GA GUARDIAN

246 Club

Core Review

Security Assessment

November 7th, 2025

Summary

Audit Firm Guardian

Prepared By Robert Regeida, Zdravko Hristov, Michael Lett

Client Firm 246 Club

Final Report Date November 7, 2025

Audit Summary

246 Club engaged Guardian to review the security of their 246 Club Core and Re-A-Token. From the 2nd of September to the 15th of September, a team of 3 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.

✅ 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 Information

Project Overview 5

Audit Scope & Methodology 6

Smart Contract Risk Assessment

Findings & Resolutions 9

Addendum

Disclaimer 47

About Guardian 48

Project Overview

Project Summary

Project Name	246 Club
Language	Solidity
Codebase	https://github.com/246club/246-core, https://github.com/246club/re-a-token
Commit(s)	Main Review commit: 8ddde79919492f79b80932a88ae62c3daf79303a Remediation Review commit: dfc41c8b3eaf3a11053c7eb7f0f9dc2e36d82520

Audit Summary

Delivery Date	November 7, 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	1	0	0	0	0	1
Medium	4	0	0	2	1	1
Low	9	0	0	9	0	0
Info	20	0	0	16	3	1

Audit Scope & Methodology

Scope and details:
contract,source,total,comment
246-core/src/ACLManagerStorage.sol,41,85,31
246-core/src/Account.sol,56,104,24
246-core/src/Diamond246.sol,37,65,19
246-core/src/Storage.sol,32,76,25
246-core/src/facets/ACLManagerFacet.sol,44,80,21
246-core/src/facets/AuthorizationFacet.sol,30,50,7
246-core/src/facets/BaseFacet.sol,246,462,140
246-core/src/facets/BorrowingFacet.sol,130,189,21
246-core/src/facets/CollateralManagementFacet.sol,52,89,14
246-core/src/facets/ConfiguratorFacet.sol,255,411,51
246-core/src/facets/DiamondCutFacet.sol,9,22,11
246-core/src/facets/DiamondLoupeFacet.sol,32,59,20
246-core/src/facets/EmergencyFacet.sol,108,161,20
246-core/src/facets/InitializerFacet.sol,18,30,5
246-core/src/facets/InterestManagementFacet.sol,58,92,8
246-core/src/facets/LiquidationFacet.sol,133,216,44
246-core/src/facets/OwnershipFacet.sol,15,29,8
246-core/src/facets/RestakingFacet.sol,84,121,9
246-core/src/facets/RewardManagementFacet.sol,60,91,8
246-core/src/facets/ViewerFacet.sol,290,438,51
246-core/src/connectors/AaveV3Connector.sol,34,70,17
246-core/src/libraries/AaveV3Lib.sol,74,211,107
246-core/src/libraries/BorrowingPowerLib.sol,41,74,24
246-core/src/libraries/ConnectorCallLib.sol,20,33,9
246-core/src/libraries/ConstantsLib.sol,10,27,9
246-core/src/libraries/DataTypesLib.sol,77,103,68
246-core/src/libraries/DiamondLib.sol,197,245,28
246-core/src/libraries/ErrorsLib.sol,42,55,40
246-core/src/libraries/EventsLib.sol,108,459,298
246-core/src/libraries/MathLib.sol,30,53,13
246-core/src/libraries/PairLib.sol,15,27,8
246-core/src/libraries/SharesMathLib.sol,19,40,13
246-core/src/libraries/UtilsLib.sol,30,47,10
246-core/src/libraries/ValidationLib.sol,38,88,38
246-core/src/dependencies/aave/v3.5/TokenMath.sol,26,93,58
246-core/src/dependencies/aave/v3.5/WadRayMath.sol,83,157,53
246-core/src/dependencies/aave/v3.2/DataTypes.sol,231,323,67
246-core/src/dependencies/aave/v3.2/Errors.sol,96,103,99
246-core/src/dependencies/aave/v3.2/IPool.sol,56,191,119
246-core/src/dependencies/aave/v3.2/ReserveConfiguration.sol,207,489,243
source count: {total: 5758, source: 3164, comment: 1858, single: 689, block: 1169, mixed: 207, empty: 943, todo: 0, blockEmpty: 0, commentToSourceRatio: 0.5872313527180784}

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	Interest Can Be Griefed	Rounding	High	Resolved
M-01	Excess Club Interest Repayment Can't Be Utilized	Logical Error	Medium	Acknowledged
M-02	Increasing Delegation From Disabled Restaking	Validation	Medium	Partially Resolved
M-03	coverDelegationAsset Can Orphan Debt	Logical Error	Medium	Acknowledged
M-04	Club Repayments/Liquidations Can Be Blocked	DoS	Medium	Resolved
L-01	Users Should Be Able To Skip Nonces	Signatures	Low	Acknowledged
L-02	0 Cap Permanently Applies baseBorrowRate2	Configuration	Low	Acknowledged
L-03	Some Positions Cannot Be Closed	Rounding	Low	Acknowledged
L-04	Missing Global Reentrancy Guard	Best Practices	Low	Acknowledged
L-05	Refunds Are Sent To feeRecipient	Logical Error	Low	Acknowledged
L-06	Missing Bad Debt Resolution	Logical Error	Low	Acknowledged
L-07	Repay Can Leave Residual Debt Dust	Rounding	Low	Acknowledged
L-08	Aave Repayments Desync Internal Scaled Debt	Logical Error	Low	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
L-09	Disabling An In-use Restaking Asset	Logical Error	Low	Acknowledged
I-01	Rounding Mismatch	Rounding	Info	Acknowledged
I-02	Pair Cap Hard-cap Placeholders	Configuration	Info	Acknowledged
I-03	Ownership Is Transferred At Once	Best Practices	Info	Acknowledged
I-04	EOAs Can Be Set As Facets	Validation	Info	Acknowledged
I-05	DiamondLib Loops Are Not Optimized	Gas Optimization	Info	Acknowledged
I-06	initializeDiamondCut() Can Invoke Non-facets	Validation	Info	Acknowledged
I-07	Initializing Multiple Facets Is Not Supported	Configuration	Info	Acknowledged
I-08	Potential Division By Zero In _borrowRate	Validation	Info	Resolved
I-09	Missing Explicit Collateral Balance Check	Validation	Info	Acknowledged
I-10	Potential DOS Due To Unbounded Accrual Loops	DoS	Info	Partially Resolved
I-11	Trapped Collateral In Paused Pool If Liquidated	Informational	Info	Partially Resolved
I-12	Utilization Can Exceed 100%	Informational	Info	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
I-13	convertToRedeem Doesn't Adjust For Max Amount	Unexpected Behavior	Info	Acknowledged
I-14	Reserve Flags And Isolation Mode Are Ignored	Validation	Info	Acknowledged
I-15	Aave Liquidations Cause Restaker Losses	Unexpected Behavior	Info	Partially Resolved

H-01 | Interest Can Be Griefed

Category	Severity	Location	Status
Rounding	High	BaseFacet.sol: 144-145	Resolved

Description PoC

The interest mechanism is divided in 2 main steps - accrual and distribution. The accrual part is done by BaseFacet._accrueInterest(). It computes the generated interest from the last accrual and adds it towards the pool.pendingInterest.

The second step, handled by BaseFacet.__distributePendingInterest(), distributes this pendingInterest to all active pairs for that pool. After that pendingInterest is reset to 0. The distribution stores the accumulated interest per scaled amount since the beginning of the pair in the accInterestPerScaledAmount variable.

```
uint256 weight = power.wDivDown(totalPower);
restaking.accInterestPerScaledAmount +
pool.pendingInterest.wMulDown(weight).wDivDown(restaking.totalScaledSupply).toUint128();
```

This calculation is prone to precision loss. The first part - pool.pendingInterest.wMulDown(weight) - will be in the debt token decimals because weight is in WAD and wMulDown is used. The result is then divided by restaking.totalScaledSupply which is in pair.asset decimals.

Imagine the scenario where we have the following pair:
• asset: WETH
• debt: USDC

Then, if the first part is 10 and the second part is 11e18, the end result will be 10 * 1e18 / 11e18 which is 0.

In result, the interest for that period was accrued, but not stored in the accumulator variable accInterestPerScaledAmount. Therefore, that interest is forever lost and cannot be claimed. An attacker can execute InterestManagementFacet.accrueAndDistributeInterest() every block to cause interest griefing.

Recommendation

Consider using a higher precision for the accInterestPerScaledAmount

Resolution

246 Club Team: The issue was resolved in commit 7dcd262.

M-01 | Excess Club Interest Repayment Can't Be Utilized

Category	Severity	Location	Status
Logical Error	Medium	BorrowingFacet.sol: 151C1-158C10	Acknowledged

Description

The first branch in `BorrowingFacet.repay()` handles the case when the user repaying has already fully repaid their Aave debt.

```
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), amount);
}
```

The `_pairTotalDebt()` functions reports the total debt accrued by a given pair

```
return amount + pair.totalDebtInterest;
```

Here `amount` is the Aave debt and `pair.totalDebtInterest` is the Club246 interest.

Because of the share mechanism, users that hold shares will bear a part of the accruing Aave debt of the pair, even if they don't have any active positions. When they later call `repay()`, the execution flow will enter the `if` statement from above and the debt value will be repaid as club interest.

When the rest of the users repay their debts, they will pay the full Aave debt because of their scaled amounts, but they will pay less club interest, because a part of it was paid by the previous user.

However, if during repayment `amount > pair.totalDebtInterest`, the excess amount will be transferred in the contract, but because `zeroFloorSub()` is used, this amount won't be subtracted from `_pairTotalDebt()`. In result, the same Aave debt has to be paid twice - users will be overcharged, but the excess amount won't be utilized.

Recommendation

Consider transferring only the needed amount from the user in that `if` block instead of the whole `amount`.

Resolution

246 Club Team: Acknowledged.

M-02 | Increasing Delegation From Disabled Restaking

Category	Severity	Location	Status
Validation	Medium	EmergencyFacet.sol: 39-54	Partially Resolved

Description PoC

EmergencyFacet.adjustDelegation() allows managing active delegations when the aDebt balance of the given account changes.

When the debt has increased and there is a need for more power delegated, the code doesn't check if the current restaking is enabled. In result users will be able to increase the delegation provided by a disabled restaking.

This will increase the restaking's totalScaledUsage, which will lead to unexpected increase in that restaking's _power() and further influence the interest distribution.

The comment in the following snippet from RestakingFacet.unstake() explains how thescaledAmount > restaking.totalScaledSupply - restaking.totalScaledUsage check is enough to prevent users from withdrawing a potential delegation power for disabled restaking.

This is because it's expected the totalScaledUsage won't be increasing, which turns out to be a

```
if (scaledAmount > restaking.totalScaledSupply - restaking.totalScaledUsage) {
revert ErrorsLib.InsufficientLiquidity();
}
{
uint256 unstakeBorrowingPower =
AAVE_V3_PROVIDER.borrowingPower(delegationPairAssets, amount, restaking.creditBufferRatio);
// if restaking is disabled, it will be caught above.
```

Recommendation

Consider not allowing adjustDelegation() to increase the current delegation if the restaking is disabled.

Resolution

246 Club Team: The issue was resolved in commit b4f7295.

M-03 | coverDelegationAsset Can Orphan Debt

Category	Severity	Location	Status
Logical Error	Medium	LiquidationFacet.sol; BorrowingFacet.sol	Acknowledged

Description

EmergencyFacet.coverDelegationAsset shrinks a delegation's restaking usage but never checks whether the pool still has outstanding debt. The function currently does:

```
(uint256 totalPower, uint256[] memory powers) = _totalPower(delegationPairAssets.debt);
_accrueInterest(pool, delegationPairAssets.debt, totalPower);
_distributePendingInterest(pool, delegationPairAssets.debt, totalPower, powers);
uint128 scaledAmount = AAVE_V3_PROVIDER.getATokenTransferScaledAmount(underlyingAsset,
amount).toUint128();
restaking.totalScaledUsage =
restaking.totalScaledUsage.zeroFloorSub(scaledAmount).toUint128();
```

If this call targets the last restaking asset backing a debt pool, restaking.totalScaledUsage can become zero while pool.totalDebt still records an outstanding loan.

Subsequent _totalPower calls return 0, so _accrueInterest immediately exits with interest: 0, leaving the pool ledger stale even though viewerFacet.pairTotalDebt shows that debt remains.

Rates and utilization are now computed against a debt figure that no longer grows and accounting/invariants break (GLOB-02 & GLOB-06 invariants were triggered in the fuzzing suite).

Recommendation

Before subtracting usage, compute the new value and revert if it would drop to zero while debt remains. For example:

```
uint128 newUsage = restaking.totalScaledUsage.zeroFloorSub(scaledAmount).toUint128();
if (newUsage = 0) {
require(viewerFacet.pairTotalDebt(pairAssets) = 0, "delegation still backs debt");
}
restaking.totalScaledUsage = newUsage;
```

Resolution

246 Club Team: Acknowledged.

M-04 | Club Repayments/Liquidations Can Be Blocked

Category	Severity	Location	Status
DoS	Medium	BaseFacet.sol: 163-165	Resolved

Description

The repayment flow is vulnerable to the following DOS vector:
• Alice has a position with 1000 Aave debt
• She wants to repay a part of her position, so she calls BorrowingFacet.repay() to repay 600 debt
• An attacker frontruns her transaction with a direct repayment to Aave of 400.
• Alice's transaction is executed and the following two calls are performed in the else branch

The connectorCall will repay her debt and the aDebtToken balance of her account will become 0. Then _adjustDelegation will set position.delegationAsset = address(0).

Even though the Aave debt has been repaid directly, Alice position still holds position.debtShares which have to be repaid.

The next time she calls BorrowingFacet.repay(), the same else branch runs and _adjustDelegate will try fetching the balance of the delegation asset.

However, because delegationAsset is 0 due to the previous repayment, the transaction will revert and Alice won't be able to exit the system.

She has to borrow new amount from the club in order to change her delegation asset back to a real token and repay afterwards. However, this may not be possible in some cases - if there is not enough liquidity, the borrow cap is hit, etc. Even if borrowing is possible, the victim may be liquidated by the attacker before that if the health the position allows it.

Another possible path for exploiting it is if an Aave liquidation happened - then anyone can repay a minimal amount to the club to reset the delegationAsset.

Even worse, the LiquidationFacet logic is similar to the repayment flow, which means liquidations can be blocked as well causing an ever increasing bad debt. And because the delegation asset is 0, EmergencyFacet.migrateDelegationAsset() cannot be called.

Recommendation

```
function _adjustDelegation(Position storage position, address debtAsset) internal {
address asset = position.delegationAsset;
+       if (asset = address(0)) return;
}
```

Resolution

246 Club Team: The issue was resolved in commit b94e4fe.

L-01 | Users Should Be Able To Skip Nonces

Category	Severity	Location	Status
Signatures	Low	AuthorizationFacet.sol: 28-29	Acknowledged

Description

AuthorizationFacet.setAuthorizationWithSig() allows the execution of ordered authorization transactions signed by the authorizer. For a transaction to be successful, the signed nonce must match the current onchain record for the user nonce, which increases by 1 each time a successful authorization is executed.

There is no way for the user to skip nonces, which has two implications:
• If a signature with nonce N expires, all signatures with nonces N + 1 are blocked
• the authorizer doesn't have a valid way to invalidate an already signed signature

Technically, both issues are currently solvable if the users acts properly. For the first one, they can sign a new message with the same nonce and execute it to unblock the queue, but that may not be practical.

For the second they can execute all the prior transactions until the faulty one, then sign a new empty approval with the nonce of the approval they want to cancel and execute it. This, again, is not practical because the user has to pay for the execution of all these transactions and sign a message with the same nonce.

Recommendation

Consider adding the following function to AuthorizationFacet so users can safely skip nonces.

```
function updateNonce(uint256 newNonce) external {
if (newNonce < nonce[msg.sender]) {
revert ErrorsLib.InvalidNonce();
}
nonce[msg.sender] = newNonce;
emit EventsLib.IncrementNonce(msg.sender, msg.sender, newNonce - 1);
}
```

Resolution

246 Club Team: Acknowledged.

L-02 | 0 Cap Permanently Applies baseBorrowRate2

Category	Severity	Location	Status
Configuration	Low	BaseFacet.sol	Acknowledged

Description

In `BaseFacet._borrowRate`, the base rate adds the "buffer" component whenever `pairTotalDebt > cap`:

```
// if the borrow cap exceeds, it does mean the buffer allocation is on and the
baseBorrowRate2 is applied
uint256 baseBorrowRate = pairTotalDebt > pairParams.cap
: pairParams.baseBorrowRate1 + pairParams.baseBorrowRate2
: pairParams.baseBorrowRate1;
```

When `cap = 0` (commonly used to mean "no cap"), any non-zero `pairTotalDebt` makes `pairTotalDebt > cap` true, so the buffer component (`baseBorrowRate2`) is applied permanently from the first wei of debt onward.

This is inconsistent with the rest of the codebase, which treats "cap enabled" as `cap > 0`. For example, in `BorrowingFacet` the cap check is gated by `if (pairParams.cap > 0 ...)`, meaning a cap of zero disables cap logic.

Recommendation

Align `_borrowRate` with the "cap enabled" semantics used elsewhere. Only apply `baseBorrowRate2` if the cap is enabled and exceeded.

Resolution

246 Club Team: Acknowledged.

L-03 | Some Positions Cannot Be Closed

Category	Severity	Location	Status
Rounding	Low	BorrowingFacet.sol; LiquidationFacet.sol	Acknowledged

Description

During `repay()` and `liquidate()`, the borrowing account of the position will call `Aave.repay()` to repay a given amount of assets.

Looking at the `repay()` function, that amount is being converted to `scaledAmount`

```
(noMoreDebt, reserveCache.nextScaledVariableDebt) = IVariableDebtToken(
reserveCache.variableDebtTokenAddress
).burn({
from: params.onBehalfOf,
scaledAmount: paybackAmount.getVTokenBurnScaledAmount(reserveCache.nextVariableBorrowIndex),
index: reserveCache.nextVariableBorrowIndex
});
```

If the `scaledAmount` is `0`, the variable debt token will revert. Because of this, very small repayments will be reverting, potentially blocking `repay()` and `liquidate()`.

Since there is no minimum amount enforced for the positions in the club, it's possible to hold a position with a small debt value and minimum collateral to support it. As time passes, the position will accrue debt, but if it's still rounding down to 0, liquidations will be impossible.

By the time the debt becomes a meaningful value, it may already have surpassed the collateral, which makes the liquidation revert forever as described in `H-01`.

Recommendation

Consider implementing checks for `borrow()` or `repay()` is called that don't allow a position to be left with a debt that rounds down to 0 scaled amount.

Resolution

246 Club Team: Acknowledged.

L-04 | Missing Global Reentrancy Guard

Category	Severity	Location	Status
Best Practices	Low	Global	Acknowledged

Description

Several user-facing flows mutate protocol accounting and then hand control to attacker-controlled callbacks before their state transitions close. For example, `BorrowingFacet.repay` reduces `position.debtShares`, `pair.totalDebtShares` and `pool.totalDebt`, then invokes `I246RepayCallback(msg.sender).on246Repay(amount, data)` while the pool still expects to pull amount from the repayer.

`LiquidationFacet.liquidate` similarly updates borrower debt, collateral and pool totals before calling `I246LiquidateCallback`. Collateral entry points such as `supplyCollateral` credit the user's position and immediately execute optional callbacks and the emergency delegation helpers adjust restaking weights before finishing. None of these functions share a reentrancy latch, so a malicious borrower or liquidator can re-enter other facets while the first invocation is mid-flight.

Today, each path still demands the owed token transfer or the transaction reverts, and the authorized-account checks plus Aave connector usage keep obvious drains at bay: the attacker can't exit with extra assets as long as those post-callback transfers succeed.

However, the windows leave state momentarily inconsistent (e.g., debts already decremented, restaking usage updated) and rely on every path continuing to completion. A future refactor that inserts a new external call could become instantly exploitable.

Recommendation

Introduce a diamond-wide non-reentrancy guard. A simple pattern is a `NonReentrantFacet` that stores a single lock flag and exposes `_nonReentrantBefore`/`_nonReentrantAfter`.

Wrap every external function that performs callbacks or touches user funds (borrow, repay, liquidate, `supplyCollateral`, emergency delegation helpers, etc.) so the guard is held across the entire mutation, ensuring no facet can be re-entered until the original call fully settles.

Resolution

246 Club Team: Acknowledged.

L-05 | Refunds Are Sent To feeRecipient

Category	Severity	Location	Status
Logical Error	Low	LiquidationFacet.sol; BorrowingFacet.sol	Acknowledged

Description PoC

When repaying debt through the Aave connector, any amount supplied in excess of the actual variable debt is refunded by the connector to a receiver address supplied by the caller. The connector's logic is:

```
function repay(address token, uint256 amount, address receiver) external {
uint256 debtAmount = IERC20(PROVIDER.debtTokenAddr(token)).balanceOf(address(this));
if (amount > debtAmount) {
SafeTransferLib.safeTransfer(token, receiver, amount - debtAmount);
amount = debtAmount;
}
if (amount = 0) return;
IPool pool = IPool(PROVIDER.getPool());
SafeTransferLib.safeApprove(token, address(pool), amount);
pool.repay(token, amount, INTEREST_RATE_MODE, address(this));
}
```

The problem is at both call sites (user re-pay and Aave-branch liquidation) the "receiver" argument is set to the protocol's feeRecipient, not the actual payer who provided the funds.

As a result, whenever a payer over-supplies relative to the outstanding variable debt the connector returns the excess to feeRecipient instead of the payer. This silently diverts user funds to protocol fees.

Recommendation

Use the payer address as the refund receiver for overpayments, not feeRecipient. In user repay, pass msg.sender. In liquidation, pass the liquidator (also msg.sender).

Resolution

246 Club Team: Acknowledged.

L-06 | Missing Bad Debt Resolution

Category	Severity	Location	Status
Logical Error	Low	LiquidationFacet.sol	Acknowledged

Description PoC

The liquidation flow cannot fully resolve positions once their collateral is exhausted and the 246Club protocol lacks any write-off or backstop mechanism to absorb the remainder. In LiquidationFacet.liquidate, liquidation proceeds in two mutually exclusive modes. In the repaidShare path, the contract derives the required seizedAmount and enforces the close factor.

If the requested repayment exceeds what is allowed by close factor it reverts and if seizedAmount exceeds available collateral, subtracting from position.collateral underflows and reverts:

```
if (uint256(position.debtShares).wMulDown(vars.closeFactor) < repaidShare) {
revert ErrorsLib.ExceedMaxLiquidatableDebt();
}
position.collateral - seizedAmount; // reverts if seizedAmount > collateral
```

In the seizedAmount path, the caller proposes how much collateral to seize, then the contract computes the corresponding repaidShare and applies the same close-factor bound. If seizedAmount exceeds available collateral the subtraction also reverts. After a series of partial liquidations, once position.collateral reaches zero, both paths become impossible: any positive repayment implies seizing strictly positive collateral, which can no longer be subtracted from zero, so liquidation reverts and cannot progress further.

The remaining position.debtShares/position.scaledDebt persist on the position. Interest continues to accrue at the pool level via BaseFacet._accrueInterest as long as pool.totalDebt > 0 and totalPower > 0, increasing utilization and pending interest. On the restaking side, the viewer logic explicitly prevents unstaking when the pool is overutilized, for example, ViewerFacet._calculateSafeUnstakeAmount immediately returns zero when totalDebt > totalPower.

As a result, residual unsecured debt can permanently peg utilization, cause interest to accrue on uncollectible balances and block restakers from exiting. Although EmergencyFacet exposes supplyInterestReserve, there is no path that consumes this reserve to repay or extinguish insolvent positions. Only BorrowingFacet.repay can reduce debt, which provides no incentive once collateral is gone.

Because of this, insolvent accounts become permanent “zombies” that keep accruing interest and can drive utilization above available power leading to stuck restakers.

Recommendation

Consider introducing a controlled function that can zero a position whose collateral is already 0 by paying its remaining debt.

Resolution

246 Club Team: Acknowledged.

L-07 | Repay Can Leave Residual Debt Dust

Category	Severity	Location	Status
Rounding	Low	BorrowingFacet.sol; LiquidationFacet.sol	Acknowledged

Description PoC

In the amount path of BorrowingFacet.repay, the repayment amount is converted to shares using floor rounding against virtual supply constants, then those shares are burned while pool.totalDebt is decremented by the raw user amount. This guarantees a residual remainder of dust debt in position.debtShares and prevents a position from being fully closed via the amount-based API.

Relevant code in repay:
• If the caller supplies an amount, the code derives a share count using toSharesDown:

```
if (amount > 0) share = amount.toSharesDown(_pairTotalDebt(pair, pairAssets), pair.totalDebtShares);
```

• Then it burns those shares and reduces pool debt by the raw amount:

```
position.debtShares - share.toUint128();
pool.totalDebt = pool.totalDebt.zeroFloorSub(amount).toUint128();
```

SharesMathLib.toSharesDown is:

```
return amount.mulDivDown(totalShares + VIRTUAL_SHARES, totalAmount + VIRTUAL_AMOUNT);
```

Both VIRTUAL_SHARES and VIRTUAL_AMOUNT are non-zero. This makes the conversion intentionally conservative. For any nominal “full repayment” amount A that a user computes in amount space, the computed share = floor(A · (T+VS)/(D+VA)) is biased downward versus the exact proportional value. As a result, even when A equals the user’s entire borrow converted via amounts, share will typically be strictly less than the user’s outstanding position.debtShares.

The function then:
• Burns fewer shares than the position holds (leaving position.debtShares > 0).
• Decreases pool.totalDebt by the full amount provided by the user (not by the effective amount that those burned shares represent under the share model).

This leaves a non-zero residual of position.debtShares (and corresponding dust debt in the pair) that continues to accrue interest. The user sees that their “full” amount repayment did not actually close the position. The dust remains until they explicitly repay by shares. Over time, the dust accrues interest, can continue to affect health checks and can block full collateral withdrawal until a share-based cleanup is performed.

The option of increasing the amount passed as parameter to the repay function is not valid as that would cause a revert by underflow here:

```
position.debtShares - share.toUint128();
```

Recommendation

Provide an explicit “repay all” path that clears the position by shares, or make the amount path upgrade itself to a share-close when the supplied amount is sufficient.

Two viable patterns:
• Add a convenience that callers can use to deterministically zero debt:
• previewRepayAllAmount(pairAssets, onBehalf): returns toAmountUp(position.debtShares, pairTotalDebt, totalDebtShares).
• repayAll(pairAssets, onBehalf): internally routes to repay(pairAssets, 0, position.debtShares, onBehalf, "").
• In repay(amount), after computing share with toSharesDown, check if amount > toAmountUp(position.debtShares, pairTotalDebt, totalDebtShares). If true, set share = position.debtShares and proceed on the share-based path, ensuring the position is completely cleared without residual debt.

Resolution

246 Club Team: Acknowledged .

L-08 | Aave Repayments Desync Internal Scaled Debt

Category	Severity	Location	Status
Logical Error	Low	BorrowingFacet.sol	Acknowledged

Description PoC

The protocol's debt accounting assumes that all reductions of Aave variable debt occur through BorrowingFacet.repay/LiquidationFacet.liquidate. If a third party (or the borrower) repays directly on Aave to the position's Account, the actual Aave vDebt goes down but the protocol's internal scaled debt stays unchanged:

• Per-pair debt is derived from the internal pair.totalScaledDebt (plus pair.totalDebtInterest).
• Interest accrual and utilization use that value and utilization also uses pool.totalDebt, which only changes inside protocol flows.

A direct Aave repay lowers the real Aave vDebt, but does not decrease position.scaledDebt, pair.totalScaledDebt, or pool.totalDebt. Until the next in-protocol touch.
• The protocol over-estimates debt and over-accrues interest/fees (higher borrowUsageRatio, higher borrowRate, larger interestAccrual).
• Users may be blocked from borrowing earlier than warranted (artificially high utilization).

The desynchronization creates a second, more acute issue when someone later repays through the protocol. BorrowingFacet.repay computes amount/share against _pairTotalDebt(...) (which reads the stale pair.totalScaledDebt) and then sends that full amount to the Account and instructs the connector to repay Aave

The Aave connector then measures the actual Aave debt and refunds any excess to the receiver passed in the calldata which is hard-coded as feeRecipient.

Because the protocol over-estimated _pairTotalDebt, it frequently sends more than Aave will accept. The resulting over-payment is refunded to feeRecipient instead of the payer. The same pattern is present in LiquidationFacet.liquidate, which also passes feeRecipient as the refund receiver when constructing the connector calldata.

The impact of this issue is that the incorrect debt/interest/utilization accounting persists until the position is fully closed. Also, subsequent protocol-level repays/liquidations will over-charge users and divert the surplus to feeRecipient.

Recommendation

Consider implementing a public/admin "sync" function to realign internal scaled debt with Aave for a position/pair in case off-protocol repayments occur and consider calling it inside emergency/maintenance flows. Any refund on any over-repay should be sent to the payer, not feeRecipient.

Resolution

246 Club Team: Acknowledged.

L-09 | Disabling An In-use Restaking Asset

Category	Severity	Location	Status
Logical Error	Low	Global	Acknowledged

Description

In `ConfiguratorFacet.enableRestaking` the diamond allows a restaking market to be toggled off even when borrowers still delegate it. Disabling sets `restaking.enabled = false` after the usual accounting (`_accrueInterest` / `_distributePendingInterest`).

Once enabled is `false`, `_power` in the `BaseFacet` no longer values the asset by its total restaked supply but instead uses only the delegated amount (already haircut by `creditBufferRatio`). `_totalPower` then reports a sharply reduced pool power while `Pool.totalDebt` remains unchanged.

During multiple fuzz runs we observed utilization `totalDebt * 1e18 / totalPower` exceed `1.05e18`, violating the protocol's requirement that utilization stay at or below 105%.

The root cause is that the contract permits disabling a restaking asset that is still underwriting outstanding debt, so the backing power suddenly collapses while the debt stays outstanding.

Recommendation

Consider performing this step, and right after, if possible atomically, migrate borrowers to another enabled restaking asset.

Resolution

246 Club Team: Acknowledged.

I-01 | Rounding Mismatch

Category	Severity	Location	Status
Rounding	Info	BorrowingFacet.sol; BaseFacet.sol	Acknowledged

Description

The pool-level debt cache (`pool.totalDebt`) is updated using caller-facing “amounts”, while per-pair debt is tracked in Aave’s scaled units and converted via the variable index.

This mixed basis causes `pool.totalDebt` to temporarily diverge from the true sum of pairs and in `repay`/`liquidation` branches it can be driven to zero while residual pair debt remains.

When that happens, the next accrual step treats the pool as having no debt and advances time without accruing interest, undercharging borrowers and underpaying restakers/fees for the elapsed interval. Pair debt is computed with Aave’s index and includes the protocol interest pot.

The `repay`/`liquidate` paths reduce the pool cache by the full user input (principal + interest portion), regardless of what Aave actually burns under its rounding rules.

Because `amount` is not guaranteed to equal the Aave principal delta (due to scaled rounding), the pool cache can undershoot the sum of pairs immediately after a `repay`/`liquidation`.

A test showed that after liquidating the only position of a pair, `pool.totalDebt = 0` while `pairTotalDebt > 0` and Aave `vDebt > 0`. The next `_accrueInterest` call sees zero usage, advances `lastUpdated` and accrues 0 for the whole interval despite outstanding pair debt.

Health checks and share math are unaffected because they use pair-level values. The liquidity guard if (`pool.totalDebt > totalPower`) becomes slightly more lenient during the drift window.

Recommendation

Consider updating `pool.totalDebt` using the exact Aave principal delta, not the caller “amount”.

Resolution

246 Club Team: Acknowledged.

I-02 | Pair Cap Hard-cap Placeholders

Category	Severity	Location	Status
Configuration	Info	Re7WethWeth.s.sol; GauntletWethCoreWeth.s.sol; FalconUsdcGho.s.sol	Acknowledged

Description

Multiple production-targeted pair deployment/update scripts configure pairs with a placeholder hard cap of 1 base unit and keep them enabled while `baseBorrowRate2` is zero. This configuration makes the market unborrowable and causes borrows to revert once total pair debt exceeds 1 unit of the debt token.

Examples:
- `script/deploy/chain/mainnet/pair/Re7WethWeth.s.sol`
- `script/deploy/chain/mainnet/pair/GauntletWethCoreWeth.s.sol`
- `script/deploy/chain/mainnet/pair/FalconUsdcGho.s.sol`

Batch update scripts broadcast every .sol in the directory with no allowlisting or sanity gates:
- `script/deploy/updateAllPairs.sh`

On-chain cap enforcement in `BorrowingFacet.borrow`: `vars.pairTotalDebt` includes the newly-added scaled debt and interest. With `cap=1` and `baseBorrowRate2=0`, any borrow that takes the pair over 1 base unit reverts.

Because cap is denominated in debt token units, cap=1 means 1 wei for 18-decimals (e.g., WETH) or 1 “micro-unit” for 6-decimals (e.g., USDC), which is effectively zero capacity.

The scripts simultaneously set `enabled = true`, presenting these markets as active while they cannot be borrowed.

Therefore, if these placeholder configs are ever pushed to production via `updateAllPairs.sh`, they will freeze borrowing on affected markets (all borrow attempts revert), despite the pair being enabled.

Recommendation

Consider revisiting the config values of the listed pairs above.

Resolution

246 Club Team: Acknowledged.

I-03 | Ownership Is Transferred At Once

Category	Severity	Location	Status
Best Practices	Info	OwnershipFacet.sol: 21	Acknowledged

Description

`OwnershipFacet.transferOwnership()` changes the owner of the Diamond in 1 step. It's typically advised to have this feature executed in 2 steps - the first sets the new owner as a pending owner and the second allows the pending owner to accept the ownership.

This way the risk of giving the ownership to an invalid or a malicious address is decreased.

Recommendation

Consider implementing the 2 step ownership transfer

Resolution

246 Club Team: Acknowledged.

I-04 | EOAs Can Be Set As Facets

Category	Severity	Location	Status
Validation	Info	DiamondLib.sol: 251	Acknowledged

Description

Before adding facet via `DiamondLib.addFacet()` or calling it by `DiamondLib.enforceHasContractCode()`, the `enforceHasContractCode()` function will revert if the target's `codesize` is 0.

Since `EIP7702` EOAs can set a delegate contract which will be executed once their address is called. EOAs that use this feature have `codesize = 23`.

This will satisfy the performed check and allow an EOA to be added as a facet or called for initialization.

Recommendation

Modify the check in`enforceHasContractCode()` and revert if the first three bytes of the code of the target are `0xef0100` - these are reserved for `EIP7702`.

Resolution

246 Club Team: Acknowledged.

I-05 | DiamondLib Loops Are Not Optimized

Category	Severity	Location	Status
Gas Optimization	Info	DiamondLib.sol: 121,148	Acknowledged

Description

The for loops in `DiamondLib.addFunctions()` and `DiamondLib.replaceFunctions()` update the `selectorIndex` in the loop body.

Because of this, the Solidity compiler will not optimize them by applying `unchecked` math when they are incremented.

Recommendation

To benefit from the implicit gas optimization, move the `++selectorIndex` to the iteration part of the loop.

```
•       for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;) {
+       for (uint256 selectorIndex; selectorIndex < _functionSelectors.length;
++selectorIndex) {
...
•           ++selectorIndex;
}
```

Resolution

246 Club Team: Acknowledged.

I-06 | initializeDiamondCut() Can Invoke Non-facets

Category	Severity	Location	Status
Validation	Info	DiamondLib.sol: 96C9-96C29	Acknowledged

Description

The call to `initializeDiamondCut()` in `DiamondLib.diamondCut()` will execute the desired calldata on the `init` address. It's not guaranteed that this address is a valid facet, it can be any EVM address.

While this may be desirable since it allows having different contracts, separate from the facets, which handle the initialization, it can also lead to unexpected behaviors if a wrong address is provided.

Recommendation

Either acknowledge and be careful with the addresses provided, or enforce that `init` is a facet in the code.

Resolution

246 Club Team: Acknowledged.

I-07 | Initializing Multiple Facets Is Not Supported

Category	Severity	Location	Status
Configuration	Info	DiamondLib.sol: 96C9-96C29	Acknowledged

Description

When `DiamondLib.diamondCut()` is called, each cut can handle different facets, but the `initializeDiamondCut()` calls only 1 of them.

The rest of the facets will have to be manually initialized which increases the risk of an adversary taking control over them.

Recommendation

Consider reworking the `initializeDiamondCut()` function so it supports calls to multiple facets.

Resolution

246 Club Team: Acknowledged.

I-08 | Potential Division By Zero In _borrowRate

Category	Severity	Location	Status
Validation	Info	BaseFacet.sol	Resolved

Description

In BaseFacet._borrowRate, the above-optimal branch computes the slope-2 contribution by dividing by (WAD - optimalUsageRatio):

```
if (borrowUsageRatio < optimalUsageRatio) {
// Calculate rate using slope1
rate = baseBorrowRate + borrowUsageRatio.mulDivDown(pairParams.slope1, optimalUsageRatio);
} else {
// Calculate rate using slope2
uint256 excessUsage = borrowUsageRatio - optimalUsageRatio;
rate =
baseBorrowRate + pairParams.slope1 + excessUsage.mulDivDown(pairParams.slope2, WAD - optimalUsageRatio);
}
```

If the pool is configured with optimalUsageRatio = WAD (i.e., 100%), the denominator of the above-optimal branch becomes zero.

While borrows are gated by pool.totalDebt <= totalPower, the utilization used in _borrowRate comes from borrowUsageRatio = pool.totalDebt.wDivDown(totalPower), where totalPower is recomputed on the fly and can drop below pool.totalDebt without a new borrow (e.g., after disabling a restaking asset, migration/usage shifts or other supply/usage changes).

In those situations, borrowUsageRatio can exceed WAD, the code takes the above-optimal path and division by (WAD - optimalUsageRatio) with optimalUsageRatio = WAD triggers a division-by-zero revert.

Because _borrowRate is called inside _accrueInterest, this revert DoS-es interest accrual (and any flow that accrues interest), blocking protocol operations until the configuration is corrected.

Recommendation

Disallow optimalUsageRatio = WAD at configuration time and assert the invariant defensively in _borrowRate. In ConfiguratorFacet.setPoolOptimalUsageRatio add the following require check: require(newOptimalUsageRatio < WAD, ErrorsLib.InvalidInputRatio());

Resolution

246 Club Team: The issue was resolved in commit 8cc59ba.

I-09 | Missing Explicit Collateral Balance Check

Category	Severity	Location	Status
Validation	Info	CollateralManagementFacet.sol	Acknowledged

Description

In `CollateralManagementFacet.withdrawCollateral`, the function subtracts the requested amount from `position.collateral` without first checking that the position has enough collateral:

```
position.collateral - amount;
```

If `amount > position.collateral`, Solidity 0.8's checked arithmetic triggers a generic underflow "panic" revert before your custom error `ErrorsLib.InsufficientCollateral()` is reached.

This yields a poor DX (ambiguous revert reason) and makes it harder to diagnose whether the failure is from "not enough collateral balance" vs. failing the post-withdraw health check.

Recommendation

Add an explicit balance check before the subtraction and revert with a descriptive custom error. Consider a new, precise error (e.g., `InsufficientCollateralBalance`) to distinguish from the risk-based `_isHealthy` failure (`InsufficientCollateral`).

For example:

```
if (position.collateral < amount) revert ErrorsLib.InsufficientCollateral();
position.collateral - amount;
```

Resolution

246 Club Team: Acknowledged.

I-10 | Potential DOS Due To Unbounded Accrual Loops

Category	Severity	Location	Status
DoS	Info	BaseFacet.sol	Partially Resolved

Description

The protocol couples interest accrual and power computation to many state-changing flows and performs unbounded iteration over pool data structures, creating a gas-exhaustion vector that can revert borrow, repay, liquidation, restake/unstake and configuration operations as the pool grows.

In `BaseFacet._accrueInterest` the function iterates over every pair in the pool to compute per-pair interest and update accounting.

This is `O(P)` with P = number of pairs in the pool. The caller always passes a fresh `totalPower` that is computed via `BaseFacet._totalPower`, which iterates over all restaking assets in the pool.

This is `O(A)` with A = number of restaked assets and `_power` includes external reads to Aave indices and further arithmetic per asset.

These two loops run on hot paths: `BorrowingFacet.borrow`/`repay`, `LiquidationFacet.liquidate`, `RestakingFacet.restake`/`unstake`, and several Configurator/Emergency functions call `_accrueInterest` (and often `_totalPower`) before proceeding.

There are no on-chain bounds on `pool.pairIdList` or `pool.assetList` sizes. As they grow, the gas cost scales roughly linearly and can exceed block gas limits, making essential operations revert out-of-gas.

Recommendation

Introduce hard upper bounds on the number of pairs and restaked assets per pool at the configurator level.

Resolution

246 Club Team: The issue was resolved in commit ede7de2.

I-11 | Trapped Collateral In Paused Pool If Liquidated

Category	Severity	Location	Status
Informational	Info	CollateralManagementFacet.sol	Partially Resolved

Description

`CollateralManagementFacet.supplyCollateral()` allows supplying collateral to a paused pool. This lets the users improve the health of their positions.

```
// do not check if the pool or pair are enabled or not
// user can make a position healthy by supplying more collateral even though the pool or pair are disabled
```

If a `liquidation` call happens before the `supply`, the collateral will be successfully added, but any attempts to withdraw it will be failing since the pool is paused.

This will trap the collateral in the contract until the pool is unpaused, even if that collateral doesn't back any debt (if full liquidation happened).

Recommendation

Inform the users about this behavior. They can use the `on246SupplyCollateral()` to check if they were liquidated.

Resolution

246 Club Team: The issue was resolved in commit 1e33b82.

I-12 | Utilization Can Exceed 100%

Category	Severity	Location	Status
Informational	Info	Global	Acknowledged

Description

The club utilization for a given pool is `totalDebt / totalPower`, where `totalPower` is the maximum borrowable amount from Aave for the available restaking amount, and `totalDebt` includes both the Aave debt and the club interest.

Because of this, it's possible to have `totalDebt > totalPower`, especially when the assets borrowed from Aave are close to the total power.

Recommendation

Acknowledge the issue and configure the pools appropriately to incentivize repayments when `utilization > 100%`

Resolution

246 Club Team: Acknowledged.

I-13 | convertToRedeem Doesn't Adjust For Max Amount

Category	Severity	Location	Status
Unexpected Behavior	Info	PendleReATokenAdapter.sol: 45	Acknowledged

Description

`PendleReATokenAdapter.convertToRedeem()` withdraws assets from Aave and returns how many were received by calling `previewConvertATokenToUnderlying()`. That function returns the unchanged amount being withdrawn.

However, Aave supports max withdrawals. Users can pass `type(uint256).max` and it will be treated as their full balance.

Because this is not accounted for in `previewConvertATokenToUnderlying()`, the returned result will be `2^256 - 1`.

Recommendation

There is no need to call `previewConvertATokenToUnderlying()` at all. You can `AAVE_V3_POOL.withdraw()` returns the withdrawn amount, use that instead.

Resolution

246 Club Team: Acknowledged.

I-14 | Reserve Flags And Isolation Mode Are Ignored

Category	Severity	Location	Status
Validation	Info	AaveV3Lib.sol	Acknowledged

Description

The helper `AaveV3Lib.isRestakable` function currently returns true solely when the reserve's LTV is greater than zero. This is too permissive. It ignores critical reserve state flags such as active/frozen/paused and isolation/silo constraints exposed by Aave's `ReserveConfiguration`.

A reserve can have `LTV > 0` while being frozen or paused (e.g., during risk events or deprecations), or while being in isolation mode that imposes significant restrictions.

In these cases, treating the asset as "restakable" allows creation of restaking pairs and acceptance of user restakes that may later stall or revert during delegation, borrowing or migrations.

The restaking and migration flows rely on `isRestakable` to greenlight an asset, so approving assets that are paused/frozen or under isolation can lead to operational inconsistencies. For example, the asset can:

• Be included in `assetList` and contribute power/weight for interest distribution and capacity even while the reserve is paused/frozen.

• Allow restakes to proceed while later delegation/borrowing paths fail or are blocked by Aave reserve constraints.

• Require emergency cleanups or migration when the reserve can not be used, stranding user `aTokens` temporarily and degrading UX.

Recommendation

Update the `isRestakable` function to require both `LTV > 0` and healthy reserve flags and explicitly reject isolation-mode assets if unsupported in your protocol.

Resolution

246 Club Team: Acknowledged.

I-15 | Aave Liquidations Cause Restaker Losses

Category	Severity	Location	Status
Unexpected Behavior	Info	Global	Partially Resolved

Description

Typically, the club liquidation parameters will be configured in a way that club liquidation happen before Aave liquidations.

However, if the price of a delegation asset drops and results in unhealthy position at Aave, this will not be reflected at the club level because there the health is determined only by their collateral and the debt.

It's expected that `EmergencyFacet.adjustDelegation()` will be called frequently to not allow Aave liquidations, but there is no guarantee this can always happen.

Whenever such liquidation happens, the restaker will have lost all their assets and the borrower can just repay their current debt to withdraw their collateral, resulting in a loss for the restakers.

Furthermore, if the protocol don't call `EmergencyFacet.coverDelegationAsset()`, the `restaking.totalScaledSupply` and `restaking.totalScaledUsage` values will be left inflated.

Recommendation

Ensure the restakers are aware of this risk and they perform frequent calls to `adjustDelegation()`.

Resolution

246 Club Team: The issue was resolved in commit 57978f9.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
I-01	Diamond Missing IERC165 Flag	Best Practices	Info	Acknowledged
I-02	Connector Can Break On Non-Standard Approvals	Best Practices	Info	Acknowledged
I-03	Diamond Accepts Unrecoverable ETH	Best Practices	Info	Acknowledged
I-04	ERC20 May Not Support Decimals()	Best Practices	Info	Acknowledged
I-05	ACC_INTEREST_PRECISION Can Be Improved	Best Practices	Info	Acknowledged

I-01 | Diamond Missing IERC165 Flag

Category	Severity	Location	Status
Best Practices	Info	DiamondLoupeFacet.sol	Acknowledged

Description

DiamondLoupeFacet.supportsInterface simply returns ds.supportedInterfaces[_interfaceId], yet no constructor, initializer, or facet ever writes to that mapping in DiamondLib.DiamondStorage.

As deployed, the diamond therefore reports false for type(IERC165).interfaceId (0x01ffc9a7) and every other interface, violating ERC-165 and breaking external contracts that relies on standard introspection.

Contracts that expects ERC-165 compliance will treat the diamond as non-standard.

Recommendation

During the initial diamond cut or initializer, explicitly set the IERC165 bit, for example:

```
DiamondLib.diamondStorage().supportedInterfaces[type(IERC165).interfaceId] = true;
```

and register any additional interface IDs that the diamond intends to advertise.

Resolution

246 Club Team: Acknowledged.

I-02 | Connector Can Break On Non-Standard Approvals

Category	Severity	Location	Status
Best Practices	Info	AaveV3Connector.sol	Acknowledged

Description

`AaveV3Connector.repay` grants allowance to the Aave pool via `SafeTransferLib.safeApprove(token, address(pool), amount)`.

Solady's `safeApprove` just forwards the direct `approve(amount)` call without first clearing the previous allowance. For tokens that enforce the zero-first approval pattern (e.g. USDT), any attempt to change an existing non-zero allowance to another non-zero value reverts.

After the first repayment, if the entire allowance was not consumed, the next approval attempt can revert, blocking borrowers from repaying and exposing them to liquidation risk.

```
function repay(address token, uint256 amount, address receiver) external {
uint256 debtAmount = IERC20(PROVIDER.debtTokenAddr(token)).balanceOf(address(this));
if (amount > debtAmount) {
SafeTransferLib.safeTransfer(token, receiver, amount - debtAmount);
amount = debtAmount;
}
if (amount = 0) return;
IPool pool = IPool(PROVIDER.getPool());
SafeTransferLib.safeApprove(token, address(pool), amount);
pool.repay(token, amount, INTEREST_RATE_MODE, address(this));
}
```

Recommendation

Adopt an approval flow that supports zero first semantics by switching to `SafeTransferLib.safeApproveWithRetry`.

Resolution

246 Club Team: Acknowledged.

I-03 | Diamond Accepts Unrecoverable ETH

Category	Severity	Location	Status
Best Practices	Info	Diamond.sol	Acknowledged

Description

`Diamond246` leaves both `fallback` and `receive()` payable, so anyone can push plain ETH into the proxy. The current facet set exposes no function to sweep native assets, meaning any ETH sent this way is stuck until governance cuts in a new facet.

```
fallback() external payable {
DiamondLib.DiamondStorage storage ds;
bytes32 position = DiamondLib.DIAMOND_STORAGE_POSITION;
assembly {
ds.slot = position
}
address facet = ds.selectorToFacetAndPosition[msg.sig].facetAddress;
if (facet = address(0)) {
revert DiamondLib.FunctionDoesNotExist();
}
assembly {
calldatacopy(0, 0, calldatasize())
let result := delegatecall(gas(), facet, 0, calldatasize(), 0, 0)
returndatacopy(0, 0, returndatasize())
switch result
case 0 { revert(0, returndatasize()) }
default { return(0, returndatasize()) }
}
}
receive() external payable {}
```

Recommendation

Disallow raw ETH by making the entry points non-payable, automatically wrap inbound ETH into WETH if that user flow is intentional, or add an admin-only sweep in a facet so governance can recover accidental deposits.

Resolution

246 Club Team: Acknowledged.

I-04 | ERC20 May Not Support Decimals()

Category	Severity	Location	Status
Best Practices	Info	AccInterestLib.sol: 44	Acknowledged

Description

`AccInterestLib.isPrecisionRequired()` calls `decimals()` on the `asset` and `debt` tokens unconditionally, but the function is not part of the `ERC20` standard, so it's optional. If working with such tokens, the call to `decimals()` will cause a revert.

Recommendation

Acknowledge if such tokens are not going to be supported, otherwise wrap the call in a `try/catch` and use default decimals if it fails.

Resolution

246 Club Team: Acknowledged.

I-05 | ACC_INTEREST_PRECISION Can Be Improved

Category	Severity	Location	Status
Best Practices	Info	AccInterestLib.sol	Acknowledged

Description

The calculation for `interestPerScaledAmount` is

```
(pendingInterest.wMulDown(weight) * ACC_INTEREST_PRECISION).wDivDown(totalScaledSupply)
```

Or

```
pendingInterest * weight / 1e18 * ACC_INTEREST_PRECISION * 1e18 / totalScaledSupply
```

There is a division before multiplication resulting in a precision loss. The expression is first divided by `1e18` and then multiplied by the same number, which means they would be cancelled out. The value of the expression is equivalent to the following

```
pendingInterest * weight * ACC_INTEREST_PRECISION / totalScaledSupply
```

The same is true for the case without precision.

```
pendingInterest.wMulDown(weight).wDivDown(totalScaledSupply);
```

Can be rewritten to

```
pendingInterest * weight / totalSupply
```

Also note that it's possible for a precision loss to happen for tokens with the same decimals as well if the amount and weight are small. For example `amount = 5` and `weight = 0.1e17`. This would also round down to 0.

If you want to further prevent this issue, you can either:
• skip resetting the `pool.pendingInterest` in `_distributePendingInterest` to 0 if NONE of the pairs accrued interest
• or recalculate the added `interestPad` after each pair accrual and subtract it from `pool.pendingInterest` instead of resetting it to 0.

Recommendation

Consider whether each of the proposed optimization is desirable and implement them if needed.

Resolution

246 Club Team: Acknowledged.

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