QuillAudits

AUDIT REPORT

December 2025

For

Bean

Table of Content

Executive Summary

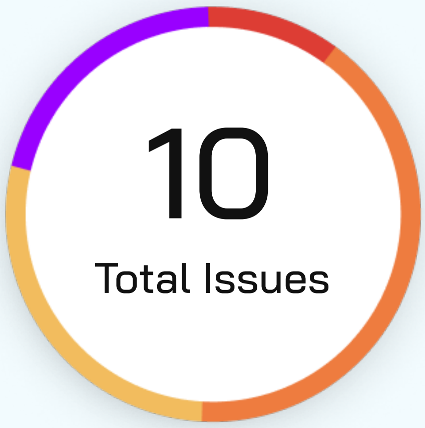
Project Name	Bean Exchange
Protocol Type	Prep Dex
Project URL	https://bean.exchange/
Overview	Bean Exchange is a prepetual dex and a fork of GMX that enables users to trade perpetual futures and spot cryptocurrencies with high leverage directly from their non-custodial wallets.
Audit Scope	The scope of this Audit was to analyze the Bean Exchange Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/BeanExchange/strategy-vault-audit/tree/perp
Branch	Main
Contracts in Scope	contracts/interfaces/markets/ISecondaryMarket.sol contracts/interfaces/markets/IPerpMarket.sol contracts/interfaces/vaults/IPerpVault.sol contracts/interfaces/externals/IWMON.sol contracts/interfaces/externals/IShMonad.sol contracts/interfaces/oracles/IPriceOracle.sol contracts/markets/lsd/ShMonadHandler.sol contracts/markets/perp/PerpHandler.sol contracts/PerpVault.sol
Commit Hash	efb94ca71b380ea7b1541403d399ecf54e3231de
Branch	Prep
Language	Solidity
Blockchain	Monad
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	24th November 2025 - 3rd December 2025
Updated Code Received	8th December 2025

Review 2	8th December 2025
Fixed In	https://github.com/BeanExchange/strategy-vault-audit/tree/dbeb5006a0273860838273169ffc38d9c90e0316

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0.0%)
🟥	High	1 (10.0%)
🟧	Medium	4 (40.0%)
🟨	Low	3 (30.0%)
🟪	Informational	2 (20.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	**1**	**4**	**3**	**2**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Wrong Index Price Used in _getPerpValue Calculations	High	Resolved
2	Precision Loss in Token Amount Calculations Enables Vault Accounting Manipulation	Medium	Resolved
3	Missing upper bound validation for fee parameters allows setting fees up to 100%	Medium	Resolved
4	Disabling a perp or secondary market permanently locks invested funds and excludes their value from portfolio calculations	Medium	Resolved
5	Use SafeERC20 methods throughout the codebase to handle non-standard token behavior	Medium	Resolved
6	Withdraw Event / Burn Mismatch	Low	Resolved
7	Vault share pricing depends on unvalidated external market data and oracle, exposing all depositors to manipulation risk	Low	Resolved
8	Missing zero address validation for owner parameter renders vault permanently non-administrable	Low	Resolved

Issue No.	Issue Title	Severity	Status
9	Unused state variable and dead code in asset accounting increase contract complexity without functional benefit	Informational	Resolved
10	Missing zero address validation in market configuration allows invalid entries in market lists	Informational	Resolved

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw
- ✓ Using inline assembly
- ✓ Style guide violation
- ✓ Unsafe type inference
- ✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

🟥 Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

🟥 High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

🟧 Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

🟨 Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

🟪 Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open Security vulnerabilities identified that must be resolved and are currently unresolved.	**Resolved** These are the issues identified in the initial audit and have been successfully fixed.
Acknowledged Vulnerabilities which have been acknowledged but are yet to be resolved.	**Partially Resolved** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Likelihood \ Impact	🟥 High	🟧 Medium	🟨 Low
🟥 High	Critical	High	Medium
🟧 Medium	High	Medium	Low
🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Wrong Index Price Used in _getPerpValue Calculations

Resolved

Path

PerpVault.sol

Function Name

`_getPerpValue`

Description

The _getPerpValue function mistakenly uses the token's own price (price) as the indexPrice parameter when calling IPerpMarket.getVaultInfo. If a perpetual market's index token differs from the long/short tokens, this mispricing occurs. In other words, it passes the long/short token price instead of the actual market index token price to the Perp market. This yields completely incorrect valuation of positions in that market.

```
if (marketState.asset0 == token && marketState.enabled) {
    uint256 indexPrice = price;  // BUG: using the asset's own price as the
index price
    uint256 longPrice = _getTokenPrice(marketState.asset0);
    uint256 shortPrice = _getTokenPrice(marketState.asset1);

    (, , , , uint256 longTokenAmount, ) = IPerpMarket(market).getVaultInfo(
        indexPrice,  // Wrong! Should be the index token's price
        longPrice,
        shortPrice
    );
    totalUsd += _tokenToUsd(marketState.asset0, longTokenAmount, price);
}
```

Notice indexPrice is set to the passed-in price (the vault token's price), not the price of the Perp market's index token. This misaligns the getVaultInfo inputs.

POC

Consider a BTC/USD perpetual market where:

Index token: WBTC, price = $60,000
Long token: WETH, price = $3,000
Short token: USDC, price = $1

If _getPerpValue(WETH, 3000) is called (calculating for WETH), the code sets indexPrice = 3000 (WETH price) instead of $60,000. As a result, getVaultInfo calculates P&L using the wrong price, leading to hugely incorrect USD values for WETH positions.

Recommendation

Look up the actual index token for each Perp market and use its price. For example:

```
Params.MarketInfo memory marketInfo = IPerpMarket(market).getMarketInfo();
uint256 indexPrice = _getTokenPrice(marketInfo.indexToken);  // Correct index
price
uint256 longPrice  = _getTokenPrice(marketState.asset0);
uint256 shortPrice = _getTokenPrice(marketState.asset1);

(,,,, uint256 longTokenAmount, uint256 shortTokenAmount) =
    IPerpMarket(market).getVaultInfo(indexPrice, longPrice, shortPrice);

if (marketState.asset0 == token) {
    totalUsd += _tokenToUsd(marketState.asset0, longTokenAmount, price);
}
if (marketState.asset1 == token) {
    totalUsd += _tokenToUsd(marketState.asset1, shortTokenAmount, price);
}
```

This ensures the correct index token price is used in vault calculations.

Medium Severity Issues

Precision Loss in Token Amount Calculations Enables Vault Accounting Manipulation

Resolved

Path

contracts/markets/perp/PerpHandler.sol:239-258

Function Name

`getVaultInfo`

Description

The getVaultInfo function calculates token amounts through division-before-multiplication operations. Line 243 divides poolValue by lpSupply before multiplying by lpBalance, line 246 calculates share price through direct division, and lines 255-256 convert USD values to token amounts via sequential divisions without scaling. This violates fixed-point arithmetic principles where multiplication must precede division to preserve precision in integer operations.

However, Solidity's integer division truncates fractional components, causing permanent precision loss when division occurs first. For instance, if poolValue is 1000e30 and lpSupply is 3e18, the operation poolValue / lpSupply yields 333333333333333333333333333333, permanently losing the 0.333 fractional remainder. This loss compounds through lines 255-256 where USD values undergo additional unscaled divisions by token prices. Fuzz testing confirms that small positions return zero tokens despite having economic value, with a vault holding $1 worth of assets computing to zero ETH and zero USDC due to cascading truncation errors.

Consequently, users with small positions receive systematically undervalued calculations, and attackers can exploit this by timing deposits to maximize truncation effects and withdrawing more tokens than entitled. The vault's accounting drifts from actual balances as precision losses accumulate, enabling value extraction through predictable rounding errors. Since getVaultInfo provides authoritative portfolio valuation throughout the system, corrupted values propagate to deposits, withdrawals, fees, and share prices in the parent PerpVault contract, with precision loss reaching 100 basis points or complete loss for certain input combinations.

Recommendation

We recommend restructuring calculations to perform multiplication before division by ensuring line 243 computes the full numerator first, and replacing lines 255-256 with scaled operations such as longTokenAmount = (vaultLongUsd * 1e18) / longPrice to preserve precision through divisions. For maximum safety, implement OpenZeppelin's FullMath library or equivalent fixed-point arithmetic, replacing all divisions with FullMath.mulDiv(numerator, multiplier, denominator) to guarantee mathematically accurate results across all input ranges while preventing overflow.

Missing upper bound validation for fee parameters allows setting fees up to 100%

Resolved

Path

contracts/PerpVault.sol:828-833

Function Name

`_setFeeParams`

Description

In contracts/PerpVault.sol:828-833, the _setFeeParams function validates that maxFeeBps is greater than or equal to baseFeeBps but does not enforce an upper bound on fee values. The configFeeParams function at line 228 allows the admin to set baseFeeBps, maxFeeBps, and slopeBps to any value up to 10000 basis points (100%).

Setting fee parameters to 100% would result in all deposited funds being transferred to the fee collector, leaving users with zero shares. While this requires admin action, the absence of a hardcoded maximum fee constant within the contract reduces transparency. Users cannot verify the maximum possible fee exposure by inspecting the contract code alone and must trust the admin will not configure excessive fees.

Recommendation

We recommend defining a constant such as uint256 private constant MAX_FEE_BPS = 1000 (10%) and adding validation in _setFeeParams to ensure params_.maxFeeBps <= MAX_FEE_BPS. This establishes a transparent, immutable upper bound that users can verify on-chain, increasing confidence in the protocol's fee structure.

Disabling a perp or secondary market permanently locks invested funds and excludes their value from portfolio calculations

Resolved

Path

contracts/PerpVault.sol:471-476

Function Name

`divestPerp`

Description

The divestPerp function enforces a market enabled check before allowing the operator to withdraw funds from a perp market. The function retrieves the PerpMarketState from storage and reverts with “MARKET_DISABLED” if the enabled flag is false. Similarly, in line 582-588, the divestMarket function performs the same enabled check for secondary markets before permitting fund withdrawal.

However, the _getPerpValue function in line 918-922 and _getSecondaryValue function in line 948-952 only include market positions in the total portfolio value calculation when marketState.enabled is true. This creates a critical inconsistency: when an admin disables a market via configPerpMarkets or configSecondaryMarkets, any funds previously invested in that market are simultaneously excluded from the portfolio value calculation while becoming impossible to withdraw. The operator cannot call divestPerp or divestMarket to recover the funds because these functions revert for disabled markets.

Consequently, shareholders suffer an immediate and permanent loss of value. The total portfolio value reported by _totalPortfolioValue drops by the entire amount invested in the disabled market, causing existing share prices to decrease proportionally. New depositors receive shares at this artificially deflated valuation, effectively extracting value from existing shareholders. The funds remain locked in the external market with no recovery mechanism available, even though the underlying assets may still be fully recoverable from the external protocol.

Consequently, this vulnerability enables both accidental and malicious fund loss scenarios. An admin attempting to disable a compromised market for safety reasons inadvertently locks all invested funds permanently. A malicious admin can disable markets containing the majority of vault assets, causing the portfolio value to collapse while the locked funds remain inaccessible. Even if the market is later re-enabled, the value extraction through dilution of existing shareholders has already occurred and cannot be reversed.

Recommendation

We recommend modifying _upsertPerpMarket and _upsertSecondaryMarket to automatically withdraw all funds from a market before allowing it to be disabled, or alternatively, removing the enabled check from divestPerp and divestMarket functions to permit fund recovery from disabled markets while retaining the check only in the invest functions to prevent new deposits into disabled markets.

Fix Summary

Client introduced the check where only empty markets can be disabled while it is possible to overwrite this check if admin pass force flag value true. However, Bean exchange team responded with below comment.

“We’ve enhanced it with the force flag. Since this action inherently relies on an admin-controlled operation, the risk falls under administrative behaviour, so it can be considered acceptable”

Use SafeERC20 methods throughout the codebase to handle non-standard token behavior

Resolved

Path

contracts/PerpVault.sol
contracts/markets/lsd/ShMonadHandler.sol

Description

Throughout the codebase, standard ERC20 methods transfer, transferFrom, and approve are used directly instead of their SafeERC20 counterparts. In contracts/PerpVault.sol, the approve function is called at lines 413, 437, 455, 458, 492, 493, 569, 582, 588, and 600. In contracts/markets/lsd/ShMonadHandler.sol, similar patterns exist for token operations.

Standard ERC20 methods return a boolean success value that is not checked in these calls. Some tokens such as USDT do not return a value at all, causing these calls to revert unexpectedly. Other tokens may return false on failure instead of reverting. Using SafeERC20 wrappers (safeTransfer, safeTransferFrom, safeApprove, or forceApprove) handles these edge cases by checking return values and reverting on failure, ensuring consistent behavior across all ERC20 implementations.

Recommendation

We recommend replacing all direct approve calls with safeApprove from OpenZeppelin's SafeERC20 library. The contract already imports SafeERC20 and applies using SafeERC20 for IERC20 at line 28, so the safe methods are available but not consistently used. For approval patterns that first approve a non-zero amount, consider using forceApprove which handles tokens that require approval to be set to zero before changing to a non-zero value.

Low Severity Issues

Withdraw Event / Burn Mismatch

Resolved

Path

PerpVault.sol

Function Name

`withdraw`

Description

In the withdraw flow, the user requests a share amount to burn. The code may reduce this to actualShareAmount if not enough buffer is available, then burns actualShareAmount from the user's balance. However, the Withdrawn event logs the original requested params.shareAmount (instead of actualShareAmount). Thus the event's "shareAmount" field can be different from the amount actually burned, misleading off-chain monitors or users.

In _previewWithdrawal, the code adjusts the share amount:

```
if (actualShareAmount > bufferShare) {
    actualShareAmount = bufferShare;
}
```

Then in withdraw:

```
_burn(msg.sender, actualShareAmount);
…
emit Withdrawn(
    msg.sender,
    params.receiver,
    params.token,
    params.shareAmount,    // requested amount (not actualShareAmount)
    calc.amountNet,
    calc.feeTokens,
    calc.feeBps
);
```

Here, _burn uses actualShareAmount, but the event emits params.shareAmount.

POC

Suppose a user requests to withdraw 100 shares, but only 50 can be covered by the buffer. The contract will burn 50 shares (actualShareAmount = 50) but emit an event showing 100 shares requested. Off-chain tools relying on the event may think the user withdrew 100 shares, causing an apparent discrepancy.

Recommendation

Change the Withdrawn event to log the actual burned share amount. For example, emit actualShareAmount instead of params.shareAmount, or include both in the event for clarity.

Vault share pricing depends on unvalidated external market data and oracle, exposing all depositors to manipulation risk

Resolved

Path

contracts/PerpVault.sol

Function Name

`_mintShares()`

Description

In contracts/PerpVault.sol:886-933, the _mintShares function calculates share amounts using totalBefore = _totalPortfolioValue(), which aggregates values from three external sources without validation. The _totalPortfolioValueRaw function at lines 1028-1040 loops through all enabled assets and calls _assetValueWithPrice at line 1037, which in turn invokes _getPerpValue at lines 916-933 and _getSecondaryValue at lines 935-951. These functions make external calls to IPerpMarket(market).getVaultInfo() at line 927 and ISecondaryMarket(market).vaultInfo() at line 946, respectively, returning token amounts that are converted to USD values and summed into the total portfolio value. Additionally, _getTokenPrice at lines 1042-1056 calls IPriceOracle(source.feed).latestRoundData() at line 1049 to obtain token prices used throughout the valuation chain. The current implementation at lines 1049-1054 validates only that the price answer is positive and optionally checks staleness via maxHeartbeat, but performs no deviation bounds checking.

However, this design creates systemic dependency on external system integrity that extends beyond first-depositor attacks to affect all vault operations. When oracle feeds experience flash crashes, extreme volatility, or manipulation, the share price calculations become distorted for every deposit and withdrawal transaction, not merely initial deposits. Chainlink oracles have historically exhibited brief price anomalies. Similarly, even whitelisted perpetual markets like GMX rely on their own oracle systems and complex pool accounting that may contain bugs or become targets of exploitation. The getVaultInfo method returns longTokenAmount and shortTokenAmount values that the vault accepts unconditionally at lines 927 and 940, with no cross-validation against actual token balances held by the market contract or sanity checks on reported value increases. Secondary markets present analogous risks where the vaultInfo function returns position arrays containing totalValueInAsset fields at line 948 that are similarly trusted without bounds verification. Because these external values directly determine the denominator in the share calculation formula shares = Precision.mulDiv(netUsd, supply, totalBefore) at line 897, any inflation in totalBefore proportionally reduces shares minted to depositors.

Consequently, the vault inherits the security properties of all connected external systems, creating multiple attack vectors and operational failure modes. The total value at risk scales with vault TVL and could represent millions of dollars across hundreds of depositors, all dependent on the continuous integrity of multiple external protocols operating beyond the vault administrator’s control.

Recommendation

We recommend implementing a multi-layered defense strategy that combines technical safeguards with operational monitoring. First, add price deviation bounds check in the _getTokenPrice function Second, implement minimum initial deposit requirements by adding a constant such as MIN_INITIAL_DEPOSIT_USD = 1000e30 representing $1000 in 30-decimal scale and enforcing require(netUsd >= MIN_INITIAL_DEPOSIT_USD) when supply == 0 in the _mintShares function to prevent dust-based precision attacks. Third, track how much value has been added in perp and secondary market and validate the deviation from the getVaultInfo return value. Fourth, establish rigorous vetting procedures for all external integrations by requiring comprehensive security audits of perpetual markets and secondary markets before whitelisting, Implement off-chain monitoring infrastructure that tracks oracle prices, market-reported values, and share price calculations in real-time, automatically triggering alerts when values deviate from expected ranges or historical patterns.

Fix Summary

Bean exchange introduced the MIN_INITIAL_DEPOSIT_USD variable to avoid dust deposits, while the price deviation check bounds in Oracle prices have not been implemented for a given reason below from the Bean exchange team

“We don’t think changing the design is necessary at this stage. The current implementation works as intended and does not introduce any issues.”

Missing zero address validation for owner parameter renders vault permanently non-administrable

Resolved

Path

contracts/PerpVault.sol

Function Name

`_mintShares()`

Description

In contracts/PerpVault.sol:120-121, the initialize function grants DEFAULT_ADMIN_ROLE to params.owner without validating that the address is non-zero. The _grantRole function from OpenZeppelin AccessControl accepts any address including address(0) and assigns the role without reverting. Unlike the operator and feeCollector parameters which are validated at lines 124-131 before role assignment, the owner parameter bypasses this check entirely.

However, granting DEFAULT_ADMIN_ROLE to address(0) renders the vault permanently non-administrable. All administrative functions including configOracle, configAssets, configPerpMarkets, configSecondaryMarkets, configFeeParams, configCooldown, configBlacklist, configRoles, setOperator, setFeeRecipient, and enableEmergency are protected by _enforceAdmin at line 885 which requires the caller to hold DEFAULT_ADMIN_ROLE. Since no externally owned account or contract can call functions from address(0), these functions become permanently inaccessible once initialized with a zero owner address.

Recommendation

We recommend adding an explicit zero address check for params.owner before granting DEFAULT_ADMIN_ROLE by adding if (params.owner == address(0)) { revert IPerpVault.InvalidAddress(); } immediately before the _grantRole(DEFAULT_ADMIN_ROLE, params.owner) call at line 121, consistent with the validation pattern already used for operator and feeCollector parameters.

Informational Issues

Unused state variable and dead code in asset accounting increase contract complexity without functional benefit

Resolved

Path

contracts/interfaces/vaults/IPerpVault.sol:91-97

Description

In contracts/interfaces/vaults/IPerpVault.sol:91-97, the AssetState struct declares a managedValueUsd field at line 96 that is intended to track the USD value of assets deployed to external markets. The corresponding AssetInfo struct at lines 189-196 also includes a managedValueUsd field at line 195 for view function responses. In contracts/PerpVault.sol, the investPerp function at lines 400-456 calculates usdValue0 and usdValue1 at lines 408 and 432 respectively by calling _getTokenPrice and _tokenToUsd, but the subsequent assignments to asset0.managedValueUsd at line 411 and asset1.managedValueUsd at line 435 are commented out. Similarly, the investMarket function at lines 543-573 calculates usdValue at line 564 but the assignment to asset.managedValueUsd at line 566 is also commented out. The divestPerp function at lines 459-540 contains comments at lines 512 and 528 explicitly noting that the price conversion is unnecessary because managedValueUsd is not updated. Despite this, the price fetching and USD conversion operations at lines 514-515 and 530-531 remain in the codebase, executing gas-consuming external oracle calls that serve no purpose.

However, the presence of this unused variable and associated dead code creates several maintenance and readability concerns. The unused struct field occupies a storage slot in the AssetState mapping that will never be written to after deployment, representing wasted contract bytecode and potential storage overhead if the field were ever accidentally written. The dead code performing USD conversions in the investment functions consumes approximately 2,600 gas per price fetch due to the external oracle call, multiplied by the number of assets involved in each transaction, accumulating unnecessary costs over the contract lifetime.

Recommendation

We recommend removing the managedValueUsd field from the AssetState struct in contracts/interfaces/vaults/IPerpVault.sol at line 96 and removing the corresponding field from the AssetInfo struct at line 195, then updating the vaultInfo function to exclude this field from its return value. Additionally, remove the dead code and non-required USD conversion within the codebase.

Missing zero address validation in market configuration allows invalid entries in market lists

Resolved

Path

contracts/PerpVault.sol:107-117

Function Name

`initialize`

Description

In contracts/PerpVault.sol:107-117, the initialize function iterates over params.perpMarkets and params.secondaryMarkets arrays and calls _upsertPerpMarket at line 109 and _upsertSecondaryMarket at line 116 without validating that the market address is non-zero. The _upsertPerpMarket function at lines 847-870 only validates market configuration when enabled && market != address(0) at line 849, meaning a configuration with market = address(0) and enabled = true bypasses asset validation. Similarly, _upsertSecondaryMarket at lines 872-880 performs no address validation. Both functions add the market address to their respective arrays at lines 864 and 876 when certain conditions are met, potentially including zero addresses.

However, permitting zero address entries in the perpMarketList or secondaryMarketList arrays introduces operational risks and unexpected behavior. The _getPerpValue function at lines 904-943 iterates over perpMarketList and calls IPerpMarket(market).getVaultInfo at line 927, which would revert or behave unpredictably when invoked on the zero address. The _getSecondaryValue function at lines 945-961 similarly calls ISecondaryMarket(market).vaultInfo at line 955. These functions are called during every deposit and withdrawal via _assetValueWithPrice and _totalPortfolioValue, meaning a single zero address entry would cause all user operations to fail. The configPerpMarkets and configSecondaryMarkets functions at lines 206-216 also lack zero address checks, allowing administrators to inadvertently add invalid entries post-initialization.

Consequently, an administrative error during initialization or configuration that includes a zero address market entry would render the vault inoperable for all users. Deposits and withdrawals would revert when calculating portfolio value, and the only remediation would be to disable the invalid market entry through a separate admin transaction. The lack of input validation places unnecessary burden on administrators to manually verify all configuration parameters and increases the risk of deployment failures or operational incidents.

Recommendation

We recommend adding explicit zero address validation in both _upsertPerpMarket and _upsertSecondaryMarket functions by requiring market != address(0) when enabled is true before proceeding with any state changes. Additionally, validate that asset0 and asset1 for perp markets and asset for secondary markets are non-zero when the market is enabled, ensuring all required addresses are valid before adding entries to the market lists.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Bean Exchange. We performed our audit according to the procedure described above.

Issues of High, Medium, Low and Informational severity were found, which has been resolved by Bean Exchange Team.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

December 2025

For

Bean

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com