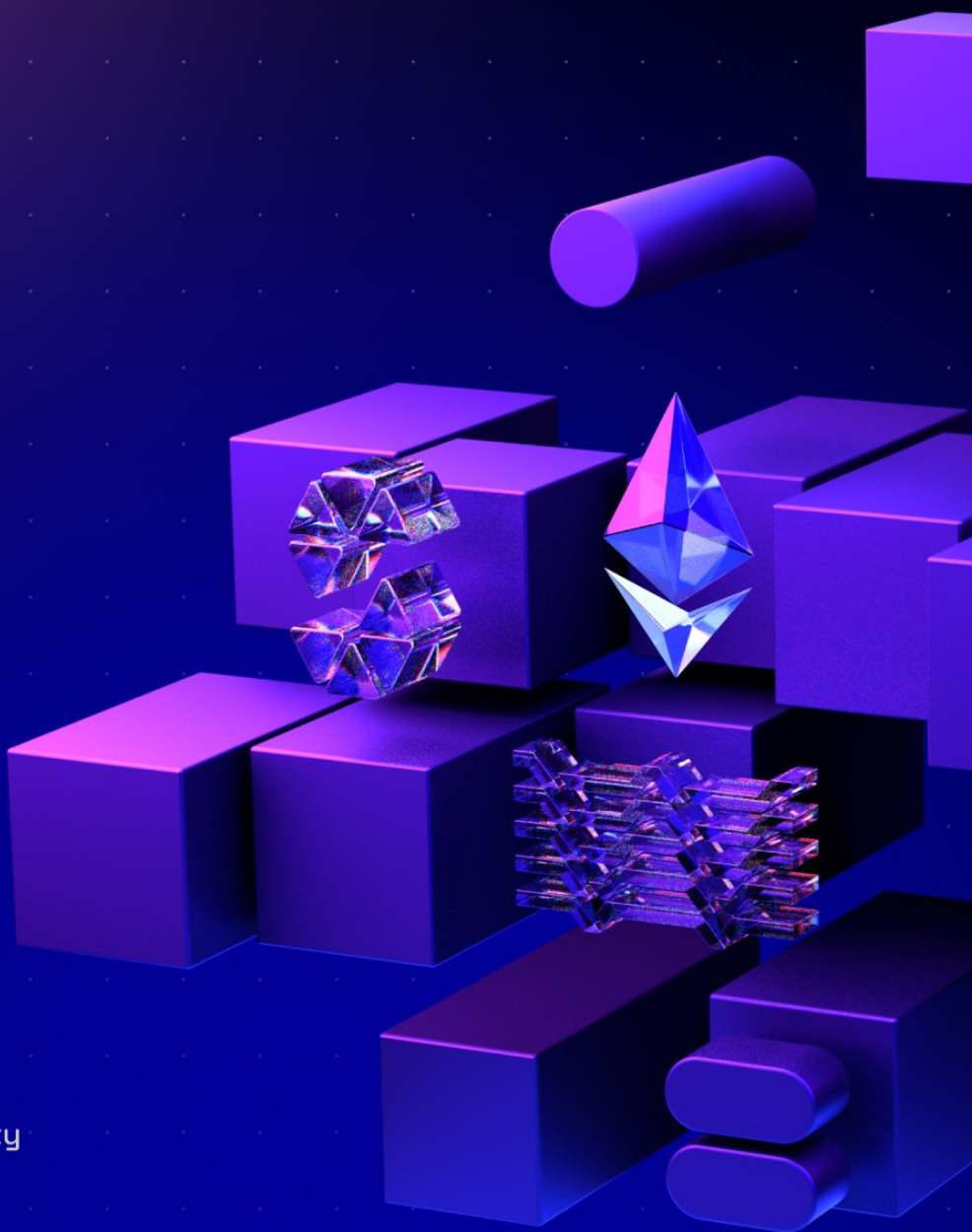
ackee
blockchain security

Aave

Umbrella

19.5.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	26.02.2025
1.0	Final Report	03.03.2025
1.1	Fix Review	04.03.2025
1.2	Fix Review	13.03.2025
1.2	Fix Review	18.03.2025
1.3	Updated commit	19.05.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

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* (system settings or parameters, such as deployment scripts, compiler configurations, using 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**	**N/A**
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*, 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* 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

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Jan Kalivoda	Lead Auditor
Naoki Yoshida	Auditor
Dmytro Khimchenko	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

Umbrella, a new version of the Aave Safety Module, is introduced to help address bad debt managment within the Aave protocol.

Revision 1.0

BGD engaged Ackee Blockchain Security to perform a security review of the Aave protocol with a total time donation of 19 engineering days in a period between February 10 and February 26, 2025, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `a2ad2ff`[1] and the scope was the `umbrella`, `stakeToken` and `rewards` folders (for a detailed list, see the Appendix B).

We began our review using static analysis tools, including Wake. This yielded the I2 finding. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we used the Wake testing framework. We implemented additional unit tests that helped us analyze the shares inflation possibility (M1) and arithmetic errors (L1). We also implemented an additional set of fuzz tests, however, a full fuzzing campaign was not in the scope of this report. The fuzz tests discovered potential integration issues with the price oracle availability (L2). During the review, we paid special attention to:

- analyzing ERC-4626 shares inflation and checking for compliance with the standard;
- ensuring the slashing mechanism could not be abused;
- checking correctness of the rewards distribution;
- ensuring the arithmetic of the system was correct;
- detecting possible reentrancies and unprotected calls in the code;

- ensuring access controls were not too relaxed or too strict; and
- looking for common issues such as data validation.

Our review resulted in 9 findings, ranging from Info to Medium severity. The most severe finding is M1, which identified an issue with shares inflation. Due to the slashing mechanism, shares can grow rapidly, making the correct functioning of the system significantly dependent on configuration. StakeToken vaults that undergo full slashing due to small deficit offsets or higher pool deficits can enter a denial-of-service state. The state can be entered by an attacker in a single transaction due to the permissionless nature of slashing and deposits. The cost of the attack is determined by the underlying token (it can be as low as a few cents).

Ackee Blockchain Security recommends BGD:

- set up off-chain monitoring for the purposes described in the M1 finding; and
- address all other reported issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

The fix review was done on the given commit `de990c5`[2]. The M1 and L1 findings were acknowledged due to the unlikely likelihood that these issues occur. For a detailed acknowledgment statement from the client, see the findings. For the result of the other issues, see the Findings Summary with updated statuses.

Revision 1.2

The team provided an updated commit with final changes before the release. The review was conducted on commit `5b987d2`[3]. No issues were identified

during this review.

Revision 1.3

The team provided an updated commit (`ff7fa2d`) with minor changes from the previous revision. The internal repository of this review was mirrored into the public repository under the Aave DAO organization. The new repository commit `62f3850`[4] was verified to match the state of the internal repository.

The review was conducted on the new commit `62f3850`[5]. The review resulted in one issue (W3) that was directly related to the code change from the previous revision. The team acknowledged the issue.

[1] full commit hash: `a2ad2ff56917861e9c3bbe23010c5f0164d41ac3`

[2] full commit hash: `de990c5c7b5c46d52eccab838dabc224adac8b8f`

[3] full commit hash: `5b987d222355a1a8fa4b475e7f31968f66dd2394`

[4] full commit hash: `62f3850816b257087e92f41a7f37a698f00fa96e`

[5] full commit hash: `62f3850816b257087e92f41a7f37a698f00fa96e`

4. Findings Summary

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

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	1	2	3	4	10

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
M1: Possible shares inflation	Medium	1.0	Acknowledged
L1: Frequent claiming of rewards can lead to losses	Low	1.0	Acknowledged
L2: The `latestAnswer` function reverts after slashing configuration removal	Low	1.0	Fixed
W1: Inconsistent usage of `_msgSender()` over `msg.sender`	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W2: Missing validation of the upper bound in `validateTargetLiquidity`	Warning	1.0	Fixed
I1: Typos	Info	1.0	Fixed
I2: Unused using-for directive	Info	1.0	Fixed
I3: Permit error handling	Info	1.0	Acknowledged
I4: The same suffix is used for name and symbol	Info	1.0	Acknowledged
W3: Missing data validation during token creation	Warning	1.3	Acknowledged

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Naoki Yoshida	Auditor
Dmytro Khimchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Umbrella is a new version of the Aave Safety Module that addresses bad debt management within the Aave protocol. It comes with an upgrade of Aave v3.3 which introduces logging of bad debts, called deficits. Umbrella consists of three main components: RewardsController, which is responsible for distributing rewards to users staking within the Umbrella system; StakeToken, which is an ERC-4626 compatible vault token that holds assets; and the Umbrella contract, which orchestrates the system and serves as the entry-point for addressing deficits through the slashing mechanism. Users staking within the Umbrella system receive rewards for being exposed to the risk of slashing to cover deficits.

Trust Model

While the permissions within the system are carefully designed to limit the potential impact of any single component, users should trust the `DEFAULT_ADMIN_ROLE` (which should be granted to Aave governance) to correctly configure the system and act honestly.

Findings

The following section presents the list of findings discovered in this revision.

For the complete list of all findings, Go back to Findings Summary

M1: Possible shares inflation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	UmbrellaStakeToken.sol	Type:	Logic error

Description

The UmbrellaStakeToken is an ERC-4626 vault. The ERC-4626 standard is known to be potentially vulnerable to share inflation when the conversion rate between shares and assets can be significantly affected. As a result, shares can grow exponentially over time up to the maximum numbers for storing integers. Once this happens, the ERC-4626 vault enters a denial-of-service state due to overflows/underflows caused by huge integers.

In the case of the UmbrellaStakeToken, the conversion rate calculation is defined by the OpenZeppelin implementation; however, it can be significantly affected by the slashing mechanism.

The following simulation demonstrates how the shares can grow over time:

1. Deploy StakeToken for WETH
2. 10 users deposit 0.1 WETH each to the StakeToken
3. The conversion rate is 1:1
4. The StakeToken is slashed for 1 WETH (full slashing)
5. Currently 1 WETH is equivalent to 999999000001000000000000000000 shares (so just with the first full slashing the value moved from 10^18 to 10^30)
6. 10 users deposit 0.1 WETH each again and get slashed for the full amount repeatedly 4 times

7. At this point, the shares amount for 1 WETH will raise an underflow error (for 0.1 WETH it is 99999500001899995000011099979500033899948700072799900200126799844300155699844 (~10^77))

With larger amounts, the shares will grow even faster. For example, with 100,000 tokens with 18 decimals, only 3 full slashes and redeposits of this amount are required. Additionally, an 18 decimal token can be much cheaper than WETH. An example of a reputable token with 18 decimals is USDC on the BNB chain. In this case, depositing $1 and slashing it 5 times (while repeating the $1 deposit) would make the contract unusable.

Moreover, the described steps can be performed in one transaction. Once the suitable conditions are met, anyone can broadcast a transaction that will make the contract unusable. Because the issue is sensitive to the specific configuration, the likelihood of the issue is considered low.

Exploit scenario

Alice repeatedly deposits and slashes the vault during the lifetime of the contract. As a result, the vault becomes unusable due to share inflation.

Recommendation

In general, be aware of the share inflation, set up off-chain monitoring, and migrate before the shares become too large, since it is not possible to fully mitigate the issue with the current design.

More specifically, introduce a cooldown period on slashing. This way, it won't be possible to enter the denial-of-service state after one transaction. Since the act of slashing is necessity for growing shares rapidly under normal conditions, once the slashing is triggered, the protocol maintainers can react by pausing the vault or setting a different deficit offset thanks to the off-chain monitoring and alerting infrastructure. The cooldown period length is

subject to discussion, but it should be at least as long as the time needed to react to the alert and pause the vault.

Acknowledgment 1.1

> *We are aware of this problem and the system provides for periodic replacement of UmbrellaStakeToken, there are also additional tools to avoid a sharp increase in shares at the initial stages within Umbrella contract. However, even if a token is disabled as a result of an attack, we see absolutely no consequences, given that for such a situation to occur, very little liquidity must be blocked in the token. If the attacker deliberately provokes such a situation, then his losses are colossally greater than the system's losses.*
>
> — BGD Team

Go back to Findings Summary

L1: Frequent claiming of rewards can lead to losses

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	*	Type:	Arithmetics

Description

The indexes for users are updated in the `_updateUserData` function. This function is called within the `_updateData` function if the user address is not zero. This function is callable with a valid user address when claiming rewards or calling the `handleDeposit` hook from the StakeToken contract.

Due to calculation imprecisions (rounding, divisions before multiplications) in the `EmissionMath` library, users can potentially receive more rewards if the index is not updated frequently (see Exploit Scenario). Also, until the user balance and indexes difference is less than the scaling factor (10^18), no rewards are accrued (which is a known issue).

Listing 1. Excerpt from EmissionMath

```
103 function calculateAccrued(
104   uint152 newRewardIndex,
105   uint152 oldUserIndex,
106   uint256 userBalance
107 ) internal pure returns (uint112) {
108   return ((userBalance * (newRewardIndex - oldUserIndex)) /
    SCALING_FACTOR).toUint112();
109 }
```

A potential problem can arise if a malicious actor starts sending small amounts of StakeToken to the victim (as low as 1 wei). This can lead to an update of indexes for each block (for the victim) because the `handleDeposit` hook is called with the victim's address. As a consequence of such griefing,

the user (victim) can accrue fewer rewards than expected. Moreover, the victim is not the only one affected by this issue, it affects all the users of the pool. The size of the lost amount highly depends on several parameters, which are examined further in the Exploit Scenario.

Measurement of the calculation imprecision

For a stake token, there is a configured RT1 reward token with 10 * 10^6 emission per second and 18 decimals. The target liquidity is set to 10000 * 10^18. The following script is used for measurement.

```
print(f"\nPercentage Differences - Multiple Updates vs Single Update
(reference):")
with chain.change_automine(False):
    for blocks_num in block_nums:
        with chain.snapshot_and_revert():
            print(f"\n\nTesting with {blocks_num} blocks:")
            print("============================")

            # Store results from both execution paths
            multiple_updates_rewards = None
            single_update_rewards = None

            with chain.snapshot_and_revert():
                for i in range(blocks_num):
                    tx = stake_token_usdc.transfer(alice, 1, from_=griefer,
confirmations=0)
                    chain.mine(lambda x: x + 1)

                chain.mine(lambda x: x + 1)
                same_timestamp = chain.blocks["pending"].timestamp
                multiple_updates_rewards = get_balances()

            chain.mine_many(blocks_num, 1)
            stake_token_usdc.transfer(alice, 1, from_=griefer, confirmations=0)
            chain.mine(lambda x: x + 1)
            chain.set_next_block_timestamp(same_timestamp)
            single_update_rewards = get_balances()

            # Calculate and print percentage differences
            for key in multiple_updates_rewards:
                multiple_val = multiple_updates_rewards[key]
```

```
                single_val = single_update_rewards[key]
                pct_diff = (abs(multiple_val - single_val) / single_val) * 100

                if pct_diff < 0.0001:  # Less than 0.0001% difference is
considered "no change"
                    print(f"{key}: No change")
                else:
                    direction = "+" if multiple_val > single_val else "-"
                    print(f"{key}: {direction}{pct_diff:.4f}%")
```

First, it snapshots the state of the chain to perform changes with the applied griefing, and then continues with the previous clean state to update the indexes only once. With some additional checks to make anvil behave as expected, the results are saved and evaluated as a percentage difference relative to the one-time reward update.

The following tables will show results for different deposits.

25% of the target liquidity: Alice: 1,000 * 10^18; Bob: 1,500 * 10^18

Blocks	Alice RT1	Bob RT1
100	-0.0566%	-0.0566%
300	-0.0570%	-0.0570%
1000	-0.0571%	-0.0571%
3000	-0.0571%	-0.0571%
6000	-0.0571%	-0.0571%
10000	-0.0571%	-0.0571%

Table 4. Percentage Differences - Multiple Updates vs referential Single Update

90% of the target liquidity: Alice: 6,000 * 10^18; Bob: 3,000 * 10^18

Blocks	Alice RT1	Bob RT1
100	-0.0900%	-0.0900%

Blocks	Alice RT1	Bob RT1
300	-0.0906%	-0.0906%
1000	-0.0908%	-0.0908%
3000	-0.0909%	-0.0909%
6000	-0.0909%	-0.0909%
10000	-0.0909%	-0.0909%

Table 5. Percentage Differences - Multiple Updates vs referential Single Update

900% of the target liquidity: Alice: 60,000 * 10^18; Bob: 30,000 * 10^18

Blocks	Alice RT1	Bob RT1
100	-0.9914%	-0.9914%
300	-0.9979%	-0.9979%
1000	-0.9991%	-0.9991%
3000	-0.9998%	-0.9998%
6000	-1.0000%	-1.0000%
10000	-0.9999%	-0.9999%

Table 6. Percentage Differences - Multiple Updates vs referential Single Update

9000% of the target liquidity: Alice: 600,000 * 10^18; Bob: 300,000 * 10^18

Blocks	Alice RT1	Bob RT1
100	-9.9220%	-9.9220%
300	-9.9813%	-9.9813%
1000	-9.9921%	-9.9921%
3000	-9.9981%	-9.9981%
6000	-9.9996%	-9.9996%

Blocks	Alice RT1	Bob RT1
10000	-9.9992%	-9.9992%

Table 7. Percentage Differences - Multiple Updates vs referential Single Update

Root cause of the issue

This behavior is caused by the precision loss in the following line of code:

Listing 2. Excerpt from EmissionMath

```
91 uint256 indexIncrease = (currentEmission * timeDelta) /
   extraParamsForIndex.totalSupply;
```

When the `currentEmission` value is small enough (in the example above, it is `10 * 10^6` when the token has `18` decimals) and the total supply is already large enough, this line of code experiences precision loss. For example, in the test described above, during execution of this line after sending 1 wei to the victim, the values of the variables are:

```
currentEmission = 10 * 10^6
totalSupply = 900_000 * 10^18
timeDelta = 1
```

It should be mentioned that `targetLiquidity` is reached, and emission is flat.

```
indexIncrease = (10 * 10**6 * 1) // 900_000 * 10**18;
```

As a result, the index increase is lost in the current calculation, and the user will not receive emissions for this index increase. This scenario is realistic for pools with small emissions and large total supply. It is unlikely to have pools that would exceed the target liquidity so extremely. However, for pools that don't reach the `targetLiquidity`, the precision loss is still present (see results

above). If we increase the emission from the 90% target liquidity example above to 10 * 10**8, then we lose only -0.0009%, so the configuration of this variable is crucial for this issue to occur.

Exploit scenario

Alice deposits her tokens into the StakeToken to receive rewards for staking them. Bob is a malicious actor who wants to damage all stakers in the pool. He starts sending 1 wei of StakeToken to Alice every block. As a result, Alice receives fewer rewards than expected.

Recommendation

Be aware that these configurations can lead to a precision loss. Consider implementing constraints against configurations that can lead to precision loss and possible griefing attacks to protect users from losing rewards.

Acknowledgment 1.1

> *We accept that there is some calculation error, but we believe that it is insignificant. In a situation where extreme values are not chosen, we assume that this error will not exceed 0.01%, which is a good result for most cases.*
>
> — BGD Team

Go back to Findings Summary

L2: The `latestAnswer` function reverts after slashing configuration removal

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	UmbrellaStakeToken.sol	Type:	Logic error

Description

When a slashing configuration is removed via the `removeSlashingConfigs` function in the Umbrella contract, it deletes the `stakesData`, which also includes the `underlyingOracle` address. This causes `latestUnderlyingAnswer()` in the UmbrellaStakeToken contract to revert due to a missing address in the Umbrella contract. In cases where other contracts or protocols rely on the `latestAnswer()` function from the UmbrellaStakeToken contract, removing the slashing configuration can disrupt the operation of these contracts. Otherwise, the token remains functional.

Listing 3. Excerpt from UmbrellaConfiguration.removeSlashingConfigs

```
135 delete $.stakesData[removalPairs[i].umbrellaStake];
```

Listing 4. Excerpt from UmbrellaStakeToken.latestAnswer

```
39 uint256 underlyingPrice = uint256(
40   IUmbrellaConfiguration(owner()).latestUnderlyingAnswer(address(this))
41 );
```

Exploit scenario

Alice (third party integrator) utilizes the `latestAnswer` function of the token within her protocol. Bob (the umbrella admin) removes the slashing configuration, causing the `latestAnswer` function to revert and Alice's

protocol to malfunction.

Recommendation

Be aware of this behavior and inform the integrators or maintain the underlying oracle information even after configuration removal.

Fix 1.1

The issue is fixed by deleting only the `reserve` information from the `stakesData` array and thus keeping the `underlyingOracle` address active even after configuration removal.

```
delete $.stakesData[removalPairs[i].umbrellaStake].reserve;
```

Go back to Findings Summary

W1: Inconsistent usage of `_msgSender()` over `msg.sender`

Impact:	Warning	Likelihood:	N/A
Target:	*	Type:	Logic error

Description

The codebase predominantly uses `_msgSender()` instead of `msg.sender` for sender address retrieval. However, in specific places in the inheritance chain, `msg.sender` is used. While this is not an issue in the current scope, since the `_msgSender()` function returns exactly the `msg.sender` value, this inconsistency should be addressed for possible future changes.

The following code listings show the usage of `msg.sender` in in-scope files that are already using `_msgSender()`.

Listing 5. Excerpt from UmbrellaConfiguration

```
310 function _checkRescueGuardian() internal view override {
311   _checkRole(RESCUE_GUARDIAN_ROLE, msg.sender);
312 }
```

Listing 6. Excerpt from RewardsDistributor

```
57 modifier onlyAuthorizedClaimer(address user) {
58   require(isClaimerAuthorized(user, msg.sender),
   ClaimerNotAuthorized(msg.sender, user));
```

Listing 7. Excerpt from RewardsDistributor

```
70 function claimAllRewards(
71   address asset,
72   address receiver
73 ) external returns (address[] memory, uint256[] memory) {
74   return _claimAllRewards(asset, msg.sender, receiver);
```

Listing 8. Excerpt from RewardsDistributor

```
94 bytes32 structHash = keccak256(
95   abi.encode(CLAIM_ALL_TYPEHASH, asset, user, receiver, msg.sender,
   _useNonce(user), deadline)
96 );
```

Listing 9. Excerpt from RewardsDistributor

```
109 return _claimSelectedRewards(asset, rewards, msg.sender, receiver);
```

Listing 10. Excerpt from RewardsDistributor

```
140 msg.sender,
141 nonce,
142 deadline
```

Listing 11. Excerpt from RewardsDistributor

```
154 address receiver
```

Listing 12. Excerpt from RewardsDistributor

```
166 /// @inheritdoc IRewardsDistributor
```

Recommendation

Unify the usage of `_msgSender()` over `msg.sender` (or the opposite) throughout the codebase.

Fix 1.1

The issue was addressed and the `msg.sender` usage is replaced with `_msgSender()` in the UmbrellaConfiguration contract. In the RewardsDistributor contract, the `msg.sender` is kept due to the potential issues with inheritance chain.

Go back to Findings Summary

W2: Missing validation of the upper bound in `validateTargetLiquidity`

Impact:	Warning	Likelihood:	N/A
Target:	EmissionMath.sol, RewardsController.sol	Type:	Data validation

Description

The `validateTargetLiquidity` function does not validate against the specified upper bound for target liquidity.

Listing 13. Excerpt from EmissionMath

```
174 function validateTargetLiquidity(uint256 targetLiquidity, uint8 decimals)
    internal pure {
175   require(targetLiquidity >= 10 ** decimals, TargetLiquidityTooLow());
176 }
```

As a result, the target liquidity in the RewardsController can be set to the `uint160` maximum value.

Listing 14. Excerpt from RewardsController

```
540 function _updateTarget(
541   InternalStructs.AssetData storage assetData,
542   address asset,
543   uint256 newTargetLiquidity
544 ) internal {
545   EmissionMath.validateTargetLiquidity(newTargetLiquidity,
    IERC20Metadata(asset).decimals());
546
547   assetData.targetLiquidity = newTargetLiquidity.toUint160();
548
549   emit TargetLiquidityUpdated(asset, newTargetLiquidity);
550 }
```

The documentation specifies the following constraints for `targetLiquidity`:

> *Minimum: 1 asset token.*
>
> *Maximum: ~1e35. (The upper bound is indirectly provided by the further validation performed on the minimum value required for the maxEmissionPerSecond. maxEmissionPerSecond must be <= 1e21 but also >= targetLiquidity * 1e3 / 1e18.)*

Recommendation

Implement the upper bound validation for the `targetLiquidity` parameter.

Fix 1.1

The upper bound validation is added to the `validateTargetLiquidity` function.

```
require(targetLiquidity >= 10 ** decimals && targetLiquidity <= 1e36,
TargetLiquidityTooLow());
```

Go back to Findings Summary

I1: Typos

Impact:	Info	Likelihood:	N/A
Target:	Umbrella.sol	Type:	Code quality

Description

There is a typo in the comment:

Listing 15. Excerpt from Umbrella

```
172 // Due to rounding error (cause of index growth), it is possible that we
    receive some wei less then expected
```

The word ”then” should be ”than” in the comparison context.

Recommendation

Fix the typo.

Fix 1.1

The typo is fixed.

Go back to Findings Summary

I2: Unused using-for directive

Impact:	Info	Likelihood:	N/A
Target:	ERC4626StakeTokenUpgradeable.sol	Type:	Code quality

Description

The `ERC4626StakeTokenUpgradeable` contract contains the following unused using-for directive.

Listing 16. Excerpt from ERC4626StakeTokenUpgradeable

```
30 using Math for uint256;
```

Recommendation

Remove the unused using-for directive.

Fix 1.1

The unused using-for directive is removed.

Go back to Findings Summary

I3: Permit error handling

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

Description

The `permit` function is correctly implemented in the `try/catch` clause to prevent permit front-running denial of service. However, when the `permit` function call fails, the error is not emitted and the transaction fails in later stages, such as on insufficient allowance. The error handling could be improved by adding a check for the desired allowance in the `catch` block. This way, if the allowance was not set with permit, the user would be notified (by reverting with a descriptive error message) that the issue is with the `permit` function call.

An example of the failure case occurs when `permit` is not supported on the `StakeToken` asset and there is no prior allowance set. In this case, the transaction will always fail due to insufficient allowance without indicating that the `permit` call failed.

Listing 17. Excerpt from StakeToken

```
77 try
78   IERC20Permit(asset()).permit(
79     _msgSender(),
80     address(this),
81     assets,
82     deadline,
83     sig.v,
84     sig.r,
85     sig.s
86   )
87 {} catch {}
88
89 return deposit(assets, receiver);
```

Recommendation

Add a check to the `catch` block to continue execution if the allowance is sufficient; otherwise, emit a descriptive error message about the failed `permit` call.

Acknowledgment 1.1

> *Adding such a check will increase the cost of the transaction, although it will not lead to significant improvements, since the transaction will revert anyway as a result of a transfer error. We believe that most users will not try to send invalid signatures (or with an expired deadline), so optimizing for the majority of cases has a higher priority than for the minority.*
>
> — BGD Team

Go back to Findings Summary

I4: The same suffix is used for name and symbol

Impact:	Info	Likelihood:	N/A
Target:	UmbrellaStkManager.sol	Type:	Configuration

Description

When creating the stake token, the same `suffix` string is used for both the token name and symbol.

