lacks proper documentation. The whole documentation consists of:

- basic README,
- basic overview of the external function,
- few high-level diagrams.

Additionally, a high number of comments provide almost no information, see:

```
// totalAutocompoundBalance - origin active deposited balance
uint256 rewardsBalance = _shareToAmount(totalShare,
autoCompoundShareIndex, autoCompoundTotalShare) -
originActiveDepositedBalance;

// Case when amount <= 1 share
if (share == 0) {

// Flash data
delete _slotPendingStakers()[activeRound];

// Check that amount fully interchanged
if (interchangeWithPendingDeposits > 0) revert
Errors.InvalidValue("withdraw");
```

Such comments provide almost no value and are rather just distractions.

Documentation should work as a specification and allow reasoning about the code in more high-level abstractions. The protocol contains some quirks, see eg the issue on upgradeability, which are not documented. As such, they can be forgotten and cause issues in the future.

Additionally, the documentation allows external entities to understand the protocol more quickly. It also forces developers to explicitly explain the design decisions, which can help to avoid mistakes.

Recommendation

It is recommended to document the protocol in more detail. The final version of the protocol should be documented using nat-spec. This would include describing all the functions, storage variables, function parameters, etc.

Additionally, document all the quirks and idiosyncrasies of the protocol that can be easily forgotten.

When writing comments, try to explain the why and not the what. The what is already explained by the code, the why is the most important part. See the following example:

```
// Case when amount <= 1 share
if (share == 0) {
```

It can be easily seen that the `share` is checked for 0 from the code. The comment states the same - so it is a what comment. Instead, the why comment would explain why we check for 0, what could happen if we didn't, etc.

Fix 2.0

A lot of nat-spec comments were added. Also, the naming of variables and functions was greatly improved. However, comprehensive documentation is still not present.

Go back to Findings Summary

6. Report revision 2.0

6.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Contracts

Contracts we find important for better understanding are described in the following section.

The system design is almost the same as in the previous revision, see the previous System Overview.

One significant change was that the treasuries were made upgradeable.

Actors

This part describes actors of the system, their roles, and permissions.

The actors are the same as in the previous revision, see the previous System Overview.

6.2. Trust Model

The trust model is almost the same as in the previous revision, see the previous System Overview.

One significant change was that the treasuries were made upgradeable. As a result, the treasuries can be upgraded to an arbitrary logic and thus the owner of the proxy must be trusted. Additionally, a pausing rewards mechanism which affects the majority of functions was added.

ackee | blockchain security

H5: Withdrawal of autocompoundBalanceOf amount reverts

High severity issue

Impact:	High	Likelihood:	Medium
Target:	Accounting.sol, AutocompoundAccounting.sol	Type:	Rounding, DoS

Description

The conversion functions for converting the amount to shares and vice versa are based on integer division and introduce rounding errors. These errors are expected and are accounted for in most places in the codebase. However, in case of withdrawing the user's full position, i.e. calling `pool.unstake(autocompoundBalanceOf(user))` the withdrawal process can revert and the user can't unstake.

The issue doesn't manifest itself in each withdrawal as it is dependent on the pool's balances, but it can happen during normal protocol operations. It is based on the discrepancy between how `autocompoundBalanceOf` is calculated and how are the withdrawal amounts calculated.

The `autocompoundBalanceOf` is calculated as:

```
(uint256 totalPoolBalance,,, uint256 activePendingRound,) =
_simulateAutocompound();
return _userActiveBalance(account, totalPoolBalance, activePendingRound,
ACTIVATED_ROUNDS_NUM_POSITION.getStorageUint256());
```

And in `_userActiveBalance` the following calculation is performed:

```
return _shareToAmount(userTotalShare,
TOTAL_MINTED_SHARE_POSITION.getStorageUint256(), totalPoolBalance);
```

Notice that in the calculation we use the `totalPoolBalance`. The withdrawal amounts are calculated as:

```
depositedWithdrawAmount =  _shareToAmount(share,  totalMintedShare,
(totalPoolBalance - pendingRestakedValue));
withdrawFromPendingAmount = _shareToAmount(share,  totalMintedShare,
(pendingRestakedValue));
```

Here we calculate the amounts separately for `totalPoolBalance - pendingRestakedValue` and for `pendingRestakedValue`.

The issue is that when we calculate the amounts separately we can have two rounding errors, while when we calculate the `autocompoundBalanceOf` we have only one rounding error. See the `shareToAmount` function:

```
return share * totalPoolBalance / totalMintedShare;
```

By dividing by `totalMintedShare` we can get a fraction, if we calculate the amounts separately we can get two fractions. However, if we added the two numbers, the fractions (the fractional parts of the result) would add up and could be bigger than 1 which would reduce the rounding error. For this reason, the `autocompoundBalanceOf` can be higher.

If the `autocompoundBalanceOf` is higher than the sum of the withdrawal amounts, the withdrawal will revert in the `withdraw` function on the following line:

```
if ((withdrawFromActiveBalanceAmount + withdrawFromPendingAmount) < value)
revert Errors.InvalidValue("withdraw");
```

It reverts because the amount to be withdrawn is smaller than the requested amount.

Vulnerability scenario

The following PoC demonstrates that the withdrawal can revert:

```
def test_withdrawal_of_autocompound_balance_of_revert(a : Accounts):
    default_chain.default_tx_account = a.alice

    init_deposit_data(dwo.rewards_treasury)
    pool_c.setPendingValidators(deposit_data[:14, from_=a.governor)

    #stake
    pool_c.stake(SOURCE, value=BN_BEACON, to=dwo.pool, from_= a.alice)
    pool_c.stake(SOURCE, value=BN_BEACON, to=dwo.pool, from_= a.bob)

    #activate validators
    acc_c.activateValidators(2, from_=a.governor)

    #add rewards
    rewards_treasury.transact(value=(35*BN_ETH), from_=a.owner)

    acc_c.autocompound()

    #activate validators
    acc_c.activateValidators(1, from_=a.governor)

    #stake large amount from charlie
    pool_c.stake(SOURCE, value=10*BN_BEACON, from_= a.charlie)

    #activate validators
    acc_c.activateValidators(10, from_=a.governor)

    #add rewards
    rewards_treasury.transact(value=(BN_ETH), from_=a.owner)
    #add rewards
    acc_c.autocompound()

    pool_c.unstake(1, True, SOURCE, from_=a.bob)
```

```
    with must_revert(Errors.InvalidValue("withdraw")):
        pool_c.unstake(acc_c.autocompoundBalanceOf(a.alice), True, SOURCE,
from_=a.alice)
```

Recommendation

This issue was sent to the Everstake team as soon as it was discovered and they fixed it without us providing a recommendation. The fix is mentioned in the next section.

Fix 2.1

The `autocompoundBalanceOf` is now calculated in two steps, which mimics the two-step `withdrawal` process:

```
function _autocompoundBalanceOf(address account, uint256 totalPoolBalance,
uint256 pendingRestaked, uint256 activePendingRound) internal view returns
(uint256){
        uint256 activatedRoundsNum =
ACTIVATED_ROUNDS_NUM_POSITION.getStorageUint256();
        // active amount + pending amount
        return _userActiveBalance(account, totalPoolBalance -
pendingRestaked, activePendingRound, activatedRoundsNum) +
_userActiveBalance(account, pendingRestaked, activePendingRound,
activatedRoundsNum);
    }
```

This works because the rounding errors are now introduced also in the `autocompoundBalanceOf` function.

[Go back to Findings Summary]

M3: simulateAutocompound checks only balance diff

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Accounting.sol	Type:	Contract logic

Description

The `simulateAutocompound` is a `view` function that is supposed to simulate the `autocompound` function without performing the corresponding storage updates.

The actual `_autocompound` function bases the autocompounding process on the rewards storage slot:

```
_update();

uint256 rewards = REWARDER_REWARDS_POSITION.getStorageUint256();
// Autocompound only if amount gt or eq than min stake amount
if (rewards < MIN_RESTAKE_POSITION.getStorageUint256()){
    return;
}
```

The `_simulateAutocompound` function bases the autocompounding process on the balance diff of the rewards treasury:

```
uint256 balanceDiff = REWARDS_TREASURY_POSITION.getStorageAddress().balance
- REWARDER_BALANCE_POSITION.getStorageUint256();
if (balanceDiff == 0) {
    return (totalPoolBalance, pendingRestaked, pendingAmount,
activePendingRound, queue);
}
```

Additionally, the protocol allows to call the `update` function independently of the autocompounding process, i.e., only the rewards storage slots are updated.

If the `_update` is called independently, then due to the updated rewards storage slot, the balance diff will be 0. At the same time, the rewards are still not autocompounded (the autocompound process is update + autocompound rewards). But because the `_simulateAutocompound` function only checks the balance diff it will prematurely return the (incorrect) values.

This can lead to returning incorrect values in the `view` functions that utilize the `_simulateAutocompound`. The most severe scenario found can lead to user transactions reverting. If the autocompounding would activate a round it could decrease the actual pending amount. This is because the current pending amount would be used to activate the current round and the remaining amount (which is lower than 32 ETH) could be lower than the previous pending amount.

If a user creates a `withdrawPending` transaction assuming that the pending amount is the value reported by the `_simulateAutocompound` his transaction could revert because the actual pending amount could be lower (i.e. the protocol doesn't have enough pending amount to cover the withdrawal).

Vulnerability scenario

The following PoC demonstrates the said issue. It is demonstrated using chain snapshots (state of the chain at the time of the snapshot) which are explained inside comments of the PoC.

```
def test_simulate_autocompound_wrong_return_due_to_previous_update(a :
Accounts, dwo : Deployments):
    default_chain.default_tx_account = a.alice

    init_deposit_data(dwo.rewards_treasury)
```

```
    pool_c.setPendingValidators(deposit_data[:2], from_=a.governor)

    #stake the beacon amount so that we have some shares
    pool_c.stake(SOURCE, value=BN_BEACON, from_= a.bob)

    #activate validators
    acc_c.activateValidators(1, from_=a.governor)

    rewards_treasury.transact(value=BN_ETH, from_=a.owner)
    acc_c.update()
    rewards_treasury.transact(value=2, from_=a.owner)
    acc_c.update()
    acc_c.autocompound()
    rewards_treasury.transact(value=BN_ETH//2, from_=a.owner)

    #snapshot of the chain before calling the independent update
    before_update = default_chain.snapshot()

    acc_c.update()

    correct_autocompound_balance_of_bob = 0
    #snapshot of the chain before calling the autocompound function
    # - autocompound has two parts: update + autocompound of rewards, so
the following
    #   call to autocompound will finish the autocompounding process and we
will
    #   be able to get the correct balance of bob
    before_autocompound = default_chain.snapshot()
    dwo.acc_c.autocompound()
    correct_autocompound_balance_of_bob =
acc_c.autocompoundBalanceOf(a.bob)

    #we revert to the state before the call to autocompound and call the
autcompoundBalanceOf which
    #uses _simulateAutocompound under the hood. Because the function
contains the said bug
    #the returned amount will be different from the value computed using
the non-simulated autocompound
    default_chain.revert(before_autocompound)
    assert acc_c.autocompoundBalanceOf(a.bob) !=
correct_autocompound_balance_of_bob
```

```
    #we revert to the state before the independent update and call the
autcompoundBalanceOf which
    #now correctly accounts for the accumulated rewards and correctly
computes the autocompound balance
    default_chain.revert(before_update)
    assert acc_c.autocompoundBalanceOf(a.bob) ==
correct_autocompound_balance_of_bob
```

Recommendation

Implement the same logic for the `_simulateAutocompound` as for the `_autocompound` function. This means that the `_simulateAutocompound` should also consider the rewards storage slot, not only the balance diff.

Fix 2.1

The `_simulateAutocompound` function was modified to account for the `unclaimedRewards`:

```
if (balanceDiff == 0 && unclaimedReward == 0) {
    return (totalPoolBalance, pendingRestaked, pendingAmount,
activePendingRound, queue);
}
```

The PoC now throws an error on the assert:

```
assert dwo.acc_c.autocompoundBalanceOf(a.bob) !=
correct_autocompound_balance_of_bob
```

That implies that the values are now the same.

Go back to Findings Summary

L10: Pending deposited can't be withdrawn

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Pool.sol, Accounting.sol, AutocompoundAccounting.sol	Type:	Protocol design

Description

Users' stake can be in multiple states - pending, pending deposited and active. If the stake is in the states pending or active, withdrawal requests can be immediately issued. But if the stake is in the state pending deposited (i.e., the stake is deposited to the validator but is waiting for the activation), the user can't initiate the withdrawal process of this stake. If he does so, the transaction will revert due to an insufficient amount of shares (which are minted during activation).

The activation process can take days which forces the users to wait to long time to initiate the withdrawal process.

The protocol also allows for batch deposits, which batch the stake amount and set the `depositRound` to `activePendingRound - 1`. So if the stake was staked sequentially, the user could be minted shares in an earlier round. Due to batching, he gets the shares in the last round relative to the batch amount. In such a case, the user would be allowed to withdraw even further.

See the following PoC:

```
default_chain.default_tx_account = a.alice
#stake
pool_c.stake(SOURCE, value=4 * BN_BEACON, from_= a.alice)
pool_c.stake(SOURCE, value=BN_BEACON, from_= a.bob)
```

```
pool_c.stake(SOURCE, value=2 * BN_ETH, from_= a.alice)
pool_c.stake(SOURCE, value=BN_BEACON, from_= a.bob)

acc_c.activateValidators(1, from_=a.governor)

rewards_treasury.transact(value=BN_ETH, from_=a.owner)

pool_c.stake(SOURCE, value=BN_ETH, from_= a.alice)

balance_alice = a.alice.balance
pool_c.unstakePending(BN_ETH, from_= a.alice)
assert a.alice.balance == balance_alice + BN_ETH


with must_revert(Errors.InvalidValue("withdrawable balance")):
    balance_alice = a.alice.balance
    pool_c.unstake(BN_ETH, True, SOURCE, from_= a.alice)
    assert a.alice.balance == balance_alice + BN_ETH

#activate validators -> this will finally enable alice to unstake
acc_c.activateValidators(3, from_=a.governor)

#alice finally unstakes
balance_alice = a.alice.balance
pool_c.unstake(BN_ETH, True, SOURCE, from_= a.alice)
assert a.alice.balance == balance_alice + BN_ETH
```

Recommendation

This limitation is imposed by the ETH staking process. Consider adding a mechanism that would allow the user to make withdrawal requests even if it is in the pending deposited state. If this would be complicated, ensure that users are aware of this limitation and that this information is clearly stated in the documentation.

Fix 2.1

A notice was added to the code documentation:

```
/// @notice Pending deposited can't be unstaked till validator activation
PendingBalance[] pendingDepositedBalances;
```

Go back to Findings Summary

L11: Lack of call to disableInitializers()

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Pool.sol, Accounting.sol, RewardsTreasury.sol, WithdrawTreasury.sol	Type:	Front-running

Description

The upgradeable pattern was rewritten to use `initializer` and `onlyInitializing` modifiers. The previous version disallowed initialization of the logic contract due to a special quirk, se L8 (this no longer applies in the current version).

The logic contracts don't have explicit constructors (and thus don't have calls to `initializer` or `disableInitializers` modifiers). As such the logic contracts are vulnerable to front-running the `initialize` transaction.

The current logic contracts can't be exploited if the `initialize` transaction is front-run. However, allowing the attacker to initialize the logic contract is a bad practice and should be avoided (at least for the sake of reputation).

Exploit scenario

1. Alice deploys the logic contracts and calls the `initialize` function on them.
2. Eve watches the mempool and sees the said transaction and makes the same one by herself.
3. Eve adds a higher gas price to her transaction and gets it accepted first and thus she now owns the privileged roles in the logic contract.

Recommendation

This guide on upgradeability by OpenZeppelin explains this very issue and is recommended to be followed. Especially follow the part `Initializing the Implementation Contract`.

Fix 2.1

The upgradeable contracts received an explicit constructor that calls the `disableInitializers()` function. This prevents an attacker from being able to call `initialize` on the logic contract because it has the `initializer` modifier.

Go back to Findings Summary

L12: Lack of 0 shares check in simulateAutocompound

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Accounting.sol	Type:	Contract logic

Description

The `simulateAutocompound` is a `view` function that is supposed to simulate the `autocompound` function without performing the corresponding storage updates.

The `autocompound` function has two parts:

1. update the rewards balances,
2. autocompound the updated rewards.

In the `update` part there is the following check present:

```
// Not update if nothing on deposit
if (TOTAL_BALANCE_POSITION.getStorageUint256() == 0){
    return;
}
```

This check is present to avoid updating rewards before any shares are minted (which would cause some of the rewards to be missed).

The `_simulateAutocompound` function lacks this if-statement and thus can incorrectly autocompound these rewards and thus return incorrect values. These wrong values shouldn't impact the user view function and thus the impact is low.

Vulnerability scenario

The following PoC demonstrates that the lack of the if-statement can lead to reporting incorrect values:

```
def test_simulate_autocompound_missing_0_shares_check(a : Accounts, dwo :
Deployments):
    default_chain.default_tx_account = a.alice

    init_deposit_data(dwo.rewards_treasury)
    dwo.pool_c.setPendingValidators(deposit_data[:2], from_=a.governor)

    dwo.rewards_treasury.transact(value=BN_ETH, from_=a.owner)

    dwo.acc_c.update()

    assert dwo.acc_c.balance == 0

    assert dwo.acc_c.pendingBalance() != 0 and
dwo.acc_c.pendingRestakedRewards() != 0
```

Recommendation

Implement the simulation of the said if-statement also in the `_simulateAutocompound` function.

Fix 2.1

The following if-statement was added to the `_simulateAutocompound` function:

```
// Not update if nothing on active balance
if (TOTAL_BALANCE_POSITION.getStorageUint256() == 0){
    return (totalPoolBalance, pendingRestaked, pendingAmount,
activePendingRound, queue);
}
```

The PoC now throws an error.

Go back to Findings Summary

L13: Lack of 0 shares check in feeBalance

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Accounting.sol	Type:	Contract logic

Description

The `feeBalance` is a `view` function that is supposed to return the unclaimed balance fee.

As was the case with _simulateAutocompound 0 shares check, the function doesn't take into consideration the following if-statement in the update function:

```
// Not update if nothing on deposit
if (TOTAL_BALANCE_POSITION.getStorageUint256() == 0){
    return;
}
```

This check is present to avoid updating rewards before any shares are minted (which would cause some of the rewards to be missed).

The `feeBalance` function calls `_balanceDiffWithoutClosedValidators` which has the following body:

```
balanceDiff = REWARDS_TREASURY_POSITION.getStorageAddress().balance -
REWARDER_BALANCE_POSITION.getStorageUint256();
uint256 closedValidatorsNum = _calculateValidatorClose(balanceDiff);
balanceDiff -= (closedValidatorsNum * BEACON_AMOUNT);
return balanceDiff;
```

As can be seen, diff between `balance` and `REWARDER_BALANCE` is taken. But if the shares are equal to zero, the rewards can't be updated which this function doesn't take into consideration.

Vulnerability scenario

The following PoC demonstrates that the lack of the if-statement can lead to reporting incorrect values:

```
def test_fee_balance_missing_0_shares_check(a : Accounts, dwo :
Deployments):
    default_chain.default_tx_account = a.alice

    init_deposit_data(dwo.rewards_treasury)
    dwo.pool_c.setPendingValidators(deposit_data[:2], from_=a.governor)

    dwo.rewards_treasury.transact(value=BN_ETH, from_=a.owner)

    assert dwo.acc_c.feeBalance() != 0
```

Recommendation

Implement the said if-statement also in the `_balanceDiffWithoutClosedValidators` function.

Fix 2.1

The if-statement was added to the `_balanceDiffWithoutClosedValidators` function:

```
 function _balanceDiffWithoutClosedValidators() private view returns
(uint256 balanceDiff) {
    if (TOTAL_BALANCE_POSITION.getStorageUint256() == 0) {
        return 0;
    }
```

Go back to Findings Summary

ackee | blockchain security

W6: Withdraw can return by 1 wei more than requested

Impact:	Warning	Likelihood:	N/A
Target:	Accounting.sol, AutocompoundAccounting.sol	Type:	Integer division rounding

Description

In certain protocol states the user can end up withdrawing by 1 wei more than what was requested.

This is caused by increasing the share count in _withdraw function in AutocompoundAccounting:

```
if (amount <= withdrawAmount || share == totalShare) {
        break;
    }

    share += 1;
} while (true);
```

The share is increased by 1 for each non-breaking iteration and the share amount is then used in the 2 conversions for depositedWithdrawAmount and withdrawFromPendingAmount. For both cases, the amount can increase by 1 and thus the resulting amount is higher by 1 wei.

Recommendation

Ensure that this behavior is intended and document it.

Fix 2.1

The issue was acknowledged as part of the share accounting system. A notice was added to the code to document the behavior.

Go back to Findings Summary

W7: Withdrawal revert due to rounding

Impact:	Warning	Likelihood:	N/A
Target:	Accounting.sol, AutocompoundAccounting.sol	Type:	DoS, rounding

Description

In certain protocol states users' withdrawals can revert. The main issue lies in the following two facts:

1. a user tries to withdraw more than is his autocompound balance (this obviously should revert),
2. at the same time the user provides a withdrawal amount that, after conversion, corresponds to all his shares, i.e., the user only tries to withdraw an amount corresponding to his share balance (this should not revert).

Assume the `_withdraw` function in `AutocompoundAccounting`: normally, if the amount is less or equal to withdrawAmount, then a new iteration of the do-while would be taken. But if all the shares are used, then the loop breaks.

```
if (amount <= withdrawAmount || share == totalShare) {
        break;
    }

    share += 1;
} while (true);
```

If the loop breaks in a situation where amount < withdrawAmount, then the function `withdraw` in `Accounting` will revert because of:

```
if ((withdrawFromActiveBalanceAmount + withdrawFromPendingAmount) < value)
revert Errors.InvalidValue("withdraw");
```

and that would revert the whole withdrawal process.

The conclusion is that there exist protocol states, where the user can't withdraw all his pool share.

Vulnerability scenario

```
default_chain.tx_callback = print_tx
    default_chain.default_tx_account = a.alice
    init_deposit_data(dwo.rewards_treasury)
    pool_c.setPendingValidators(deposit_data[:14], from_=a.governor)

    #stake
    pool_c.stake(SOURCE, value=BN_BEACON, from_= a.alice)
    pool_c.stake(SOURCE, value=BN_BEACON, from_= a.bob)

    #activate validators
    dwo.acc_c.activateValidators(2, from_=a.governor)

    #add rewards
    rewards_treasury.transact(value=(35*BN_ETH), from_=a.alice)

    acc_c.autocompound()

    #activate validators
    acc_c.activateValidators(1, from_=a.governor)

    #stake large amount from charlie
    pool_c.stake(SOURCE, value=10*BN_BEACON, from_= a.charlie)

    #activate validators
    acc_c.activateValidators(10, from_=a.governor)

    #charlie can unstake everything but his pending balance
    charlie_pending_balance = acc_c.pendingBalanceOf(a.charlie)
    with must_revert(Errors.InvalidValue("withdraw")):
```

```
        #revert due to an off-by-one error:
        #      ├──── console.log('withdrawFromActiveBalanceAmount',
3173499999999999999, 'withdrawFromPendingAmount', 0) ✓ [STATICCALL]
        #      └──── console.log('value', 3173500000000000000) ✓
[STATICCALL]
        # will revert here:  if ((withdrawFromActiveBalanceAmount +
withdrawFromPendingAmount) < value) revert Errors.InvalidValue("withdraw");
        pool_c.unstake(10*BN_BEACON - charlie_pending_balance, True,
SOURCE, from_=a.charlie)
```

Recommendation

Ensure that this behavior is intended and document it.

[Go back to Findings Summary]

ackee | blockchain security

W8: unstakePending and activateBalance can revert due to bad timing

Impact:	Warning	Likelihood:	N/A
Target:	Pool.sol, Accounting.sol	Type:	DoS, timing

Description

Due to bad timing, the call to `unstakePending` and `activateStake` can revert. Both the `unstakePending` and `activateBalance` and dependent on the user having a pending balance:

- in the case of `unstakePending` the user has to have sufficient pending balance,
- in the case of `activateBalance` the user has to have a non-zero pending balance.

If these conditions aren't met then the transactions will revert.

If a user makes a `pendingDeposit` transaction but this transaction is mined after new deposit transactions that activate a round (and make the pending balance lower), the transaction will revert. The problem for `activateBalance` is analogical.

Vulnerability scenario

1. Alice makes a `withdrawPending` transaction.
2. Bob makes a deposit transaction with an amount sufficient to activate a round. As a result, the pending balance goes to the pending deposited state.
3. Bob's transaction is mined before Alice's.

4. Alice’s transaction reverts, she loses gas and the pending amount is not withdrawn.

Recommendation

Ensure that this behavior is intended and document it.

Go back to Findings Summary

I7: Code duplication for ownership

Impact:	Info	Likelihood:	N/A
Target:	TreasuryBase.sol, OwnableWithSuperAdmin.sol	Type:	Code duplication

Description

Both the `TreasuryBase` and `OwnableWithSuperAdmin` contracts implement the logic for ownable contracts with 2-step ownership transfers.

This increases the probability of copy-paste errors and makes the code harder to maintain.

Recommendation

Create a new separate contract that would implement the said logic. Make `TreasuryBase` and `OwnableWithSuperAdmin` inherit from it.

Go back to Findings Summary

ackee | blockchain security

I8: Typos in code and comments

Impact:	Info	Likelihood:	N/A
Target:	WithdrawRequests.sol, AutocompoundAccounting.sol, Accounting.sol	Type:	Code quality

Description

The code contains multiple typos both in the code and in the comments. See:

1. `WithdrawRequests`: REQESTS (instead of REQUESTS):

```
uint8 constant MAX_REQESTS_LEN = 4;
```

2. `AutocompoundAccounting`: _refreshedAccount (instead of _refreshAccount)

```
function _refreshedAccount(address sourceAccount, uint256
activePendingRound, uint256 activatedRoundsNum) internal returns
(StakerAccount storage staker)
```

3. `Accounting`: roung (instead of round)

```
// Close current pending roung
```

4. `Accounting`: exected (instead of expected)

```
* @notice Return num of validators exected to close
```

Typos make the code harder to read and make the code look less professional.

Recommendation

Fix the typos in the code and comments.

Fix 2.1

The typos were fixed.

[Go back to Findings Summary]

I9: Array length validation

Impact:	Info	Likelihood:	N/A
Target:	ValidatorList.sol	Type:	Data validation

Description

The `reorderPending` function allows reordering the pending validators based on their index in the main queue. The function is called from the `Pool` contract without any prior validation.

The body of the function is implemented as:

```
quickSort(set._values, set._activePendingElementIndex, set._values.length -
1);
```

If the length is 0, then the call will revert due to underflow.

Recommendation

Add data validation or error handling to allow for more transparent behavior.

Fix 2.1

New validation to prevent the mentioned case was added:

```
if (set._values.length == 0 || set._activePendingElementIndex ==
set._values.length) revert Errors.InvalidValue("empty list");
```

[Go back to Findings Summary](#)

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Everstake: Ethereum Staking Protocol, 5.9.2023.

ackee | blockchain security

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Superclass/Ancestor of C

A contract that C inherits/derives from.

Subclass/Child of C

A contract that inherits/derives from C.

Syntactic contract

A Solidity contract. May have an inheritance chain, and may be deployed.

Deployed contract

An EVM account with non-zero code. If its source was written in Solidity, it was created through at least one syntactic contract. If that contract had superclasses (parents), it would be composed of multiple syntactic contracts.

Init/initialization function

A non-constructor function that serves as an initializer. Often used in upgradeable contracts.

External entrypoint

A `public` or `external` function.

Public/Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Mutating function

A non-`view` and non-`pure` function.

Appendix C: Woke fuzz tests

During the audit it was uncovered that the library `ValidatorList` contained bugs. The library is optimized for lower gas usage and has higher complexity. To validate its logic, we wrote a differential fuzz test that compares the output of the library with an output of an independent implementation that we built.

We built the alternative implementation in Python and used the Woke framework to test that the invariants we defined hold across randomized sequences of transactions and inputs.

C.1. Tests

The following test shows the differential fuzz test of the `ValidatorList` library. The `@flow` decorator defines the functions that are used to make state changes. The `@invariant` decorator defines the properties that must hold after each state change.

To enable testing the library independently of the rest of the system, we mocked it into a contract.

Further information about fuzzing in Woke can be found in the documentation.

Ouput

We ran the tests with a high number of different random seeds, where each of the runs contained hundreds of state changes. The output of the library and our implementation were the same.

Test code

The following snippet contains the class that holds the state and implements the flow functions and the invariants.

```
class ValidatorFuzzTest(FuzzTest):
    remaining_validators : Set[int]
    pending_validators: Set[int]

    def __init__(self):
        self.deposit_data = []

    def pre_sequence(self):
        self.rewards_treasury =
RewardsTreasury.deploy(default_chain.accounts[0],
from_=default_chain.accounts[0])
        self.validator_list =
MockValidatorList.deploy(from_=default_chain.accounts[0])
        if not self.deposit_data:
            self.deposit_data = get_deposit_data(str(
self.rewards_treasury.address))
        #init testing variables
        self.remaining_validators = set()
        for i in range(len(self.deposit_data)):
            self.remaining_validators.add(i)
        #indexes of pending validators
        self.pending_validators = set()
        #head of the list, points to the first active validator
        self.active_pending_index = 0
        #list of indexes of validators in the list
        #we would usually index here by active_validator_index
        #the result is the index of the validator in the deposit_data
        self.validator_indexes = []
        #total number of validators added to the list
        self.total_validator_num = 0
        #maps status of each validator
        #key is the index of the validator in the deposit_data, value is
the status
        self.statuses = {}
        for index in range(len(self.deposit_data)):
            self.statuses[index] = ValidatorList.ValidatorStatus.Unknown
        #tracks the indexes of the validators as they were added to the
list
        self.pubkey_ordering = []
```

```
        self.active_validator_index = 0
        self.exited_validators = set()


    def add_index(self, index):
        if self.active_validator_index > 0:
            self.active_validator_index -= 1
            list_index = self.active_validator_index
            self.pubkey_ordering[list_index] = index
            #assert self.active_pending_index > 0
        else:
            self.pubkey_ordering.append(index)
            self.total_validator_num += 1
        if self.active_pending_index > 0:
            self.active_pending_index -= 1
            assert index not in self.validator_indexes
            self.validator_indexes[self.active_pending_index] = index
        else:
            assert index not in self.validator_indexes
            self.validator_indexes.append(index)

    @flow()
    def add_validators(self) -> None:
        if len(self.remaining_validators) <= 0:
            with must_revert(Errors.InvalidParam("validator known")):
                self.validator_list.add(self.deposit_data[random.randint(0,
len(self.deposit_data) - 1)])
        num : int = random.randint(0, 3)
        num = min(num, len(self.remaining_validators))
        for _ in range(num):
            index = random.choice(tuple(self.remaining_validators))
            assert self.validator_list.getStatus(
self.deposit_data[index].pubkey) == ValidatorList.ValidatorStatus.Unknown
            self.validator_list.add(self.deposit_data[index])
            self.pending_validators.add(index)
            self.remaining_validators.remove(index)
            self.add_index(index)
            self.statuses[index] = ValidatorList.ValidatorStatus.Pending

    @flow()
    def shift_validators(self) -> None:
        if len(self.pending_validators) > 0:
```

```
            num : int = random.randint(1, 3)
            num = min(num, len(self.pending_validators))
            for _ in range(num):
                self.validator_list.shift()
                self.pending_validators.remove(self.validator_indexes[
self.active_pending_index])
                self.statuses[self.validator_indexes[
self.active_pending_index]] = ValidatorList.ValidatorStatus.Deposited
                self.active_pending_index += 1
        else:
            with must_revert("Pending validator"):
                self.validator_list.shift()


    @flow()
    def replace_validators(self) -> None:
        if len(self.remaining_validators) <= 0:
            return
        replace_with = random.choice(tuple(self.remaining_validators))
        if len(self.pending_validators) > 0:
            replace_at = random.randint(0, len(self.pending_validators) -
1)
            assert replace_at + self.active_pending_index < len(
self.validator_indexes)
            assert self.statuses[self.validator_indexes[replace_at +
self.active_pending_index]] == ValidatorList.ValidatorStatus.Pending
            assert self.validator_list.getStatus(self.deposit_data[
self.validator_indexes[replace_at + self.active_pending_index]].pubkey) ==
ValidatorList.ValidatorStatus.Pending
            self.validator_list.replace(replace_at,
self.deposit_data[replace_with])
            #remove the replaced validator from pending and add the new one
            self.pending_validators.remove(
self.validator_indexes[replace_at + self.active_pending_index])
            self.pending_validators.add(replace_with)
            #the replaced validator can now be used again
            self.remaining_validators.add(self.validator_indexes[replace_at
+ self.active_pending_index])
            self.remaining_validators.remove(replace_with)
            #update status
            self.statuses[replace_with] =
ValidatorList.ValidatorStatus.Pending
```

```
            self.statuses[self.validator_indexes[replace_at +
self.active_pending_index]] = ValidatorList.ValidatorStatus.Unknown
            for idx, i in enumerate(self.pubkey_ordering):
                if i == self.validator_indexes[replace_at +
self.active_pending_index]:
                    self.pubkey_ordering[idx] = replace_with
                    break

            assert self.validator_list.getStatus(self.deposit_data[
self.validator_indexes[replace_at + self.active_pending_index]].pubkey) ==
ValidatorList.ValidatorStatus.Unknown
            assert replace_with not in self.validator_indexes
            self.validator_indexes[replace_at + self.active_pending_index]
= replace_with
            assert self.validator_list.getStatus(
self.deposit_data[replace_with].pubkey) ==
ValidatorList.ValidatorStatus.Pending

        else:
            #no pending validators, active element index must be equal to
length
            #thus we should revert on out-of-bounds: uint256 validatorIndex
= set._values[index].index;
            with
must_revert(Panic(PanicCodeEnum.INDEX_ACCESS_OUT_OF_BOUNDS)):
                assert self.active_pending_index ==
self.validator_list.valuesLength()
                self.validator_list.replace(0,
self.deposit_data[replace_with])


    @flow()
    def mark_as_exited(self) -> None:
        num : int = random.randint(0, 3)
        for _ in range(num):
            if self.total_validator_num <= 0:
                with must_revert(Errors.InvalidValue("index")):
                    self.validator_list.markAsExited(1)
                return
            if self.active_validator_index >= len(self.pubkey_ordering):
                return
            index = self.pubkey_ordering[self.active_validator_index]
```

```
            if self.statuses[index] !=
ValidatorList.ValidatorStatus.Deposited:
                with must_revert(Errors.InvalidValue("status")):
                    self.validator_list.markAsExited(1)
            else:
                assert self.statuses[index] ==
ValidatorList.ValidatorStatus.Deposited
                assert self.validator_list.getStatus(
self.deposit_data[index].pubkey) == ValidatorList.ValidatorStatus.Deposited
                self.validator_list.markAsExited(1)
                self.statuses[index] = ValidatorList.ValidatorStatus.Exited
                self.exited_validators.add(index)
                self.active_validator_index += 1
                assert self.validator_list.getStatus(
self.deposit_data[index].pubkey) == ValidatorList.ValidatorStatus.Exited

    @invariant()
    def remaining_are_unknown(self) -> None:
        for i in self.remaining_validators:
            assert self.statuses[i] ==
ValidatorList.ValidatorStatus.Unknown
            assert self.validator_list.getStatus(
self.deposit_data[i].pubkey) == ValidatorList.ValidatorStatus.Unknown


    @invariant()
    def active_validator_index(self):
        assert self.active_validator_index <= self.total_validator_num
        assert self.active_validator_index ==
self.validator_list.activeValidatorIndex()

    @invariant()
    def validator_statuses(self):
        for i in range(len(self.deposit_data)):
            assert self.validator_list.getStatus(
self.deposit_data[i].pubkey) == self.statuses[i]

    @invariant()
    def validator_length(self) -> None:
        assert self.total_validator_num == self.validator_list.length()
        assert len(self.pubkey_ordering) == self.total_validator_num
```

```python
    @invariant()
    def get_pending_validator(self) -> None:
        for i in range(len(self.pending_validators)):
            pubk = self.validator_list.getPending(i)
            assert pubk == self.deposit_data[self.validator_indexes[i +
self.active_pending_index]].pubkey
            assert self.statuses[self.validator_indexes[i +
self.active_pending_index]] == ValidatorList.ValidatorStatus.Pending
            status = self.validator_list.getStatus(pubk)
            assert status == ValidatorList.ValidatorStatus.Pending

    @invariant()
    def get_all_validators(self) -> None:
        for i in range(self.total_validator_num):
            pubk, status = self.validator_list.get(i)
            assert pubk == self.deposit_data[
self.pubkey_ordering[i]].pubkey
            assert status == self.statuses[self.pubkey_ordering[i]]

    @invariant()
    def active_index(self) -> None:
        assert self.active_pending_index ==
self.validator_list.activeElementIndex()

    @invariant()
    def pending_validator_length(self) -> None:
        assert len(self.pending_validators) ==
self.validator_list.pendingLength()

    @invariant()
    def total_len_lt_pending(self) -> None:
        assert self.total_validator_num >=
self.validator_list.pendingLength()

    @invariant()
    def active_index_le_total_length(self) -> None:
        assert self.active_pending_index <= self.validator_list.length()
```

Quick sort test

The following test shows a differential fuzz test of the quick sort algorithm which sorts the pending queue based on the index in the main queue. The test is based on a mock ValidatorSort contract which allows adding ValidatorListElements with random indexes and then exposes the sort function (which is used in the actual ValidatorList). The test creates a model of the queue in Python, sorts it in Python and then compares the result with the result of the sort function in the contract.

```
def add_data(vs: ValidatorSort) -> List[int]:
    data = set()
    data_list = []
    num_of_elems = random_int(1, 100, min_prob=0.1, max_prob=0.1)
    while len(data) != num_of_elems:
        index = random_int(0, 2**32, min_prob=0.1)
        if index not in data:
            data.add(index)
            data_list.append(index)
            vs.add(index)
    active_pending = random_int(0, len(data_list)-1)
    return data_list, active_pending


def validate(data: List[int], vs: ValidatorSort, active_pending) -> bool:
    data = data[active_pending:]
    data.sort()

    vs.sort(active_pending)
    vs_data = vs.get_data()[active_pending:]

    assert len(data) == len(vs_data)
    for i in range(len(data)):
        assert data[i] == vs_data[i].index


@default_chain.connect()
def test_default():
    default_chain.set_default_accounts(default_chain.accounts[0])
    default_chain.tx_callback = tx_callback
    vs = ValidatorSort.deploy()
    for _ in range(100):
```

```
        data, active_pending = add_data(vs)
        validate(data, vs, active_pending)
        vs.clear()
```

ackee blockchain security

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain