GA GUARDIAN

Bounce

Leveraged Token Protocol Review

Security Assessment

September 23rd, 2025

Summary

Audit Firm Guardian

Prepared By Wafflemakr, Ayoub Benterki, Kirkeelee, Ali Shehab

Client Firm Bounce

Final Report Date September 23, 2025

Audit Summary

Bounce engaged Guardian to review the security of their Bounce Leveraged Token Protocol. From the 1st of September to the 18th of September, a team of 4 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Confidence Ranking

Given the lack of critical issues detected and minimal code changes following the main review, Guardian assigns a Confidence Ranking of 4 to the protocol. Guardian advises the protocol to consider periodic review with future changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

🔗 Blockchain network: **HyperEVM**

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

📊 PoC test suite: https://github.com/GuardianOrg/bounce-contracts-team1-1756500079996

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	Bounce
Language	Solidity
Codebase	https://github.com/bounce-tech/bounce-contracts
Commit(s)	Initial commit: 2bfeb6138e7081463a1210629df628ca8e4b5d30 Final commit: e9c3cbdd097bab8858e0cce65384613bf194c17a

Audit Summary

Delivery Date	September 23, 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	3	0	0	0	0	3
Medium	9	0	0	1	0	8
Low	13	0	0	6	0	7
Info	10	0	0	2	0	8

Audit Scope & Methodology

Scope and details:

contract,source,total,comment
bounce-contracts/src/Airdrop.sol,44,56,1
bounce-contracts/src/Bounce.sol,17,22,1
bounce-contracts/src/Factory.sol,82,97,1
bounce-contracts/src/GlobalStorage.sol,159,187,1
bounce-contracts/src/HyperliquidHandler.sol,38,48,1
bounce-contracts/src/LeveragedToken.sol,278,324,3
bounce-contracts/src/LeveragedTokenProxy.sol,12,18,1
bounce-contracts/src/Locker.sol,119,142,1
bounce-contracts/src/Ownable.sol,13,18,1
bounce-contracts/src/Referrals.sol,89,111,1
bounce-contracts/src/Staker.sol,179,208,1
bounce-contracts/src/Vesting.sol,113,141,1
bounce-contracts/src/utils/ScaledNumber.sol,29,37,1
bounce-contracts/src/constants/Addresses.sol,4,6,1
bounce-contracts/src/constants/Config.sol,28,34,2
source count: {
  total: 1449,
  source: 1204,
  comment: 18,
  single: 18,
  block: 0,
  mixed: 1,
  empty: 228,
  todo: 0,
  blockEmpty: 0,
  commentToSourceRatio: 0.014950166112956811
}

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	Incorrect Spot Index Used	Logical Error	High	Resolved
H-02	Perp Losses Are Absorbed	Unexpected Behavior	High	Resolved
H-03	Step Wise Jump After Bridging Funds	Logical Error	High	Resolved
M-01	Fee Accrual Can Cause Mint And Redeem	DoS	Medium	Resolved
M-02	Final Payout May Be Below User's Min Amount	Unexpected Behavior	Medium	Acknowledged
M-03	redeployLt Can Redeploy An Unintended Token	Unexpected Behavior	Medium	Resolved
M-04	DoS In LeveragedToken	DoS	Medium	Resolved
M-05	Inability To Bridge In	Logical Error	Medium	Resolved
L-01	Unfair Lock Duration	Best Practices	Low	Acknowledged
L-02	Claim And Recover Overlap	Validation	Low	Resolved
L-03	Checkpoint Order Causes Wrong Reverts	Error	Low	Resolved
L-04	Missing Validation For Perp Owner Update	Validation	Low	Resolved
L-05	Unnecessary Approval	Best Practices	Low	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
L-06	Superfluous Max Redemption Check	Superfluous Code	Low	Resolved
L-07	Precision Loss In Reward Calculation	Rounding	Low	Acknowledged
L-08	Streaming Fee Avoided Due To Rounding	Rounding	Low	Resolved
L-09	Potential Rate Volatility Bridging To Core	Gaming	Low	Acknowledged
L-10	No Vesting Token Recovery Mechanism	Logical Error	Low	Resolved
I-01	Underwater Accounts Can Cause Partial DoS	DoS	Info	Resolved
I-02	Incorrect Revert Message	Error	Info	Resolved
I-03	Users Could Spam Prepare Redeem Events	Best Practices	Info	Acknowledged
I-04	Unused Imports And Errors	Best Practices	Info	Resolved
I-05	Min Lock Never Set During Deployment	Configuration	Info	Resolved
I-06	Missing Validation On Vesting Duration	Validation	Info	Resolved
I-07	Mutable Base Asset Address	Configuration	Info	Resolved
I-08	Some Functions Are Missing Event Emits	Best Practices	Info	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
I-09	Immutable Variables Can Be Cached	Gas Optimization	Info	Resolved

H-01 | Incorrect Spot Index Used

Category	Severity	Location	Status
Logical Error	High	HyperliquidHandler.sol: 32	Resolved

Description PoC

The `HyperliquidHandler` uses the `PrecompileLib` to calculate spot and perps values, including the `spotAssetValue` used by the `LeveragedToken` to calculate the `baseAsset` value.

However, this function calls `PrecompileLib.normalizedSpotPx` with the `tokenId_` (token index) instead of the actual `spotIndex`.

Consequently, every call to `spotAssetValue` will revert, causing a major break on the `LeveragedToken` functionality.

This is due to the fact that for the USDT asset the token index is 268 but there are only 219 spot pairs when fetching price.

Recommendation

Calculate the spot index using the `PrecompileLib.getSpotIndex` with the base asset address. Then, use this index in the `normalizedSpotPx` call.

Resolution

Bounce Team: The issue was resolved in PR#62.

H-02 | Perp Losses Are Absorbed

Category	Severity	Location	Status
Unexpected Behavior	High	LeveragedToken.sol: 153-158	Resolved

Description PoC

When users redeem LT for base assets, small redemptions can use the atomic `redeem`. Otherwise they use the two-step `prepareRedeem` → `executeRedeem`. In `prepareRedeem`, the contract precomputes the user's payout in base units at the current rate and credits it:

```
uint256 baseAmount_ = ltToBaseAmount(ltAmount_);
// ... snip ...
_burn(msg.sender, ltAmount_);
_addCredit(msg.sender, baseAmount_);
```

Later, `executeRedeem` transfers the credited base amount if available. Because the LT's value depends on perp positions on Core, losses that occur after `prepareRedeem` are not applied to users who already prepared: their credited base is fixed.

Those losses are effectively shifted to users who redeem or prepare after the drawdown (or to remaining holders), creating unfair loss socialization and potential solvency pressure if many users queue before a loss.

Recommendation

Consider refactoring the redemption flow that snapshots LT and base amounts at `prepareRedeem`, then finalizes them at `executeRedeem` based on user inputs (`redeemIndex`, `ltAmount`).

The payout is capped to the lower of the prep exchange rate and the current exchange rate, ensuring losses are reflected without exceeding the initial snapshot. Global debt is treated as a computed field (`min(debtBaseAmount, ltToBaseAmount(ltToRedeem))`) to stay consistent with protocol losses.

Resolution

Bounce Team: The issue was resolved in PR#72.

H-03 | Step Wise Jump After Bridging Funds

Category	Severity	Location	Status
Logical Error	High	LeveragedToken.sol: 220	Resolved

Description PoC

When LTs are minted, base assets are deposited into the contract, increasing the `LeveragedToken.totalValue` by the token balance scaled to 18 decimals.

On the other side, when these assets are bridged out, the `LeveragedToken.totalValue` will rely on the spot value of the bridged USDT0 assets to Core, multiplied by the spot USDT/USDC price.

Due to the fact that this spot price is not exactly 1:1, it creates a stepwise jump in the total value and therefore, the exchange rate of the LT.

Even though the spot USDT/USDC price is usually close to $1.00, there are days when volatility kicks in, moving the price +/- 2 %.

However, the `redemptionFee` is set to 0.3%, opening up the possibility of extracting funds by minting and immediately redeeming from the LT. Keep in mind this step wise jump also applies when bridging from Core to EVM.

Recommendation

Consider converting USDT (base asset) amounts into USDC, including those held by the EVM contract, preventing step wise jumps in the exchange rate when bridging assets.

The base amount param during minting should also need to be adjusted in terms of the spot price, so all calculations are in USDC terms.

Resolution

Bounce Team: The issue was resolved in PR#71.

M-01 | Fee Accrual Can Cause Mint And Redeem

Category	Severity	Location	Status
DoS	Medium	LeveragedToken.sol: 255-269	Resolved

Description

The current implementation of `\_checkpoint()` computes the streaming fee owed since the last checkpoint and immediately attempts to pay the full amount by calling `\_payStreamingFee()`. This requires the contract to have a sufficient `baseAssetBalance()`.

If the owed fee exceeds the available balance, the fee transfer reverts, which in turn causes all user-facing functions that invoke `\_checkpoint()` (such as `mint()`, `redeem()`, and `prepareRedeem()`) to revert.

As a result, normal user operations may become blocked when liquidity is low, effectively bricking the contract until additional funds are bridged in.

Recommendation

Instead of requiring immediate full payment, track fees as an accrued liability in storage.

Resolution

Bounce Team: The issue was resolved in PR#58.

M-02 | Final Payout May Be Below User's Min Amount

Category	Severity	Location	Status
Unexpected Behavior	Medium	LeveragedToken.sol: 154	Acknowledged

Description PoC

When a user calls the `prepareRedeem` function, they specify the minimum amount they wish to receive via the `minBaseAmount_` parameter.

However, the protocol performs its slippage check on the amount before fees are deducted, allowing the transaction to pass even if the final payout is insufficient.

For example, a user might request a minimum of $1,000, but because the redemption fee is calculated later in the `executeRedeem` function, they may only receive $950. As a result, the user receives less than they were expecting/specified.

Recommendation

Consider deducting the fees within the `prepareRedeem` function so that the user's minimum amount is checked against the final value after fees have been deducted. This also matches the logic done in the redeem function.

Resolution

Bounce Team: Acknowledged.

M-03 | redeployLt Can Redeploy An Unintended Token

Category	Severity	Location	Status
Unexpected Behavior	Medium	Factory.sol: 46-48	Resolved

Description PoC

If you want to redeploy a leverage token that has already been deployed, you can't call `createLt`. Instead, you must call `redeployLt`, passing the address of the token you want to redeploy. This function will attempt to swap or move the token, remove the old one, and then call `_createLT` to deploy a new instance.

However, the issue is that `redeployLt` does not update the `_ltIndex` mapping. If the owner tries to redeploy a token that's in the middle of the `_lts` array, it will result in the last two tokens sharing the same index.

This can cause unintended behavior: when you try to redeploy another middle token later, the function will incorrectly reference the last element instead of the intended one, causing to redeploy unintended token.

Further Explanation

Imagine we have three tokens: A, B, and C, all deployed. Now, we decide to redeploy B:
The code finds its index, which is 1.
It moves the last token, C, into B's spot. The list now looks like [A, C, C].
It then pops the last element, so the list becomes [A, C]. (Notice that `_ltIndex` was not updated.)
A new instance of token B (let's call it newB) is deployed and added to the list, with `_ltIndex` set to 2.

At this point, the list looks like:
[A, C, newB]

Next, we try to redeploy token C:
The code searches for its index, which is 2.
It tries to move the last token (newB) into that spot, but since newB is already there, nothing happens.

It then pops the last element, which is newB, and redeploys a new token. Now the list looks like:
[A, C, newC]
As you can see, token C was not removed — instead, newB was removed by mistake.

Recommendation

We must update also the `_ltindex` of the moved token not just the spot of it in the `_lts` array.

Resolution

Bounce Team: The issue was resolved in PR#59.

M-04 | DoS In LeveragedToken

Category	Severity	Location	Status
DoS	Medium	Staker.sol: 135	Resolved

Description

The LeveragedToken contract relies on the Staker contract for fee distribution via the _payFees function, which calls staker_.donateFees(baseAmount_). In the Staker contract's donateFees function, there is a check if (div_ = 0) revert ZeroBalance(); where div_ = totalStaked - totalUnstaking.

If there are no effective stakers (i.e., totalStaked = totalUnstaking), this revert occurs, causing _payFees to fail. This revert propagates to key LeveragedToken functions that call _payFees directly or indirectly: _checkpoint(): Called in mint(), redeem(), prepareRedeem(), executeRedeem(), bridgeOut(), and bridgeIn(). A revert here blocks all these operations.

_payStreamingFee(): Called in _checkpoint(), causing streaming fee payments to fail. _payRedemptionFee(): Called in redeem() and executeRedeem(), blocking redemptions. When totalStaked = totalUnstaking (e.g., all users have unstaked or are in the process of unstaking), the contract enters a DoS state for core functionalities like minting, redeeming.

Recommendation

Add a retry mechanism:

```
uint256 internal _pendingFees;  // Track unpaid fees
function _payFees(uint256 baseAmount_) internal {
LeveragedTokenStorage storage $ = _getLeveragedTokenStorage();
if (baseAmount_ = 0) return;
IStaker staker_ = $.globalStorage.staker();
uint256 amount_ = baseAmount_ + _pendingFees;
_baseAsset().safeIncreaseAllowance(address(staker_), amount_);
try staker_.donateFees(amount_) {
_pendingFees = 0;
} catch {
_pendingFees = amount_;
}
}
```

Resolution

Bounce Team: The issue was resolved in PR#60.

M-05 | Inability To Bridge In

Category	Severity	Location	Status
Logical Error	Medium	LeveragedToken.sol: 201	Resolved

Description

The current `bridgeIn` flow will calculate the `coreAmount_` using `evmToWei` helper function. This converts an EVM amount to a Core (wei) amount.

However, the flow later uses `bridgeToEvm` which expects an EVM amount, not Core. Consequently, the value passed will be converted again.

For USDT0, the double conversion will increase the value 100 times (2 EVM extra decimals so multiplied by 10*2).

Recommendation

When using Core amounts, make sure to use the correct `bridgeToEvm` signature, with `isEvmAmount = false` as follows:

`CoreWriterLib.bridgeToEvm(tokenId_, coreAmount_, false);`.

Alternatively, avoid converting from EVM to core and just pass the `amount_` function param:

`CoreWriterLib.bridgeToEvm(tokenId_, amount_);`.

Resolution

Bounce Team: The issue was resolved in PR#61.

L-01 | Unfair Lock Duration

Category	Severity	Location	Status
Best Practices	Low	Locker.sol: 50	Acknowledged

Description

The `lock()` function sets the unlock time as `block.timestamp + lockDuration` for all users, regardless of when they join. However, rewards accrue linearly from `_startTime` (set by the first locker) to `_startTime + lockDuration`.

Late joiners must wait the full `lockDuration` to unlock, but if they join after `_startTime + lockDuration`, they earn no rewards yet are locked for the entire period. This discourages late participation and may lead to bad user experience.

Recommendation

```
function lock(uint256 amount_) external override {
// ... existing checks ...
_startDistribution();
_checkpoint(msg.sender);
_bounce().safeTransferFrom(msg.sender, address(this), amount_);
_locked[msg.sender] = amount_;
// Set unlock time to the end of reward distribution for fairness
uint256 rewardEndTime = _startTime + lockDuration;
if (block.timestamp > rewardEndTime) {
// If rewards have ended, allow immediate unlock or reject lock
revert RewardsEnded();
}
_unlockTime[msg.sender] = rewardEndTime;  // Align with reward period end
totalLocked = amount_;
emit Lock(msg.sender, amount_);
}
```

Resolution

Bounce Team: Acknowledged.

L-02 | Claim And Recover Overlap

Category	Severity	Location	Status
Validation	Low	Airdrop.sol: 48	Resolved

Description

Users can claim their airdrop via `Airdrop:claim(amount, merkleProof)`. Claims are blocked only when `now > deadline`:

```
if (block.timestamp > deadline) revert DeadlinePassed();
```

Unclaimed tokens can be recovered by the owner via `recover`, which blocks recovery only when `now < deadline`:

```
if (block.timestamp < deadline) revert AirdropOngoing();
```

At `now = deadline`, both conditions pass: `claim` still allows claims (since `>` is false) and `recover` also allows recovery (since `<` is false).

This creates a race at the exact deadline and can lead to unexpected reverts or inconsistent outcomes depending on call ordering.

Recommendation

Consider refactoring the claim condition to be inclusive and include the deadline, so that the airdrop ends when now > deadline.

Resolution

Bounce Team: The issue was resolved in PR#55.

L-03 | Checkpoint Order Causes Wrong Reverts

Category	Severity	Location	Status
Error	Low	LeveragedToken.sol: 166-167	Resolved

Description

In executeRedeem, the balance check is performed before _checkpoint():

```
if (baseAssetBalance() < baseAmount_) revert NotEnoughBaseAsset();
_checkpoint();
```

This ordering means the balance validation can use outdated state. If _checkpoint() reduces the available balance by applying fees, the transfer may later fail with a low-level ERC20 revert (ERC20: transfer amount exceeds balance) instead of the intended custom error NotEnoughBaseAsset.

This issue is also present in bridgeOut. This leads to misleading error messages. Suppose baseAssetBalance() returns 1,000. A user calls executeRedeem with baseAmount_ = 1,000. The initial check passes since 1,000 is not less than 1,000.

Immediately after, _checkpoint() runs and deducts fees, reducing the actual available balance below 1,000. The subsequent safeTransfer then reverts with a generic ERC20 insufficient balance error, rather than the contract's intended NotEnoughBaseAsset error.

Recommendation

Move _checkpoint() before the balance check so the validation always reflects the current state.

Resolution

Bounce Team: The issue was resolved in PR#56.

L-04 | Missing Validation For Perp Owner Update

Category	Severity	Location	Status
Validation	Low	LeveragedToken.sol: 176	Resolved

Description

The `LeveragedToken` will be deployed with a `perpOwner` which is the user holding the leveraged position in Hyperliquid.

The current implementation allows global owner to update this address via `setPerpOwner` without any validation, creating a stepwise jump in `LeveragedToken._hyperliquidValue` and `HyperliquidHandler.marginUsed`

Recommendation

Avoid perp owner change if there are active positions.

Resolution

Bounce Team: The issue was resolved in PR#57.

L-05 | Unnecessary Approval

Category	Severity	Location	Status
Best Practices	Low	LeveragedToken.sol: 286	Acknowledged

Description

In order to pay referral fees, the `LeveragedToken` contract gives the `Referrals.sol` contract an allowance.

However, the contract grants an allowance for the entire `baseAmount_` (the redemption fee), even though only a small portion is needed to pay the rebates.

Recommendation

Reset the allowance, after calling the `donateRebates` function in the `LeveragedToken` contract.

Resolution

Bounce Team: Acknowledged.

L-06 | Superfluous Max Redemption Check

Category	Severity	Location	Status
Superfluous Code	Low	LeveragedToken.sol: 275	Resolved

Description

In order to calculate `_redemptionFee`, the function will choose the minimum value between the current redemption fee times the LT leverage, with the max redemption fee set in the config file.

The max leverage in Hyperliquid is 40x, and the current deployment configuration suggests that redemption fee is 0.3%, while the max redemption fee is 50%.

The `_redemptionFee` will calculate the following:

`return redemptionFee_.min(baseAmount_.mul($.globalStorage.maxRedemptionFeeShare()));`

However, this will never return the max fee calculation, as `12% < 50%`, adding unnecessary checks and wasting gas.

Recommendation

Remove the max redemption fee check. Alternatively, adjust the configuration values to make sure the fee will be capped at some point.

Resolution

Bounce Team: The issue was resolved in PR#69.

L-07 | Precision Loss In Reward Calculation

Category	Severity	Location	Status
Rounding	Low	Locker.sol: 126	Acknowledged

Description PoC

In the Locker contract, reward distribution uses integer division. This causes precision loss, leaving small amounts (dust) unclaimed in the contract.

In attached `test_DustLeftover_after_claims`, with 2 users (1000e18 and 500e18 locked, 1000e18 rewards), total claimed is 999999999999999999000 (999.999e20), leaving 1000 wei unclaimed.

Recommendation

Add a `rescueFunds` function for owner to recover unclaimed amounts.

Resolution

Bounce Team: Acknowledged.

L-08 | Streaming Fee Avoided Due To Rounding

Category	Severity	Location	Status
Rounding	Low	LeveragedToken.sol: 266	Resolved

Description

When `totalAssets` value is low, the `periodFee_` calculation can round to 0 if `_checkpoint` is called frequently (low time elapsed).

However, the `lastCheckpoint` is still updated even if `_payFees` early returns due to `baseAmount_` being zero.

Recommendation

Do not update `lastCheckpoint` timestamp if no fees are paid, unless `totalAssets` are 0.

Resolution

Bounce Team: The issue was resolved in PR#70.

L-09 | Potential Rate Volatility Bridging To Core

Category	Severity	Location	Status
Gaming	Low	LeveragedToken.sol: 220	Acknowledged

Description

According to the Hyperliquid docs, Transfers from `HyperEVM` to `HyperCore` happen in the same L1 block as the `HyperEVM` block, immediately after the `HyperEVM` block is built.

However, after some research in docs, code and Hyperliquid discord, there's no guarantee that the reflection takes place in the next EVM block, not even a clear statement in the docs to support that the next EVM block will have all previous Core queued actions processed.

Consequently, the `LeveragedToken` may be temporarily land in an invalid state, when tokens are in transit during bridging.

Recommendation

Be aware of this issue and document it for users. Will be wise to monitor the exchange rate for any stepwise jumps during normal operation

Resolution

Bounce Team: Acknowledged.

L-10 | No Vesting Token Recovery Mechanism

Category	Severity	Location	Status
Logical Error	Low	Vesting.sol: 13	Resolved

Description

The Vesting contract has no owner-only recovery function. If the contract is overfunded, ends with dust due to rounding, or the vesting setup changes (e.g., revocations) leaving surplus tokens, these tokens cannot be retrieved and remain stuck forever.

Recommendation

Add an owner-only sweep function to recover arbitrary `ERC20` tokens (at minimum the Bounce token) to a specified address. Optionally restrict recovery to when no claimable amount remains or include safety checks.

Resolution

Bounce Team: The issue was resolved in PR#73.

I-01 | Underwater Accounts Can Cause Partial DoS

Category	Severity	Location	Status
DoS	Info	HyperliquidHandler.sol: 39	Resolved

Description

The `perpValue()` function reverts when `accountValue` returned from `PrecompileLib.accountMarginSummary()` is negative. In practice, this state can occur when a user's perpetual positions are underwater.

As a result, `perpValue()` (and consequently `hyperliquidValue()`) becomes unusable for such accounts, preventing integrations from retrieving portfolio values and causing a partial DoS to the `LeveragedToken` contract.

Recommendation

Instead of reverting, consider returning zero.

Resolution

Bounce Team: The issue was resolved in PR#53.

I-02 | Incorrect Revert Message

Category	Severity	Location	Status
Error	Info	Staker.sol: 126	Resolved

Description

The `Staker.claim()` validates if claiming is enabled, but throws the revert message `AlreadyEnabled`, which is used during `enableClaiming`.

Recommendation

Consider updating the revert message to clearly show that claiming is not enabled.

Resolution

Bounce Team: The issue was resolved in PR#54.

I-03 | Users Could Spam Prepare Redeem Events

Category	Severity	Location	Status
Best Practices	Info	LeveragedToken.sol: 196	Acknowledged

Description

The protocol implements a minimum transaction amount to forbid users from spamming small mints/redeems.

However, if a user is down to deal with the minimum amount, pay the gas fees, and handle that loss; can spam the prepare redeem function, which will send prepare events to the off-chain relayer.

Recommendation

Ensure that the off-chain relayer doesn't trigger bridging calls for every emitted event, but rather batches them together, either depending on a threshold or a time interval.

Resolution

Bounce Team: Acknowledged.

I-04 | Unused Imports And Errors

Category	Severity	Location	Status
Best Practices	Info	ILeveragedToken.sol: 11	Resolved

Description

• Unused import `import {HyperliquidHandler} from "./HyperliquidHandler.sol";` in `src/Factory.sol`

• Unused import `import {IHyperliquidHandler} from "./interfaces/IHyperliquidHandler.sol";` in `src/LeveragedToken.sol`

• Unused errors in `src/interfaces/ILeveragedToken.sol`:

• `error NoCredit();`

• `error UnexpectedSize();`

• `error NotIsolated();`

• Unused errors in `src/interfaces/IReferrals.sol`:

• `error NotLeveragedToken();`

Recommendation

Remove unused imports and errors

Resolution

Bounce Team: The issue was resolved in PR#63.

I-05 | Min Lock Never Set During Deployment

Category	Severity	Location	Status
Configuration	Info	Tokenomics.s.sol: 30	Resolved

Description

During deployment, the `deployAndConfigureGlobalStorage` will set config variables in `GlobalStorage`, but it's missing the min lock amount.

Therefore, user's will be able to lock 1 wei in `Locker` contract, potentially opening new attack vectors.

Recommendation

Be sure to set the min lock amount during deployment:

`globalStorage.setMinLockAmount(Config.MIN_LOCK_AMOUNT);`

Resolution

Bounce Team: The issue was resolved in PR#64.

I-06 | Missing Validation On Vesting Duration

Category	Severity	Location	Status
Validation	Info	Vesting.sol: 28	Resolved

Description

The `Vesting` constructor initialized the `vestingDuration`. However, if the duration is 0, created vest won't be able to claim due to division by 0.

Recommendation

Validate `vestingDuration >0` during `Vesting` constructor.

Resolution

Bounce Team: The issue was resolved in PR#65.

I-07 | Mutable Base Asset Address

Category	Severity	Location	Status
Configuration	Info	GlobalStorage.sol: 55	Resolved

Description

Contracts fetch the base asset address from an external `GlobalStorage` contract at call time. Due to the fact that this token address is upgradable/mutable by owner, it will cause new calls to operate on a different token, locking user funds and opening DoS vectors.

Recommendation

Consider to only set the `baseAsset` once in `GlobalStorage` just like the bounce token.

Resolution

Bounce Team: The issue was resolved in PR#66.

I-08 | Some Functions Are Missing Event Emits

Category	Severity	Location	Status
Best Practices	Info	Vesting.sol: 112-117	Resolved

Description

Some functions, such as `cancelTransfer` and `setPerpOwner` are missing event emits despite changing state.

Recommendation

Make sure to add an event emit for `cancelTransfer` and `setPerpOwner`

Resolution

Bounce Team: The issue was resolved in PR#67.

I-09 | Immutable Variables Can Be Cached

Category	Severity	Location	Status
Gas Optimization	Info	Global	Resolved

Description

Some contracts constantly read from the `GlobalStorage` state, even though the values will never change.

This is the case for `bounce` and `baseAsset`, which should never change after being set. Although Hyperliquid fees are very low, it's best practice to avoid these calls and optimize gas.

Recommendation

Consider caching the `bounce` and `baseAsset` token address, using immutable variables if possible.

Resolution

Bounce Team: The issue was resolved in PR#68.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
M-01	Wrong Decimal Comparison	Validation	Medium	Resolved
M-02	Fees For Periods Of Inactivity Paid	Unexpected Behavior	Medium	Resolved
M-03	Checkpoint Function Permissionless	Access Control	Medium	Resolved
M-04	Infinite Exchange Rate	Logical Error	Medium	Resolved
L-01	Streaming Fee Should Be Capped	Validation	Low	Resolved
L-02	Direct Redeemers Can DOS executeRedeemers	DoS	Low	Acknowledged
L-03	Unsupported Base Assets	Logical Error	Low	Acknowledged
I-01	Hardcoded Config Addresses	Configuration	Info	Acknowledged

M-01 | Wrong Decimal Comparison

Category	Severity	Location	Status
Validation	Medium	Config.sol: 22	Resolved

Description

When the owner attempts to update the perps owner, the contract checks that the current owner does not hold any value on Core.

This check is performed by comparing `_hyperliquidAssets` against a constant defined in global storage, `perpOwnerHoldsFundsThreshold`.

The issue arises because `_hyperliquidAssets` returns values with 6 decimals (matching the base asset's decimals), while `perpOwnerHoldsFundsThreshold` is defined with 18 decimals.

Specifically, it is set in `src/constants/Config.sol` as `1e18` and later applied in the deployment script. As a result, the condition will never revert as the assets will be lower than 1e18:

```
if (_hyperliquidAssets() > $.globalStorage.perpOwnerHoldsFundsThreshold()) revert
PerpOwnerHoldsFunds();
```

Recommendation

Ensure that both values use consistent decimal precision. Since `_hyperliquidAssets` returns values in 6 decimals, update the constant in `src/constants/Config.sol` to `1e6` instead of `1e18` and acknowledge that future changes to the threshold should maintain this decimal consistency.

Resolution

Bounce Team: The issue was resolved in PR#74.

M-02 | Fees For Periods Of Inactivity Paid

Category	Severity	Location	Status
Unexpected Behavior	Medium	LeveragedToken.sol: 273	Resolved

Description PoC

In case there are periods of inactivity(most likely right after deployment) with no users, new users can be forced to pay the streaming fees for that period.

Let's assume `lastCheckpoint` is set by calling `checkpoint()` and there are no users. Attacker donates 1 wei of base asset to LT contract which makes `totalAssets() > 0` and calls `checkpoint`.

Fee calculation for 1 wei will round down to 0. The following will be true and checkpoint calls will return early: `if (periodFee_ = 0 totalAssets() > 0) return;`

If a new user mints and redeems, they will pay the streaming fee from the `lastCheckpoint` that was set at the beginning. This makes them pay fees for the period they did not participate.

Recommendation

Consider minting shares for protocol owned address right after deployment so that `totalAssets` and `totalSupply` are > 0.

This will also fix the other issue of exchange rate calculation which will happen when a user mints and redeems leaving at least 1 wei behind.

This will make `(totalSupply() > _INITIAL_SUPPLY)` and `totalAssets > 0`, so `exchangeRate()` reaches infinity DoSing new mints.

Resolution

Bounce Team: The issue was resolved in PR#75.

M-03 | Checkpoint Function Permissionless

Category	Severity	Location	Status
Access Control	Medium	LeveragedToken.sol: 216-222	Resolved

Description

The new public `checkpoint` function allows any user to execute it at any given time. As long as the time elapsed and period fee are greater than 0, it will pay streaming fees.

Previously, the checkpoint was only meant to be executed during minting and redeeming, which involved a min transaction size.

However, now users can execute it in short intervals, causing rounding errors to reduce the final streaming fees paid.

Additionally, users can execute the checkpoint with an old timestamp, increasing the likelihood of rounding issues, as they just need to pass a value 1 second older than the latest checkpoint.

Recommendation

Consider making the `checkpoint` function permissioned, only callable by the keeper.

Resolution

Bounce Team: The issue was resolved in PR#75.

M-04 | Infinite Exchange Rate

Category	Severity	Location	Status
Logical Error	Medium	LeveragedToken.sol: 236	Resolved

Description PoC

When the `LeveragedToken` is launched, depositing base assets will mint LT at a one to one ratio. However, if the user immediately redeems LT balance minus 1 wei, the new exchange rate calculation will dramatically increase.

This is due to the fact that total assets is scaled to 18 decimals, and the `div` function will add another 18 decimals, before dividing by `_INITIAL_SUPPLY + 1`

Recommendation

Consider minting protocol owned liquidity right after deployment so that `totalAssets` and `totalSupply` are not zero.

Resolution

Bounce Team: The issue was resolved in PR#77.

L-01 | Streaming Fee Should Be Capped

Category	Severity	Location	Status
Validation	Low	GlobalStorage.sol: 155-178	Resolved

Description

The redemption fee is now capped at 2% (0.02e18). However, the streaming fee and rebates still used a max value of 100%.

Recommendation

Consider adding a max value for streaming fee and rebates.

Resolution

Bounce Team: The issue was resolved in PR#76.

L-02 | Direct Redeemers Can DOS executeRedeemers

Category	Severity	Location	Status
DoS	Low	LeveragedToken.sol: 141	Acknowledged

Description

`prepareRedeem` is used for the case when the current contract balance is not enough to cover a redemption.

The shares of those users are kept in the LT contract until funds arrive via bridge. When funds arrive other user can call `redeem` and deplete funds of the contract temporarily DoS'ing the users who have prepared a redeem.

This can have more impact during times of market stress when users are rushing to redeem.

Recommendation

Add the following check in redeem:

```
uint256 currentBalance = _baseAsset().balanceOf(address(this));
uint256 reserved = ltToBaseAmount($.credit); // Convert total prepared LT to base assets
if (currentBalance > reserved) {
// Funds have arrived: Check if redemption fits in available balance after reserving
if (currentBalance - reserved < baseAmount_) revert InsufficientBalance();
} else {
// Funds not fully arrived: Skip reserved check, just ensure redemption fits in current
balance
if (currentBalance < baseAmount_) revert InsufficientBalance();
}
```

Resolution

Bounce Team: Acknowledged.

L-03 | Unsupported Base Assets

Category	Severity	Location	Status
Logical Error	Low	HyperliquidHandler.sol: 16	Acknowledged

Description

The `HyperliquidHandler` functions accept a base asset param, so values are returned in terms of the asset value and decimals.

The spot decimals are hardcoded to 8 and base asset decimals to 6. Although the base asset will initially be USDT0, not all assets have these decimals. For example, USDHL has 6 token decimals, but spot value has 7 decimals, instead of 8.

Recommendation

If the `HyperliquidHandler` should only support USDT0, it will be better to hardcode or set it at deployment time, instead of requiring a `baseAsset` param every time.

Alternatively, if the idea is to support any `baseAsset` and reuse the handler functions, consider fetching the spot decimals instead of using the hardcoded values.

Resolution

Bounce Team: Acknowledged.

I-01 | Hardcoded Config Addresses

Category	Severity	Location	Status
Configuration	Info	Config.sol: 8-10	Acknowledged

Description

The `MARKET_MAKER`, `DAO`, and `TREASURY` addresses are hardcoded to 0x...02/03/04. These are almost certainly not controlled by any known private keys or multisigs.

If the system transfers funds to these addresses (e.g., allocations, fees), the assets will be permanently lost with no recovery path.

Recommendation

Do not hardcode placeholder addresses. Parameterize these addresses at deployment and enforce they are nonzero and owned multisigs/contracts.

Resolution

Bounce 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