Listing 18. Excerpt from UmbrellaStkManager._getStakeNameAndSymbol

```
270     isSuffixNotEmpty ? string(abi.encodePacked(' ', suffix)) : ''
271   )
272 );
273
274 // `stk+symbol+.+suffix` or `stk+symbol`
275 string memory symbol = string(
276   abi.encodePacked(
277     'stk',
278     IERC20Metadata(underlying).symbol(),
279     isSuffixNotEmpty ? string(abi.encodePacked('.', suffix)) : ''
```

It might be better to create different suffixes for the name and symbol. For example, for the following name:

- Name: "Umbrella Stake ERC4626-Wrapped Aave v3 USDC Version 1.0"

The current implementation produces this symbol:

- Symbol: "stkwaUSDC.Version 1.0"

A more appropriate symbol would potentially be:

- Symbol: "stkwaUSDC.v1.0"

Recommendation

Modify the function to accept separate suffix parameters for name and symbol.

Acknowledgment 1.1

> *Suffixes were intended as short names that were reduced to the minimum. They should not contain full names of words or markets.*
>
> — BGD Team

Go back to Findings Summary

Report Revision 1.3

Revision Team

Revision team is the same as in Report Revision 1.0.

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.3.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

W3: Missing data validation during token creation

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

Description

In the previous revision, in the `_createStakeToken` function the `umbrellaStakeToken` address was retrieved from the `createDeterministic` call, however, in the current revision it uses value from the `predictCreateDeterministic` call.

```
address umbrellaStakeToken =
TRANSPARENT_PROXY_FACTORY().predictCreateDeterministic(
  UMBRELLA_STAKE_TOKEN_IMPL(),
  SUPER_ADMIN(),
  creationData,
  ''
);

if (umbrellaStakeToken.code.length == 0) {
  // name and symbol inside creation data is considered as unique, so using
different salts is excess
  // if for some reason we want to create different tokens with the same name
and symbol, then we can use different `cooldown` and `unstakeWindow`
  TRANSPARENT_PROXY_FACTORY().createDeterministic(
    UMBRELLA_STAKE_TOKEN_IMPL(),
    SUPER_ADMIN(),
    creationData,
    ''
  );
}

_getUmbrellaStkManagerStorage().stakeTokens.add(umbrellaStakeToken);
```

Since there are no guarantees that the `predictCreateDeterministic` function is not a state-changing call and does not return the same address as the `createDeterministic` call, the invariant check is missing. In a case where the creation call returns a different address than the predicted one, the

predicted address will be saved into the storage instead of the actual one.

Recommendation

Implement an invariant check to verify that the returned address from the creation call is the same as the predicted one.

Acknowledgment 1.3

> *We plan to add it in a future update. But at this point, the contracts have already been deployed, and implementing the check would require redeployment, which we don't think it's worth it now.*
>
> — BGD Team

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Aave: Umbrella, 19.5.2025.

Appendix B: Scope

The following files were included in the scope of the audit.

```
src/contracts
├── rewards
│   ├── RewardsController.sol
│   ├── RewardsDistributor.sol
│   ├── interfaces
│   │   ├── IRewardsController.sol
│   │   ├── IRewardsDistributor.sol
│   │   └── IRewardsStructs.sol
│   └── libraries
│       ├── EmissionMath.sol
│       └── InternalStructs.sol
├── stakeToken
│   ├── StakeToken.sol
│   ├── UmbrellaStakeToken.sol
│   ├── extension
│   │   └── ERC4626StakeTokenUpgradeable.sol
│   └── interfaces
│       ├── IERC4626StakeToken.sol
│       ├── IOracleToken.sol
│       ├── IStakeToken.sol
│       └── IUmbrellaStakeToken.sol
└── umbrella
    ├── Umbrella.sol
    ├── UmbrellaConfiguration.sol
    ├── UmbrellaStkManager.sol
    └── interfaces
        ├── IUmbrella.sol
        ├── IUmbrellaConfiguration.sol
        └── IUmbrellaStkManager.sol
```

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz