GA GUARDIAN

246 Club

re-a-token

Security Assessment

August 5th, 2025

Summary

Audit Firm Guardian

Prepared By Roberto Reigada, Zdravko

Client Firm 246 Club

Final Report Date August 5, 2025

Audit Summary

246 Club engaged Guardian to review the security of their 246 Club's re-a-token. From the 29th of July to the 5th of August, a team of 2 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Confidence Ranking

Given the number of non-critical issues detected and code changes following the main review, Guardian assigns a Confidence Ranking of 3 to the protocol. Guardian advises the protocol to address issues thoroughly and consider a targeted follow-up audit depending on code changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

🔗 Blockchain network: **Sonic**

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

Guardian Confidence Ranking

Confidence Ranking	Definition and Recommendation	Risk Profile
5: Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. **Recommendation:** Code is highly secure at time of audit. Low risk of latent critical issues.	0 High/Critical findings and few Low/Medium severity findings.
4: High Confidence	Code is clean, well-structured, and adheres to best practices. Only Low or Medium-severity issues were discovered. Design patterns are sound, and test coverage is reasonable. Small changes, such as modifying rounding logic, may introduce new vulnerabilities and should be carefully reviewed. **Recommendation:** Suitable for deployment after remediations; consider periodic review with changes.	0 High/Critical findings. Varied Low/Medium severity findings.
3: Moderate Confidence	Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases. **Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1 High finding and ≥ 3 Medium. Varied Low severity findings.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities (~2/week). Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. **Recommendation:** Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week.
1: Very Low Confidence	Code has systemic issues. Multiple High/Critical findings (≥5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured. **Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Overview

Project Summary

Project Name	246 Club
Language	Solidity
Codebase	https://github.com/246club/re-a-token
Commit(s)	Initial commit: 56eef1dcda0f15b9306dfde407bf47dc7d62ad27

Audit Summary

Delivery Date	August 5, 2025
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
Critical	0	0	0	0	0	0
High	4	0	0	0	0	4
Medium	7	0	0	2	3	2
Low	2	0	0	0	0	2
Info	6	0	0	3	2	1

Audit Scope & Methodology

Scope and details:

The main focus of our review is the Re-a-token scope:

contract,source,total,comment
re-a-token/src/ReAToken.sol,153,255,44
re-a-token/src/ReATokenFactory.sol,28,49,10
re-a-token/src/libraries/ErrorsLib.sol,4,6,1
source count: {
  total: 310,
  source: 185,
  comment: 55,
  single: 22,
  block: 33,
  mixed: 0,
  empty: 70,
  todo: 0,
  blockEmpty: 0,
  commentToSourceRatio: 0.2972972972972973}

As a part of this review, the following facets from the 246 core repo were reviewed:

contract,source,total,comment
./src/facets/ViewerFacet.sol,227,347,40
./src/facets/RestakingFacet.sol,84,121,9
./src/facets/InterestManagementFacet.sol,58,92,8
source count: {
  total: 560,
  source: 369,
  comment: 57,
  single: 35,
  block: 22,
  mixed: 1,
  empty: 135,
  todo: 0,
  blockEmpty: 0,
  commentToSourceRatio: 0.15447154471544716}

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	🔴 Critical	🟠 High	🟡 Medium
Likelihood: *Medium*	🟠 High	🟡 Medium	🟢 Low
Likelihood: *Low*	🟡 Medium	🟢 Low	🟢 Low

Impact

High Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.

Medium A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.

Low Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.

Medium An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.

Low Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	ReAToken.mint Is Permanently Unusable	DoS	High	Resolved
H-02	Deposits To ReAToken Fail If Interest Is 0	DoS	High	Resolved
H-03	Missing UNDERLYING_ASSET Approval	DoS	High	Resolved
H-04	claimInterest Tries To Transfer Yet-To-Be-Paid Interest	Logical Error	High	Resolved
M-01	ReATokenFactory.createReAToken Can Be Front-run	Unexpected Behavior	Medium	Resolved
M-02	Potential Inflation Attack In ReAToken	Frontrunning	Medium	Partially Resolved
M-03	maxRedeem/maxWithdraw Returns Inaccurate Values	Warning	Medium	Resolved
M-04	Unprotected External Callback Enables Reentrancy	Warning	Medium	Acknowledged
M-05	accrueAndDistributeInterest Calls Could Revert	DoS	Medium	Partially Resolved
M-06	migrateDelegationAsset Is Permissionless	Unexpected Behavior	Medium	Partially Resolved
M-07	Wrong Interest Repayment Logic	Logical Error	Medium	Acknowledged
L-01	Stale Liquidity Index Usage	Warning	Low	Resolved
L-02	Missing DelegationPairAssets Validation	Validation	Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
I-01	Pool Pausing In 246 Protocol	DoS	Info	Partially Resolved
I-02	Redundant Import Statement	Best Practices	Info	Resolved
I-03	Missing Global Reentrancy Guard	Best Practices	Info	Acknowledged
I-04	Pool Utilization Increases After Some Operations	Warning	Info	Partially Resolved
I-05	Lack Of Event Emissions In Setter Functions	Best Practices	Info	Acknowledged
I-06	Club246 Interest Impacts Available Power	Informational	Info	Acknowledged

H-01 | ReAToken.mint Is Permanently Unusable

Category	Severity	Location	Status
DoS	High	ReAToken.sol	Resolved

Description

The `ReAToken.mint` function calls OpenZeppelin's `ERC4626` pre-flight check `require(shares < maxMint(receiver))`. `maxMint` is implemented as `_convertToShares(_maxDeposit(), Math.Rounding.Floor)`.

When the underlying restaking pair is enabled, `_maxDeposit` returns the literal maximum `uint256`. Passing this value into `_convertToShares` makes the library multiply by the current `totalSupply`, which is equal to 1e12 initially due to the `+ 10 ** _decimalsOffset()` addition (for aUSDC/USDC pair), which inevitably overflows with a `panic: arithmetic underflow or overflow (0x11)` error.

As a consequence every external call to `mint` (and any higher-level flow that relies on it, such as front-end integrations or automated strategies) reverts even when the vault is perfectly solvent and its supply cap on Aave has room.

Recommendation

Given that there is no max. cap if the asset is restakable, consider updating the `ReAToken.maxMint` function to:

```
function maxMint(address) public view override(ERC4626, IERC4626) returns (uint256) {
return _maxDeposit();
}
```

Resolution

246 Club Team: The issue was resolved in commit c596260.

H-02 | Deposits To ReAToken Fail If Interest Is 0

Category	Severity	Location	Status
DoS	High	ReAToken.sol	Resolved

Description

Every call to ReAToken._deposit (deposit, mint and their ERC-4626 helpers) executes the sequence:

```
super._deposit(caller, receiver, assets, shares);   // share accounting
_supplyClub246(assets);                             // forward aTokens to Diamond(Club246)
_accrueInterest();                                  // ← fatal if no interest is ready
```

Inside _accrueInterest the vault consults the Diamond(Club246) expectedRestakeAmount in its ViewerFacet:

```
(_, uint256 expectedInterest) =
IViewerFacet(DIAMOND246).expectedRestakeAmount(delegationPairAssets, address(this));
bool restakable = IViewerFacet(DIAMOND246).isRestakable(delegationPairAssets);
if (restakable || expectedInterest > maxAssetsSuppliableToAave()) return; // no guard for zero
uint256 interest =
IInterestManagementFacet(DIAMOND246).claimInterest(delegationPairAssets, address(this), address(this));
```

• IInterestManagementFacet.claimInterest reverts with ZeroInterest when expectedInterest is zero.
• Even if that revert were removed, the next line _supplyAave(interest) would call Aave V3's IPool.supply with amount = 0, which would also reverts

Hence any call arriving while expectedInterest = 0 fails outright. That situation is not limited to a brand-new vault:
• Two users depositing in the same block see the second transaction revert (interest cannot accrue between calls).
• Two different users would not be able to deposit and withdraw in the same block.
• Any long-lived vault whose interest has just been swept and who's following deposits occur before another accrue cycle will fail likewise.

In effect, the contract is only usable in blocks where a positive, claimable interest balance already exists. Because the vault reverts any deposit or mint when there is no immediately claimable interest, it effectively renders the contract unusable whenever the interest buffer drops to zero. Integrations and front-ends that batch multiple operations in the same block, such as Zaps, will fail unpredictably. In practice, in active ReAToken vaults, users will encounter failed transactions even though the pool is fully funded and restakable.

Recommendation

Short-circuit _accrueInterest when expectedInterest = 0 to prevent the zero-interest path from invoking either claimInterest or _supplyAave. The most localized patch is to extend the early-return guard:

```
if (isRestakable || expectedInterest = 0 || expectedInterest > maxAssetsSuppliableToAave()) return;
```

With this change a fresh vault accepts the first deposit and interest will be claimed only once it has accrued to a positive value.

Resolution

246 Club Team: The issue was resolved in commit 32f9666.

H-03 | Missing UNDERLYING_ASSET Approval

Category	Severity	Location	Status
DoS	High	ReAToken.sol	Resolved

Description

The ReAToken vault's constructor approves the restaking asset (e.g., aUSDC) to the Aave V3 pool and the Diamond246 protocol for transfers during deposits and restakes, but fails to approve the underlying asset (e.g., USDC, retrieved via IAToken(_delegationPairAssets.asset).UNDERLYING_ASSET_ADDRESS()) to the Aave V3 pool, which is required for supplying claimed interest back to Aave in the compounding process, causing the supply operation to revert due to insufficient allowance and blocking the entire yield compounding loop after the first interest accrual.

This issue occurs because interest claimed from the 246 protocol is denominated in the debt asset (which matches the underlying asset for typical stablecoin pairs like aUSDC/USDC), and the _accrueInterest function in ReAToken attempts to compound it by calling _supplyAave(interest) (line: AAVE_V3_POOL.supply(UNDERLYING_ASSET, assets, address(this), 0);), which internally requires an ERC20 approval from the vault to the pool for the underlying asset to execute the supply, but no such approval is set in the constructor (only for the aToken itself).

As a result, once interest is claimed (via IInterestManagementFacet(DIAMOND246).claimInterest, transferring underlying tokens to the vault), the subsequent supply to Aave fails with an ERC20 "insufficient allowance" revert, halting compounding and all the underlying ReAToken deposit and withdrawal operations.

This vulnerability is triggered in normal operation after the first borrow-generated interest becomes claimable and it persists because the approval is only set once at deployment, with no mechanism to approve the underlying dynamically.

Recommendation

To resolve this approval gap, add an explicit approval for all the UNDERLYING_ASSET to the AAVE_V3_POOL in the ReAToken constructor using IERC20(UNDERLYING_ASSET).forceApprove(address(AAVE_V3_POOL), type(uint256).max);, ensuring the vault can supply claimed interest without reverts.

Resolution

246 Club Team: The issue was resolved in commit f49a780.

H-04 | claimInterest Tries To Transfer Yet-To-Be-Paid Interest

Category	Severity	Location	Status
Logical Error	High	ReAToken.sol	Resolved

Description

InterestManagementFacet.claimInterest pays restakers by executing:

```
SafeTransferLib.safeTransfer(delegationPairAssets.debt, receiver, interest);
```

but the function never first checks whether the Diamond actually holds the required interest amount of the debt-asset (e.g., USDC).

The contract's balance is only topped-up indirectly when borrowers or liquidators transfer debt tokens in the "repay to 246" code paths, or when the DAO manually calls supplyInterestReserve.

If a restaker (or, in practice, any ReAToken vault action) calls claimInterest before sufficient tokens have arrived, the safeTransfer reverts, bubbling all the way up to the ERC-4626 call that triggered it (e.g., deposit, withdraw).

Because every ReAToken operation begins with an internal _accrueInterest → claimInterest step, a single shortfall permanently blocks all user deposits and withdrawals for that vault until the debt-asset balance is replenished.

This leaves funds stranded, halts yield generation and exposes users to indefinite lock-up.

Recommendation

Consider tracking the amount of interest paid through repayments, liquidations or supplyInterestReserve calls. Cap the transfer to this available balance while carrying any remainder forward.

Resolution

246 Club Team: The issue was resolved in commit 12cb728.

M-01 | ReATokenFactory.createReAToken Can Be Front-run

Category	Severity	Location	Status
Unexpected Behavior	Medium	ReATokenFactory.sol	Resolved

Description

The ReATokenFactory deploys new vault tokens with CREATE2:

```
function createReAToken( DelegationPairAssets memory delegationPairAssets,
string memory name, string memory symbol, bytes32 salt)
external returns (IReAToken reAToken){reAToken = IReAToken(address(new ReAToken{ salt: salt
}(delegationPairAssets, DIAMOND246, AAVE_V3_PROVIDER, name, symbol))
);
isReAToken[address(reAToken)] = true;
emit CreateReAToken(
msg.sender, address(reAToken), delegationPairAssets.id(), delegationPairAssets, name, symbol, salt
);
}
```

Because the address is deterministic, hash(0xFF, factory, salt, keccak256(init_code)), anyone who sees the pending transaction in the mempool can relay a twin transaction that uses the exact same salt and constructor arguments while bidding a higher gas price. If their transaction lands first, the vault is deployed to the expected address, isReAToken[address(reAToken)] is set by the attacker's call and all later attempts with that salt revert with "contract already deployed."

A deployment script that queues follow-up operations (e.g. setWhitelist, stakeInitial or external approvals) in the same bundle will then fail mid-flight, leaving the system in an unexpected state. Although no assets are stolen, the disruption could be enough to halt releases, cause CI/CD pipelines to fail and force manual operator intervention.

Recommendation

Derive the salt on-chain from immutable data that the attacker cannot control, such as salt = keccak256(abi.encode(msg.sender, delegationPairAssets)). This makes an identical front-run impossible because the attacker can not become msg.sender for that salt derivation.

Resolution

246 Club Team: The issue was resolved in commit c4e33d6.

M-02 | Potential Inflation Attack In ReAToken

Category	Severity	Location	Status
Frontrunning	Medium	ReAToken.sol	Partially Resolved

Description PoC

The inflation attack vulnerability in the ReAToken vault enables a malicious actor to frontrun an honest user's deposit by making a minimal initial deposit followed by a large donation of restaking assets on behalf of the ReAToken vault directly to the Diamond246 protocol, artificially inflating the share price within the 246 restaking system and causing the victim's deposit to mint zero or near-zero vault shares, effectively donating their assets to the attacker who can then withdraw a portion of the inflated value.

Although the vault's totalAssets relies on expectedRestakeAmount from the ViewerFacet, which queries the 246 protocol's stored position rather than the vault's direct token balance, this does not fully mitigate the attack because the donation targets the 246 protocol itself (where the assets are restaked), manipulating the underlying scaled supply and power calculations that feed into the preview functions, leading to an overstated exchange rate during the victim's deposit.

To dissect this, recall that in a standard ERC4626 vault, the attack exploits rounding in share minting by donating to the vault contract to bloat totalAssets without increasing totalSupply. In ReAToken, totalAssets calls expectedRestakeAmount, which unscales the vault's scaledSupply in 246 (stored in restakers[delegationPairId][vaultAddress].scaledSupply), adds previewed interest and ignores the vault's actual balanceOf(asset). Deposits in ReAToken mint shares using this previewed rate, then restake via _supplyClub246 (calling RestakingFacet's restake), which transfers aTokens to Diamond246 and updates scaledSupply.

Instead of donating to ReAToken, the attacker donates to Diamond246 after a small restake, inflating 246's aToken balance. But restake only updates scaledSupply on explicit calls, direct transfers to Diamond increase its balanceOf(aToken) but not scaledSupply (as unscaling uses scaled, not balance). In the POC provided, the attacker (user1) begins by supplying 3e18 WETH to Aave to obtain aWETH, then deposits a minimal 1 wei-equivalent to the ReAToken vault, minting 1 share and establishing themselves as the initial shareholder.

Next, they approve 2e18 aWETH to the Diamond246 and call restake with 2e18-1 on behalf of the ReAToken vault itself, this transfers the large amount of aWETH directly to the Diamond246, which scales it (dividing by Aave's liquidity index) and adds it to the vault's scaledSupply in 246 (restaker.scaledSupply += scaledAmount.toUint128();), effectively "donating" the assets to inflate the vault's position in the protocol without minting additional ReAToken shares, as the restake call is external and bypasses the vault's minting logic. At this point, the vault's totalSupply remains 1 (from the attacker's tiny deposit), but the 246 protocol's scaledSupply for the vault has ballooned to approximately the scaled equivalent of 2e18, causing expectedRestakeAmount to preview around 2e18 assets.

When user2 (the victim) deposits 1e18 aWETH, the previewDeposit computes shares as (1e18 * 1) / 2e18 = 0.5, which floors to 0 due to integer division and downward rounding, resulting in zero shares minted, with the deposit effectively donated to the protocol (transferred to Diamond246 during the internal restake call). The logs confirm this: user2's ReAToken balance remains 0 after the deposit. The attacker then withdraws 1.5e18 from the vault, reclaiming a portion of their initial contribution plus part of the donated value.

The POC results show they end up with approximately 2.5e18-1 aWETH (from an initial 3e18 supplied to Aave), indicating a net loss of about 0.5e18 rather than a profit, the victim's 1e18 donation is captured by the protocol as inflated scaledSupply, but the attacker's single share only entitles them to a fraction of it upon redemption, with the remainder effectively trapped or diluted across the vault's future users. This makes the attack primarily a griefing mechanism: it denies user2 shares and usability without netting the attacker a gain, mostly to disrupt adoption, force users to over-deposit to overcome the inflation.

Recommendation

To eliminate this, enforce a minimum decimals offset in _decimalsOffset (e.g., +6) for precision.

Resolution

246 Club Team: The issue was resolved in commit c990d5d.

M-03 | maxRedeem/maxWithdraw Returns Inaccurate Values

Category	Severity	Location	Status
Warning	Medium	ReAToken.sol	Resolved

Description

The maxRedeem and maxWithdraw functions in the ReAToken vault provide previews of the maximum redeemable shares or withdrawable assets for a user, but these values are inaccurately inflated because they rely on the expectedRestakeAmount from the ViewerFacet, which includes previewed interest in its calculations even though this interest is not immediately available as it must be realized through future borrower repayments or liquidations before it can be claimed and compounded, leading to an overestimation of available liquidity and causing attempted operations to potentially return less than expected or fail during execution.

Specifically, maxWithdraw computes the user's pro-rata assets via _convertToAssets (inherited from ERC4626, using totalAssets which calls expectedRestakeAmount) and caps it against a simulated maxUnstakeAmount (or maxUnstakeAmountAfterRestake if conditions allow), where expectedRestakeAmount unscales the principal and adds simulated interest after previewing accrual and distribution, treating the interest as readily claimable despite it being contingent on actual inflows from repayments or liquidations.

In reality, interest exists only as virtual accruals in pool.pendingInterest and pair.totalDebtInterest and the Diamond246 contract's balance of the debt asset (from which claims transfer) may not yet hold the corresponding tokens if repayments lag behind accruals.

This discrepancy means the previews optimistically assume the interest is already funded and claimable, but when the actual claim occurs in _accrueInterest, if the protocol lacks the balance, the transfer in claimInterest would fail, misaligning with the preview.

The actual impact for users of the protocol is overestimated withdrawal limits that cannot be fully realized, resulting in failed or partial redemptions.

Recommendation

To ensure accurate previews, modify the ViewerFacet's expectedRestakeAmount to distinguish between realized (claimable) and unrealized (previewed) interest by adding a separate return value for each, and update ReAToken's maxRedeem and maxWithdraw to conservatively use only the realized portion plus safely unstakable principal, while always claiming available interest in _accrueInterest without assumptions of full realization.

Resolution

246 Club Team: The issue was resolved in commit 12cb728.

M-04 | Unprotected External Callback Enables Reentrancy

Category	Severity	Location	Status
Warning	Medium	RestakingFacet.sol	Acknowledged

Description

In `RestakingFacet.restake` the contract:
1. Updates all critical accounting (`totalScaledSupply`, `scaledSupply`, `interestPad`…).
2. Then calls an arbitrary hook controlled by `msg.sender`:

```
if (data.length > 0) I246RestakeCallback(msg.sender).on246Restake(amount, data);
```

Because this hook is invoked before the actual token transfer (`safeTransferFrom`) and without a reentrancy guard, a malicious contract can re-enter any publicly-accessible 246 function while the vault's accounting already reflects the new deposit but no tokens have been received.

An attacker can, for example, call `restake` a second time (inflating their weight) or attempt an `unstake` that relies on balances which are not yet backed by real aTokens. Although many secondary calls will revert when the subsequent `safeTransfer` finally fails, the pattern opens the door to future logic changes that might make re-entrancy profitable.

This behavior also enables free flash loans. Users can call `restake` with amount set as the whole balance of the contract. Their `restaker` struct will be populated with the desired amount, but before the tokens are transferred, the facet will give them the execution flow. The user can then call `unstake`, which will successfully decrease their balance and will send them all of the tokens held in the contract. The user can then do whatever action they wish with the funds before returning them.

Recommendation

Move the external callback after a successful `safeTransferFrom`, following the Checks-Effects-Interactions pattern:

```
// 1. EFFECTS
... accounting updates ...
// 2. INTERACTIONS – first pull the tokens
SafeTransferLib.safeTransferFrom(delegationPairAssets.asset, msg.sender, address(this), amount);
// 3. OPTIONAL CALLBACK
if (data.length = 0) {
I246RestakeCallback(msg.sender).on246Restake(amount, data);
}
```

and protect the function with a `nonReentrant` modifier to forbid nested calls.

Resolution

246 Club Team: Acknowledged. This callback pattern is intentional and it allows complex actions like flash loan or flash repay. Also, any reentrancy attempt would revert if the subsequent safeTransferFrom fails.

M-05 | accrueAndDistributeInterest Calls Could Revert

Category	Severity	Location	Status
DoS	Medium	InterestManagementFacet.sol, BaseFacet.sol	Partially Resolved

Description PoC

The 246 protocol's `_accrueInterest` function can revert with an arithmetic overflow error that can be triggered under conditions of prolonged bad debt accumulation or high pool utilization, causing subsequent calls to the accrual mechanism to revert permanently.

This effectively bricks core protocol operations such as borrowing, repaying, unstaking, liquidations and interest claims, as they rely on up-to-date interest calculations.

The overflow arises from unchecked multiplications in the Taylor series approximation used for compounding interest when pool utilization exceeds 100%, leading to exponential debt growth that overwhelms `uint256` limits within a finite number of accrual steps.

In the provided test script, this issue is demonstrated by simulating a borrowing scenario followed by 137 days of daily accruals without repayments or liquidations, allowing utilization to creep above 1 due to unaddressed interest, after which the debt explodes quadratically and triggers the revert on the next accrual attempt.

This revert blocks all future accruals, as the function cannot complete without overflowing and, therefore, leaves the protocol in a totally broken state requiring an upgrade. This scenario is very unlikely, as the position becomes liquidatable way earlier. However, it is something that can easily happen if the pool is paused as liquidations/repayments revert in this case:

```
function liquidate(
PairAssets memory pairAssets,
address borrower,
uint256 seizedAmount,
uint256 repaidShare,
bytes calldata data
) external returns (uint256, uint256) {
LiquidateVars memory vars;
Pool storage pool = _pools[pairAssets.debt];
if (borrower == address(0)) revert ErrorsLib.ZeroAddress();
if (!UtilsLib.exactlyOneZero(seizedAmount, repaidShare)) revert ErrorsLib.InvalidInput();
if (pool.paused) revert ErrorsLib.PoolPaused(); // <----------
...
}
```

Recommendation

To mitigate this vulnerability, introduce a hard cap on the annual borrow rate in the `_borrowRate` function of `BaseFacet`, limiting it to a reasonable maximum such as 5e18 wad (500% APR) after computation but before returning, which prevents the quadratic feedback loop by enforcing linear growth at extreme utilizations and ensures calculations remain within `uint256` bounds even under prolonged bad debt scenarios. Additionally, cap utilization inputs to `_borrowRate` at 2e18 wad (200%) to bound excess calculations.

Resolution

246 Club Team: The issue was resolved in commit 5a4797c.

M-06 | migrateDelegationAsset Is Permissionless

Category	Severity	Location	Status
Unexpected Behavior	Medium	EmergencyFacet.sol	Partially Resolved

Description

The `migrateDelegationAsset` function in `EmergencyFacet` is declared as external without any access modifiers or authorization checks, allowing any external caller to invoke it on any borrower's position.

This function migrates the delegation asset (a restaked `aToken` used as collateral in Aave V3) for a specified borrower's position by calculating the current borrowing power, selecting a new delegation asset via a pseudo-random search in `_findDelegationAssetAndAmount`, delegating the equivalent power from the new asset to the borrower's contract account and undelegating the old asset back to the protocol.

The lack of validation, such as checking if `msg.sender` is the borrower, an authorized party (e.g., via `isAuthorized[borrower][msg.sender]`), or a privileged role (e.g., guardian or dev), exposes all positions to interference.

Due to the pseudo-random selection an attacker can time or spam calls to influence selection towards assets with higher utilization ratios, manipulating borrow rates.

Recommendation

To mitigate this, add access controls to restrict calls to authorized parties only. Implement a modifier like `onlyGuardianOrDev` (from `ACLManagerStorage`) for emergency use, and/or add an ownership check such as `require(msg.sender = borrower || isAuthorized[borrower][msg.sender], "Unauthorized caller");`

Resolution

246 Club Team: The issue was resolved in commit 991c822.

M-07 | Wrong Interest Repayment Logic

Category	Severity	Location	Status
Logical Error	Medium	BorrowingFacet.sol	Acknowledged

Description

This report is about an issue that was found during the review of a file that's not part of the scope - BorrowingFacet.sol. The repay() function allows users to repay their debt and when they do it, Aave debt is favored. This means that until the Aave interest is not fully paid out, none of the repayments will go towards the club interest and all to Aave.

The problem is in the first if statement

```
if (vars.positionScaledDebt = 0) {
// repay to 246
pair.totalDebtInterest = pair.totalDebtInterest.zeroFloorSub(amount).toUint128();
if (data.length > 0) I246RepayCallback(msg.sender).on246Repay(amount, data);
SafeTransferLib.safeTransferFrom(pairAssets.debt, msg.sender, address(this), 3965002);
}
```

It fires when the user has previously paid their full Aave debt which would make their positionScaledDebt to 0. In the meantime, they will keep accruing Aave debt even though their position is closed because of the shares mechanism. This means that the new Aave debt will be split proportionally between the current user and all the other users.

Later, when a full repayment happens and the if is fired, positionScaledDebt will be 0, even though Aave debt was accrued. The whole amount then will be subtracted from the club debt. If the club debt is less than the current user's club debt + aave debt, the surplus will be still repaid by the user, but it won't be repaid to Aave.

Because of this, the Aave positions will never be fully repaid and they will continue accruing debt that cannot be cleared unless a liquidation at the Aave level happens. This will not stop user withdrawals since position.debtShares will be 0 and they will be considered healthy.

Recommendation

Consider reworking the repayment mechanism.

Resolution

246 Club Team: Acknowledged. The case you described where users with zero scaledDebt still pay interest based on the pair's Aave debt is intentional. One thing to note is that users borrow from us (246 Club), not directly from Aave. It is not considered users with zero scaledDebt socialize others Aave debt.

L-01 | Stale Liquidity Index Usage

Category	Severity	Location	Status
Warning	Low	ReAToken.sol	Resolved

Description

The `maxAssetsSuppliableToAave` function determines the remaining supply capacity for the underlying asset in Aave V3 by subtracting the current supply from the configured supply cap, but it relies on potentially outdated values from the `getReserveData` call, specifically the `liquidityIndex` and `accruedToTreasury` fields, which are not dynamically updated and may reflect stale state at the time of the last reserve interaction rather than the current block, resulting in an underestimation or overestimation of the actual supplyable amount that could lead to failed supplies when nearing the cap or unintended cap breaches if treasury accruals have increased since the last update.

Recommendation

To address this, replace the direct usage of `reserveData.liquidityIndex` and manual computation of current supply with a call to Aave's `getReserveNormalizedIncome` function on the pool contract to fetch the real-time normalized liquidity index, and adjust the current supply calculation accordingly to incorporate up-to-date accruals, ensuring the function always reflects the precise remaining capacity.

Resolution

246 Club Team: The issue was resolved in commit 43afad9.

L-02 | Missing DelegationPairAssets Validation

Category	Severity	Location	Status
Validation	Low	ReATokenFactory.sol	Resolved

Description

The `ReATokenFactory` contract permits the unrestricted creation of `ReAToken` `ERC4626` vaults for any `delegationPairAssets` comprising a restaking asset and debt asset without verifying whether the pair is properly configured, enabled or active within the underlying 246 protocol.

In these vaults, any call to deposit/mint will revert as `_supplyClub246` internal call will always fail if the `delegationPairAssets` are not supported.

Recommendation

To remediate this deployment validation deficiency, integrate a pre-creation check in `ReATokenFactory`'s `createReAToken` by calling `if (IViewerFacet(DIAMOND246).isRestakable(delegationPairAssets)) revert NotSupportedPair();` immediately after input validation.

Resolution

246 Club Team: The issue was resolved in commit 6be06e6.

I-01 | Pool Pausing In 246 Protocol

Category	Severity	Location	Status
DoS	● Info	ReAToken.sol, InterestManagementFacet.sol	Partially Resolved

Description

The pausing mechanism in the 246 protocol's pools, intended as an emergency control to halt risky operations like borrowing or restaking during incidents, inadvertently creates a denial-of-service vulnerability for integrated systems such as the ReAToken vault, where all deposits and withdrawals become blocked due to the inability to claim and compound accrued interest when a pool is paused.

This stems from a strict revert in the claimInterest function of the InterestManagementFacet, which checks the pool's paused state and prevents any interest transfers, freezing the vault's core compounding loop and causing cascading failures in user interactions that rely on up-to-date yield accrual.

Pool pausing is managed in ConfiguratorFacet's pausePool(address debtAsset, bool paused), which sets pool.paused = paused. This flag is a boolean toggle, controllable by guardians or devs (via modifiers like onlyGuardianOrDev), designed to revert mutative functions across facets (e.g., borrow, repay, restake, unstake, liquidate) to prevent further risk exposure during crises like oracle failures or exploits.

The critical point of failure occurs in InterestManagementFacet's claimInterest, which includes:

```
if (pool.paused) revert ErrorsLib.PoolPaused();
```

This check is executed before any accrual, distribution or transfer, meaning that whenever a pool is paused, no interest can be claimed at all. The function proceeds to accrue interest (_accrueInterest), distribute it (_distributePendingInterest), compute the restaker's claimable amount and transfer the debt asset (e.g., USDC) to the receiver, but the pause revert blocks the entire process.

Now, this becomes problematic in the ReAToken vault, which relies on claiming interest from 246 to compound yields as part of its ERC4626 operations.

• If restakable and within Aave supply cap conditions are met, it calls claimInterest to transfer the interest to the vault, supplies it to Aave to mint more aTokens, and restakes those to 246 for compounding.
• When the pool is paused, claimInterest reverts with PoolPaused, causing _accrueInterest to fail and propagate the revert to the calling deposit or withdrawal.
• For deposits: Users cannot add assets, as the compounding step fails, halting new liquidity inflows.
• For withdrawals: Users cannot redeem shares, as the preview and unstake rely on up-to-date accrual, trapping funds in the vault during pauses.

This creates a full DoS: In a paused pool, the ReAToken vault becomes non-functional for all users, as every mutative operation (deposit, mint, withdraw, redeem) triggers _accrueInterest, which will attempt a claim and revert. Which will also block the withdrawal as the if (pool.paused) revert ErrorsLib.PoolPaused(); is also present there.

Recommendation

Merely informative. Consider documenting this behaviour so users are aware of this behaviour.

Resolution

246 Club Team: The issue was resolved in commit 4a250b3.

I-02 | Redundant Import Statement

Category	Severity	Location	Status
Best Practices	Info	ReAToken.sol, BaseFacet.sol	Resolved

Description

The codebase contains several redundant import statements, where libraries or symbols are imported multiple times or in ways that duplicate functionality already available through other imports or definitions, such as the double import of the `Math` library in `ReAToken.sol` and the potential overlap of `WAD` constant imported from `MathLib.sol` in `BaseFacet.sol` despite its definition in `ConstantsLib.sol`, which could lead to unnecessary compilation overhead and reduced code clarity without affecting runtime behavior.

These redundancies do not introduce runtime vulnerabilities but increase compilation time slightly and make the code harder to maintain by obscuring dependencies.

Recommendation

Consider removing the redundant imports.

Resolution

246 Club Team: The issue was resolved in commit 812f12a.

I-03 | Missing Global Reentrancy Guard

Category	Severity	Location	Status
Best Practices	Info	Global	Acknowledged

Description

The protocol introduces significant reentrancy risks by incorporating optional callbacks in several key functions, which are triggered when non-empty data is provided and executed via external calls to the `msg.sender` before completing critical interactions such as token transfers.

This design violates the Checks-Effects-Interactions pattern, where external calls should ideally occur last to prevent malicious contracts from reentering the system and exploiting transient states.

Specifically, the callbacks flagged include the `on246Restake` in `RestakingFacet.restake` (called after updating scaled supplies and interest pads but before transferring `aTokens` to the diamond), the `on246SupplyCollateral` in `CollateralManagementFacet.supplyCollateral` (invoked after increasing position collateral but prior to transferring collateral to the account), the `on246Repay` in `BorrowingFacet.repay` (executed following debt reduction and delegation adjustments yet before debt token transfers), and the `on246Liquidate` in `LiquidationFacet.liquidate` (triggered after adjusting debt and collateral but preceding debt token transfers).

These callbacks, absent any reentrancy guards, allow attackers to reenter functions like `unstake`, `borrow`, or `claimInterest` during the callback, manipulating phantom supplies or health factors before the original transaction's token movements finalize.

Recommendation

To mitigate these risks, implement a dedicated reentrancy guard facet using a `nonReentrant` modifier pattern, such as that provided by `OpenZeppelin`, applied globally to all functions invoking callbacks or handling external calls, ensuring that state modifications and interactions are fully resolved before any callback execution.

Resolution

246 Club Team: Acknowledged. This callback pattern is intentional and it allows complex actions like flash loan or flash repay. Also, any reentrancy attempt would revert if the subsequent safeTransferFrom fails.

I-04 | Pool Utilization Increases After Some Operations

Category	Severity	Location	Status
Warning	Info	Global	Partially Resolved

Description PoC

The 246 protocol allows certain administrative and emergency operations that inadvertently increase pool utilization by reducing the effective borrowing power without proportionally decreasing debt, which can push utilization above the optimal ratio, inflating borrow rates.

This occurs because borrowing power calculations shift from total scaled supply to total scaled usage when restaking is disabled and coverage operations further decrease usage without impacting debt, both of which shrink the denominator in the utilization formula of `totalDebt` divided by `totalPower`, leading to higher utilization.

In the provided test script, we can see how the utilization increases from 6.26% to 6.46% once `enableRestaking(delegationPairAssets, false)` is called. Finally, we can also see how the utilization raises again when `coverDelegationAsset` is called, from 6.46% to 6.91%.

Recommendation

Merely informative. Be aware that certain operations, and not exclusively `borrow` calls, do also increases the utilization and therefore the borrowing rates.

Resolution

246 Club Team: The issue was resolved in commit bbed4e0.

I-05 | Lack Of Event Emissions In Setter Functions

Category	Severity	Location	Status
Best Practices	Info	Global	Acknowledged

Description

Throughout the codebase, several setter functions modify critical global state variables (e.g., configuration parameters, ratios and addresses) without emitting corresponding events.

This omission hinders off-chain monitoring, auditing and user notifications, as external observers (e.g., frontends, indexers, or security tools) rely on events to track changes efficiently.

For instance, while some setters like `setConnector` emit `SetConnector`, others such as internal updates in `BaseFacet` for authorization mappings or nonce increments, lack events, making it difficult to detect unauthorized or unexpected modifications.

In high-stakes DeFi protocols, this can obscure governance actions, delay incident response and complicate historical state reconstruction.

Recommendation

Ensure all setter functions that alter mutable state emit descriptive events, including old and new values where applicable, to enhance transparency and enable reliable off-chain tracking. Adopt a consistent pattern across facets, such as using a library like `EventsLib` for all emissions.

Resolution

246 Club Team: Acknowledged. Could you provide us more specific case where we are lack of? We think authorization, nonce and others are emit events to track its state.

I-06 | Club246 Interest Impacts Available Power

Category	Severity	Location	Status
Informational	Info	ViewerFacet.sol, RestakingFacet.sol	Acknowledged

Description

In `RestakingFacet.unstake()`, the following check ensures the pool's borrowing power won't fall below its current debt after the restake is executed: `pool.totalDebt > totalPower - unstakeBorrowingPower`.

Both `totalPower` and `unstakeBorrowing` power represent the Aave borrowing power for the particular asset. On the other hand, `totalDebt` includes not only the interest accrued by Aave, but by the club as well.

Because of that, the calculations will result in less available amount for unstaking even if it's not directly backing the Aave position. The same is true in `ViewerFacet._calculateSafeUnstakeAmount()`.

Recommendation

Acknowledge this finding and reconsider if the available power calculation should be changed.

Resolution

246 Club Team: Acknowledged. Removing this validation in unstake() would cause the following borrow() calls to revert, since the same validation (pool.totalDebt > vars.totalPower) still exists in borrow().

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits