Astrid

Finance

by Ackee Blockchain

23.10.2023

Contents

1. Document Revisions

0.1	Draft report	20.10.2023
1.0	Final report	23.10.2023
1.1	Fix review	23.10.2023

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
Lukáš Böhm	Lead Auditor
Andrey Babushkin	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.

3. Executive Summary

The Astrid Finance protocol works as a service between staked ETH holders and EigenLayer. Instead of directly choosing EigenLayer operators, where staked ETH is delegated for securing actively validated services (AVSs), Astrid Finance protocol provides pools for staked ETH. By staking into the main Astrid contract, users are delegating responsibility for choosing EigenLayer operators and their staked ETH tokens. In return, users receive Astrid's Restaked ETH token.

Revision 1.0

Astrid engaged Ackee Blockchain to perform a security review of the Astrid protocol with a total time donation of 9 engineering days in a period between October 9 and October 20, 2023 and the lead auditor was Lukáš Böhm. The audit has been performed on the commit 76d1f8d [1] and the scope was the following:

- AstridProtocol.sol
- Delegator.sol
- RestakedETH.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 a local deployment, testing, and fuzzing, we have involved Woke testing framework. We implemented a differential fuzzing test for RestakedETH contract to ensure the custom ERC20-like logic works appropriately (see Appendix B with the code snippet). During the review, we paid particular attention to:

- ensuring the arithmetic of the RestakedETH is correct,
- detecting possible price manipulation,

- ensuring the logic of the contracts cannot cause DoS,
- detecting possible reentrancies in the code,
- ensuring access controls are not too relaxed or too strict,
- testing whether no unstaking path reverts,
- looking for common issues such as data validation.

Our review resulted in 16 findings, ranging from Info to Medium severity. The code quality is average, and several places in the codebase do not follow the best practices. For instance, code duplication (I2), lack of proper input data validation across the codebase(L4), lack of in-code documents, admin's power to influence the protocol (see Trust Model) etc.

The protocol is simple, and existing documentation is sufficient for understanding the logic. However, we strongly advise adding proper in-code NatSpec documentation to the codebase. The client was responsive and provided us with all the necessary information we needed during the review.

Ackee Blockchain recommends Astrid:

- validate input parameters across the codebase,
- make the protocol's logic symmetrical - adding/removing,
- add proper in-code NatSpec documentation,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

Astrid provided an updated codebase with fixes on the commit: `1592f9b`. The fix review was done on October 23, 2023.

No new changes were introduced except the ones responding to our findings.

See the summary of the findings for the current status of issues.

[1] full commit hash: 76d1f8d4f28ad1d498fd285d5b65f1b1fb0bfaec

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
M1: The item from `delegators` array cannot be removed	Medium	1.0	Fixed
M2: The array `delegators` can contain duplicates	Medium	1.0	Fixed
M3: The initial `_gonsPerFragment` is too large	Medium	1.0	Fixed
M4: Unbounded iteration over withdrawals may cause DoS	Medium	1.0	Fixed
M5: User cannot withdraw staked token if the admin changes the `whitelist` flag	Medium	1.0	Fixed

	Severity	Reported	Status
L1: Division by zero	Low	1.0	Fixed
L2: Incorrect require condition	Low	1.0	Fixed
L3: Wrong index input cause panic error	Low	1.0	Fixed
L4: Lack of basic data validation across the codebase	Low	1.0	Fixed
L5: Lack of balance data validation	Low	1.0	Fixed
W1: `queueWithdrawal` does not revert if the provided strategy address is not in the strategy list	Warning	1.0	Fixed
W2: Fixed `_gonsPerFragment` may open the possibility of arbitrage and price manipulation attacks	Warning	1.0	Fixed
I1: `DOMAIN_SEPARATOR` can be cached	Info	1.0	Acknowledged
I2: Code duplication	Info	1.0	Acknowledged
I3: Data duplication	Info	1.0	Acknowledged
I4: Missing error message	Info	1.0	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.

The Astrid protocol is decentralized and serves as a layer between EigenLayer and end users. It implements second-level liquid staking, providing alternative tokens for each staked token. Users can deposit their staked tokens (stETH, rETH, cbETH, etc.) to the protocol and receive the restaked counterparts (rstETH, rrETH, rcbETH, etc.). Staked tokens are aggregated by the protocol and staked through multiple delegators to EigenLayer's operators. Also, Astrid provides several liquidity pools for restaked tokens.

The Astrid protocol consists of three main contracts and multiple interfaces: `AstridProtocol.sol`, `Delegator.sol` and `RestakedETH.sol`.

Contracts

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

AstridProtocol

The `AstridProtocol.sol` contract is the main contract of the protocol. It manages deposits and withdrawals of staked tokens, minting and burning restaked tokens, and delegating staked tokens to delegators. Also, it holds information about the total supply of staked and restaked tokens and keeps their ratio constant through the so-called rebasing. The decision of where to allocate staked tokens, as well as the rebasing, is performed manually by the governance party. The contract is upgradeable, and the upgrade is performed by the governance party.

Delegator

Delegators are contracts responsible for communicating with EigenLayer's operators, and each Delegator is responsible for a single operator. Delegators acquire staked ETH and pass them to an EigenLayer strategy. They also have convenient functions to get the state of the stake, such as the current stake balance, the availability of withdrawals, and the queue of pending withdrawals. The contract is upgradeable, and the upgrade is performed by the governance party.

RestakedETH

The `RestakedETH.sol` contract is a base contract for all restaked tokens. It implements the ERC20 interface and provides functions for minting and burning restaked tokens. Internally, balances are represented by so-called gons, factors of weis. This representation allows scaling users' balances to change the total supply of staked tokens without needing expensive transfer operations. The total supply of staked tokens changes when the protocol receives rewards from EigenLayer or when the slashing occurs. The rebaser role performs the rebasing. The contract is upgradeable, and the upgrade is performed by the governance party.

Actors

Governance

The governance role is responsible for all system-level decisions. It can pause the protocol, upgrade contracts, and set roles for all contracts. As for the time of the audit, it is implemented as a multisig Gnosis Safe wallet. In `AstridProtocol.sol`, it is also responsible for whitelisting staked tokens, rebasing, and allocating assets to specific delegators.

EigenLayer Operator

EigenLayer operators are responsible for staking tokens, distributing rewards, and slashing. They are not part of the protocol, but they are essential for understanding the protocol's functionality. The protocol is designed to work with multiple operators, each with its own Delegator.

Rebaser

The rebaser role is responsible for rebasing restaked tokens in the `RestakedETH.sol` contract. This role is not explicitly set in initializers; however, the role must be set to the AstridProtocol deployment for correct functioning.

Minter

The minter role is responsible for minting and burning restaked tokens in the `RestakedETH.sol` contract. This role is not explicitly set in initializers; however, the role must be set to the AstridProtocol deployment for correct functioning. The role is critical since it may change the total supply of restaked tokens and influence the number of staked tokens users can withdraw.

User

The user is an end-user of the protocol. The user can deposit staked tokens, withdraw staked tokens, and swap restaked tokens for staked tokens and vice versa in Uniswap's liquidity pools provided by Astrid.

5.2. Trust Model

On the high level, Astrid works as an actively managed fund. Users deposit their staked tokens to the contract, and Astrid governance later decides where to allocate them. Users implicitly trust the Governance party to manage the system in a manner that benefits the protocol and its stakeholders. If the governance stakes tokens to a malicious or malfunctioning operator and slashes occur, the loss is handled by the staker. Also, given that the Governance can pause the protocol and upgrade contracts, there is significant trust in their decisions not to act maliciously or carelessly.

As mentioned above, the roles rebaser and Minter should be explicitly set by the Governance. These roles are essential for the proper functioning of the Astrid protocol. However, if these roles are set to a party different from the Astrid Protocol and get compromised or act maliciously, they could distort the ratio of staked to restaked tokens, thereby affecting users' potential withdrawals. The assignment and management of these roles should be handled with extreme care.

M1: The item from `delegators` array cannot be removed

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AstridProtocol	Type:	Contract logic

Description

The contract AstridProtocol contains an array `IDelegator[] public delegators` for storing Delegator contracts. The function `addDelegator` allows to add a new Delegator address to the array. However, there is no possibility to remove an item from the array. It will be a problem if one of the Delegators turns out to be malicious or broken after an upgrade, and the issue will arise when the function `rebase()` is called.

Vulnerable scenario

During the process of rebasing, all Delegators are called and their balances are summarized:

Listing 1. rebaseInfo()

```
for (uint256 i; i < delegators.length; i++) {
    _stakedTokenBackedSupply +=
delegators[i].getAssetBalance(_stakedTokenAddress,
stakedTokenMapping.eigenLayerStrategyAddress);
}
```

If a malicious/broken Delegator changes the logic of the function `.getAssetBalance()` (Delegators are upgradable), it can return any value and affect the final sum `_stakedTokenBackedSupply == info.stakedTokenBackedSupply`, which is crucial for rebasing.

Listing 2. rebase()

```
if (info.restakedTokenTotalSupply < info.stakedTokenBackedSupply) {
    _supplyDelta = info.stakedTokenBackedSupply -
info.restakedTokenTotalSupply;
    _isRebasePositive = true;
} else {
    _supplyDelta = info.restakedTokenTotalSupply -
info.stakedTokenBackedSupply;
    _isRebasePositive = false;
}
```

The final result of the rebasing process may be affected as it can end up executing the wrong code branch.

Recommendation

Add a function for removing Delegators from the array.

```
require(_delegatorIndex < delegators.length)
delegators[_delegatorIndex] = delegators[delegators.length - 1];
delegators.pop();
```

It is a good practice for smart contract development, to achieve function ”symmetry”. Add/remove, send/receive, deposit/withdraw, etc.

Fix 1.1

The following code was added to the contract AstridProtocol that fixes the issue:

```
function removeDelegator(
    uint16 _delegatorIndex
) public whenNotPaused onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_delegatorIndex < delegators.length, "AstridProtocol:
_delegatorIndex out of bounds");
```

```
    address delegatorAddress = address(delegators[_delegatorIndex]);
    delegators[_delegatorIndex] = delegators[delegators.length - 1];
    delegators.pop();

    emit DelegatorRemoved(delegatorAddress);
}
```

Go back to Findings Summary

M2: The array `delegators` can contain duplicates

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AstridProtocol	Type:	Data validation

Description

The contract AstridProtocol contains an array `IDelegator[] public delegators` for storing Delegator contracts. The function `addDelegator()` allows to add a new Delegator address to the array. However, the contract does not implement the logic to ensure there is no duplicate in the `delegators` array. If two or more duplicates are added to the array, it will be problematic when the function `rebase()` is called.

Together with the issue M1, mentioning that delegators cannot be removed from the array, the severity is even higher.

Vulnerable scenario

Similar scenario as M1. During the process of rebasing, all Delegators are called and their balances are summarized:

Listing 3. rebaseInfo()

```
for (uint256 i; i < delegators.length; i++) {
    _stakedTokenBackedSupply +=
delegators[i].getAssetBalance(_stakedTokenAddress,
stakedTokenMapping.eigenLayerStrategyAddress);
}
```

The amount of the balance in the duplicated Delegator contract will be accounted for more than once. It will affect the final sum `_stakedTokenBackedSupply == info.stakedTokenBackedSupply`, which is crucial for rebasing.

Listing 4. rebase()

```
if (info.restakedTokenTotalSupply < info.stakedTokenBackedSupply) {
    _supplyDelta = info.stakedTokenBackedSupply -
info.restakedTokenTotalSupply;
    _isRebasePositive = true;
} else {
    _supplyDelta = info.restakedTokenTotalSupply -
info.stakedTokenBackedSupply;
    _isRebasePositive = false;
}
```

The final result of the rebasing process may be affected as it can end up executing the wrong code branch.

Recommendation

Implement the logic to ensure there is no duplicate in the `delegators` array. Iterate over the array and check if the address is already present in the array. If so, revert the transaction.

Fix 1.1

The following function was added to the contract AstridProtocol:

```
function _delegatorExists(
    address _delegatorAddress
) internal view returns (bool) {
    for (uint256 i; i < delegators.length; i++) {
        if (_delegatorAddress == address(delegators[i])) {
            return true;
        }
    }
    return false;
}
```

This function is used in the function `addDelegators()` to check if the address is already present in the array. If so, the transaction is reverted.

Go back to Findings Summary

M3: The initial `_gonsPerFragment` is too large

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	RestakedETH	Type:	Arithmetics

Description

The RestakedETH contract represents users' balances in fractions of Wei called ”gons”. The ratio between gons and Weis is stored in the `_gonsPerFragment` variable. The initial setup of `_gonsPerFragment` happens during the first successful mint in the `mint()` function, and is updated after every rebase. The initial value of `_gonsPerFragment` is set to be `10^50`, and this number is too large to be used with the given maximum supply.

The maximum supply of Wei is set on line 31 and equals `2^128 - 1`. The decimals of the restaked token are equal to `18`, which means that one restaked ETH will have `10^18` restaked Wei. Balances are represented in gons and are stored as `uint256`, that is the maximum value of decimal places of total gons is given by `log10(2^256) ~= 77`.

Given that the number of gons per fragment is set to be `10^50`, the maximum supply of restaked Wei is approximately equal to `10^(77-50) = 10^27`, or `10^9` ETH. If the number of minted ETH is higher by at least one order of magnitude, the contract will not be able to represent the total supply of Wei, and the `mint()` function will fail with the overflow error. Because the contract is assumed to have the maximum supply of Wei equal to `2^128 - 1`, or approximately `10^38`, the initial number of gons per fragment `x` should fit the following inequality:

$$
\begin{aligned}
(2^{128} - 1) \cdot 10^{x} \leq 2^{256} - 1,\\
x \leq \left\lfloor \log_{10} \frac{2^{256} - 1}{2^{128} - 1} \right\rfloor,\\
x < \lfloor 38.6 \rfloor,\\
x \leq 38.
\end{aligned}
$$

To summarize, the initial number of gons per wei (`_gonsPerFragment`) should be set to a maximum of `10^38`.

Recommendation

- Modify the initial value of `_gonsPerFragment` to be equal to `10^38` or less.
- Pay attention to this value when updating the maximum total supply.

Fix 1.1

The client claims that due to the already deployed instance of the contract the `_gonsPerFragment` value cannot be changed. The change occurred in maximum total supply that was restricted to `10^27` Weis. The previous variable was marked as deprecated due to the upgradeability and a new variable, `MAXIMUM_SUPPLY = 10**27`, was introduced. The issue is fixed.

Go back to Findings Summary

M4: Unbounded iteration over withdrawals may cause DoS

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	Delegator	Type:	Denial of Service

Description

The main Astrid protocol contract has the `rebase()` function that calculates a new ratio between gons and fragments (Weis) for restaked tokens based on a change between the supply of staked tokens and restaked ones. The staked tokens are usually scattered in different contracts, including delegators, EigenLayer strategies and the main Astrid contract. Thus, to compute the total backed supply of the staked token, the main Astrid contract computes the `ReBaseInfo` structure by calling the `rebaseInfo()` function. `rebaseInfo()`, in turn, iteratively calls the `getAssetBalance()` function on all added delegators besides other operations.

The Delegator contract's `getAssetBalance()`, in simplistic terms, iterates over all pending withdrawals and calls the corresponding EigenLayer strategy to calculate the value of staked assets. The issue lies in the fact that the `getAssetBalance()` function does not have a limit on the number of withdrawals it can iterate over, and the iteration always starts from the first withdrawal. For every pending withdrawal, an external call to the EigenLayer strategy is made, which may be expensive in terms of gas. If the number of withdrawals is large, the `getAssetBalance()` function may run out of gas and fail to execute. This will cause the `rebase()` function to fail as well, effectively causing a DoS.

Exploit Scenario

If any of the delegators has a large number of pending withdrawals, or if one of the delegators is malfunctioning or malicious, the `rebaseInfo()` function will call the `getAssetBalance()` function on all delegators, and on every call, there will be an iteration over all pending withdrawals and an external call to the EigenLayer strategy. These together may cause the `rebase()` function to run out of gas and fail to execute with no way to recover. The main Astrid protocol will still be functional, but the `rebase()` function will not be able to execute, which may cause the protocol to be stuck in a state where the ratio between gons and fragments is not updated and the value of restaked tokens is not calculated correctly, which opens up the possibility of other attacks.

Listing 5. rebaseInfo()

```
for (uint256 i; i < delegators.length; i++) {
    _stakedTokenBackedSupply +=
delegators[i].getAssetBalance(_stakedTokenAddress,
stakedTokenMapping.eigenLayerStrategyAddress);
}
```

Listing 6. _getAssetBalance()

```
for (uint256 i = 0; i < withdrawals.length; ) {
    WithdrawalInfo memory withdrawalInfo = withdrawals[i];
    if (withdrawalInfo.stakedTokenAddress == _token &&
withdrawalInfo.pending) {
        balance +=
IStrategy(_eigenLayerStrategyAddress).sharesToUnderlyingView(withdrawalInfo
.shares);
    }
    unchecked {
        ++i;
    }
}
```

Recommendation

1. Implement the way to remove delegators from the delegator list, as stated in M1: The item from `delegators` array cannot be removed.
2. Consider caching values from `IStrategy.sharesToUnderlying()` and using them in `getAssetBalance()` instead of calling the strategy every time.
3. Limit the total number of delegators.

Fix 1.1

The issue is fixed by adding a limit on the number of delegators that can be added to the delegator list. A new storage variable `maxDelegators` was introduced, and the `addDelegator()` function was updated to check if the number of delegators is less than the `maxDelegators` value. If the number of delegators is greater than `maxDelegators`, the function will revert. The `maxDelegators` value is set in the initializer. Also, the `removeDelegator()` function was added to remove delegators from the delegator list.

Go back to Findings Summary

M5: User cannot withdraw staked token if the admin changes the `whitelist` flag

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	AstridProtocol	Type:	Contract logic

Description

For setting and allowing new staked tokens to be used inside the protocol, the admin has to call a function `setStakedTokenMapping`.

Listing 7. setStakedTokenMapping()

```
function setStakedTokenMapping(
    address _stakedTokenAddr,
    bool _whitelisted,
    address _restakedTokenAddr,
    address _eigenLayerStrategyAddr
) public whenNotPaused onlyRole(DEFAULT_ADMIN_ROLE) {
    stakedTokens[_stakedTokenAddr] = StakedTokenMapping({
        whitelisted: _whitelisted,
        restakedTokenAddress: _restakedTokenAddr,
        eigenLayerStrategyAddress: _eigenLayerStrategyAddr
    });

    emit StakedTokenMappingSet(_stakedTokenAddr, _whitelisted,
_restakedTokenAddr, _eigenLayerStrategyAddr);
}
```

The flag `whitelisted` is present in functions `deposit()` and `withdraw()` to check if the token can be used inside the protocol. If the admin decides to change one of the parameters from the structure `{whitelisted, restakedTokenAddress, eigenLayerStrategyAddress}`, the admin can overwrite the structure by calling the function again. If he changes the `whitelisted` flag to false, users cannot withdraw the staked token if they have already staked it.

Vulnerable scenario

The following flow results in a user not being able to withdraw the staked token.

1. Admin sets the stake token xETH, where the address of the token xETH is used as a key for mapping. Structure parameters are set `{whitelisted = True, restakedTokenAddress = 0x… , eigenLayerStrategyAddress = 0x…}` as a value for the key xETH.
2. Bob stakes xETH.
3. Admin changes the structure parameters to `{whitelisted = False, restakedTokenAddress = 0x… , eigenLayerStrategyAddress = 0x…}`.
4. Bob tries to withdraw xETH, but the function `withdraw()` will fail because the flag `whitelisted` is false.

Recommendation

Change the logic to allow users to withdraw the staked token even if the admin changes the `whitelist` flag. This behavior gives the contract admin extra power but decreases trust in the protocol.

Fix 1.1

The issue was fixed by removing the `require` statement with the check for the `whitelisted` flag from the `withdraw()` function.

Go back to Findings Summary

L1: Division by zero

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	RestakedETH	Type:	Arithmetics

Description

The RestakedETH contract has three locations in the code that may cause the division-by-zero panic.

`transferAll()` and `transferAllFrom()`

In the `transferAll()` and `transferAllFrom()` functions in the RestakedETH contract, the transferred value is computed by dividing the user's balance in gons by `_gonsPerFragment`:

Listing 8. transferAll()

```
uint256 value = gonValue.div(_gonsPerFragment);
```

Listing 9. transferAllFrom()

```
uint256 value = gonValue.div(_gonsPerFragment);
```

This variable is set to a non-zero value only after the first mint. If a user calls one of these functions before the first mint, these functions will revert with the division-by-zero panic.

`rebase()`

In the `rebase()` function in the RestakedETH contract, the new `_gonsPerFragment` value is computed by dividing the total supply in gons by the total supply in fragments, and the total supply is updated by either adding or subtracting the `supplyDelta` value:

Listing 10. rebase()

```
if (isRebasePositive) {
    _totalSupply = _totalSupply.add(supplyDelta);
} else {
    _totalSupply = _totalSupply.sub(supplyDelta);
}

if (_totalSupply > MAX_SUPPLY) {
    _totalSupply = MAX_SUPPLY;
}

_gonsPerFragment = _totalGons.div(_totalSupply);
```

If the `supplyDelta` value equals the current total supply, the division-by-zero panic will occur.

Recommendation

Transfers

One way is to add zero-value checks to the `transferAll()` and `transferAllFrom()` and `rebase()` functions to prevent the division-by-zero panic:

Listing 11. transferAll()

```
value = (_gonsPerFragment == 0) ? 0 : gonValue.div(_gonsPerFragment);
```

Listing 12. transferAllFrom()

```
value = (_gonsPerFragment == 0) ? 0 : gonValue.div(_gonsPerFragment);
```

Another option is to move the initialization of the `_gonsPerFragment` variable from `mint()` to the initializer function. This will ensure that the `_gonsPerFragment` variable is always set to a non-zero value, and the code will be cleaner.

`rebase()`

If the `supplyDelta` value equals the current total supply, this means that the total supply will be zero after the rebase. This can be the case if users have withdrawn all their staked tokens from the protocol. The behavior of the `rebase()` function in this case should be decided by the project team.

Fix 1.1

Transfers

The issue was fixed by moving the initial setup of the `_gonsPerFragment` variable in the initializer function. In this case, the `_gonsPerFragment` variable is always set to a non-zero value, and the division-by-zero panic will not occur.

`rebase()`

The issue was fixed by adding a zero-value check to the `rebase()` function for the `_totalGons` variable. The client also claims the following:

> As for when _totalGons != 0 && _totalSupply == 0, this only happens in the edge case when there are totally no more supply backing the gons at all, in the very worst case that if this happens, we're able to send LST manually to the Astrid main protocol contract and then rebase in a single call, since `rebaseInfo` will pick up this new LST amount and pass it as supplyDelta into the rebase function.

We consider the issue fixed.

Go back to Findings Summary

L2: Incorrect require condition

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	RestakedETH	Type:	Contract logic

Description

The RestakedETH contract represents users' balances in fractions of Wei called "gons". In the `burn()` function on line 113, there is a require statement that checks if the user has enough balance:

Listing 13. burn()

```
require(_gonBalances[from] >= amount, "Amount must be owned by from
address");
```

The issue with this code is that the `amount` variable is denominated in Wei, while the `_gonBalances` mapping holds values in gons. Since the number of gons is much larger than the number of Wei, the condition will always pass. Although this issue does not pose a security problem and cannot be exploited, if the balance is insufficient, the transaction will revert with an underflow error.

Recommendation

To address this issue, you can replace lines 113-115 with the following code:

```
uint256 gonValue = amount.mul(_gonsPerFragment);
require(_gonBalances[from] >= gonValue, "Amount must be owned by from
address");
```

Fix 1.1

The issue was fixed by changing the require statement on line 113 to use the `gonValue` variable instead of the `amount` variable.

Go back to Findings Summary

L3: Wrong index input cause panic error

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	AstridProtocol	Type:	Data validation

Description

The contract AstridProtocol contains the array `IDelegator[] public delegators` for storing Delegator contracts.

While calling several functions in the contract, the admin has to provide an index of the Delegator from the array he wants to operate with. If a provided index is incorrect, a transaction will end with `Panic error - index out of bounds`. Every transaction should either revert or execute successfully.

Affected functions:

- `pullDelegator`
- `restakeDelegator`
- `queueWithdrawalDelegator`
- `completeQueuedWithdrawalDelegator`

The same problem is in the function `claim()` with an array `withdrawalRequestsByUser[msg.sender]`, and the function `completeQueuedWithdrawal()` with an array `withdrawals[msg.sender]`.

Vulnerable scenario

Suppose admin calls one of the mentioned functions with `index >= delegators.length`. The index causes panic error and revert.

The same happens in the function `claim` if a user calls the function with `index >= withdrawalRequestsByUser[msg.sender].length`, or the function `completeQueuedWithdrawal()` with `index >= withdrawals[msg.sender].length`.

Recommendation

It is not an exploitable vulnerability, but contracts should **never** return such errors, and all the input parameters should be validated appropriately and contain an explanatory revert string. Add requirement `index < array.length`.

As indexes are hard to read (read the storage delegators[]), the system admin must be sure where a specific Delegator is in the array. This approach is error-prone. The function to get a specific address for an index or vice versa may be helpful.

Fix 1.1

In all affected functions, a `require`-statement with `_delegatorIndex < delegators.length` was added. The issue is fixed.

Go back to Findings Summary

L4: Lack of basic data validation across the codebase

Low severity issue

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

Description

This issue does not contain more severe data validations mentioned in separate findings such as M2, L3, L4.

The contract AstridProtocol does not implement sufficient data validation checks in the following function:

- `initialize()`
  - _governanceAddr - zero address check
  - _eigenLayerStrategyManagerAddr - contract existence check
- `setEigenLayerStrategyManagerAddress()`
  - _eigenLayerStrategyManagerAddr - contract existence check
- `setStakedTokenMapping()`
  - _stakedTokenAddr - contract existence check
  - _restakedTokenAddr - contract existence check
  - _eigenLayerStrategyAddr - contract existence check
- `addDelegators()`
  - _delegatorContracts - contract existence check
- `pullDelegator()`

    - _token - contract existence check

- `queueWithdrawalDelegator()`
    - _stakedTokenAddress - contract existence check
- `rebaseInfo()`
    - _stakedTokenAddress - contract existence check
- `rebase()`
    - _stakedTokenAddress - contract existence check

The contract Delegator does not implement sufficient data validation checks in the following function:

- `initialize()`
    - _governanceAddr - zero address check
    - _astridProtocolAddr - contract existence check
- `setAstridProtocolAddress()`
    - _astridProtocolAddr - contract existence check Other functions in the contract also do not implement sufficient data validation checks, but they should be called only by the AstridProtocol contract, where the missing checks are already mentioned.

The contract RestakedETH does not implement sufficient data validation checks in the following function:

- `initialize()`
    - _governanceAddr - zero address check
    - _stakedTokenAddress - contract existence check

In most cases, the transaction will revert during the execution, if wrong input data are provided. However, the contract will not provide sufficient information about the reason for the revert as the transaction will call unexisting contracts or access unexisting mapping indexes. Additionally, in some scenarios, it will not revert at all, and incorrect data (addresses) will be stored in the contract. Even though the contract may be redeployed (wrong data in `initialize`) or parameters may be reset (setStakedTokenMapping), it is a good practice to avoid such mistakes. These scenarios are not as severe but they may lead to unexpected behavior in the system and make debugging more difficult.

Recommendation

Implement sufficient data validation checks in the mentioned functions.

Fix 1.1

In the contract AstridProtocol, the following fixes were implemented:

- `initialize()`
  - _governanceAddr: added zero address check
  - _eigenLayerStrategyManagerAddr: implemented contract existence check
- `setEigenLayerStrategyManagerAddress()`
  - _eigenLayerStrategyManagerAddr: implemented contract existence check
- `setStakedTokenMapping()`
  - _stakedTokenAddr: implemented contract existence check
  - _restakedTokenAddr: implemented contract existence check
  - _eigenLayerStrategyAddr: implemented contract existence check
- `addDelegators()`

- _delegatorContracts: implemented contract existence check

- `pullDelegator()`
  - _token: implemented contract existence check
- `queueWithdrawalDelegator()`
  - _stakedTokenAddress: implemented contract existence check
- `rebaseInfo()`
  - _stakedTokenAddress: implemented contract existence check
- `rebase()`
  - _stakedTokenAddress: implemented contract existence check

In the contract Delegator, the following fixes were implemented:

- `initialize()`
  - _governanceAddr: added zero address check
  - _astridProtocolAddr: implemented contract existence check
- `setAstridProtocolAddress()`
  - _astridProtocolAddr: implemented contract existence check
- `restake()`
  - _stakedTokenAddress: implemented contract existence check
  - _eigenLayerStrategyManagerAddress: implemented contract existence check
  - _eigenLayerStrategyAddress: implemented contract existence check
- `queueWithdrawal()`
  - _stakedTokenAddress: implemented contract existence check
  - _eigenLayerStrategyManagerAddress: implemented contract existence check

    - _eigenLayerStrategyAddress: implemented contract existence check
- `canWithdraw()`
    - _eigenLayerStrategyManagerAddress: implemented contract existence check
- `completeQueuedWithdrawal()`
    - _eigenLayerStrategyManagerAddress: implemented contract existence check
    - _eigenLayerStrategyAddress: implemented contract existence check
- `pull()`
    - token: implemented contract existence check
- `getAssetBalances()`
    - _eigenLayerStrategyManagerAddress: implemented contract existence check
- `getAssetBalance()`
    - _token: implemented contract existence check
    - _eigenLayerStrategyAddress: implemented contract existence check

In the contract RestakedETH, the following fixes were implemented:

- `initialize()`
    - _governanceAddr: added zero address check
    - _stakedTokenAddress: implemented contract existence check

Go back to Findings Summary

L5: Lack of balance data validation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	RestakedETH	Type:	Data validation

Description

In the contract RestakedETH several functions do not perform any data validation of user-provided input parameters.

Affected functions:

- `transfer()`
- `transferFrom()`

In the mentioned functions, the balance of `msg.sender` is not validated before the subtraction. If a user tries to send more tokens than they have, the transaction will revert to the Underflow error. Even though the issue is not exploitable, error messages should not appear in the well-designed protocol all the edge case scenarios should be validated and proper error messages returned.

Vulnerable scenario

Bob has a balance of 10 RestakedETH and he calls the function `transfer(to = address of Alic, amount = 15)`. Because his balance is less than 15, the transaction will revert to the Underflow error.

Recommendation

Implement proper checks if the balance is sufficient to perform a transfer

Fix 1.1

In both affected functions, an additional check was added to verify if the balance of the sender is sufficient to perform a transfer. The issue is fixed.

Go back to Findings Summary

W1: `queueWithdrawal` does not revert if the provided strategy address is not in the strategy list

Impact:	Warning	Likelihood:	N/A
Target:	Delegator	Type:	Contract logic

Description

In `queueWithdrawal()` of the Delegator contract, the strategy index is set by iterating over the strategy list in EigenLayer's strategy manager: if the address of the strategy equals the provided in the input arguments, the `strategyIndex` variable is set. However, there is no verification if any from all strategy addresses were equal to the provided one. In this case, the if-statement inside the look will never fire, the `strategyIndex` will default to `0`, and the execution will continue as if the requested strategy was in the first position in the array:

Listing 14. queueWithdrawal()

```
uint256 strategyIndex;
uint256 strategyListLength =
IStrategyManager(_eigenLayerStrategyManagerAddress).stakerStrategyListLengt
h(address(this));
for (uint256 i; i < strategyListLength; i++) {
    if
(IStrategyManager(_eigenLayerStrategyManagerAddress).stakerStrategyList(add
ress(this), i) == _eigenLayerStrategyAddress) {
        strategyIndex = i;
        break;
    }
}
```

The behavior largely depends on the implementation in EigenLayer. It may revert if the strategy index and the strategy address do not match, or it may continue as if the strategy was in the first position in the array. Both cases are not desirable and are considered undefined behavior.

Recommendation

Introduce an additional check to verify if the provided strategy address is in the strategy list. If it is not, revert the transaction.

Fix 1.1

The initial value of `strategyIndex` was set to `type(uint256).max`. After the cycle, the `require`-statement was added to verify if the strategy index was not equal to the initial value. The issue is fixed.

Go back to Findings Summary

W2: Fixed `_gonsPerFragment` may open the possibility of arbitrage and price manipulation attacks

Impact:	Warning	Likelihood:	N/A
Target:	AstridProtocol, RestakedETH	Type:	Arithmetics

Description

In RestakedETH, the `_gonsPerFragment` variable is responsible for holding the correct ratio between the internal representation of restaked tokens in so-called gons and the tokens themselves. Users' balances are internally denominated in gons, and the value of gons is subject to change depending on the supply of staked tokens. The change may occur due to either staking rewards or slashing happening on EigenLayer. After each rebase, the supply of restaked tokens becomes similar to the circulating staked tokens supply. To maintain the real value of restaked tokens after each rebase, the `_gonsPerFragment` variable is adjusted accordingly, ensuring a consistent ratio between staked and restaked tokens. In short, rebasing allows keeping the ratio between staked and restaked tokens constant while keeping the value of users' holdings in restaked tokens the same. However, according to the Astrid development team, rebase is called manually once per day.

Also, Astrid Finance operates a liquidity pool on Uniswap V2, enabling users to trade between staked and restaked tokens. The price of restaked tokens, relative to staked tokens, is dynamically determined using the Uniswap V2 formula. This means that the ratio between staked and restaked tokens can vary and may differ from the ratio derived from the RestakedETH contract. This difference in ratios could potentially be exploited by arbitrageurs, who seek to profit from price discrepancies. However, it would be helpful to provide a concrete example of how such exploitation could occur.

Another thing that should be mentioned is that the `processWithdrawals()` function of the AstridProtocol contract may be called by anyone. This relates to this issue because this function is responsible for storing the amount of gons the user will receive during the token claim. Again, the exact scenario of how it may be exploited is not clear, however, because the contract has the additional storage boolean variable `processWithdrawalsOnWithdraw` that is responsible for automatic withdrawals, we consider the lack of access control to the `processWithdrawals()` function as an issue.

Recommendation

Consider automating the rebase process to happen more frequently than once per day. Also, consider adding access control to the `processWithdrawals()` function.

Fixed 1.1

The issue was fixed by adding an access control modifier to the `processWithdrawals()` function so that only `DEFAULT_ADMIN_ROLE` can call it.

Go back to Findings Summary

I1: `DOMAIN_SEPARATOR` can be cached

Impact:	Info	Likelihood:	N/A
Target:	RestakedETH	Type:	Gas optimization

Description

In the RestakedETH contract, the `DOMAIN_SEPARATOR()` function implements the EIP-712 domain separator that prevents signature in different domains. This function is essentially a `keccak256` hash of multiple distinctive to the current domain properties, like the chain ID, the address of the current contract, etc. The implementation directly follows the one provided in the EIP. However, since the domain separator is constant for a given domain, calculating it on every call is ineffective, and the value may be cached to save on gas. The savings may become significant if the `permit` function is used often.

Recommendation

To fix the issue, the following may be done according to the OpenZeppelin implementation:

1. In the `initialize()` function, store current `address(this)` and `block.chainid`. These variables together determine the current domain, and if they change, the domain separator must be recomputed.
2. Extract the `DOMAIN_SEPARATOR()` computation to a new `internal` function. This function will only be called on the initialization and in the case when the domain changes.
3. In the `initialize()` function, compute the separator and cache it in a separate storage value, e.g., `_cachedDomainSeparator`.
4. Change the `DOMAIN_SEPARATOR()` function's body to the following one (taken from the OZ implementation linked above):

```
function DOMAIN_SEPARATOR() public view returns (bytes32) {
    if (address(this) == _cachedThis && block.chainid == _cachedChainId) {
        return _cachedDomainSeparator;
    } else {
        return _buildDomainSeparator();
    }
}
```

Fix 1.1

The client was notified of the issue and provided with the recommendation. The client acknowledged the issue and the recommendation with the comment:

> Won't fix as contracts are deployed and will affect existing storages that are being used.

Go back to Findings Summary

I2: Code duplication

Impact:	Info	Likelihood:	N/A
Target:	RestakedETH	Type:	Best practices

Description

The code of the RestakedETH contract is often repeated in different functions. Four transfer functions perform the same operation with different values, but they are all implemented directly in the body of the corresponding external function. Such code style is error-prone.

Recommendation

Create private functions for `_transfer` and `_approve`, and call these functions with correct parameters in all external counterparts, such as transfer functions and functions that modify the allowances. For reference, one can be inspired by the design of the ERC20 tokens by OpenZeppelin. Follow the best practices to make the code more readable and maintainable. Follow the best practices to make the code more readable and maintainable.

Fix 1.1

The client was notified of the issue and provided with the recommendation. The client acknowledged the issue and the recommendation with the comment:

> Won’t fix as contracts are deployed and will affect existing storages that are being used.

Go back to Findings Summary

I3: Data duplication

Impact:	Info	Likelihood:	N/A
Target:	AstridProtocol	Type:	Best practices

Description

The structure `WithdrawalRequest` is stored in two mappings: `withdrawalRequests` and `withdrawalRequestsByUser`. In the function `_processWithdrawals`, the information there is duplicated. This design is error-prone and gas-inefficient.

Recommendation

Consider storing the structure in `withdrawalRequests` and pushing only indices of corresponding items to `withdrawalRequestsByUser[user]`. Follow the best practices to make the code more readable and maintainable.

Fix 1.1

The client was notified of the issue and provided with the recommendation. The client acknowledged the issue and the recommendation with the comment:

> Won't fix as contracts are deployed and will affect existing storages that are being used.

Go back to Findings Summary

I4: Missing error message

Impact:	Info	Likelihood:	N/A
Target:	RestakedETH	Type:	Best practices

Description

The modifier `validRecipient()` and the function `permit()` miss any error messages inside the `require` statements. Well-explanatory errors make the code more readable and help with user experience and debugging.

Recommendation

Add error strings and follow the best practices to make the code more readable and maintainable.

Fix 1.1

For all `require`-statements with missing error messages, explanatory strings were added. The issue is now fixed.

Go back to Findings Summary

6. Report revision 1.1

No significant changes were performed in the contracts, and no new vulnerabilities were found. All the changes are responding to reported issues.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Astrid: Finance, 23.10.2023.

Appendix B: Woke code snippet

```
class DifferentialRestakedETH():
    # Implementation of Python version of RestakedETH contract is skipped
in the code snippet
    def mint(self, to: Account, amount: int) -> None:
    def burn(self, from_: Account, amount: int) -> None:
    def get_rebase_positive(self) -> bool:
    def get_supply_delta(self) -> int:
    def rebase(self, epoch: int, is_rebase_positive: bool, supply_delta:
int) -> int:

class RestakedETHFuzzTest(FuzzTest):
    governance: Account
    contract: RestakedETH
    diff_contract: DifferentialRestakedETH
    _epoch: int

    def random_wei(self) -> int:
        return random_int(0, 10 * 10 ** 18)

    def pre_sequence(self) -> None:
        self.governance = default_chain.accounts[0]
        # Deploy simple ERC20 token
        staked_token = ERC20Mock.deploy("Staked Token", "STK",from_=
self.governance)

        # Deploy upgradable restake token
        restaked_token_impl     = RestakedETH.deploy()
        proxy_restaked_token    = ERC1967Proxy.deploy(restaked_token_impl,
b'')
        restaked_token          = RestakedETH(proxy_restaked_token)
        restaked_token.initialize(self.governance, staked_token, "STK")

        restaked_token.grantRole(restaked_token.MINTER_ROLE(),
self.governance, from_=self.governance)
        restaked_token.grantRole(restaked_token.REBASER_ROLE(),
self.governance, from_=self.governance)

        self.contract = restaked_token
        self.diff_contract = DifferentialRestakedETH()
        self._epoch = 1692403201

    @property
    def epoch(self) -> int:
        epoch = self._epoch
        self._epoch += 1
        return epoch

    @flow()
    def flow_mint(self) -> None:
        # call mint on contract and python model

    @flow()
    def flow_burn(self) -> None:
        # call burn on contract and python model

    @flow(weight=1)
    def flow_rebase(self) -> None:
        # call rebase on contract and python model

    @invariant(period=50)
    def check_balances(self) -> None:
        for account, expected_gons in
self.diff_contract.gon_balances.items():
            real_gons = self.contract.scaledBalanceOf(account)
            assert expected_gons == real_gons, f"[Balance] expected:
{expected_gons}, real: {real_gons}"
        assert self.diff_contract.total_supply ==
self.contract.totalSupply(), f"[Total Supply] expected: {
self.diff_contract.total_supply}, real: {self.contract.totalSupply()}"
        assert self.diff_contract.total_gons ==
self.contract.scaledTotalSupply(), f"[Total Gons] expected: {
self.diff_contract.total_gons}, real: {self.contract.totalGons()}"

    def post_sequence(self) -> None:
        for account, gons in self.diff_contract.gon_balances.items():
            amount = gons // self.diff_contract.gons_per_fragment
            self.diff_contract.burn(account, amount)
            self.contract.burn(account, amount, from_=self.governance)
            assert self.contract.balanceOf(account) == 0, f"[Balance]
expected: 0, real: {self.contract.balanceOf(account)}"
            # assert self.contract.scaledBalanceOf(account) == 0, f"[Gons]
```

```
expected: 0, real: {self.contract.scaledBalanceOf(account)}"
        assert self.diff_contract.total_supply ==
self.contract.totalSupply(), f"[Total Supply] expected: {
self.diff_contract.total_supply}, real: {self.contract.totalSupply()}"
        # assert self.contract.totalSupply() == 0, f"[Total Supply]
expected: 0, real: {self.contract.totalSupply()}"
        assert self.diff_contract.total_gons ==
self.contract.scaledTotalSupply(), f"[Total Gons] expected: {
self.diff_contract.total_gons}, real: {self.contract.totalGons()}"
        # assert self.contract.scaledTotalSupply() == 0, f"[Total Gons]
expected: 0, real: {self.contract.totalGons()}"

@default_chain.connect()
def test_restaked_eth() -> None:
    default_chain.set_default_accounts(default_chain.accounts[0])
    RestakedETHFuzzTest().run(sequences_count=10, flows_count=100000)
```

ackee | blockchain security

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain