AAVE UMBRELLA SECURITY AUDIT REPORT

May 19, 2024

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals

- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

Aave **Umbrella** is a smart contract that manages staking, slashing, and deficit coverage for Aave pools. Users stake `aToken` or `GHO` for rewards, with the risk of slashing if the pool accrues bad debt. If a deficit arises, Umbrella seizes a portion of staked assets or uses external funds to cover it, ensuring liquidity provider protection. It features methods for slashing, StakeToken management, and controlled deficit coverage, governed by role-based permissions. Slashing occurs only when deficits exceed a set threshold, preventing unnecessary asset seizures.

1.4 Project Dashboard

Project Summary

Title	Description
Client	BGD Labs
Project name	Aave Umbrella
Timeline	05.02.2025 - 15.05.2025
Number of Auditors	3

Project Log

Date	Commit Hash	Note
05.02.2025	946a220a57b4ae0ad11d088335f9bcbb0e34dcef	Commit for the audit
14.02.2025	488e165b1084b5347d1b7b8c84278accd78ef14c	Commit for the re-audit
17.02.2025	a4f0f4a89af9a4c79e7c4d0e6775d14230d19efe	Commit for the re-audit
26.02.2025	e3dde13fece757561fc199a1523470b6764f8930	Commit for the re-audit
10.03.2025	a05713ef234385a527bf0b09478d2f2ca9616cb4	Commit for the re-audit
15.05.2025	ff7fa2d3b732fa4037ef4f46ed50c662f401d5b7	Commit for the re-audit

Project Scope

The audit covered the following files:

File name	Link

File name	Link
src/contracts/umbrella/Umbrella.sol	Umbrella.sol
src/contracts/umbrella/UmbrellaConfiguration.sol	UmbrellaConfiguration.sol
src/contracts/umbrella/UmbrellaStkManager.sol	UmbrellaStkManager.sol
src/contracts/rewards/libraries/EmissionMath.sol	EmissionMath.sol
src/contracts/rewards/libraries/InternalStructs.sol	InternalStructs.sol
src/contracts/rewards/RewardsController.sol	RewardsController.sol
src/contracts/rewards/RewardsDistributor.sol	RewardsDistributor.sol
src/contracts/stakeToken/StakeToken.sol	StakeToken.sol
src/contracts/stakeToken/extension/ERC4626StakeTokenUpgradeable.sol	ERC4626StakeTokenUpgradeable.sol
src/contracts/stakeToken/UmbrellaStakeToken.sol	UmbrellaStakeToken.sol
src/contracts/helpers/UmbrellaBatchHelper.sol	UmbrellaBatchHelper.sol
src/contracts/helpers/interfaces/IUmbrellaBatchHelper.sol	IUmbrellaBatchHelper.sol
src/contracts/helpers/interfaces/IUniversalToken.sol	IUniversalToken.sol

Deployments

File name	Contract deployed on mainnet	Comment
RewardsController.sol	0x85C337...78aFb5d0	

File name	Contract deployed on mainnet	Comment
TransparentUpgradeableProxy.sol	0x4655Ce...1E38188B	RewardsController Proxy
UmbrellaStakeToken.sol	0x75e8aC...E9370197	
Umbrella.sol	0x929e21...2B72646d	
TransparentUpgradeableProxy.sol	0xD400fc...ee3881a8	Umbrella Proxy
UmbrellaBatchHelper.sol	0xCe6Ced...7b13fc2F	

1.5 Summary of findings

Severity	# of Findings
Critical	0
High	0
Medium	1
Low	2

ID	Name	Severity	Status
M-1	Underflow with `_coverDeficit()`	Medium	Fixed
L-1	Prediction inconsistencies in `_predictStakeTokenAddress()`	Low	Fixed
L-2	Potential decoding error in `try-catch`	Low	Fixed

1.6 Conclusion

Aave **Umbrella** combines staking, slashing, and deficit coverage functionalities for Aave pools. Therefore, in this audit, we paid special attention to the mechanisms underlying deficit coverage and StakeToken management.

Key vectors examined:

- Analysis of the deficit coverage logic and associated token handling.
- Review of parameter configurations affecting the slashing mechanism.
- Evaluation of the StakeToken address prediction mechanism and its interactions with potential upgrades.
- Assessment of parameter mutability and its implications for operational consistency.
- Vectors related to rounding errors and possible underflows.

In addition, our comprehensive checklist covered business logic, common ERC20 issues, interactions with external contracts, integer overflows, reentrancy attacks, access control, typecasting pitfalls, rounding errors, and other potential issues.

Comments on some vectors:

- In `ERC4626StakeTokenUpgradeable._update()` if `from` is `address(0)` and the user already has stake tokens, rewards are updated first and then balance is increased. As a result user cannot abuse rewards accumulation.
- The functions `RewardsDistributor.claimSelectedRewardsPermit()` and `RewardsDistributor.claimAllRewardsPermit()` both use the same nonce value. If a user provides two signatures at the same time and one of the functions is called, the other function cannot be called afterward because the nonce will have already been increased. However, this situation only applies if the user signs messages for both functions, which is an unlikely scenario and unrealistic in the context of this protocol's architecture.
- In `StakeToken` user cannot withdraw/redeem tokens when they out of the withdrawal window. Both functions `ERC4626StakeTokenUpgradeable.maxRedeem` and `ERC4626StakeTokenUpgradeable.maxWithdraw` are overriden so they always check that user in the withdrawal window.
- When user claims reward after `distributionEnd` timestamp, rewards are applyied only for delta between global index and user index. Global index cannot increase after the `distributionEnd` timestamp.
- If the distribution has ended and the admin reactivates it, rewards will only accrue from the reactivation point onward. For example, if `lastUpdatedTimestamp` was 100 and `distributionEnd` was also 100, and the admin reactivates distribution at 200, the `lastUpdatedTimestamp` is updated to 200. As a result, users cannot claim rewards for the period between 100 and 200.
- If rewards were not initialized using `RewardsController.configureAssetWithRewards()`, they cannot be configured via `RewardsController.configureRewards()` due to a require statement enforcing that rewards must be initialized. This means the admin must first initialize the reward before configuring it.
- If a user designates themselves as the `authorizedUser`, claiming rewards on their own behalf has the same effect as using `RewardsDistributor.claimAllRewards()` or

`RewardsDistributor.claimSelectedRewards()`. If another authorized user claims rewards on behalf of the owner using these functions, the reward owner's index is updated correctly, preventing the claimer from claiming the same rewards a second time.

- If a user passes a nonexistent reserve to `Umbrella.slashReserveDeficit()`, it cannot be slashed because the reserve's state is empty. As a result, the `newDeficit` value will be zero, causing the transaction to revert. Even if `POOL().getReserveDeficit(reserve)` returns a non-zero value, the reserve must have its configuration set by the admin for the slashing process to proceed.
- The function `Umbrella._slashAsset()` cannot return a `deficitToCover` value greater than `assetToSlash`. The `assetToSlash` value can only be reduced within `ERC4626StakeTokenUpgradeable._slash()` if it exceeds the `maxSlashable` limit.
- If an admin updates `slashingConfig` with the same `slashConfig.umbrellaStake` address as previously set, only the `liquidationBonus` is updated. The address is not duplicated in `configurationMap`.
- The pool deficit can only be eliminated (decreased) through the `Umbrella` contract. It can only be increased during `LiquidationLogic.executeLiquidationCall()` and cannot be duplicated within the same liquidation call for the same reserve.
- A user cannot claim the same reward twice. When rewards are claimed, the user's index is updated to match the global reward's index. As a result, claiming the same reward twice within a single block is not possible because `userData.index == rewardData.index`. The global index can only increase if the passed time is greater than zero. It also cannot be increased if the condition `lastUpdateTimestamp == block.timestamp` is met. The `lastUpdateTimestamp` is updated after each interaction with the `RewardController` contract, ensuring that the global index only advances based on actual time passage.
- The `RewardController` can have a maximum of 8 rewards, and this limit cannot be exceeded. The `rewardsInfo` array increases by only one element per call to `RewardController._setUpReward()`, ensuring that the reward count cannot exceed the predefined maximum.

No significant vulnerabilities were found.

Key notes:

1. The DAO can raise the Liquidation Bonus to 100% at any time, allowing it to cover more bad debt than users might expect. This is not a security issue because, ultimately, it is the decision of the DAO.
2. If a second `StakeToken` is added to a reserve in Umbrella without removing the first one, and then the first one is removed, slashing the second token may occur without offsetting bad debt and may happen immediately. This is not a security issue because, ultimately, it is the decision of the DAO.
3. Umbrella.sol#L183, there is a type conversion `uint(int)`. This conversion may cause a silent overflow; however, in this case, the oracle cannot return negative numbers. Therefore, this is not considered a security issue.
4. UmbrellaBatchHelper.sol#L326, there might be a revert on the line `abi.decode(data, (address))` if `stakeToken.asset()` has a `fallback()` function that doesn't modify the state.

The code from the final commit ff7fa2d3 completely matches the code of the public repository 62f38508.

2. FINDINGS REPORT

2.1 Critical

Not Found

2.2 High

Not Found

2.3 Medium

M-1	Underflow with `_coverDeficit()`
Severity	Medium
Status	Fixed in 488e165b

Description

An attacker can directly transfer tokens to the Umbrella contract, causing an underflow error when `Umbrella._setPendingDeficit()` or `Umbrella._setDeficitOffset()` is triggered.

This happens because the `_coverDeficit()` function returns the entire contract balance when transferring tokens, rather than just the portion sent by `msg.sender`, and hacker can increase it by a direct transfer:

```
IERC20(aToken).safeTransferFrom(_msgSender(), address(this), amount);
amount = IERC20(aToken).balanceOf(address(this));
```

Umbrella.sol#L148-L151

Governance has the ability to transfer excess tokens via an emergency function. However, an attacker can frontrun the next call to `coverDeficitOffset()` and `coverPendingDeficit()` functions, causing a DoS again.

Recommendation

We recommend considering the case where the contract already holds funds when calculating the actual received balance.

2.4 Low

L-1	Prediction inconsistencies in `_predictStakeTokenAddress()`
Severity	Low
Status	Fixed in 488e165b

Description

The `_predictStakeTokenAddress()` function calculates the StakeToken address using parameters such as `UMBRELLA_STAKE_TOKEN_IMPL()`, `SUPER_ADMIN()`, and `creationData`, which includes:

- `stakeSetup.underlying`,
- `name`,
- `symbol`,
- `stakeSetup.cooldown`,
- `stakeSetup.unstakeWindow`.

UmbrellaStkManager.sol#L239-L244

There are a number of inconsistencies in contract management and address prediction:

1. The `UmbrellaStkManager` contract stores the `umbrellaStakeTokenImpl` address at initialization and uses it when deploying new proxies for `UmbrellaStakeToken`. However, if the `UmbrellaStakeToken` implementation is upgraded, `UmbrellaStkManager.createStakeTokens()` will still create proxies with the outdated implementation, leading to inconsistencies between old and new stake tokens.

It is possible to upgrade the `Umbrella` contract to use a new `UmbrellaStakeToken` implementation, but it would make the function `_predictStakeTokenAddress()` to return incorrect addresses for older implementations.

2. Multiple tokens with the same `name`, `symbol` and `underlying` could be created using different `cooldown` and `unstakeWindow`.

3. In the created **StakeToken** contract, it is possible to update the **cooldown** and **unstakeWindow** parameters. This leads to uncertainty — when specifying these parameters along with their new values, the prediction function will return an address different from the one where the **StakeToken** was originally created, since the parameters have changed.

Moreover, considering that we can create two identical **StakeTokens** with the same symbol and underlying asset, and then modify the **cooldown** and **unstakeWindow** parameters in one of them to match the other, the uncertainty increases even further. This raises the question of what the prediction function should return in such cases.

Recommendation

If the function is intended to be used only during the deployment of new **StakeTokens**—to set the **StakeToken** address in other contracts before the actual deployment—then we recommend adding a warning in the comments stating that this function should not be used for any other purpose.

L-2	Potential decoding error in `try-catch`
Severity	Low
Status	Fixed in a05713ef

Description

In the `UmbrellaBatchHelper` contract, there is a potential issue with error handling in the following try-catch block:

```
try IUniversalToken(underlyingOfStakeToken).aToken() returns (address _aToken)
```

UmbrellaBatchHelper.sol#L324

The `aToken()` call is made using `staticcall` since it's a view function. If the `underlyingOfStakeToken` contract has a `fallback()` function **that doesn't modify state**, a decoding error could occur that won't be caught by the catch clause and the whole transaction will revert.

While the code works correctly with tokens like WETH (where the fallback function modifies state and the `staticcall` revert is caught), there's a possibility of encountering tokens with fallback functions that don't modify state. In such cases, the decoding error would not be caught, leading to unexpected behavior.

Recommendation

We recommend considering using `staticcall` directly and checking its return value.

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes