FAILSAFE

18 November 2025

Prepared for
KD
Jun
BaseVol

Prepared by
Sameh Wahab
Ari Medvinsky
FailSafe

BaseVol

Smart Contract Audit Report

Table of Contents

Executive Summary

FailSafe was engaged to perform a comprehensive, high-assurance security assessment of the BaseVol smart contracts. Our team conducted an end-to-end review to evaluate the protocol's security posture, identify potential vulnerabilities, and deliver actionable recommendations to strengthen overall resilience. Leveraging deep expertise in blockchain security and threat modeling, we executed a meticulous analysis focused on uncovering issues that could compromise user funds, system reliability, or protocol integrity.

During the assessment, we identified several significant vulnerabilities, including critical issues with material financial and operational impact. Notably, we discovered logic flaws allowing duplicate order ID submissions that could enable double-spending, as well as insufficient validation of share prices that could result in division-by-zero scenarios and a complete locking of user funds. We also observed the absence of pause protection on operator-level functions and unbounded loops within automated processing workflows, both of which introduced risks of denial-of-service conditions and irreversible protocol disruption.

We commend the BaseVol development team for their proactive and collaborative approach to security. The team responded with exceptional diligence, addressing the identified vulnerabilities promptly and effectively. By implementing the recommended fixes and design improvements outlined in this report, the BaseVol team has demonstrated a strong commitment to delivering a secure, stable, and trustworthy ecosystem for its users.

Project Details

Project	BaseVol
Website	https://www.basevol.com/
Repository	https://github.com/stvol-official/basevol-contract/commits/feat/erc7540/contracts
Blockchain	Base
Audit Type	Smart Contract Audit Report
Initial Commit	856897810724ce36ce5d665f5502c486cc57f7ac
Final Commit	5b36b2b6ab183962774890dfb4656cf2fc91273b
Timeline	7 October 2025 - 20 October 2025
	Final Report: 18 November 2025

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

⊗ **Critical**	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
● **High**	The issue affects the ability of the platform to compile or operate in a significant way.
● **Medium**	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
● **Low**	The issue has minimal impact on the platform’s ability to operate.
● **Info**	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- contracts/diamond-common/Diamond.sol
- contracts/diamond-common/facets/DiamondCutFacet.sol
- contracts/diamond-common/facets/DiamondLoupeFacet.sol
- contracts/diamond-common/libraries/LibDiamond.sol
- contracts/diamond-common/interfaces/IDiamondCut.sol
- contracts/diamond-common/interfaces/IDiamondLoupe.sol
- contracts/basevol/facets/BaseVolAdminFacet.sol - Administrative functions
- contracts/basevol/facets/BaseVolViewFacet.sol - View/query functions
- contracts/basevol/facets/OrderProcessingFacet.sol - Order matching and processing
- contracts/basevol/facets/RedemptionFacet.sol - Redemption logic
- contracts/basevol/facets/RoundManagementFacet.sol - Epoch/round lifecycle management
- contracts/basevol/facets/InitializationFacet.sol - Contract initialization
- contracts/basevol/libraries/LibBaseVolStrike.sol
- contracts/genesis-vault/facets/VaultCoreFacet.sol
- contracts/genesis-vault/facets/SettlementFacet.sol - Epoch settlement logic
- contracts/genesis-vault/facets/KeeperFacet.sol - Keeper operations
- contracts/genesis-vault/facets/GenesisVaultAdminFacet.sol - Administrative functions
- contracts/genesis-vault/facets/GenesisVaultViewFacet.sol - View functions
- contracts/genesis-vault/facets/GenesisVaultInitializationFacet.sol - Initialization
- contracts/genesis-vault/facets/ERC20Facet.sol - ERC-20 token operations
- contracts/genesis-vault/libraries/LibERC20.sol
- contracts/genesis-vault/libraries/LibGenesisVault.sol
- contracts/genesis-vault/libraries/LibGenesisVaultStorage.sol
- contracts/core/BaseVolOneMin.sol
- contracts/core/ClearingHouse.sol - Central clearing and settlement
- contracts/core/VaultManager.sol - Vault management system
- contracts/core/vault/GenesisStrategy.sol - Strategy management
- contracts/core/vault/BaseVolManager.sol - BaseVol integration manager
- contracts/core/vault/MorphoVaultManager.sol - Morpho protocol integration
- contracts/core/vault/storage/GenesisStrategyStorage.sol

- contracts/core/vault/storage/BaseVolManagerStorage.sol
- contracts/core/vault/storage/MorphoVaultManagerStorage.sol
- contracts/storage/ClearingHouseStorage.sol
- contracts/storage/VaultManagerStorage.sol
- contracts/core/vault/interfaces/IGenesisStrategy.sol
- contracts/core/vault/interfaces/IBaseVolManager.sol
- contracts/core/vault/interfaces/IMorphoVaultManager.sol
- contracts/interfaces/IClearingHouse.sol
- contracts/interfaces/IVaultManager.sol
- contracts/interfaces/IERC173.sol
- contracts/interfaces/IDiamondCut.sol
- contracts/interfaces/IDiamondLoupe.sol
- contracts/core/vault/errors/GenesisStrategyErrors.sol
- contracts/errors/BaseVolErrors.sol
- contracts/errors/ClearingHouseErrors.sol
- contracts/errors/CommonErrors.sol
- contracts/errors/VaultErrors.sol
- contracts/libraries/PythLazer.sol - Pyth oracle integration
- contracts/libraries/PythLazerLib.sol - Pyth oracle library
- contracts/libraries/PendingLib.sol - Pending request utilities
- contracts/libraries/ERC20TokenImpl.sol
- contracts/libraries/LibBaseVolStrike.sol
- contracts/libraries/LibDiamond.sol
- contracts/types/Types.sol - Common type definitions
- contracts/upgradeInitializers/DiamondInit.sol

Out of scope

- contracts/core/BaseVolStrike.sol (legacy) - Strike-based prediction market
- contracts/core/BaseVolOneHour.sol (legacy) - 1-hour prediction market
- contracts/core/vault/GenesisVault.sol (legacy) - Genesis vault implementation

- contracts/core/vault/GenesisManagedVault.sol (legacy) - Managed vault base
- contracts/core/vault/storage/GenesisVaultStorage.sol
- contracts/core/vault/storage/GenesisVaultManagedVaultStorage.sol
- contracts/storage/BaseVolStrikeStorage.sol
- contracts/storage/BaseVolOneMinStorage.sol
- contracts/core/vault/interfaces/IGenesisVault.sol
- contracts/core/vault/interfaces/IERC7540.sol *(superseded by genesis-vault implementation)*
- contracts/core/vault/errors/GenesisVaultErrors.sol
- contracts/facets/AdminFacet.sol
- contracts/facets/DiamondCutFacet.sol
- contracts/facets/DiamondLoupeFacet.sol
- contracts/facets/InitializationFacet.sol
- contracts/facets/OrderProcessingFacet.sol
- contracts/facets/RedemptionFacet.sol
- contracts/facets/RoundManagementFacet.sol
- contracts/facets/ViewFacet.sol
- contracts/interfaces/IMetaMorphoV1_1.sol - External Morpho protocol interface
- contracts/interfaces/IMorpho.sol - External Morpho protocol interface
- Deployment scripts and migration files
- Testing utilities and test contracts
- Frontend/backend integration code
- Node modules and third-party dependencies
- Documentation files (README.md, genesisvault-erc7540.md)

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
✖ Critical	9	-	-	-	9
❗ High	8	-	-	-	8
❗ Medium	11	-	-	-	11
❗ Low	4	-	-	-	4
ℹ Info	3	-	-	-	3
Total	35	0	0	0	35

#	Findings	Severity	Status
1	50-Epoch Claimability Window - Permanent Fund Loss	⊗ Critical	Resolved
2	Auto-Processing Unbounded Loops - Complete Settlement DoS	⊗ Critical	Resolved
3	Critical Operator Functions Missing Pause Protection	⊗ Critical	Resolved
4	Donation Attack via Strategy - Extreme Inflation Bypasses +1 Protection	⊗ Critical	Resolved
5	Duplicate Order IDs - Accounting Corruption And Double-Spending	⊗ Critical	Resolved
6	Fee Calculation Overflow	⊗ Critical	Resolved
7	Force Withdrawal Zeroes Entire Balance - Direct Fund Loss	⊗ Critical	Resolved
8	No Validation of Share Price - Can Settle with Price = 0	⊗ Critical	Resolved
9	Zero Price Manipulation	⊗ Critical	Resolved
10	Anyone Can Inflate Strategy Valuation by Depositing to BaseVolManager via Public depositTo	❗ High	Resolved
11	Base Initialization Function Lacks One-Time-Only Protection + Missing Input Validation	❗ High	Resolved
12	GenesisStrategy utilizedAssets Accounting Can Diverge from Actual Asset Positions	❗ High	Resolved
13	Infinite Token Approvals Amplify Vulnerabilities in Multiple Contracts	❗ High	Resolved
14	Memory vs Storage Bug - Order Accounting Corruption	❗ High	Resolved
15	Unbounded Loop DoS in '_settleFilledOrders'	❗ High	Resolved
16	Unbounded Loops Over User Epochs - DoS Attack	❗ High	Resolved
17	Withdrawal Request Spam - Unbounded Array Growth Causes Operational DoS	❗ High	Resolved
18	Commission Fee Mid-Epoch Change Affects Pending Orders	❗ Medium	Resolved
19	depositTo Phishing - Public Deposit Surface Attack	❗ Medium	Resolved
20	Missing Amount and Parameter Validation Across Multiple Admin Functions	❗ Medium	Resolved
21	Missing Oracle and Price Data Validation Across Settlement Flow	❗ Medium	Resolved
22	Missing Zero-Address Validation Across Multiple Functions	❗ Medium	Resolved
23	No Maximum Fee Validation - DoS via Fee Misconfiguration	❗ Medium	Resolved
24	Single Operator Centralization Without Safeguards	❗ Medium	Resolved
25	Strategy Rebalancing Transfer Failure - Missing Self-Allowance (Simple 1-Line Bug)	❗ Medium	Resolved

#	Findings	Severity	Status
26	Systematic Input Validation Missing in Initialization Functions	Medium	Resolved
27	Unbounded Loops and O(n) Operations Cause DoS at Scale	Medium	Resolved
28	Vault Shares Transferable When Paused - Design Ambiguity	Medium	Resolved
29	ETH Transfers Use Deprecated transfer() with 2300 Gas Stipend	Low	Resolved
30	Missing Input Validation for Critical Admin Parameters Across Multiple Functions	Low	Resolved
31	Missing or Improperly Timed Event Emissions for Critical Admin Operations	Low	Resolved
32	View Functions Iterate Unbounded Arrays Without Pagination	Low	Resolved
33	Critical Admin Operations Lack Timelock Protection for Both Parameter Changes and Upgrades	Info	Resolved
34	Empty Order Array Causes Array Out-of-Bounds Error	Info	Resolved
35	Missing Input Validation on View Function Parameters	Info	Resolved

Finding 1: 50-Epoch Claimability Window - Permanent Fund Loss

Severity: ❌ Critical

Status: Resolved

Description:

Hardcoded `if (epoch + 50 <= currentEpoch) continue;` in all claim functions permanently skips epochs older than 50. This is an intentional design for gas optimization (README.md line 590), but creates catastrophic fund loss: BaseVolOneMin (60s epochs) = 50 minutes window, BaseVolOneHour (3600s epochs) = 50 hours window. User funds remain in storage but are permanently inaccessible. `sweep()` function exists but requires totalSupply==0, making individual recovery impossible.

Impact:

- Users lose 100% of deposited funds if they don't claim within 50 minutes (1-min vault) or 50 hours (1-hour vault).
- Funds remain in storage and contract holds tokens, but users can never access them.
- No individual recovery mechanism exists while vault is active.

Source:

- contracts/genesis-vault/facets/VaultCoreFacet.sol (deposit line 346, mint line 406, withdraw line 481, redeem line 564, maxDeposit line 647, maxRedeem line 674, maxWithdraw line 701)
- contracts/genesis-vault/facets/GenesisVaultViewFacet.sol (view functions lines 413, 442, 471, 504)
- contracts/genesis-vault/facets/GenesisVaultAdminFacet.sol (sweep line 201)

Code:

```
// VaultCoreFacet.sol:346 (and 6 other locations)
for (uint256 i = 0; i < userEpochs.length && remainingAssets > 0; i++) {
    uint256 epoch = userEpochs[i];
    if (epoch + 50 <= currentEpoch) continue;  // ⊠ Permanently skips old epochs

    // Process epoch...
}

// Impractical recovery (GenesisVaultAdminFacet.sol:201)
function sweep(address receiver) external onlyOwner {
    if (s.totalSupply != 0) revert TotalSupplyNotZero();  // ⊠ Requires empty vault
    uint256 balance = s.asset.balanceOf(address(this));
    s.asset.safeTransfer(receiver, balance);
}
```

Proof of Concept:

Attack Steps (Natural - No Attacker Needed):

1. Friday 9 AM (Epoch 1000): Alice deposits 10,000 USDC via `requestDeposit()`
2. Vault stores: `userEpochDepositAssets[alice][1000] = 10,000`
3. Friday 5 PM (Epoch 1008): Epoch 1000 settles → `roundData[1000].isSettled = true`
4. Alice goes on weekend vacation (Friday-Sunday)
5. Epochs advance: 1009, 1010, … 1050, 1051, … 1060
6. Monday 9 AM (Epoch 1072, 72 hours later): Alice returns
7. Alice calls `deposit()` to claim shares
8. Function loops: Finds epoch 1000 in `userDepositEpochs[alice]`
9. Check: `if (1000 + 50 <= 1072) continue` → TRUE → SKIPS epoch 1000
10. Loop completes, returns 0 shares minted
11. Alice's 10K USDC permanently inaccessible (still in storage, but skipped forever)

Remediation:

- RECOMMENDED: Pagination (maintains gas efficiency, no fund loss)

```
function deposit(uint256 assets, address receiver, address controller, uint256 maxEpochs)
    external returns (uint256 shares) {
    uint256 epochsProcessed = 0;
    for (uint256 i = 0; i < userEpochs.length && remainingAssets > 0; i++) {
        if (epochsProcessed >= maxEpochs) break;  // ⊠ Pagination limit
        // ⊠ No expiration - all epochs accessible
        epochsProcessed++;
    }
}
```

- ALTERNATIVE: Return funds on expiry

```
if (epoch + 50 <= currentEpoch) {
    uint256 expiredAssets = s.userEpochDepositAssets[controller][epoch];
    if (expiredAssets > 0) {
        s.asset.safeTransfer(controller, expiredAssets);
        delete s.userEpochDepositAssets[controller][epoch];
    }
    continue;
}
```

- MINIMUM: Extend window (50 epochs → 2160 epochs = 90 days for 1-hour vault)

Discussion:

Developer:

The Genesis Vault currently only checks the most recent 50 epochs when calculating pending deposits and pending redeems.

Reasons: 1. Checking all epochs would exceed the gas limit and cause errors. 2. Normally, distributions occur automatically through onRoundSettled, so it's not an issue — but if onRoundSettled fails more than 50 times, it could become one. That said, if it has failed 50+ times already, that's a bigger problem in itself. 3. The functions performing this check are deposit, mint, withdraw, and redeem. In Genesis Vault, since distribution happens automatically via onRoundSettled, users typically don't call these functions directly. They exist only to comply with the ERC-7540 standard.

Auditor:

Thanks for the detailed explanation! We completely understand the gas optimization rationale - it makes sense. However, we'd like to discuss an alternative approach that maintains gas efficiency while preventing fund loss.

The Challenge We see that `deposit()`/`withdraw()`/`redeem()`/`mint()` serve as recovery when `onRoundSettled()` fails. The concern is this creates a dependency loop:

Flow:

```
onRoundSettled fails → Users need manual claim → 50-epoch limit expires → Funds become
       inaccessible
```

When onRoundSettled Can Fail: - Unbounded loops DoS (1000+ users exceeds gas limit) - Zero share price causes division-by-zero - Fee calculation overflow reverts settlement

The audit found several scenarios where this can occur, creating the exact situation mentioned.

Timeline Consideration

- **BaseVolOneMin**: 50 epochs = 50 minutes
- **BaseVolOneHour**: 50 epochs = 2.08 days

These windows might be tight for production scenarios like keeper downtime, infrastructure maintenance, or the documented DoS scenarios.

ERC-7540 Standard Consideration Since these functions exist for ERC-7540 compliance, having them expire after 50 epochs could create challenges. Users expecting standard behavior might lose access to their funds.

Suggested Solutions (Still Maintains Gas Efficiency!) **Option 1 - Pagination (Our Recommendation):**

```
function deposit(..., uint256 maxEpochs) external returns (uint256) {
    for (uint256 i = 0; i < userEpochs.length && epochsProcessed < maxEpochs; i++) {
        // Process epochs - no expiration
    }
}
```

Option 2 - Return Expired Funds:

```
if (epoch + 50 <= currentEpoch) {
    s.asset.safeTransfer(controller, expiredAssets);  // Auto-return
}
```

Option 3 - Extend Window:

```
if (epoch + 2160 <= currentEpoch) continue;  // 90 days
```

Why This Matters: - Fund loss with no recovery path - Short timeframes (50 min - 2 days) for real-world scenarios - Could affect users during infrastructure issues

Our recommendation: Option 1 (pagination) would be ideal - it keeps your gas optimization while completely eliminating the fund loss risk. It's a win-win!

Developer:

Please check 5ff318b2289d924af381063d4d7b179223f42aa0 commit

Auditor:

VERIFIED FIXED(commit 5ff318b2289d924af381063d4d7b179223f42aa0).

Finding 2: Auto-Processing Unbounded Loops - Complete Settlement DoS

Severity: ❌ Critical

Status: Resolved

Description:

Auto-processing functions loop through ALL users for an epoch without gas limits or pagination. When epoch has 1000+ users (easily achievable with 1 wei deposits), `onRoundSettled()` exceeds block gas limit and reverts. Settlement permanently fails, locking all user funds for that epoch. No recovery mechanism exists.

Impact:

Complete settlement DoS when epoch has 1000+ users. All funds locked permanently for that epoch. Attack is cheap (1 wei deposit per user = ~$300-$3000 for 1000 users), trivial to execute, and undetectable until settlement fails.

Source:

- contracts/genesis-vault/facets/SettlementFacet.sol (`_autoProcessEpochDeposits` , lines 279-289)
- contracts/genesis-vault/facets/SettlementFacet.sol (`_autoProcessEpochRedeems` , lines 291-301)
- contracts/genesis-vault/facets/SettlementFacet.sol (`onRoundSettled` , line 85)

Code:

```
function _autoProcessEpochDeposits(uint256 epoch) internal {
    address[] memory users = s.epochDepositUsers[epoch];
    for (uint256 i = 0; i < users.length; i++) {  // NO GAS LIMIT
        _autoProcessUserDeposit(users[i], epoch);
    }
}

function _autoProcessEpochRedeems(uint256 epoch) internal {
    address[] memory users = s.epochRedeemUsers[epoch];
    for (uint256 i = 0; i < users.length; i++) {  // NO GAS LIMIT
        _autoProcessUserRedeem(users[i], epoch);
    }
}

// Attacker creates 1000+ users easily
function requestDeposit(...) external {
    _addUserToEpochDepositList(controller, currentEpoch);  // No limit
}
```

Proof of Concept:

Attack Steps:

1. Attacker creates 1000 wallet addresses (gas: ~$300-$3000)
2. For each wallet: Calls `requestDeposit(1 wei)` -> Adds to `epochDepositUsers[currentEpoch]`
3. Epoch 1000 now has 1000+ users in array
4. Epoch 1000 ends, keeper calls `onRoundSettled(1000)`
5. Function calls `_processRoundSettlement()` -> `_autoProcessEpochDeposits(1000)`
6. Loop iterates: `for (i = 0; i < 1000; i++)` -> Each iteration: ~100K gas
7. Total gas needed: 1000 x 100K = 100M gas (exceeds 30M block limit)
8. Transaction reverts with “Out of Gas”
9. Epoch 1000 can NEVER settle (no pagination, no recovery)
10. All legitimate user funds for epoch 1000 permanently locked

Remediation:

```
// Implement batch-based settlement with pagination
function settleEpochBatch(uint256 epoch, uint256 startIndex, uint256 batchSize) external onlyKeeper {
    require(startIndex + batchSize <= s.epochDepositUsers[epoch].length, "Invalid batch");

    for (uint256 i = startIndex; i < startIndex + batchSize; i++) {
        _autoProcessUserDeposit(s.epochDepositUsers[epoch][i], epoch);
    }

    // Track progress
    s.epochProcessedCount[epoch] = startIndex + batchSize;
}

// Multiple transactions can settle an epoch
function isEpochFullySettled(uint256 epoch) external view returns (bool) {
    return s.epochProcessedCount[epoch] >= s.epochDepositUsers[epoch].length;
}
```

Discussion:

> *Developer:*
>
> At the end of each round, automatic distribution loops through all users who made a request deposit or request redeem for that round. If someone maliciously sends requests from thousands of accounts, the loop could run out of gas while processing them.
>
> This finding is valid — a deliberate spam attack could indeed exhaust gas. We’ll likely need to implement pagination, splitting the distribution process across multiple transactions instead of a single one. Not urgent right now, but good to add as a TODO item.

> *Auditor:*
>
> Thanks for acknowledging this is valid and for planning to implement pagination! We’d like to respectfully

suggest this should be prioritized before production launch rather than as a TODO item. Here's why:

Attack Economics

- **Attack :** (1000 addresses × 1 wei deposits)
- **Impact:** Permanent DoS + fund loss (via 50-epoch expiration)
- **Affected:** ALL users in that epoch
- **Recovery:** No operational workaround available

This represents a very low-cost attack with significant impact on users.

This unbounded loop issue could break that assumption:

```
Attacker DoS epoch → onRoundSettled fails permanently →
Users need manual claim → 50-epoch limit expires →
100% permanent fund loss
```

Timeline (BaseVolOneHour): - T+0: attack poisons epoch - T+50 hours: Window expires
- Result: ALL users lose 100% of funds permanently

Why We Recommend Pre-Launch Priority

1. **No operational workaround** - If it happens, settlement is stuck
2. **Repeatable** - Could affect multiple epochs
3. **Difficult recovery** - Would need contract upgrade with state migration

Implementation Suggestion

```
function settleEpochBatch(uint256 epoch, uint256 startIdx, uint256 batchSize)
    external onlyKeeper {
    // Process users[startIdx : startIdx+batchSize]
    // Track progress: s.epochProcessedCount[epoch]
}
```

> **Urgency:** We strongly recommend fixing before production launch
>
> **Reasoning:** - Low attack cost vs. high impact on users - Limited operational workarounds - Could compound with the 50-epoch limit issue - You've already acknowledged it's valid - implementing pagination would address this completely
>
> **Bottom line:** Since you're already planning pagination, we'd love to see this prioritized for the launch version rather than post-launch.

> *Developer:*
>
> Please check a822c89533b22a22949935d0876ed874e2161165 commit.
>
> Change planned so keeper calls getRemainingSettlementCount after onRoundSettled repetitively until isComplete is confirmed!

> *Auditor:*
>
> **VERIFIED FIXED**(commit a822c89533b22a22949935d0876ed874e2161165).

Finding 3: Critical Operator Functions Missing Pause Protection

Severity: ❌ Critical

Status: Resolved

Description:

Four critical operator functions lack `whenNotPaused` modifier. During emergencies when admin calls `pause()`, operators can continue submitting, closing, and settling orders. Emergency pause mechanism is completely bypassed for the most critical operations. Same issue affects BaseVolOneHour.sol.

Impact:

Admin calls `pause()` during detected exploit, but operator functions continue processing orders. Ongoing exploit continues uninterrupted. User funds remain at risk during emergency. Pause mechanism useless for most critical functions.

Source:

- contracts/core/BaseVolOneMin.sol (updatePrice line 109, submitOneMinOrders line 121, closeOneMinOrders line 175, settleOneMinOrders line 310)

Code:

```
// All missing whenNotPaused modifier
function updatePrice(...) external payable onlyOperator {  // Missing whenNotPaused
function submitOneMinOrders(...) external nonReentrant onlyOperator {  // Missing whenNotPaused
function closeOneMinOrders(...) public onlyOperator {  // Missing whenNotPaused
function settleOneMinOrders(...) public onlyOperator returns (...) {  // Missing whenNotPaused

// Compare with BaseVolStrike.sol:108 which HAS protection
function executeRound(...) external payable whenNotPaused onlyOperator {  // Has whenNotPaused
```

Proof of Concept:

Attack Steps:

1. Normal operations: Operator submits orders via `submitOneMinOrders()`
2. T=10:00 AM: Admin detects price oracle manipulation exploit in progress
3. T=10:01 AM: Admin immediately calls `pause()` to halt all operations
4. Contract state: `paused = true`
5. T=10:02 AM: Attacker (compromised operator) calls `submitOneMinOrders([...])`

6. Function executes successfully (NO `whenNotPaused` check)
7. Attacker calls `settleOneMinOrders([...])` -> Processes fraudulent orders
8. T=10:05 AM: $500K drained while contract is "paused"
9. Admin believes protocol is safe, but exploit continued during pause
10. Emergency pause mechanism completely useless

Remediation:

```
// Add whenNotPaused modifier to all 4 functions
function submitOneMinOrders(...) external nonReentrant onlyOperator whenNotPaused {  // Add this
function closeOneMinOrders(...) public onlyOperator whenNotPaused {  // Add this
function settleOneMinOrders(...) public onlyOperator whenNotPaused returns (...) {  // Add this
function updatePrice(...) external payable onlyOperator whenNotPaused {  // Add this
```

Discussion:

> *Developer:*
>
> This refers to the 1-minute contract. Some functions remain operational even when the admin pauses the system. We've never actually used pause, and in an emergency, we could simply take down the server, so it's not a major concern.

> *Auditor:*
>
> We understand the operational approach! Just wanted to flag that server shutdown might not cover all scenarios, particularly if operator keys are compromised.
>
> **The Situation** Four operator functions in BaseVolOneMin don't have the `whenNotPaused` modifier: - `submitOneMinOrders()` - Processes user orders - `closeOneMinOrders()` - Closes positions - `settleOneMinOrders()` - Executes settlement - `updatePrice()` - Updates prices
>
> **The concern:** If admin calls `pause()` during an incident, these functions would still work.
>
> **The Compromised Key Scenario** If operator keys get compromised (phishing, breach, etc.): - Attacker can use them from their own infrastructure - Server shutdown wouldn't help because they don't need your server - On-chain permissions remain active

Potential timeline:

```
T+1: Admin calls pause() → paused = true
T+2: Admin starts server shutdown process
T+3-5: Attacker (with stolen keys) keeps calling functions from elsewhere
T+10: Server is down, but attacker already done
```

There's a gap where `pause()` is active but doesn't actually stop these operations.

Architectural Inconsistency **BaseVolOneHour:** Has pause protection (inherits from BaseVolStrike) **BaseVolOneMin:** Missing pause protection

Since they have the same threat model, matching protection would be consistent.

The "Never Used" Point We hear you on not having used pause! The concern is more about "what if": - Security controls are like fire extinguishers - rarely needed, but critical when they are - If pause exists, it's good for it to work fully (avoids false confidence) - Standard OpenZeppelin pause behavior is expected

Super Simple Fix

```
// Add whenNotPaused to all 4 functions (4 lines):
function submitOneMinOrders(...) external onlyOperator whenNotPaused {
function closeOneMinOrders(...) public onlyOperator whenNotPaused {
function settleOneMinOrders(...) public onlyOperator whenNotPaused {
function updatePrice(...) external payable onlyOperator whenNotPaused {
```

Effort: <5 minutes (literally just adding the modifier to 4 functions!)

Our Take **Good news:** BaseVolOneHour is already protected!

Reasoning: - Provides defense-in-depth against compromised keys - Makes pause work as expected - Matches BaseVolOneHour's design - Super quick fix

Suggestion: Adding these modifiers would complete the pause protection across all contracts.

Developer:

Please check commit 6bd7fb2326d58047b575d5d6f7fc0682573e449b

Auditor:

VERIFIED FIXED(commit 6bd7fb2326d58047b575d5d6f7fc0682573e449b).

Finding 4: Donation Attack via Strategy - Extreme Inflation Bypasses +1 Protection

Severity: ❌ Critical

Status: Resolved

Description:

`totalAssets()` includes strategy's raw token balance via `strategy.totalAssetsUnderManagement()` -> `balanceOf(address(strategy))`. Anyone can donate tokens directly to strategy to inflate this balance. While OpenZeppelin's +1 formula provides basic protection, extreme donations cause victim deposits to round to zero shares. Attacker profits: deposit 1 wei (1 share) -> donate 1M USDC -> victims deposit 8M (receive 0 shares each) -> attacker withdraws 4.5M USDC = 3.5M profit.

Impact:

Leading to Complete Victim Fund Loss**

Phase 1: Attacker Setup (Empty Vault Exploitation)

- Attacker monitors for newly deployed vault or vault with zero supply
- Deposits minimal amount (1 wei) to become first depositor
- Receives first share, establishing 100% ownership of vault
- No special permissions required - uses standard deposit function

Phase 2: Balance Inflation (Donation Attack)

- Attacker transfers large amount of tokens directly to strategy contract
- Strategy's `balanceOf()` inflates dramatically
- Vault's `totalAssets()` includes this inflated strategy balance
- Total supply remains at 1 share (attacker's ownership)
- Share price artificially inflated to extreme ratio

Phase 3: Victim Trap (Zero Share Minting)

- Multiple victims deposit substantial amounts during normal vault operation
- Each victim's share calculation: `(depositAmount * 2) / inflatedTotalAssets`
- Despite OpenZeppelin +1 protection, extreme inflation ratio causes rounding to zero

- Victims receive exactly 0 shares for their full deposits
- Victim funds enter vault but no ownership tokens minted
- Each victim loses 100% of deposited amount

Phase 4: Attacker Extraction (Profitable Exit)

- Attacker redeems single share for withdrawal
- Withdrawal calculation distributes vault's total assets proportionally
- Attacker receives significant portion of total vault balance
- Net result: Substantial profit from victims' trapped funds

Critical Characteristics:

- **No Admin Required:** Pure permissionless exploit using public functions
- **Undetectable Setup:** Appears as normal first deposit + external transfer
- **Guaranteed Success:** Mathematical certainty victims receive zero shares
- **Irreversible Loss:** Victims cannot withdraw without shares
- **Permanent Damage:** Remaining funds stuck in vault with no owner
- **Scalable Impact:** More victims = higher attacker profit
- **Timing Window:** Effective immediately after vault deployment or emptying

Attack Vector Amplification:

- Works on epoch-based deposits (victims don't immediately see zero shares)
- Compatible with all ERC4626-style vaults using balanceOf() for strategy assets
- Bypasses standard inflation protection when donation ratio is extreme
- Creates permanent insolvency (vault holds funds but insufficient shares to claim)

Source:

- contracts/genesis-vault/libraries/LibGenesisVault.sol (totalAssets, lines 74-76)
- contracts/core/vault/GenesisStrategy.sol (getStrategyIdleAssets, lines 688-690)
- contracts/genesis-vault/libraries/LibGenesisVault.sol (convertToShares, line 131)

Code:

```
// Strategy uses raw balance (GenesisStrategy.sol:689)
function getStrategyIdleAssets() public view returns (uint256) {
    return IERC20(asset()).balanceOf(address(this));  // ⊠ Donation vulnerable
}

// Vault includes strategy (LibGenesisVault.sol:75)
uint256 strategyAssets = s.strategy != address(0)
  ? IGenesisStrategy(s.strategy).totalAssetsUnderManagement()  // ⊠ Inflated by donations
  : 0;

// Shares calculation (LibGenesisVault.sol:131)
return (assets * (supply + 1)) / (totalAssets_ + 1);  // ⊠ +1 insufficient for extreme inflation
```

Proof of Concept:

Attack Steps:

1. Attacker deposits 1 wei to empty vault -> Receives 1 share (100% ownership)
2. totalSupply = 1, totalAssets = 1
3. Attacker sends 1,000,000 USDC directly to strategy contract (donation)
4. totalAssets inflates to 1,000,000,000,001 (strategy balance included)
5. Victim 1 deposits 400,000 USDC -> Shares = `(400K * 2) / 1M = 0.8` -> Rounds to 0
6. Victims 2-20 each deposit 400K USDC -> Each receives 0 shares
7. totalAssets now: 1 + 1M + 8M = 9,000,000 USDC, totalSupply still 1
8. Attacker withdraws 1 share -> Receives `(1 * 9M) / 2 = 4.5M USDC`
9. Result: Attacker profit = 3.5M, Victims lose 8M, 4.5M stuck in vault

Remediation:

- PRIMARY FIX: Track expected balance in strategy

```
uint256 public expectedStrategyBalance;

function _depositToStrategy(uint256 amount) internal {
    expectedStrategyBalance += amount;
}

function getStrategyIdleAssets() public view returns (uint256) {
    return expectedStrategyBalance;  // ⊠ Ignores donations
}
```

- SECONDARY: Minimum first deposit

```
uint256 constant MINIMUM_FIRST_DEPOSIT = 1000e6;  // 1000 USDC
require(s.totalSupply > 0 || assets >= MINIMUM_FIRST_DEPOSIT, "Too small");
```

- TERTIARY: Prevent zero shares

```
uint256 shares = (assets * (supply + 1)) / (totalAssets_ + 1);
require(shares > 0 || assets == 0, "Deposit too small");
```

Discussion:

> *Developer:*
>
> This issue should already be resolved in the latest Strategy implementation.

> *Auditor:*
>
> Great to hear this is resolved! To verify and close this out, could you please share:
>
> 1. **Commit hash** where the fix was implemented
> 2. **File path(s)** of the changed files
>
> We'll take a quick look and confirm everything looks good!

> *Developer:*
>
> Please check 0affbd0657704773ff1455cd11c7fae21e242c51 commit.
>
> To-do: call initializeExpectedBalance() after strategy upgrade

> *Auditor:*
>
> **WHAT IS FIXED**
>
> The team implemented an `expectedBalance` tracking mechanism to prevent donation attacks:
>
> 1. **Storage Variable Added** (GenesisStrategyStorage.sol:46):
>
>    ```
>    uint256 expectedStrategyBalance;
>    ```
>
> 2. **Tracking Functions Added** (GenesisStrategy.sol:818-841):
>
>    ```
>    function _increaseExpectedBalance(uint256 amount) internal
>    function _decreaseExpectedBalance(uint256 amount) internal
>    ```
>
> 3. **14 out of 15 Transfer Functions CORRECTLY Call `_decreaseExpectedBalance()`:**

- Line 187: `_decreaseExpectedBalance(idleBalance);`
- Line 357: `_decreaseExpectedBalance(transferAmount);`
- Line 396: `_decreaseExpectedBalance(fromIdle);`
- Line 1130: `_decreaseExpectedBalance(adjusted);` (BaseVol deposit)
- Line 1145: `_decreaseExpectedBalance(adjusted);` (Morpho deposit)
- Line 1377: `_decreaseExpectedBalance(transferAmount);`
- Line 1407: `_decreaseExpectedBalance(amount);`
- Line 1447: `_decreaseExpectedBalance(amount);`
- Line 1493: `_decreaseExpectedBalance(amount);`
- Line 1571: `_decreaseExpectedBalance(transferAmount);`
- Line 1601: `_decreaseExpectedBalance(amount);`
- Line 1637: `_decreaseExpectedBalance(amount);`
- Line 1661: `_decreaseExpectedBalance(amount);`
- Plus several `_increaseExpectedBalance()` calls for incoming funds

WHAT IS NOT FIXED - CRITICAL BUG

File: `GenesisStrategy.sol`
Function: `processAssetsToWithdraw()` (Lines 290-297)
Bug: Missing `_decreaseExpectedBalance()` call

Current Broken Code:

```
function processAssetsToWithdraw() public whenIdle {
  address _asset = asset();
  uint256 _assetsToWithdraw = getStrategyIdleAssets();

  if (_assetsToWithdraw > 0) {
    IERC20(_asset).safeTransfer(vault(), _assetsToWithdraw);  //  Transfers assets
    //  MISSING: _decreaseExpectedBalance(_assetsToWithdraw);
  }
}
```

REQUIRED REMEDIATION

Fix Location: GenesisStrategy.sol, Line 296 (after the safeTransfer)

Required Change:

```
function processAssetsToWithdraw() public whenIdle {
  address _asset = asset();
  uint256 _assetsToWithdraw = getStrategyIdleAssets();

  if (_assetsToWithdraw > 0) {
    IERC20(_asset).safeTransfer(vault(), _assetsToWithdraw);
    _decreaseExpectedBalance(_assetsToWithdraw);  //  ADD THIS LINE
  }
}
```

ADDITIONAL ISSUE: Manual Initialization Required

The `expectedStrategyBalance` is NOT automatically initialized on upgrade.

Required Action:

```
// After upgrade, owner must call:
strategy.initializeExpectedBalance();
```

Location: GenesisStrategy.sol:852

```
function initializeExpectedBalance() external onlyOwner {
  GenesisStrategyStorage.Layout storage $ = GenesisStrategyStorage.layout();
  uint256 actualBalance = IERC20(asset()).balanceOf(address(this));
  $.expectedStrategyBalance = actualBalance;
  emit ExpectedBalanceInitialized(actualBalance);
}
```

Developer:

Fixed in commit e49af150b97cfcfaba493fe1bbafbac360c4acd0

Auditor:

VERIFIED FIXED(commit e49af150b97cfcfaba493fe1bbafbac360c4acd0).

Finding 5: Duplicate Order IDs - Accounting Corruption And Double-Spending

Severity: Critical

Status: Resolved

Description:

`submitFilledOrders()` only validates the first transaction's ID (line 25) but doesn't check if subsequent transactions have sequential IDs or if order IDs already exist. Operator can submit duplicate orders with the same idx/epoch, adding them multiple times to storage (line 53: `orders.push(order)`). This corrupts accounting and allows double-spending of escrowed funds.

Impact:

- Compromised operator can submit same order multiple times causing users to: (1) lock funds 2x-10x, (2) receive double/multiple payouts if winning, (3) pay multiple fees if losing. Protocol accounting becomes corrupted and insolvent.

Source:

- contracts/basevol/facets/OrderProcessingFacet.sol (submitFilledOrders, lines 21-62)

Code:

```
function submitFilledOrders(FilledOrder[] calldata transactions)
    external nonReentrant onlyOperator {

    // Line 25: ONLY validates FIRST transaction
    if (bvs.lastFilledOrderId + 1 > transactions[0].idx) revert LibBaseVolStrike.InvalidId();

    for (uint i = 0; i < transactions.length; i++) {
        FilledOrder calldata order = transactions[i];

        bvs.clearingHouse.lockInEscrow(...overUser, overAmount...);
        bvs.clearingHouse.lockInEscrow(...underUser, underAmount...);

        // Line 53: No duplicate check before pushing
        FilledOrder[] storage orders = bvs.filledOrders[order.epoch];
        orders.push(order);  // ⊠ Can push same order multiple times
    }
}
```

Proof of Concept:

Attack Steps:

1. Compromised operator submits batch: `[{idx: 100, epoch: 1, ...}]` -> Accepted

2. Operator submits batch: `[{idx: 101, ...}, {idx: 100, ...}]` -> First tx validates (101 > 100), second tx is duplicate
3. Order #100 now exists TWICE in `filledOrders[1]` array
4. When settlement occurs, loop processes order #100 twice
5. If Alice wins: Receives 2x payout (double-spending from escrow)
6. Protocol accounting corrupted, funds drained

Remediation:

- Fix validation logic (line 25)
- Validate ALL transactions are sequential
- Use mapping instead of array to prevent duplicates

Discussion:

> *Developer:*
>
> This occurs when a malicious operator submits an order with an already-used ID during BaseVol order filling. However, this shouldn't be a concern on our end as the operators are whitelisted

> *Auditor:*
>
> We totally get that operators are trusted! The concern here is less about malicious operators and more about protecting against operational hiccups.
>
> **Real-World Scenarios** **1. Network Glitches:** - RPC timeout → operator script retransmits → accidental duplicate - Network hiccup → uncertain if transaction went through → resubmission - These happen even with careful operators!
>
> **2. Compromised Keys (Defense in Depth):** - Even with whitelisting, keys can get phished or breached - Stolen keys have same permissions as legitimate ones
>
> **3. What Could Happen:** - Users' funds get locked multiple times for same order - Accounting gets messy - Could lead to double payouts or other issues

The Defense-in-Depth Approach This is standard practice in financial systems - validate inputs even for trusted roles. It's not about not trusting your team, it's about having safety nets for when things go wrong.

Fix

```
// 5 lines of code:
require(transactions[0].idx == bvs.lastFilledOrderId + 1, "Invalid ID");
for (uint i = 1; i < transactions.length; i++) {
    require(transactions[i].idx == transactions[i-1].idx + 1, "Not sequential");
}
```

Effort: <10 minutes (quick and straightforward!)

Why It Matters: - Catches operational errors (network issues, retries) - Provides defense if keys ever get compromised - Prevents accounting issues - Really simple to add

Bottom line: This is standard defensive programming - catches honest mistakes before they become problems.

Developer:

Commit 2b26ea8926e1a766d37a85482b1173545138bfb5

Auditor:

VERIFIED FIXED(commit 2b26ea8926e1a766d37a85482b1173545138bfb5).

Finding 6: Fee Calculation Overflow

Severity: ✖ Critical

Status: Resolved

Description:

SettlementFacet uses `(10 ** s.decimals)` as divisor when calculating performance and management fees, but fee rates (performanceFee, managementFee) are configured with FLOAT_PRECISION (1e18) scaling. For 6-decimal assets like USDC, this creates a **1 trillion times (1e12) overcharge error**. When any user redeems in a profitable epoch during auto-settlement, the calculated fee is astronomically large, causing `transferFeesToRecipient()` to revert and blocking ALL users' redemptions for that epoch indefinitely. VaultCoreFacet has the CORRECT implementation using FLOAT_PRECISION (lines 776, 779), and developer documentation (genesisvault-erc7540.md lines 527, 531) explicitly shows the correct formula, proving SettlementFacet contains a critical bug. SettlementFacet doesn't even define FLOAT_PRECISION constant, indicating incomplete code migration from monolithic to Diamond pattern.

Impact:

- **CRITICAL - Complete Vault DoS on Any Profitable Redemption**

  Mathematical Impact (USDC Example):

  - User has 100 USDC profit (100e6)
  - performanceFee configured as 0.2e18 (20%)
  - **Expected fee:** `(100e6 * 0.2e18) / 1e18 = 20e6 = 20 USDC`
  - **ACTUAL fee (bug):** `(100e6 * 0.2e18) / 1e6 = 20e18 = 20,000,000,000,000,000,000 USDC`
  - **Multiplier:** 1e18 / 1e6 = **1 trillion times overcharge**

  System-Wide Impact:

  - Every auto-processed redemption in profitable epoch triggers revert
  - `asset.safeTransfer(recipient, 20_billion_USDC)` reverts: "ERC20: transfer amount exceeds balance"
  - Entire `onRoundSettled()` transaction reverts
  - ALL users' redemptions for that epoch permanently blocked
  - No recovery mechanism - requires contract upgrade

- Vault becomes completely unusable in any profit scenario

Attack Vector:

- Unprivileged - any user redemption triggers bug
- Attack cost: ~$1 (minimal redemption request)
- Impact: 100% DoS of all vault withdrawals
- Repeatability: Every profitable epoch fails
- No timelock or delay - immediate permanent impact

Management Fee Impact:

- Management fee shares minted 1 trillion times larger than intended
- Massively dilutes all user shares
- Can overflow uint256, causing additional DoS
- Occurs at every settlement, compounding damage

Source:

- contracts/genesis-vault/facets/SettlementFacet.sol (_calculateAndChargePerformanceFee, line 463)
- contracts/genesis-vault/facets/SettlementFacet.sol (_mintManagementFeeShares, line 396)
- contracts/genesis-vault/facets/VaultCoreFacet.sol (_calculateAndChargePerformanceFee, lines 776, 779) [CORRECT reference implementation]

Code:

```
// SettlementFacet.sol line 463 (BUGGY - Performance Fee)
function _calculateAndChargePerformanceFee(
    address user,
    uint256 withdrawShares,
    uint256 currentSharePrice
  ) internal returns (uint256 feeAmount) {
    // ...
    uint256 profitPerShare = currentSharePrice - userData.waep;
    uint256 totalProfit = (profitPerShare * withdrawShares) / (10 ** s.decimals);

    // ⊠ BUG: Uses (10 ** s.decimals) = 1e6 for USDC instead of FLOAT_PRECISION = 1e18
    feeAmount = (totalProfit * s.performanceFee) / (10 ** s.decimals);

    if (feeAmount > 0) {
      LibGenesisVault.transferFeesToRecipient(feeAmount, "performance");
    }
}

// SettlementFacet.sol line 396 (BUGGY - Management Fee)
function _mintManagementFeeShares() internal {
    // ...
    uint256 feeRate = (s.managementFee * timeElapsed) / (365 days);

    // ⊠ BUG: Uses (10 ** s.decimals) = 1e6 for USDC instead of FLOAT_PRECISION = 1e18
    uint256 feeShares = (currentTotalSupply * feeRate) / (10 ** s.decimals);

    if (feeShares > 0) {
      LibERC20._mint(recipient, feeShares);
    }
}

// VaultCoreFacet.sol lines 776, 779 (CORRECT - For comparison)
uint256 internal constant FLOAT_PRECISION = 1e18;  // Line 22

function _calculateAndChargePerformanceFee(...) internal returns (uint256 feeAmount) {
    // ...
    if (hurdleRateValue > 0) {
        uint256 hurdleThresholdPerShare = (userData.waep * hurdleRateValue) / FLOAT_PRECISION;
        if (profitPerShare > hurdleThresholdPerShare) {
            uint256 excessProfit = (excessProfitPerShare * withdrawShares) / (10 ** s.decimals);
            feeAmount = (excessProfit * s.performanceFee) / FLOAT_PRECISION;  // ⊠ CORRECT!
        }
    } else {
        feeAmount = (totalProfit * s.performanceFee) / FLOAT_PRECISION;  // ⊠ CORRECT!
    }
}

// LibGenesisVault.sol line 347 (No validation)
function transferFeesToRecipient(uint256 amount, string memory feeType) internal {
    if (amount == 0) return;
    // ⊠ NO validation on amount size
    s.asset.safeTransfer(recipient, amount);  // Will revert if insufficient balance
}
```

Proof of Concept:

Attack Steps:

1. Deploy GenesisVault Diamond with USDC (6 decimals)
2. Configure performanceFee = 0.2e18 (20%), managementFee > 0
3. User A deposits 1000 USDC → Receives shares at WAEP = 1.0e6
4. Vault appreciates: totalAssets increases, sharePrice becomes 1.1e6 (10% gain)
5. User A calls `requestRedeem(all shares)` → Request recorded for current epoch
6. Epoch ends, keeper calls `onRoundSettled(epoch)`
7. Settlement flow: `_processRoundSettlement()` -> `_autoProcessEpochRedeems()` -> `_autoProcessUserRedeem(user_A, epoch)`
8. Line 347: Calls `_calculateAndChargePerformanceFee(user_A, shares, 1.1e6)`
9. Line 460: `totalProfit = (0.1e6 * shares) / 1e6 = 0.1e6` (10 cents profit per share)
10. Line 463: **BUGGY calculation:** `feeAmount = (0.1e6 * 0.2e18) / 1e6 = 0.2e18 = 200 trillion USDC`
11. Line 467: `transferFeesToRecipient(200_trillion)` called
12. Line 347 (LibGenesisVault): `asset.safeTransfer(feeRecipient, 200_trillion)` -> **REVERT** (vault balance ~1000 USDC)

13. Entire `onRoundSettled()` reverts, epoch settlement fails
14. ALL users' redemptions for this epoch permanently frozen
15. Repeat for ANY profitable epoch → protocol completely unusable

Remediation:

```
// IMMEDIATE FIX - Add FLOAT_PRECISION constant to SettlementFacet.sol
uint256 internal constant FLOAT_PRECISION = 1e18;

// Fix line 463 (Performance Fee)
// BEFORE:
feeAmount = (totalProfit * s.performanceFee) / (10 ** s.decimals);
// AFTER:
feeAmount = (totalProfit * s.performanceFee) / FLOAT_PRECISION;  // ⊠

// Fix line 396 (Management Fee)
// BEFORE:
uint256 feeShares = (currentTotalSupply * feeRate) / (10 ** s.decimals);
// AFTER:
uint256 feeShares = (currentTotalSupply * feeRate) / FLOAT_PRECISION;  // ⊠

// Add sanity checks to transferFeesToRecipient
function transferFeesToRecipient(uint256 amount, string memory feeType) internal {
    if (amount == 0) return;

    // Sanity check: fee cannot exceed reasonable bounds
    uint256 vaultBalance = s.asset.balanceOf(address(this));
    require(amount <= vaultBalance, "Fee exceeds vault balance");

    // Additional check for performance fees (should never exceed total profit)
    // This catches fee calculation bugs early

    s.asset.safeTransfer(recipient, amount);
    emit FeesTransferred(recipient, amount, feeType);
}

// Centralize fee calculation into shared library
// Move _calculateAndChargePerformanceFee to LibGenesisVault to prevent future divergence

// Add comprehensive unit tests
// Test performance fee with 6-decimal assets (USDC) and various performanceFee values
// Test management fee minting with realistic time periods
// Verify feeAmount <= totalProfit and feeShares <= reasonable percentage of totalSupply

// Document that all fee rates are scaled by FLOAT_PRECISION (1e18), NOT asset decimals
// Add comments: "performanceFee is in 1e18 precision (0.2e18 = 20%)"
```

Discussion:

Developer:

This has been fixed recently.

Auditor:

Awesome! To verify and close this out, could you please share:

1. **Commit hash** where the fix was implemented
2. **File path(s)** of the changed files (specifically SettlementFacet.sol lines 396, 463)

We'll take a quick look to confirm!

Developer:

Commit 3cef75e70b4c9817759123005f2de6db80740270

Auditor:

WHAT IS FIXED

File: `SettlementFacet.sol`

1. **Constant Added** (Line 18):

```
uint256 internal constant FLOAT_PRECISION = 1e18;
```

2. **Performance Fee FIXED** (Line 506):

```
//  CORRECT - Uses FLOAT_PRECISION
feeAmount = (totalProfit * s.performanceFee) / FLOAT_PRECISION;
```

WHAT IS NOT FIXED - PRODUCTION BLOCKER

File: `SettlementFacet.sol`
Function: `_chargeManagementFee()` (Line 437)
Bug: Still uses wrong divisor - causes 1 TRILLION times overcharge

Current Broken Code:

```
// Line 436-437
uint256 feeRate = (s.managementFee * timeElapsed) / (365 days);
uint256 feeShares = (currentTotalSupply * feeRate) / (10 ** s.decimals);  //  WRONG
```

REQUIRED REMEDIATION

Fix Location: SettlementFacet.sol, Line 437

Required Change:

```
// CHANGE FROM:
uint256 feeShares = (currentTotalSupply * feeRate) / (10 ** s.decimals);

// CHANGE TO:
uint256 feeShares = (currentTotalSupply * feeRate) / FLOAT_PRECISION;
```

Developer:

Fixed in commit 40ead7ceff97fa3c7f1101cf391cb86088e1dc40

Auditor:

VERIFIED FIXED(commit 40ead7ceff97fa3c7f1101cf391cb86088e1dc40).

Finding 7: Force Withdrawal Zeroes Entire Balance - Direct Fund Loss

Severity: ❌ Critical

Status: Resolved

Description:

`executeForceWithdraw()` sets user's ENTIRE balance to zero (line 335: `$.userBalances[msg.sender] = 0`) instead of subtracting withdrawal amount. Three critical scenarios: (1) Balance increases during 24h delay → user loses additional funds, (2) Balance decreases → withdrawal reverts and user stuck, (3) Orphaned funds accumulate, which operator can extract via `addUserBalance()` ($100 per call, unlimited calls).

Impact:

- User loses all additional funds received during 24h delay.
- If balance decreases, withdrawal fails (DoS).
- Orphaned funds accumulate from multiple users, which malicious operator can extract by repeatedly calling `addUserBalance(attacker, 100)` to create phantom accounting, then withdrawing.

Source:

- contracts/core/ClearingHouse.sol (executeForceWithdraw, line 335)
- contracts/core/ClearingHouse.sol (addUserBalance, line 471)

Code:

```
// Line 335 - CRITICAL BUG
function executeForceWithdraw() external nonReentrant {
  ForceWithdrawalRequest storage request = $.forceWithdrawalRequests[requestIdx];

  if ($.userBalances[msg.sender] < request.amount + $.withdrawalFee)
        revert InsufficientBalance();

  request.processed = true;
    $.userBalances[msg.sender] = 0;  // ⊠ Should subtract, not zero
  $.treasuryAmount += $.withdrawalFee;
  $.token.safeTransfer(msg.sender, request.amount);
}

// Operator can extract orphaned funds (line 467)
function addUserBalance(address user, uint256 amount) external nonReentrant onlyOperator {
    if (amount > MAX_OPERATOR_CHANGE_AMOUNT) revert AmountExceedsLimit();  // $100 limit
    $.userBalances[user] += amount;  // ⊠ Creates accounting without tokens
}
```

Proof of Concept:

Attack Steps (Scenario 1 - Balance Increase):

1. Alice has 1000 USDC, calls `requestForceWithdraw()` → `request.amount = 999.9`
2. During 24h delay: Bob sends Alice 500 USDC via `depositTo(alice, 500)`
3. Alice balance now: 1500 USDC
4. After 24h: Alice calls `executeForceWithdraw()`
5. Check passes: `1500 >= 999.9 + 0.1` ⊠
6. Line 335: `$.userBalances[alice] = 0` ⊠ (should subtract 1000)
7. Alice receives 999.9 USDC, but balance zeroed completely
8. Result: Alice loses 500 USDC (orphaned in contract)

Attack Steps (Scenario 3 - Operator Extraction):

1. 10 users lose funds via Scenario 1 → 5000 USDC orphaned
2. Malicious operator notices: contract balance > sum of user balances
3. Operator calls `addUserBalance(attackerWallet, 100)` × 50 times
4. Creates 5000 USDC accounting for attacker (no tokens deposited)
5. Attacker calls `withdraw(5000e6)` → Receives orphaned USDC
6. Result: Operator steals 5K from victims' losses

Remediation:

- CRITICAL FIX - Line 335 (one line change) $.userBalances[msg.sender] -= (request.amount + $.withdrawalFee); // ⊠ Subtract instead of zero
- Additional improvements
    1. Add balance change detection uint256 balanceAtRequest = request.amount + $.withdrawalFee; uint256 currentBalance = $.userBalances[msg.sender]; if (currentBalance > balanceAtRequest) { // Return excess to user or keep in balance uint256 excess = currentBalance - balanceAtRequest; emit ExcessFundsDetected(msg.sender, excess); }
    2. Remove or restrict addUserBalance() to prevent phantom accounting

Discussion:

> *Developer:*
>
> Fixed!

Auditor:

Great! To verify and close this out, could you please share:

1. **Commit hash** where the fix was implemented
2. **File path(s)** of the changed files (specifically ClearingHouse.sol line 335)

We'll review quickly to confirm everything looks good!

Auditor:

hash: 02c90058aa0bf06dcfb815fd38d0bb2aea224e49 Issue no longer exists in the codebase

Finding 8: No Validation of Share Price - Can Settle with Price = 0

Severity: ✖ Critical

Status: Resolved

Description:

Zero Share Price Settlement

`onRoundSettled()` calculates sharePrice via `_calculateCurrentSharePrice()` but performs NO validation. If `totalAssets()` returns 0 (due to `trySub` underflow when `claimable > idle + strategyAssets`), sharePrice becomes 0. This zero price is stored without checks, causing division-by-zero in auto-processing functions, permanently preventing epoch settlement.

Impact:

If sharePrice settles as 0, all auto-processing reverts with division-by-zero. Epoch settlement permanently fails. All user funds for that epoch are locked with no recovery. Can occur naturally from strategy losses causing `totalAssets()` underflow.

Source:

- contracts/genesis-vault/facets/SettlementFacet.sol (onRoundSettled, lines 77-79)
- contracts/genesis-vault/facets/SettlementFacet.sol (_calculateCurrentSharePrice, lines 96-122)
- contracts/genesis-vault/libraries/LibGenesisVault.sol (totalAssets, line 82)

Code:

```
// SettlementFacet.sol:77-79
function onRoundSettled(uint256 epoch) external onlyKeeper {
    uint256 sharePrice = _calculateCurrentSharePrice();
    roundData.sharePrice = sharePrice;  // NO VALIDATION - can be 0
    roundData.isSettled = true;
    _processRoundSettlement(epoch);  // Will fail if sharePrice = 0
}

// SettlementFacet.sol:112-122
function _calculateCurrentSharePrice() internal view returns (uint256) {
    uint256 vaultTotalAssets = LibGenesisVault.totalAssets();  // Can be 0
    uint256 vaultTotalSupply = s.totalSupply + _totalPendingRedeemShares();
    return (vaultTotalAssets * (10 ** s.decimals)) / vaultTotalSupply;  // No +1 protection
}

// LibGenesisVault.sol:82
(, uint256 totalAssetsResult) = (idle + strategyAssets).trySub(claimable);
// If claimable > (idle + strategyAssets), trySub returns (false, 0) -> totalAssets = 0

// Auto-processing division by zero (SettlementFacet.sol:314)
uint256 sharesToMint = (claimableAssets * (10 ** s.decimals)) / roundData.sharePrice;  // Division by
    0
```

Proof of Concept:

Attack Steps (Natural Occurrence):

1. Vault state: idle = 500K, strategyAssets = 500K, claimable = 1.1M
2. Keeper calls `onRoundSettled(epoch 1000)`
3. Function calls `_calculateCurrentSharePrice()`
4. Calculates `totalAssets()`: `(idle + strategyAssets).trySub(claimable)`
5. Calculation: `(500K + 500K).trySub(1.1M)` -> `1M.trySub(1.1M)` -> underflow
6. `trySub` returns `(false, 0)` -> `totalAssets = 0`
7. `sharePrice = (0 * 1e6) / totalSupply = 0`
8. Line 79: `roundData.sharePrice = 0` -> NO VALIDATION, stored
9. User tries to claim: `shares = (assets * 1e6) / 0` -> DIVISION BY ZERO
10. All users locked out for epoch 1000, funds permanently stuck

Remediation:

```
function onRoundSettled(uint256 epoch) external onlyKeeper {
    uint256 sharePrice = _calculateCurrentSharePrice();

    // ADD VALIDATION
    require(sharePrice > 0, "Share price cannot be zero");
    require(sharePrice >= MIN_SHARE_PRICE, "Share price too low");
    require(sharePrice <= MAX_SHARE_PRICE, "Share price too high");

    // Optional: Deviation check
    if (s.lastSharePrice > 0) {
        uint256 deviation = /* calculate deviation */;
        require(deviation <= 50, "Deviation exceeds 50%");
    }
    s.lastSharePrice = sharePrice;

    roundData.sharePrice = sharePrice;  // Now guaranteed valid
    // ... rest of function
}
```

Discussion:

> *Developer:*
>
> This has been fixed recently.

> *Auditor:*
>
> Excellent! To verify and close this out, could you please share:

> 1. **Commit hash** where the fix was implemented
> 2. **File path(s)** of the changed files (specifically SettlementFacet.sol lines 77-79, 96-122)
>
> We'll take a quick look and confirm!

> *Developer:*
>
> commit ffb163ac36dd3c77856742d35a1791fcd54490bd

> *Auditor:*
>
> **VERIFIED FIXED**(commit ffb163ac36dd3c77856742d35a1791fcd54490bd).

Finding 9: Zero Price Manipulation

Severity: ✖ Critical

Status: Resolved

Description:

Two price-setting mechanisms with different security levels:

(1) **Manual Override**: `setManualRoundEndPrices()` accepts ANY value without validation (line 130: `problemRound.endPrice[data.productId] = data.price`). Operator can set price=0 (all "Under" win) or max (all "Over" win). Product 0 has partial protection in `settleFilledOrders()`, but products 1+ are fully vulnerable. (

2) **Pyth Oracle**: Signature verified but NO timestamp/staleness checks, accepting hours/days-old valid data.

Impact:

- Operator can force 100% of one side to lose by setting extreme prices.
- Manual override bypasses all validation.
- Stale oracle data creates unfair settlements.
- Multi-product vulnerability allows targeting non-BTC markets while BTC appears normal.

Source:

- contracts/basevol/facets/RoundManagementFacet.sol (setManualRoundEndPrices, line 130)
- contracts/basevol/facets/RoundManagementFacet.sol (_processPythLazerPriceUpdate, lines 236-304)
- contracts/libraries/PythLazer.sol (verifyUpdate, lines 67-103)
- contracts/basevol/facets/OrderProcessingFacet.sol (settleFilledOrders, line 70)

Code:

```
// Manual Override (NO VALIDATION)
function setManualRoundEndPrices(PriceData[] calldata priceData, ...) external onlyOperator {
    for (uint i = 0; i < priceData.length; i++) {
        problemRound.endPrice[data.productId] = data.price;  // ANY value accepted
    }
}

// Partial Protection (Product 0 only)
function settleFilledOrders(uint256 epoch, ...) public onlyOperator {
    if (round.startPrice[0] == 0 || round.endPrice[0] == 0)  // Checks product 0
        revert InvalidRoundPrice();
    // Products 1+ not checked
}

// Oracle (Missing Timestamp Check)
function _processPythLazerPriceUpdate(...) internal {
    (bytes memory payload, ) = bvs.pythLazer.verifyUpdate(...);  // Signature verified
    // NO timestamp validation - accepts stale data
    if (priceFound && price > 0) {  // Zero filtered for oracle
        tempData[validCount] = PriceData({...});
    }
}
```

Proof of Concept:

Attack Steps (Manual Override - Zero Price):

1. Round 500 has 1000 users (500 OVER, 500 UNDER) on ETH (product 1)
2. Malicious operator has 200 UNDER positions
3. Real ETH price: $3,100 → OVER should win
4. Operator calls `setManualRoundEndPrices([{productId: 1, price: 0}])`
5. Settlement calculates: `strikePrice > 0` → UNDER wins (all UNDER positions win)
6. 500 OVER users lose funds, 500 UNDER users (including operator) win
7. Operator profits unfairly, ETH protection bypassed (only BTC has checks)

Attack Steps (Stale Oracle):

1. BTC was $60,000 at 10:00 AM (Pyth signed data valid)
2. BTC now $55,000 at 11:00 AM (8% drop)
3. Most users bet OVER expecting recovery
4. Operator submits 1-hour-old Pyth data ($60k) via `executeRound()`
5. Signature valid, Signer valid, NO timestamp check
6. Settlement uses $60k instead of $55k
7. Unfair advantage to OVER positions

Remediation:

- Add validation to manual override
- Add multi-sig requirement for manual override
- Add timelock (24 hours) for manual prices
- Add timestamp validation to oracle

- Fix multi-product protection

Discussion:

> *Developer:*
>
> This relates to BaseVol — it only becomes an issue with a malicious operator, so it doesn't really affect us.

> *Auditor:*
>
> Thanks for the response! We'd like to break this down into two separate issues - they have different implications.
>
> **Vector 1: Manual Override (Trust-Based)**
>
> ```
> function setManualRoundEndPrices(PriceData[] calldata priceData, ...) {
>     problemRound.endPrice[data.productId] = data.price;  // Accepts any value
> }
> ```
>
> - Currently accepts any price value (including 0 or max)
> - Product 0 has checks, but products 1+ don't
>
> This one is more about trusting the operator, we hear you on that!
>
> **Vector 2: Stale Oracle (Operational Issue)**
>
> ```
> function _processPythLazerPriceUpdate(...) {
>     (bytes memory payload, ) = bvs.pythLazer.verifyUpdate(...);
>     // Missing timestamp freshness check
> }
> ```
>
> **This can happen without any bad intent:** - Operator accidentally uses cached/old Pyth data - Network delays cause stale data to be submitted - Timing issues between price fetch and submission - Creates unintentionally unfair settlements

Why Vector 2 Matters Timestamp validation is pretty standard in oracle-using protocols. It's less about trust and more about catching operational timing issues before they affect users.

Fixes **Vector 1:**

```
require(data.price > 0, "Invalid price");
uint256 maxDeviation = (startPrice * 50) / 100;
require(data.price >= startPrice - maxDeviation, "Deviation exceeds 50%");
```

Vector 2:

```
uint64 publishTime = _extractPublishTime(payload);
require(block.timestamp - publishTime <= MAX_PRICE_AGE, "Stale price");
```

Effort: 10-15 minutes (straightforward additions)

Why We Think This Matters: - Vector 2 (stale oracle) - standard best practice, catches timing issues - Vector 1 (manual override) - adds bounds checking for safety - Protects products 1+ which currently have no checks - Quick to implement

Our recommendation: Vector 2 (timestamp check) is really the key one - it's standard practice and catches honest timing mistakes. Vector 1 is more optional depending on your trust model.

Developer:

Commit 720c9513d16fd94ebdc9fa91457cfd8bd80b0658

Auditor:

VERIFIED FIXED(commit 720c9513d16fd94ebdc9fa91457cfd8bd80b0658).

Finding 10: Anyone Can Inflate Strategy Valuation by Depositing to BaseVolManager via Public depositTo

Severity: ❗ High

Status: Resolved

Description:

The strategy's `getBaseVolAssets()` returns `baseVolManager.totalClearingHouseBalance()` which includes BOTH normal deposits (tracked by strategy) AND external deposits (via public `depositTo()`). Since Clearing-House's `depositTo()` is public and unrestricted, anyone can deposit tokens directly under BaseVolManager's account: `clearingHouse.depositTo(baseVolManager, amount)`. This inflates `totalClearingHouseBalance()` -> inflates `getBaseVolAssets()` -> inflates vault's `totalAssets()` -> manipulates share price.

Combined with H20 (accounting divergence), this creates a direct manipulation vector where attackers can:

(1) Inflate share price before depositing -> Receive more shares than deserved -> Withdraw after price corrects -> Profit from other users,

(2) Deflate share price before redeeming (if possible to reduce) -> Receive more assets per share -> Drain vault. The attack requires capital (tokens must be deposited) but attacker controls BaseVolManager's balance without strategy's knowledge.

Impact:

HIGH - Direct Share Price Manipulation Leading to Fund Theft. Attacker with capital can manipulate vault share price:

(1) Inflation Attack: Deposit 100K USDC to baseVolManager -> Share price increases -> Attacker deposits 1M USDC to vault -> Receives fewer shares than deserved (other users benefit) -> Withdraw 100K from ClearingHouse -> Share price corrects -> Loss to attacker BUT if timed with victim deposits, can profit.

(2) Arbitrage: Monitor share price vs actual assets -> When divergence occurs, exploit for profit.

(3) Accounting Chaos: Combined with H20, creates permanent accounting errors.

(4) Rebalancing Errors: Strategy makes wrong decisions based on inflated BaseVol balance. Capital requirement: Moderate (requires depositing tokens but can withdraw later). Profit potential: Depends on timing and other users' deposits/withdrawals during manipulation window.

Source:

- contracts/core/ClearingHouse.sol (depositTo, lines 163-169)
- contracts/core/vault/BaseVolManager.sol (totalClearingHouseBalance, line 237)
- contracts/core/vault/GenesisStrategy.sol (getBaseVolAssets, line 755)

Code:

```
// ClearingHouse.sol line 163-169 - PUBLIC, unrestricted
function depositTo(address user, uint256 amount) external nonReentrant {
  ClearingHouseStorage.Layout storage $ = ClearingHouseStorage.layout();
  $.token.safeTransferFrom(msg.sender, address(this), amount);
  $.userBalances[user] += amount;  // ⊠ Can target baseVolManager
  emit Deposit(user, msg.sender, amount, $.userBalances[user]);
}

// BaseVolManager.sol line 237 - Includes ALL balances
function totalClearingHouseBalance() public view returns (uint256) {
  return clearingHouse.totalUserBalances(address(this));
  // = userBalances[this] + userEscrowBalances[this]
  // ⊠ Includes externally deposited amounts
}

// GenesisStrategy.sol line 752-756 - Used for share price
function getBaseVolAssets() public view returns (uint256) {
  if (address($.baseVolManager) == address(0)) return 0;
  return $.baseVolManager.totalClearingHouseBalance();
  // ⊠ Returns inflated value if external deposits exist
}

// Share price calculation uses this
totalAssets = idle + getBaseVolAssets() + getMorphoAssets();
sharePrice = (totalAssets * 1e6) / totalSupply;
```

Proof of Concept:

Steps to Reproduce (Share Price Manipulation):

1. Initial state: Vault operational with totalAssets = 1M USDC, totalSupply = 1M shares, sharePrice = 1.0
2. Actor with capital calls `ClearingHouse.depositTo(baseVolManager, 100_000e6)` (public unrestricted function)
3. Transaction executes: 100K USDC transferred to ClearingHouse, credited to baseVolManager account
4. BaseVolManager balance increases: Now has 100K USDC balance (not tracked by strategy)
5. Strategy's getBaseVolAssets() called during next deposit
6. Returns: `baseVolManager.totalClearingHouseBalance()` = 100K USDC
7. Vault's totalAssets calculation: idle + getBaseVolAssets() + getMorphoAssets() = 1M + 100K = 1.1M
8. Share price calculation: `1.1M * 1e6 / 1M shares = 1.1e6` (inflated by 10%)
9. Inflated sharePrice active: 1.1 instead of actual 1.0
10. Legitimate user deposits 1.1M USDC expecting fair share allocation
11. Shares minted calculation: `1.1M assets / 1.1 price = 1M shares`

12. User underpaid: Receives 1M shares instead of deserved 1.1M shares (100K share shortfall)
13. Actor withdraws external deposit: Calls `ClearingHouse.withdraw()` to retrieve 100K
14. BaseVolManager balance returns to 0 -> Vault totalAssets returns to normal = 1.1M actual
15. Share price recalculates: `1.1M / (1M old + 1M new shares) = 1.1 / 2 = 0.55`
16. User loss: Deposited 1.1M but received shares worth less due to temporary price inflation
17. Compounding with H9: If combined with accounting divergence (H9), creates permanent share price distortion
18. Attack timing: Can be coordinated with known large deposits or vault rebalancing events

Remediation:

```
// OPTION 1: Track only strategy-deposited amounts (best)
// In BaseVolManager, maintain separate accounting:
uint256 public strategyDepositedAmount;  // Only track what strategy deposited

function depositToClearingHouse(uint256 amount) external authCaller(strategy()) {
  strategyDepositedAmount += amount;
  // ... existing logic ...
}

function withdrawFromClearingHouse(uint256 amount) external authCaller(strategy()) {
  strategyDepositedAmount -= amount;
  // ... existing logic ...
}

function totalClearingHouseBalance() public view returns (uint256) {
  return strategyDepositedAmount;  // ⊠ Ignore external deposits
}

// OPTION 2: Whitelist depositTo recipients (prevents targeting baseVolManager)
mapping(address => bool) public depositToAllowed;

function depositTo(address user, uint256 amount) external nonReentrant {
  require(depositToAllowed[user] || user == msg.sender, "Recipient not allowed");
  // ... existing logic ...
}

// OPTION 3: Restrict depositTo to operator only
function depositTo(address user, uint256 amount) external nonReentrant onlyOperator {
  // ... existing logic ...
}

// OPTION 4: Use withdrawable balance only (exclude escrowed)
function totalClearingHouseBalance() public view returns (uint256) {
  return clearingHouse.userBalances(address(this));
  // Excludes escrowed, but still includes external deposits
  // Combine with Option 1 for full protection
}

// RECOMMENDED: Implement Option 1 + add monitoring
function getExternalDepositAmount() external view returns (uint256) {
  uint256 actual = clearingHouse.totalUserBalances(address(this));
  uint256 tracked = strategyDepositedAmount;
  return actual > tracked ? actual - tracked : 0;
}

emit ExternalDepositDetected(getExternalDepositAmount());
```

Discussion:

Developer:

It's technically true that if someone deposits directly into baseVolManager through the ClearingHouse, it artificially raises the share price.

But as it is not possible for the attacker to withdraw from baseVolManager, there are no adverse effects

Auditor:

Good point about the withdrawal restriction! However, we'd like to discuss a few concerns that remain even without direct profit extraction.

What Still Concerns Us 1. Share Price Manipulation - External deposit inflates `totalClearingHouseBalance()` - Share price temporarily increases - Users depositing during this window receive fewer shares - Value gets redistributed even if attacker can't profit directly

2. Accounting Synchronization - Strategy's internal `utilizedAssets` tracking ≠ actual `totalClearingHouseBalance()` - This creates reporting discrepancies - Could affect rebalancing decisions - Compounds with another accounting issue we found

3. Possible Scenarios - Griefing: Someone sacrificing capital to harm protocol (we've seen this in other DeFi projects) - Competitive disruption: While shorting protocol elsewhere - Accidental: Users depositing to wrong address - Timing attacks: Coordinating with large known deposits

The Economic Irrationality Point We hear you that it's economically irrational! The concern is that DeFi has seen cases where griefing attacks, competitive tactics, or even mistakes create these exact scenarios.

Fix

```
uint256 public strategyDepositedAmount;

function totalClearingHouseBalance() public view returns (uint256) {
    return strategyDepositedAmount;  // Ignores external deposits
}
```

Effort: <30 minutes | **Benefit:** Cleans up accounting and prevents manipulation

Why We Think This Matters: - Share price can still be manipulated temporarily - Accounting stays synchronized - Prevents griefing and accidental issues - Quick fix

Our take: The withdrawal restriction definitely reduces the severity, but the accounting fix would still be valuable to keep things clean!

Developer:

As depositTo() function as been removed, this is no longer an issue

Auditor:

VERIFIED FIXED(commit 720c9513d16fd94ebdc9fa91457cfd8bd80b0658).

Finding 11: Base Initialization Function Lacks One-Time-Only Protection + Missing Input Validation

Severity: ● High

Status: Resolved

Description:

The `DiamondInit.init()` function has TWO critical issues:

(1) Missing Input Validation: NO validation on any of 8 input parameters. If `intervalSeconds = 0`, causes **division-by-zero** in epoch calculations, completely DoSing the protocol. If addresses are `address(0)`, causes silent failures.

(2) Re-initialization Vulnerability (Design Flaw): The `init()` function has NO one-time-only protection (no OpenZeppelin `initializer` modifier, no initialized flag check). While the Diamond standard (EIP-2535) allows `diamondCut` to call initialization functions multiple times during upgrades (this is by design), **base initialization functions should be protected to run only once**. Best practice is to have separate init functions for upgrades (e.g., `InitV2`, `InitV3`) that only initialize NEW storage/facets. However, this codebase uses a SINGLE `init()` function that can be called repeatedly via `diamondCut([], diamondInit, newCalldata)` to **overwrite all critical base parameters** (token, oracle, admin, operator, timestamps). This creates both a **recovery mechanism** (owner can fix bad init) AND a **critical vulnerability** (compromised owner can brick live protocol by resetting parameters maliciously or accidentally). **Note:** The ability for `diamondCut` to accept init parameters is STANDARD Diamond behavior for upgrades; the BUG is that the base init function doesn't have proper one-time-only guards.

Impact:

HIGH - Protocol Bricking + Re-initialization Attack Surface.

Impact 1 - Bad Initialization: If `intervalSeconds = 0`: All calls to `getCurrentEpoch()` revert with division by zero. Every user function (deposit, withdraw, trade) becomes unusable. Recovery possible via re-initialization (see Impact 2). Other risks: `address(0)` parameters cause silent failures, excessive `commissionfee` drains all user funds, wrong `startTimestamp` breaks epoch logic.

Impact 2 - Re-initialization Risk: Owner can call `diamondCut()` with `DiamondInit` anytime to overwrite ALL critical parameters:

- Change token address -> breaks all balances, deposits fail
- Change oracle address -> breaks price feeds, settlements fail
- Change admin/operator -> access control takeover

- Change `intervalSeconds` -> breaks epoch calculations
- Change `startTimestamp` -> corrupts all epoch data
- Positive: Can fix bad initialization
- Negative: If owner compromised or makes error, can brick live protocol with user funds locked

Source:

- contracts/upgradeInitializers/DiamondInit.sol (init function, lines 23-78)
- contracts/facets/DiamondCutFacet.sol (line 14-21)

Code:

```
// ISSUE 1: No Input Validation (DiamondInit.sol lines 23-78)
function init(
    address _usdcAddress,
    address _oracleAddress,
    address _adminAddress,
    address _operatorAddress,
    uint256 _commissionfee,
    address _clearingHouseAddress,
    uint256 _startTimestamp,
    uint256 _intervalSeconds
) external {
    // NO VALIDATION AT ALL!
    // NO access control (anyone can call directly)
    // NO initializer protection (can be called multiple times)

    bvs.token = IERC20(_usdcAddress);  // Can be address(0)!
    bvs.oracle = IPyth(_oracleAddress);  // Can be address(0)!
    bvs.clearingHouse = IClearingHouse(_clearingHouseAddress);  // Can be address(0)!
    bvs.intervalSeconds = _intervalSeconds;  // Can be 0! -> DIVISION BY ZERO!
    bvs.startTimestamp = _startTimestamp;  // Can be far future!
    bvs.commissionfee = _commissionfee;  // No MAX check (can be 100%+)!
}

// ISSUE 2: Re-initialization Possible (DiamondCutFacet.sol lines 14-21)
function diamondCut(
    FacetCut[] calldata _diamondCut,
    address _init,
    bytes calldata _calldata
) external override {
    LibDiamond.enforceIsContractOwner();  // Only owner
    LibDiamond.diamondCut(_diamondCut, _init, _calldata);
    // Can call with DiamondInit address MULTIPLE TIMES
    // Overwrites all storage values on each call
}

// LibDiamond.sol line 153 - Executes init via delegatecall
(bool success, ) = _init.delegatecall(_calldata);
// NO check if already initialized!
```

Proof of Concept:

Steps to Reproduce (Division-by-Zero Scenario):

1. Prepare deployment: Diamond contract ready to deploy

2. Create initialization parameters with `_intervalSeconds = 0` (error in deployment script or manual override)
3. Deploy Diamond with `DiamondInit.init()` call via `diamondCut`
4. Initialization executes: Line 198: `bvs.intervalSeconds = 0` (NO validation)
5. Deployment completes successfully - no errors during init
6. User attempts to interact: calls `deposit()` or any function requiring epoch
7. Internal call chain: `deposit()` -> `getCurrentEpoch()` -> Line 203: `(block.timestamp - startTimestamp) / intervalSeconds`
8. Division by zero: `(block.timestamp - startTimestamp) / 0` -> Transaction reverts
9. Every user function reverts: deposit, withdraw, requestDeposit, requestRedeem, settlement, all view functions
10. Protocol completely unusable - no user can interact
11. Admin realizes error: Checks if can fix via re-initialization
12. Discovery: Owner CAN call `diamondCut` again with correct parameters (see Scenario 2)

Scenario 2 - Re-initialization Attack (Compromised Owner):

1. Protocol live in production: Users have deposited $10M USDC
2. Diamond deployed at address 0xDiamond, currently operational
3. Original initialization: `intervalSeconds = 3600` (1 hour), token = USDC, oracle = Pyth
4. Attack: Owner's private key compromised OR owner makes critical error
5. Attacker/malicious owner prepares new init calldata with malicious params:
   - `_intervalSeconds = 0` (DoS)
   - `_oracleAddress = attackerContract` (price manipulation)
   - `_adminAddress = attackerAddress` (access control takeover)
6. Calls `diamond.diamondCut([], diamondInitAddress, maliciousCalldata)`
7. Transaction executes: `LibDiamond.enforceIsContractOwner()` passes (owner/attacker)
8. initializeDiamondCut performs delegatecall to `DiamondInit.init(maliciousParams)`
9. ALL STORAGE OVERWRITTEN:
   - intervalSeconds: 3600 -> 0 (breaks epoch calculation)
   - oracle: Pyth -> attackerContract (breaks price feeds)
   - admin: legitimate -> attacker (access control taken over)
10. Next user interaction: ANY function calling `getCurrentEpoch()` -> Division by zero

11. Protocol completely DoSed - $10M locked
12. Attacker controls admin functions via new admin address
13. Result: Live protocol with user funds bricked by re-initialization

Remediation:

Separate base initialization from upgrade initialization + Add input validation:

```
// UNDERSTANDING: Diamond standard allows calling init functions via diamondCut multiple times
// This is CORRECT for upgrades. The FIX is to protect BASE init from re-initialization.

// FIX 1: Add initialization flag to BaseVolStrikeStorage
struct DiamondStorage {
    bool initialized;  // Add this
    IERC20 token;
    // ... rest of storage
}

// FIX 2: Protect BASE init() from re-initialization (ONE-TIME ONLY)
function init(...) external {
    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();

    // CRITICAL: Ensure this is first-time initialization only
    require(!bvs.initialized, "Base config already initialized");

    // Validate addresses
    require(_usdcAddress != address(0) && _usdcAddress.code.length > 0, "Invalid USDC");
    require(_oracleAddress != address(0) && _oracleAddress.code.length > 0, "Invalid Oracle");
    require(_clearingHouseAddress != address(0), "Invalid ClearingHouse");
    require(_adminAddress != address(0) && _operatorAddress != address(0), "Invalid admin/operator");

    // Validate critical numeric params
    require(_intervalSeconds >= 60 && _intervalSeconds <= 7 days, "Invalid interval");
    require(_startTimestamp >= block.timestamp - 7 days && _startTimestamp <= block.timestamp + 1 days
    , "Invalid startTime");
    require(_commissionfee <= 1000, "Fee too high"); // Max 10%

    // ... rest of initialization

    // CRITICAL: Mark as initialized (prevents future calls)
    bvs.initialized = true;
}

// ALTERNATIVE: Use OpenZeppelin Initializable pattern
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract DiamondInit is Initializable {
    function init(...) external initializer {  // OZ prevents re-initialization
        // ... validation and initialization
    }
}

// FIX 3: Create SEPARATE init functions for upgrades (BEST PRACTICE)
// contracts/upgradeInitializers/DiamondInitV2.sol
contract DiamondInitV2 {
    function initV2(address newFacet, uint256 newParameter) external {
        // This initializes ONLY NEW features/facets
        // Does NOT touch base configuration (token, oracle, etc.)

        LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();
        require(bvs.initialized, "Base init must run first");  // Ensure base is initialized

        // Initialize new features only
        bvs.newFeatureParameter = newParameter;
        // ... other new-feature initialization
    }
}

// USAGE:
// Initial deployment: diamondCut([facets], diamondInit, init_calldata)  <- Runs once
// Upgrade 1: diamondCut([newFacets], diamondInitV2, initV2_calldata)   <- Safe to run
// Upgrade 2: diamondCut([newFacets], diamondInitV3, initV3_calldata)   <- Safe to run

// FIX 4: Add emergency recovery ONLY if absolutely necessary
function emergencyReset(...) external {
    LibDiamond.enforceIsContractOwner();
    require(isPaused(), "Must pause first");  // Require protocol paused
    require(totalValueLocked() == 0, "Must have 0 TVL");  // Require funds withdrawn
    require(getUserCount() == 0, "Must have no users");  // Require no active users

    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();
    bvs.initialized = false;  // Allow re-initialization only in EXTREME emergency

    emit EmergencyResetAuthorized(msg.sender, block.timestamp);
    // WARNING: This should ONLY be used in catastrophic scenarios
}

// KEY TAKEAWAY:
// diamondCut accepting init functions = STANDARD Diamond behavior (correct)
// Base init() without one-time-only protection = BUG (needs fixing)
// Separate init functions for upgrades (InitV2, InitV3) = BEST PRACTICE
```

Discussion:

> *Developer:*
>
> This would only be exploitable if an attacker already had deployment privileges. If that's compromised, the system is already effectively compromised.

> *Auditor:*
>
> We understand the access control point! We'd like to break this into two separate concerns - one's about access (you're right on that), the other's about catching honest mistakes.
>
> **Issue 1: Missing Input Validation (The Key One)** **Current Code:**
>
> ```
> function init(..., uint256 _intervalSeconds) external {
>     bvs.intervalSeconds = _intervalSeconds;  // No validation
> }
> ```
>
> **The concern:** If `intervalSeconds = 0` gets passed, it causes division-by-zero in all epoch calculations - protocol would be DoS'd from launch.

How this could happen (without any compromise): - Typo in deployment script - Wrong config file loaded - Copy-paste error from testnet settings - Environment variable mix-up - Human error during deployment

Impact: Protocol unusable from day 1, would need upgrade to fix.

This is standard defensive programming - catch configuration mistakes before they become problems, regardless of who's deploying!

Issue 2: Re-initialization Risk (Secondary Concern) **Current Code:**

```
// Can be called multiple times via diamondCut
diamondCut([], diamondInit, newCalldata)  // Could overwrite base parameters
```

The scenario: Base init could accidentally get called again during upgrades, resetting live parameters.

How this could happen: - Owner makes mistake during upgrade → Calls wrong init function - Operational confusion about which init to use - (Less likely but possible: compromised owner key)

Impact: Could reset live protocol parameters (token, oracle, intervals).

The Defense-in-Depth Perspective The "if owner compromised" point is valid! Our thinking is more about: - Catching **accidental** mistakes during upgrades - Standard OpenZeppelin pattern (Initializable) does this - Protects operations team from honest errors - It's less about malicious compromise, more about operational safety

Fix

```
// Add initialization flag
bool initialized;

function init(..., uint256 _intervalSeconds) external {
    require(!initialized, "Already initialized");
    require(_intervalSeconds >= 60 && _intervalSeconds <= 7 days, "Invalid interval");
    require(_usdcAddress != address(0), "Invalid USDC");
    // ... other validations
    initialized = true;
}
```

> **Effort:** 10 minutes | **Benefit:** Catches deployment mistakes and prevents accidental re-init
>
> **Why We Think This Matters:** - Input validation catches deployment mistakes (division-by-zero = instant DoS) - Init guard prevents operational upgrade errors - Standard defensive programming practices - Really quick to add
>
> **Bottom line:** Issue #1 (input validation) is the critical one - catches honest configuration mistakes. Issue #2 (re-init) is more of a safety net for upgrade operations.

> *Developer:*
>
> Commit 202dba37d1597088453570078a528204bc5b48a5

> *Auditor:*
>
> **VERIFIED FIXED**(commit 202dba37d1597088453570078a528204bc5b48a5).

Finding 12: GenesisStrategy utilizedAssets Accounting Can Diverge from Actual Asset Positions

Severity: ❗ High

Status: Resolved

Description:

The strategy tracks `$.utilizedAssets` as internal accounting for total assets deployed across BaseVol and Morpho. However, this accounting can diverge from actual positions due to:

(1) **External Deposits**: Anyone can call `depositTo(baseVolManager, amount)` in ClearingHouse to inflate BaseVolManager's balance without updating strategy's utilizedAssets,

(2) **Profit/Loss Reconciliation Timing**: The `baseVolWithdrawCompletedCallback()` (lines 977-996) attempts to reconcile by comparing `actualClearingHouseBalance` vs `$.strategyBalance`, then updating `$.utilizedAssets`. However, if external deposits occurred, this reconciliation treats the inflated balance as "profit" and increases utilizedAssets incorrectly,

(3) **Concurrent Operations**: Multiple operations updating utilizedAssets without atomic checks can cause race conditions,

(4) **Withdrawal Failure Recovery**: If withdrawal partially fails or callbacks are not invoked, utilizedAssets remains inflated. The result is permanent accounting divergence where reported strategy assets don't match actual on-chain positions.

Validation: Code review confirms accounting update paths in `GenesisStrategy.sol` lines 970-1000 can diverge.

Impact:

HIGH - Incorrect Share Price Calculations Leading to Value Loss/Theft. Share price depends on `totalAssets()` which includes strategy's `getTotalUtilizedAssets()` -> `utilizedAssets`.

If utilizedAssets is inflated: (1) Share price artificially high -> New depositors receive fewer shares than deserved -> Value transfer from new to existing users, (2) Share price artificially low -> Redeemers receive fewer assets than deserved -> Value remains in vault, benefiting remaining users, (3) Strategy rebalancing decisions based on wrong data -> Suboptimal capital allocation, (4) Accounting audits fail -> Reported PnL incorrect -> Protocol appears profitable when losing or vice versa. Divergence accumulates over time and becomes permanent without manual reconciliation.

Source:

- contracts/core/vault/GenesisStrategy.sol (baseVolWithdrawCompletedCallback, lines 970-1000)

- contracts/core/vault/GenesisStrategy.sol (utilizedAssets tracking, multiple locations)

Code:

```
// GenesisStrategy.sol lines 970-1000 - Reconciliation Logic
function baseVolWithdrawCompletedCallback(uint256 amount, bool success)
    external authCaller(baseVolManager()) {
  GenesisStrategyStorage.Layout storage $ = GenesisStrategyStorage.layout();

  if (success) {
    // Check actual balance for profit/loss calculation
    uint256 actualClearingHouseBalance = $.baseVolManager.totalClearingHouseBalance();

    // Update utilizedAssets considering profit or loss
    if (actualClearingHouseBalance != $.strategyBalance) {
      if (actualClearingHouseBalance > $.strategyBalance) {
        // Treats as "Profit"
        uint256 profit = actualClearingHouseBalance - $.strategyBalance;
        $.utilizedAssets += profit;  // ⊠ But this could be external deposit, not profit!
      } else {
        // Loss
        uint256 loss = $.strategyBalance - actualClearingHouseBalance;
        if (loss >= $.utilizedAssets) {
          $.utilizedAssets = 0;
        } else {
          $.utilizedAssets -= loss;
        }
      }
      $.strategyBalance = actualClearingHouseBalance;
    }

    // Then subtract withdrawal
    $.utilizedAssets -= amount;  // ⊠ Could underflow if accounting wrong
    $.strategyBalance -= amount;
  }
}
```

Proof of Concept:

Steps to Reproduce (Accounting Divergence):

1. Initial state: Strategy deployed with utilizedAssets = 1M USDC, actual BaseVol balance = 1M USDC (synchronized)
2. External actor calls `ClearingHouse.depositTo(baseVolManager, 100_000e6)` (public function)
3. Transaction executes: BaseVolManager's balance in ClearingHouse increases to 1.1M USDC
4. Strategy's internal accounting unchanged: utilizedAssets remains 1M USDC
5. **Divergence created:** Actual balance (1.1M) ≠ Tracked balance (1M)
6. Time passes: Keeper triggers normal rebalancing operation
7. Keeper calls `keeperRebalance()` -> triggers withdrawal: `withdrawFromClearingHouse(50_000e6)`
8. Withdrawal completes successfully -> Callback invoked: `baseVolWithdrawCompletedCallback(50K, true)`
9. Callback line 671: Reads `actualClearingHouseBalance = 1.1M - 50K = 1.05M`
10. Callback line 674: Compares with stored `$.strategyBalance = 1M`

11. Line 675: Condition true: `1.05M > 1M` -> Interprets as “profit”
12. Line 677: Calculates profit: `1.05M - 1M = 50K`
13. Line 678: Updates accounting: `$.utilizedAssets += 50K` -> utilizedAssets = 1.05M
14. Line 692: Subtracts withdrawal: `$.utilizedAssets -= 50K` -> utilizedAssets = 1M (final)
15. **Result:** utilizedAssets = 1M but actual BaseVol balance = 1.05M
16. **Divergence:** 50K USDC difference permanently embedded in accounting
17. Over multiple cycles and external deposits, divergence accumulates
18. **Impact:** Share price calculations use wrong utilizedAssets -> Incorrect valuations

Remediation:

```
// 1. Track expected balances explicitly, ignore external deposits
struct StrategyAccounting {
  uint256 expectedBaseVolBalance;  // What we deposited
  uint256 expectedMorphoBalance;   // What we deposited
  uint256 utilizedAssets;          // Sum of expected balances
}

// 2. On deposit, increment expected
function depositToClearingHouse(uint256 amount) {
  $.expectedBaseVolBalance += amount;
  $.utilizedAssets += amount;
  // ... execute deposit ...
}

// 3. On withdrawal, decrement expected
function withdrawFromClearingHouse(uint256 amount) {
  $.expectedBaseVolBalance -= amount;
  $.utilizedAssets -= amount;
  // ... execute withdrawal ...
}

// 4. Reconcile only tracked balances, exclude external
function getTotalUtilizedAssets() public view returns (uint256) {
  return $.utilizedAssets;  // Use tracked amount, not balanceOf()
}

// 5. Add view function to detect divergence
function getAccountingDivergence() external view returns (int256) {
  uint256 actual = getBaseVolAssets() + getMorphoAssets();
  return int256(actual) - int256($.utilizedAssets);
}

// 6. Add emergency reconciliation (admin only)
function reconcileAccounting() external onlyOwner {
  uint256 actualTotal = getBaseVolAssets() + getMorphoAssets();
  emit AccountingReconciled($.utilizedAssets, actualTotal);
  $.utilizedAssets = actualTotal;
}

// 7. Prevent external inflation by restricting baseVolManagerDeposit
// Already restricted to onlyBaseVolManager, ensure depositTo can't target it
```

Discussion:

> *Developer:*
>
> Commit e74fd4348540347d567489cfd7c08ccad8878a6a

> *Auditor:*
>
> **VERIFIED FIXED**(commit e74fd4348540347d567489cfd7c08ccad8878a6a).

Finding 13: Infinite Token Approvals Amplify Vulnerabilities in Multiple Contracts

Severity: ❗ High

Status: Resolved

Description:

TWO infinite approvals create amplified risk:

(1) Vault -> Strategy (Internal): `setStrategy()` approves `type(uint256).max` to strategy (line 133), remaining active until next strategy change. Strategy uses this approval (lines 167, 924). Any strategy vulnerability amplifies to total vault drainage.

(2) MorphoVaultManager -> Morpho (EXTERNAL - HIGHER RISK): `initialize()` approves `type(uint256).max` to external Morpho protocol (line 90). This is MORE CONCERNING because Morpho is an external, potentially upgradeable protocol. If Morpho vault is upgraded maliciously or has a vulnerability, ALL approved funds can be drained. While deposits have `maxStrategyDeposit` limit (10M USDC), the approval itself is unlimited and persists indefinitely.

Impact:

HIGH - Amplifies Vulnerabilities in Both Internal and External Contracts

Impact (1) - Vault -> Strategy (Internal):

- Amplifies any strategy bug into total vault drainage
- If strategy has reentrancy/access control/logic vulnerability -> ALL vault funds drainable
- Approval persists for months/years until strategy change
- Compromised operator can extract full balance via strategy

Impact (2) - MorphoVaultManager -> Morpho (EXTERNAL - HIGHER RISK):

- **CRITICAL EXTERNAL DEPENDENCY:** Approval to protocol outside team's control
- If Morpho vault is upgradeable -> malicious upgrade can drain all approved funds
- If Morpho has vulnerability (e.g., reentrancy, access control) -> funds at risk
- While deposits limited to 10M USDC (maxStrategyDeposit), approval is UNLIMITED
- Approval never revoked -> persists indefinitely across all deposits
- Strategy deposits ~1M, but approval allows draining entire MorphoVaultManager balance

- **Risk Amplification:** External protocol changes outside audit scope can introduce vulnerabilities

Source:

- contracts/genesis-vault/facets/GenesisVaultAdminFacet.sol (setStrategy, line 133) - Vault -> Strategy
- contracts/core/vault/GenesisStrategy.sol (lines 167, 924) - Uses vault approval
- contracts/core/vault/MorphoVaultManager.sol (initialize, line 90) - Manager -> Morpho (EXTERNAL)

Code:

```
// (1) VAULT -> STRATEGY: Infinite approval (GenesisVaultAdminFacet.sol:133)
s.asset.approve(_strategy, type(uint256).max);  // Infinite, never auto-revoked

// Strategy uses vault approval (GenesisStrategy.sol:167, 924)
_asset.safeTransferFrom(address(_vault), address(baseVolManager()), amount);
IERC20(asset()).safeTransferFrom(address(vault()), address($.morphoVaultManager), toMorpho);

// (2) MORPHO MANAGER -> MORPHO VAULT: Infinite approval (MorphoVaultManager.sol:90)
// MORE CONCERNING: Approves to EXTERNAL protocol
function initialize(address _morphoVault, address _strategy) external initializer {
    // ... initialization ...
    $.asset.approve(_morphoVault, type(uint256).max);  // Infinite to external Morpho
}

// Manager uses Morpho approval (MorphoVaultManager.sol:104)
$.morphoVault.deposit(amount, address(this));  // Uses infinite approval
// If Morpho vault upgraded maliciously -> can drain all approved funds
```

Proof of Concept:

Risk Amplification Demonstration:

Scenario 1 - Internal Strategy Vulnerability Amplification:

1. Initial state: Vault deployed with 10M USDC total value locked
2. Owner calls `setStrategy(newStrategy)` during protocol upgrade
3. Transaction executes line 133: `s.asset.approve(_strategy, type(uint256).max)`
4. Infinite approval granted and remains active indefinitely
5. Time passes: Months/years of operation
6. **Risk Event:** Strategy contract contains a vulnerability (e.g., missing access control on withdrawal function, reentrancy bug, logic error)
7. Attacker discovers and exploits the strategy vulnerability
8. Strategy contract uses the infinite vault approval to call `vault.asset.transferFrom(vault, attacker, 10_000_00`
9. Transfer succeeds due to unlimited approval

10. **Result:** Entire vault balance (10M USDC) drained - vulnerability impact amplified from strategy-scope to vault-scope

Scenario 2 - External Protocol Risk (Morpho) - HIGHER CONCERN:

1. MorphoVaultManager.initialize() called during deployment
2. Line 90 executes: `$.asset.approve(_morphoVault, type(uint256).max)`
3. Infinite approval granted to external Morpho protocol (MetaMorpho vault)
4. Protocol operates normally: Strategy deposits 5M USDC to Morpho over time
5. Approval persists: Never revoked, always unlimited despite 10M maxStrategyDeposit limit
6. **Risk Event:** Morpho protocol is upgradeable - malicious upgrade deployed OR vulnerability discovered
7. Attacker exploits Morpho vulnerability or malicious upgrade logic
8. Morpho contract calls `morphoManager.asset.transferFrom(morphoManager, attacker, balance)`
9. Transfer succeeds due to unlimited approval
10. **Result:** Entire MorphoVaultManager balance drained (potentially entire protocol's Morpho allocation)
11. **Critical Difference:** External protocol outside team's control - cannot prevent malicious upgrades or vulnerabilities

Remediation:

```
// FIX (1) - VAULT -> STRATEGY: Just-In-Time (JIT) approvals
function approveStrategyTransfer(uint256 amount) internal {
    s.asset.approve(s.strategy, amount);  // Limited approval
}

// Strategy requests approval before each operation
function utilize(uint256 amount) external onlyVault {
    vault.approveStrategyTransfer(amount);
    asset.transferFrom(vault, destination, amount);
    vault.approveStrategyTransfer(0);  // Revoke after
}

// FIX (2) - MORPHO MANAGER -> MORPHO: JIT approvals (CRITICAL for external protocol)
// MorphoVaultManager.sol - Replace infinite approval
function initialize(address _morphoVault, address _strategy) external initializer {
    // ... initialization ...
    // REMOVE: $.asset.approve(_morphoVault, type(uint256).max);
    // Don't approve during init - approve per transaction
}

function depositToMorpho(uint256 amount) external authCaller(strategy()) {
    // ... validation ...

    // JIT approval - approve exact amount needed
    $.asset.approve(address($.morphoVault), amount);
    $.morphoVault.deposit(amount, address(this));
    $.asset.approve(address($.morphoVault), 0);  // Revoke after

    // ... rest of logic
}

```

```
// ALTERNATIVE for both: Capped approval with periodic refresh
uint256 constant MAX_STRATEGY_APPROVAL = 1_000_000e6;  // 1M USDC cap
uint256 constant MAX_MORPHO_APPROVAL = 1_000_000e6;    // 1M USDC cap

// Vault -> Strategy
s.asset.approve(_strategy, MAX_STRATEGY_APPROVAL);  // Limited damage

// Morpho Manager -> Morpho
$.asset.approve(_morphoVault, MAX_MORPHO_APPROVAL);  // Limited to 1M max loss

// Add function to refresh/revoke if needed
function refreshMorphoApproval(uint256 newAmount) external onlyOwner {
    require(newAmount <= MAX_MORPHO_APPROVAL, "Exceeds max");
    $.asset.approve(address($.morphoVault), newAmount);
}

// BEST PRACTICE for external protocols:
// - JIT approvals (approve before each use, revoke after)
// - OR capped approvals with monitoring/refresh mechanism
// - NEVER infinite approval to external/upgradeable contracts
// - Consider Morpho multisig timelock before trusting infinite approval
```

Discussion:

Developer:

Commit b5be3830e6d088baff68b5ed29e101c93dc45fc4

Auditor:

VERIFIED FIXED(commit b5be3830e6d088baff68b5ed29e101c93dc45fc4).

Finding 14: Memory vs Storage Bug - Order Accounting Corruption

Severity: High

Status: Resolved

Description:

`releaseEpochEscrow()` uses `memory` instead of `storage` for FilledOrder reference (line 151), causing `order.isSettled = true` (line 179) to only modify memory copy, not storage. Orders permanently show as unsettled. ClearingHouse escrow checks prevent infinite drain, but accounting remains corrupted.

Impact:

Orders never marked as settled in storage, breaking `countUnsettledFilledOrders()` and analytics. Escrow checks prevent fund loss, but accounting corruption persists. Missing guard check allows repeated execution attempts.

Source:

- contracts/basevol/facets/RoundManagementFacet.sol (releaseEpochEscrow, line 151)

Code:

```
function releaseEpochEscrow(uint256 epoch) external onlyOperator {
    FilledOrder[] storage orders = bvs.filledOrders[epoch];

    for (uint i = 0; i < orders.length; i++) {
        FilledOrder memory order = orders[i];  // MEMORY instead of storage

        if (!order.isSettled) {
                bvs.clearingHouse.releaseFromEscrow(...);
            order.isSettled = true;  // Updates memory only, not storage
        }
    }
}
```

Proof of Concept:

- Deploy protocol and execute epoch 1 with 3 filled orders (6 users total: 3 overUser, 3 underUser)
- Orders are in escrowed state: `orders[0].isSettled = false`, `orders[1].isSettled = false`, `orders[2].isSettled = false`
- Operator calls `releaseEpochEscrow(1)` to release funds
- Function executes: Loops through orders array
- For each order: Creates memory copy of order struct (line 151: `FilledOrder memory order`)

- Sets `order.isSettled = true` in memory (line 179)
- Memory update does NOT persist to storage
- Transaction completes successfully
- Check storage: All orders still show `isSettled = false` in storage
- Operator calls `releaseEpochEscrow(1)` again (no guard check prevents this)
- ClearingHouse validates escrow balances -> Reverts with "InsufficientBalance" (escrow already empty)
- Result: Accounting permanently corrupted - orders marked as unsettled but funds already released

Remediation:

```
// Change line 151 from memory to storage
FilledOrder storage order = orders[i];  // Storage pointer

// Add guard check at function start
require(!round.isSettled, "Epoch already settled");

// Fix missing isSettled in branch 1 (line 161)
     if (order.overUser == order.underUser) {
         bvs.clearingHouse.releaseFromEscrow(...);
    order.isSettled = true;  // Add this
}
```

Discussion:

Developer:

In BaseVol, when setting order.isSettled = true;, the order is in memory, not storage, so the change isn't persisted. That's accurate — it's just a reference variable, so it doesn't affect execution. Fixed.

Auditor:

Great catch on your end! To verify and close this out, could you please share:

1. **Commit hash** where the fix was implemented
2. **File path(s)** of the changed files (specifically RoundManagementFacet.sol line 151)

We'll take a quick look to confirm!

Developer:

Commit 26c73da62d988ea7f1bc51b217e2784df9a0a64a

> *Auditor:*
>
> **VERIFIED FIXED**(commit 26c73da62d988ea7f1bc51b217e2784df9a0a64a).

Finding 15: Unbounded Loop DoS in `_settleFilledOrders`

Severity: ❗ High

Status: Resolved

Description:

The `_settleFilledOrders()` function loops through ALL orders for an epoch without pagination or gas limit checks. With 1000+ orders, the function will exceed the block gas limit (30M), causing all settlement transactions to revert and permanently preventing the round from being settled. Each order settlement costs ~100k gas (external calls to ClearingHouse). With 1000 orders, total gas required is 100M, far exceeding the block limit.

Impact:

HIGH - Settlement DoS + Permanent Fund Lock. Round cannot be settled via `executeRound()` or `setManualRoundEndPrices()`. Funds permanently locked in escrow (unless operator uses `releaseEpochEscrow`). System effectively DoS'd at scale. Maximum ~300 orders per epoch before hitting gas limit.

Source:

- contracts/basevol/facets/RoundManagementFacet.sol (`_settleFilledOrders`, lines 306-319)
- contracts/basevol/facets/RoundManagementFacet.sol (called from `executeRound` line 91, `setManualRoundEndPrices` line 141)

Code:

```
// Line 306-319
function _settleFilledOrders(Round storage round) internal {
    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();

    if (round.epoch == 0 || round.startTimestamp == 0 || round.endTimestamp == 0) return;

    uint256 collectedFee = 0;
    FilledOrder[] storage orders = bvs.filledOrders[round.epoch];

    for (uint i = 0; i < orders.length; i++) {  //  Unbounded loop
        FilledOrder storage order = orders[i];
        collectedFee += LibBaseVolStrike.settleFilledOrder(round, order);
        // Each settle: ~100k gas (external calls, state updates)
    }
    // Block gas limit: 30M -> Maximum ~300 orders before DoS
    // With 1000 orders: 100M gas -> EXCEEDS LIMIT -> REVERT
}
```

Remediation:

- Implement pagination for settlement: Add `batchSize` parameter to `_settleFilledOrders()`.

- Process orders in batches of 200-300.
- Allow multiple settlement transactions per epoch.
- Track settlement progress in storage (`settledOrderIndex`).
- Add `settleOrdersBatch(epoch, startIndex, endIndex)` function for manual batching.

Discussion:

> *Developer:*
>
> Commit 71bdee56f93625e7f8ba192e7c26dad7df343493

> *Auditor:*
>
> **VERIFIED FIXED**(commit 71bdee56f93625e7f8ba192e7c26dad7df343493).

Finding 16: Unbounded Loops Over User Epochs - DoS Attack

Severity: ❗ High

Status: Resolved

Description:

Critical functions loop over `userDepositEpochs` arrays which are unbounded and grow indefinitely. Active users accumulate 100s-1000s of epochs, causing massive gas consumption that exceeds block limit, permanently DoS-ing their ability to claim.

Impact:

HIGH - Permanent User DoS. Active users with many epochs cannot call `deposit` / `mint` / `withdraw` / `redeem` , funds become permanently inaccessible, no recovery mechanism.

Source:

- contracts/genesis-vault/facets/VaultCoreFacet.sol (`deposit` , line 342)
- contracts/genesis-vault/facets/VaultCoreFacet.sol (`mint` , line 402)
- contracts/genesis-vault/facets/VaultCoreFacet.sol (`withdraw` , line 477)
- contracts/genesis-vault/facets/VaultCoreFacet.sol (`redeem` , line 560)

Code:

```
// Line 342 - deposit() loops over ALL user epochs
for (uint256 i = 0; i < userEpochs.length && remainingAssets > 0; i++) {
    // If userEpochs.length = 1000, gas exceeds block limit
}
```

Proof of Concept:

```
**Steps to Reproduce (Active User Lockout):**

1. GenesisVault deployed and operational
2. Active user Alice interacts with vault regularly over extended period
3. Each interaction adds epoch to user's depositEpochs array:
   - Week 1: 50 deposit epochs
   - Month 1: 200 deposit epochs
   - Month 6: 1000+ deposit epochs accumulated
4. Alice attempts to claim: calls `deposit(1000e6, alice, alice)` to claim pending deposits
5. Function executes line 342: `for (uint256 i = 0; i < userEpochs.length && remainingAssets > 0; i++)`
6. userEpochs.length = 1000 -> Loop must iterate 1000 times
7. Each iteration: Read storage, process epoch, update state (~100k gas per iteration)
8. Total gas required: 1000 x 100k = 100M gas
9. Block gas limit: 30M gas (Base network standard)
10. **Transaction exceeds gas limit** -> Reverts with "Out of Gas"
11. Alice tries all claiming functions: `deposit()`, `mint()`, `withdraw()`, `redeem()` -> All revert
```

```
12. **Alice permanently locked out** - cannot access any of her deposited funds
13. No recovery mechanism: No pagination, no epoch cleanup, no admin rescue
14. **Result:** Active user funds permanently inaccessible due to accumulated epochs
```

Remediation:

Implement pagination (`maxEpochsToProcess` parameter), add epoch cleanup after claiming, or use mapping instead of arrays.

Discussion:

> *Developer:*
>
> Yes, large array loops can exceed gas limits — but these functions aren't used in our service.

> *Auditor:*
>
> We understand the service architecture! The thing is, these are public ERC4626/ERC7540 standard functions that users can call directly, regardless of how your service uses them.
>
> **The Functions We're Talking About**
>
> ```
> function deposit(uint256 assets, address receiver) public returns (uint256 shares)
> function mint(uint256 shares, address receiver) public returns (uint256 assets)
> function withdraw(uint256 assets, address receiver, address owner) public returns (uint256 shares
>     )
> function redeem(uint256 shares, address receiver, address owner) public returns (uint256 assets)
> ```
>
> These are part of the standard interface - users and integrations will expect them to work.
>
> **Why This Still Matters** **1. Users Can Call Directly** - Any user can interact with these functions on-chain - Wallet integrations use these - Aggregators and external protocols call these - No service layer needed
>
> **2. Standard Compliance** - Protocol implements ERC4626/ERC7540 - These standards require working functions - Similar to the earlier discussion about the 50-epoch limit - Users expect standard behavior
>
> **3. Active Users Hit This Naturally (Not Just Attacks)** - User makes 100+ deposits over 6 months (normal active user) - Array grows with each epoch - User calls `withdraw()` to claim - Gas limit exceeded - This happens through normal usage, not attacks!

4. **These Are Recovery Functions** - When auto-settlement has issues, users need these - If your service doesn't use them, users must be able to - Broken recovery = users stuck

Impact Natural DoS (Not Attack): - Active user with 100+ epochs accumulated - Attempts to claim via standard functions - Exceeds gas limit - Funds permanently inaccessible

Actual Attack: - Attacker intentionally creates many small deposits - Forces their own array to grow - Prevents themselves from being able to claim - Can grief other users through similar mechanisms

Fix

```
function withdraw(uint256 assets, address receiver, address owner, uint256 maxEpochs)
    public returns (uint256 shares) {
    uint256 processed = 0;
    for (uint256 i = 0; i < userEpochs.length && processed < maxEpochs; i++) {
        // Process epoch
        processed++;
    }
}
```

Effort: 30 minutes | **Benefit:** Makes standard functions work for active users

Why This Matters: - Standard functions need to work for compliance - Active users hit limits through normal usage (not attacks) - Users can call these directly regardless of service layer - These serve as recovery when auto-settlement has issues

Our recommendation: Adding pagination (`maxEpochs` parameter) would let active users claim in multiple transactions while maintaining gas efficiency!

Developer:

Commit 07bc03bc526d3e67f6f3f4e4565db7fcc8f35fe9

Auditor:

VERIFIED FIXED(commit 07bc03bc526d3e67f6f3f4e4565db7fcc8f35fe9).

Finding 17: Withdrawal Request Spam - Unbounded Array Growth Causes Operational DoS

Severity: ❗ High

Status: Resolved

Description:

requestWithdrawal() only validates current balance at request time but does NOT reserve funds or limit request count. Attacker with minimal balance (e.g., 1.1 USDC) can submit thousands of withdrawal requests for same funds, causing unbounded array growth. Each request passes validation since balance never decreases during request creation. Operator must process each request individually (approve or reject), creating asymmetric cost: attacker pays gas for request creation (~50-80k per request), operator pays gas for approval/rejection (~50k+ per request) PLUS storage bloat costs.

No batch rejection function exists - only `withdrawBatch()` for approvals. With 10,000 requests, operator faces 10,000 individual transactions or expensive batch scripts. Griefing ratio: 2.5x-25x operator cost vs attacker cost.

Impact:

HIGH - Asymmetric Griefing & Operational DoS

Attack Economics:

- Attacker capital: 100 USDC (fully recoverable)
- Attacker cost: 200M gas for 10k requests ≈ $20-$200
- Operator cost: 495M gas for 9,910 rejections ≈ $50-$500
- Griefing ratio: **2.5x - 25x** (operator pays more)
- Repeatability: Unlimited - attack repeatable daily with same capital

System Impact:

- Storage bloat: 10k requests = 1.6 MB permanent blockchain data
- Operational overhead: Manual review required for each request
- Legitimate requests buried in spam queue
- Monitoring strain: `getWithdrawalRequests()` returns max 100, requires pagination through thousands
- User experience degradation: Delayed withdrawal processing
- Trust erosion: Users experience unexplained delays

Attack Scenario (10,000 requests):

- Attacker deposits 100 USDC
- Calls `requestWithdrawal(1e6)` 10,000 times
- Each passes validation: 100e6 >= 1e6 + 0.1e6
- Array grows to 10,000 entries (balance unchanged)
- Operator approves first ~90 requests (until balance exhausted)
- Remaining 9,910 requests revert on approval
- Operator must individually reject each or leave unprocessed
- Cost: Operator pays 2.5x-25x attacker's cost

Source:

- contracts/core/ClearingHouse.sol (`requestWithdrawal` , lines 233-251)
- contracts/core/ClearingHouse.sol (`validWithdrawal` modifier, lines 103-108)
- contracts/core/ClearingHouse.sol (`approveWithdrawal` , lines 253-265)
- contracts/core/ClearingHouse.sol (`rejectWithdrawal` , lines 267-280)
- contracts/storage/ClearingHouseStorage.sol (`withdrawalRequests` array, line 24)

Code:

```
// requestWithdrawal (lines 233-251) - No reservation
function requestWithdrawal(uint256 amount)
    external nonReentrant validWithdrawal(msg.sender, amount)
    returns (WithdrawalRequest memory) {

    WithdrawalRequest memory request = WithdrawalRequest({
        idx: $.withdrawalRequests.length,
        user: msg.sender,
        amount: amount,
        processed: false,
        message: "",
        created: block.timestamp
    });

    $.withdrawalRequests.push(request);  // ⊠ Unbounded growth
    emit WithdrawalRequested(msg.sender, amount);
    return request;
}

// validWithdrawal modifier (lines 103-108) - Point-in-time check only
modifier validWithdrawal(address user, uint256 amount) {
    if (amount == 0) revert InvalidAmount();
    if ($.userBalances[user] < amount + $.withdrawalFee) revert InsufficientBalance();
    _;  // ⊠ Does NOT reserve funds, does NOT track cumulative pending
}

// Storage (ClearingHouseStorage.sol line 24) - No per-user tracking
WithdrawalRequest[] withdrawalRequests;  // ⊠ Unbounded array
// ⊠ NO: mapping(address => uint256) pendingWithdrawalTotal
// ⊠ NO: mapping(address => uint256) userRequestCount

// approveWithdrawal (lines 253-265) - Individual processing only
function approveWithdrawal(uint256 idx) external nonReentrant onlyOperator {
    if (idx >= $.withdrawalRequests.length) revert InvalidIdx();
    WithdrawalRequest storage request = $.withdrawalRequests[idx];
    if (request.processed) revert RequestAlreadyProcessed();
    if ($.userBalances[request.user] < request.amount + $.withdrawalFee)
        revert InsufficientBalance();  // ⊠ Will revert for spam requests
    // ... process withdrawal
}

// rejectWithdrawal (lines 267-280) - Individual processing only, no batch function
function rejectWithdrawal(uint256 idx, string calldata reason)
    external nonReentrant onlyOperator {
    // ⊠ Must be called individually for each spam request
    // ⊠ NO batch rejection function available
}
```

Proof of Concept:

- Initial state: ClearingHouse operational, withdrawal fee = 0.1 USDC
- Attacker deposits 100 USDC via `deposit()` -> Balance: 100 USDC
- Attacker prepares spam attack: Writes script to call `requestWithdrawal(1e6)` repeatedly
- Loop iteration 1: `requestWithdrawal(1e6)` called
- `validWithdrawal` modifier checks: `100e6 >= 1e6 + 0.1e6`
- Request created and pushed to array: `$.withdrawalRequests.length = 1`
- Balance not reserved - still 100 USDC available
- Loop iterations 2-10,000: Repeat previous steps
- Each request passes validation (balance unchanged)
- Final state: `$.withdrawalRequests.length = 10,000`, user balance still 100 USDC
- Attacker gas cost: 10,000 × 20k gas ≈ 200M gas (~$20-$200 at 10-100 gwei)
- Operator processes requests: calls `approveWithdrawal(0)`
- First ~90 requests succeed: Transfer 1 USDC each until balance exhausted
- Request 90+: `approveWithdrawal(90)` -> Check fails: `0 < 1e6 + 0.1e6` -> REVERT "InsufficientBalance"
- Remaining 9,910 requests must be rejected: Operator must call `rejectWithdrawal()` 9,910 times
- No batch rejection function available
- Operator cost: 9,910 × 50k gas ≈ 495M gas (~$50-$500 at 10-100 gwei)
- Result: Operator pays 2.5x-25x more than attacker - asymmetric griefing
- Attack repeatable: Attacker withdraws funds and repeats daily

Remediation:

```
// PRIMARY FIX: Track cumulative pending amounts per user
mapping(address => uint256) public pendingWithdrawalTotal;

modifier validWithdrawal(address user, uint256 amount) {
    if (amount == 0) revert InvalidAmount();
    uint256 requiredAmount = amount + $.withdrawalFee;
    // ⊠ Check total pending + new request
    require(
        $.userBalances[user] >= $.pendingWithdrawalTotal[user] + requiredAmount,
        "Insufficient balance for pending + new"
    );
    _;
}

function requestWithdrawal(uint256 amount) external {
    // ... validation ...
    $.withdrawalRequests.push(request);
    $.pendingWithdrawalTotal[msg.sender] += amount + $.withdrawalFee;  // ⊠ Track
    emit WithdrawalRequested(msg.sender, amount);
}

function approveWithdrawal(uint256 idx) external onlyOperator {
    // ... existing logic ...
    $.pendingWithdrawalTotal[request.user] -= request.amount + $.withdrawalFee;  // ⊠ Decrease
}

function rejectWithdrawal(uint256 idx, string calldata reason) external {
    // ... existing logic ...
    $.pendingWithdrawalTotal[request.user] -= request.amount + $.withdrawalFee;  // ⊠ Decrease
}

// SECONDARY FIX: Add per-user request limits
mapping(address => uint256) public userPendingRequestCount;
uint256 constant MAX_PENDING_REQUESTS_PER_USER = 10;

function requestWithdrawal(uint256 amount) external {
    require(
        $.userPendingRequestCount[msg.sender] < MAX_PENDING_REQUESTS_PER_USER,
        "Too many pending requests"
    );
    // ... rest of logic ...
    $.userPendingRequestCount[msg.sender]++;
}

// TERTIARY FIX: Add batch rejection function
function rejectWithdrawalBatch(
    uint256[] calldata indices,
    string calldata reason
) external nonReentrant onlyOperator {
    for (uint256 i = 0; i < indices.length; i++) {
        uint256 idx = indices[i];
        if (idx >= $.withdrawalRequests.length) continue;
        WithdrawalRequest storage request = $.withdrawalRequests[idx];
        if (request.processed) continue;

        request.processed = true;
        request.message = reason;
        $.pendingWithdrawalTotal[request.user] -= request.amount + $.withdrawalFee;
        $.userPendingRequestCount[request.user]--;
        emit WithdrawalRejected(request.user, request.amount);
    }
}

// ALTERNATIVE: Reserve balance on request creation
function requestWithdrawal(uint256 amount) external {
    // ⊠ Lock funds immediately
    $.userBalances[msg.sender] -= (amount + $.withdrawalFee);
    $.pendingWithdrawals[msg.sender] += (amount + $.withdrawalFee);
    // On approval: transfer from pending
    // On rejection: return to userBalances
}
```

Discussion:

> *Developer:*
>
> Seems to concern unused functions in ClearingHouse. Fixed.

> *Auditor:*
>
> Excellent! To verify and close this out, could you please share:
>
> 1. **Commit hash** where the fix was implemented
> 2. **File path(s)** of the changed files (specifically ClearingHouse.sol lines 233-280)
>
> We'll take a quick look and confirm everything looks good!

> *Auditor:*
>
> **VERIFIED FIXED**(commit 02c90058aa0bf06dcfb815fd38d0bb2aea224e49).

Finding 18: Commission Fee Mid-Epoch Change Affects Pending Orders

Severity: ● Medium

Status: Resolved

Description:

`setCommissionfee()` takes immediate effect without epoch transition logic. Orders submitted with expected fee rate X can settle with fee rate Y if admin changes commission mid-epoch. The commission fee is read from global storage (`bvs.commissionfee`) at settlement time, NOT stored with individual orders at submission time. This creates unfairness: Users submit orders seeing "2% commission fee" in UI, but if admin changes fee to 5% before settlement, users pay the NEW 5% fee despite submitting under the OLD 2% terms. No timelock, no epoch-based transitions, no grandfathering of pending orders. This violates user expectations and can be exploited via front-running: Admin sees large profitable settlements pending, increases fee before processing to extract more revenue.

Impact:

MEDIUM - User Deception + Unfair Treatment + Front-Running Risk. Three failure modes:

(1) User Deception: User submits order when UI shows 200 basis points (2%) commission. Admin changes fee to 500 basis points (5%) before settlement. User expects to pay 2% -> Actually pays 5% -> 2.5x more than expected. No warning, no consent.

(2) Same-Epoch Unfairness: Two users submit identical orders in same epoch at same price. Admin changes fee mid-epoch. First user settled before change pays 2%, second user settled after change pays 5% -> Unfair treatment for identical trades in same epoch.

(3) Front-Running Opportunity: Admin/operator sees pending settlements with large payouts (e.g., 1M USDC profit distribution). Admin calls `setCommissionfee(5000)` increasing fee from 2% to 50% (MAX). Immediately settles orders. Protocol extracts 50% instead of 2% = additional $480k revenue from unsuspecting users. While requires admin action (not external exploit), creates trust issue and poor UX. Users cannot predict final costs.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (setCommissionfee, lines 76-80)
- contracts/basevol/facets/OrderProcessingFacet.sol (settlement uses global fee)

Code:

```
// BaseVolAdminFacet.sol lines 76-80 - Immediate effect
function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    // ⊠ Has maximum validation (good)
    if (_commissionfee > MAX_COMMISSION_FEE) revert InvalidCommissionFee();

    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();

    // ⊠ NO EPOCH TRANSITION LOGIC
    // ⊠ NO TIMELOCK
    // ⊠ Applies immediately to ALL pending orders
    bvs.commissionfee = _commissionfee;  // Global storage

    emit CommissionFeeUpdated(_commissionfee);
}

// OrderProcessingFacet.sol - Settlement reads CURRENT fee
function settleFilledOrder(
    Round storage round,
    FilledOrder storage order
) internal returns (uint256) {
    // ... settlement logic ...

    // ⊠ Reads CURRENT global fee, not fee at order submission
    uint256 commission = (settlementAmount * bvs.commissionfee) / 10000;

    // If commissionfee changed between order submission and settlement:
    // User expected old fee but pays new fee

    return commission;
}

// FilledOrder struct - NO fee field
struct FilledOrder {
    address overUser;
    address underUser;
    uint256 productId;
    uint256 escrowAmount;
    bool isSettled;
    // ⊠ NO: uint256 commissionFeeAtSubmission
}
```

Proof of Concept:

Alice submits order when fee=2% (UI shows “2% commission”). Admin changes fee to 5% before settlement. Settlement reads current global fee (5%) not submission fee (2%) -> Alice pays 5% = 5000 USDC instead of expected 2% = 2000 USDC -> overpaid 3000 USDC (150%) without consent.

Remediation:

```
// FIX 1: Store fee with order at submission (Recommended)
struct FilledOrder {
    address overUser;
    address underUser;
    uint256 productId;
    uint256 escrowAmount;
    bool isSettled;
    uint256 commissionFeeAtSubmission;  // ⊠ Add this field
}

function submitOrder(...) external {
    // ... order creation logic ...

    // ⊠ Store fee at submission time
    order.commissionFeeAtSubmission = bvs.commissionfee;
}

function settleFilledOrder(
    Round storage round,
    FilledOrder storage order
) internal returns (uint256) {
    // ... settlement logic ...

    // ☒ Use fee from order submission, not current global fee
    uint256 commission = (settlementAmount * order.commissionFeeAtSubmission) / 10000;

    return commission;
}

// FIX 2: Epoch-based fee transitions (Alternative)
mapping(uint256 => uint256) public epochCommissionFee;  // epoch => fee

function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    if (_commissionfee > MAX_COMMISSION_FEE) revert InvalidCommissionFee();

    uint256 nextEpoch = getCurrentEpoch() + 1;

    // ☒ Apply to NEXT epoch, not current
    epochCommissionFee[nextEpoch] = _commissionfee;

    emit CommissionFeeScheduled(nextEpoch, _commissionfee, block.timestamp);
}

function settleFilledOrder(...) internal returns (uint256) {
    // ☒ Use fee for THIS epoch, set before epoch started
    uint256 commission = (settlementAmount * epochCommissionFee[round.epoch]) / 10000;

    return commission;
}

// FIX 3: Add timelock for fee changes
uint256 public constant FEE_CHANGE_TIMELOCK = 24 hours;
uint256 public pendingCommissionFee;
uint256 public feeChangeTimestamp;

function proposeCommissionFee(uint256 _commissionfee) external onlyAdmin {
    if (_commissionfee > MAX_COMMISSION_FEE) revert InvalidCommissionFee();

    pendingCommissionFee = _commissionfee;
    feeChangeTimestamp = block.timestamp + FEE_CHANGE_TIMELOCK;

    emit CommissionFeeProposed(_commissionfee, feeChangeTimestamp);
}

function executeCommissionFeeChange() external onlyAdmin {
    require(block.timestamp >= feeChangeTimestamp, "Timelock not expired");
    require(pendingCommissionFee != 0, "No pending fee change");

    bvs.commissionfee = pendingCommissionFee;

    emit CommissionFeeUpdated(pendingCommissionFee);

    pendingCommissionFee = 0;
    feeChangeTimestamp = 0;
}

// RECOMMENDED: Combination of FIX 1 + FIX 3
// - Store fee with order (immediate fairness)
// - Add timelock (gives users warning of changes)
```

Discussion:

> *Developer:*
>
> Fixed in commit 881da1c9c80941e8fd9a91ff9ef0d989f2db0466

> *Auditor:*
>
> **VERIFIED FIXED**(commit 881da1c9c80941e8fd9a91ff9ef0d989f2db0466).

Finding 19: depositTo Phishing - Public Deposit Surface Attack

Severity: ! Medium

Status: Resolved

Description:

The PUBLIC `depositTo()` function allows ANYONE to deposit funds to ANY user's account without their consent or approval. This creates a trust-exploitable attack surface enabling three distinct attack vectors:

(1) Phishing with On-Chain Proof: Attacker deposits $0.01 to 1000 unsuspecting users, then launches social engineering campaign: "You received an airdrop! Click here to claim at [phishing-site]". Users verify on-chain that they DO have balance in protocol, lending credibility to scam.

(2) Storage Bloat: Attacker can inflate `userBalances` mapping with millions of dust deposits for minimal cost (~$0.10 total), increasing storage costs for node operators and bloating state.

(3) UX Confusion: Users see unexpected balances in their accounts without initiating deposits, creating confusion and support burden. This pattern violates user expectation that their account balance only changes through their own actions.

Impact:

MEDIUM - Reputation Damage + Enables Phishing + Storage Bloat.

Three attack scenarios:

(1) Phishing Campaign [PRIMARY RISK]: Attacker deposits $0.01 (1e4 wei) to 1000 targeted users. Attack cost: 1000 × $0.01 = $10 + gas (~$5-50). Launches phishing campaign: "Congratulations! BaseVol Protocol airdropped you $X. Claim at basevol-scam[.]com". Victims check real BaseVol contract -> See non-zero balance -> Trust increases -> Click phishing link -> Lose funds from wallet approval scam. **Estimated success rate:** 5-10% of targeted users fall for scam. **Attacker profit:** 50-100 victims × avg $500 stolen = $25k-50k profit from $15 investment.

(2) Storage Bloat: Deposit $0.0001 to 10M addresses. Cost: $1000 + gas. Inflates userBalances mapping, increases state size, creates operational burden for indexers/nodes. Not immediately harmful but degrades system performance.

(3) Support Burden: Confused users create support tickets: "Why do I have balance? I didn't deposit!" Protocol reputation damaged, customer support overwhelmed, user trust eroded. **Real-world comparable:** Similar issues exploited in other protocols (e.g., token airdrops to unsuspecting users for phishing).

Source:

- contracts/core/ClearingHouse.sol (depositTo function, lines 163-169)

Code:

```
// ClearingHouse.sol lines 163-169
function depositTo(address user, uint256 amount)
    external nonReentrant {  //  PUBLIC - anyone can call

    ClearingHouseStorage.Layout storage $ = ClearingHouseStorage.layout();

    //  NO VALIDATION: Does user consent to receive?
    //  NO VALIDATION: Is msg.sender == user?
    //  NO MINIMUM AMOUNT CHECK

    $.token.safeTransferFrom(msg.sender, address(this), amount);
    $.userBalances[user] += amount;  //  Can credit ANY address

    emit Deposit(user, msg.sender, amount, $.userBalances[user]);

    // Attacker can:
    // - Deposit $0.01 to 1000 users for phishing
    // - Deposit $0.0001 to 1M users for storage bloat
    // - No opt-in, no consent required
}
```

Proof of Concept:

Attacker deposits $0.01 to 1000 users (cost: $15 total). Launches phishing: “BaseVol airdrop! Claim at fake-site”. Victims check on-chain -> see real balance -> trust scam -> connect wallet -> lose funds. Success rate: 5-10% -> $25k-50k profit from $15 investment.

Remediation:

```
// OPTION 1: Whitelist System (Recommended)
mapping(address => mapping(address => bool)) public depositAllowlist;

function setDepositAllowlist(address depositor, bool allowed) external {
    depositAllowlist[msg.sender][depositor] = allowed;
    emit DepositAllowlistUpdated(msg.sender, depositor, allowed);
}

function depositTo(address user, uint256 amount) external nonReentrant {
    // Require user has allowlisted depositor OR depositor is user
    require(
        msg.sender == user || depositAllowlist[user][msg.sender],
        "Recipient has not allowed deposits from sender"
    );

    $.token.safeTransferFrom(msg.sender, address(this), amount);
    $.userBalances[user] += amount;
    emit Deposit(user, msg.sender, amount, $.userBalances[user]);
}

// OPTION 2: Opt-In Mechanism
mapping(address => bool) public allowsUnsolicited Deposits;

function setAllowUnsolicitedDeposits(bool allow) external {
    allowsUnsolicitedDeposits[msg.sender] = allow;
    emit UnsolicitedDepositsToggled(msg.sender, allow);
}

function depositTo(address user, uint256 amount) external nonReentrant {
    require(
        msg.sender == user || allowsUnsolicitedDeposits[user],
        "Recipient does not accept unsolicited deposits"
    );

    // ... rest of logic
}

// OPTION 3: Minimum Deposit Amount
uint256 public constant MIN_DEPOSIT = 10e6;  // $10 minimum

function depositTo(address user, uint256 amount) external nonReentrant {
    // Require meaningful deposit amount
    require(amount >= MIN_DEPOSIT, "Deposit below minimum");

    // ... rest of logic
}

// OPTION 4: Rate Limiting per Sender
mapping(address => uint256) public lastDepositTime;
uint256 public constant DEPOSIT_COOLDOWN = 1 hours;

function depositTo(address user, uint256 amount) external nonReentrant {
    // Prevent rapid-fire dust deposits
    require(
        block.timestamp >= lastDepositTime[msg.sender] + DEPOSIT_COOLDOWN,
        "Deposit cooldown active"
    );

    lastDepositTime[msg.sender] = block.timestamp;

    // ... rest of logic
}

// OPTION 5: Restrict to Operator Only (Most Secure)
function depositTo(address user, uint256 amount)
    external nonReentrant onlyOperator {  // Only operator can call

    // ... rest of logic
}

// RECOMMENDED: Combination of Options 1 + 3 + 4
function depositTo(address user, uint256 amount) external nonReentrant {
    // Require consent
    require(
        msg.sender == user || depositAllowlist[user][msg.sender],
        "Recipient has not allowed deposits"
    );

    // Minimum amount
    require(amount >= MIN_DEPOSIT, "Deposit below minimum");

    // Rate limit
    require(
        block.timestamp >= lastDepositTime[msg.sender] + DEPOSIT_COOLDOWN,
        "Deposit cooldown active"
    );

    lastDepositTime[msg.sender] = block.timestamp;

    $.token.safeTransferFrom(msg.sender, address(this), amount);
    $.userBalances[user] += amount;
    emit Deposit(user, msg.sender, amount, $.userBalances[user]);
}
```

Discussion:

Developer:

Fixed in latest commit

Auditor:

VERIFIED FIXED(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Finding 20: Missing Amount and Parameter Validation Across Multiple Admin Functions

Severity: ! Medium

Status: Resolved

Description:

Five distinct parameter validation gaps in administrative functions allow inconsistent state and operational failures. **

(1) `setDepositLimits`** allows `userLimit` > `vaultLimit`, creating logical impossibility where users can individually deposit more than vault total capacity.

(2) `onRoundSettled` doesn't validate that `depositAmount` / `redeemShares` match pending requests in storage, allowing keeper to submit arbitrary amounts causing accounting corruption.

(3) `executeRound` accepts empty or oversized `priceData` arrays without validation, leading to DoS (empty) or zero-price settlements (wrong size).

(4) `setPriceInfo` allows changing price configurations mid-epoch while round is active, causing inconsistent settlements using wrong `priceId`s.

(5) `setLastFilledOrderId` can be set backwards (lower than current), causing order ID collisions and overwriting existing orders. These gaps create operational risks and state corruption from configuration errors or malicious keepers.

Impact:

MEDIUM - State Corruption + Operational Failures. Five distinct failure modes:

(1) `setDepositLimits` with `userLimit` > `vaultLimit` creates impossible deposit logic. Users attempting to deposit `userLimit` amount will fail when vault hits `vaultLimit` first. Confusing UX, broken deposit flow.

(2) `onRoundSettled` with wrong amounts corrupts accounting permanently. Example: 1000 USDC pending deposits, keeper submits 500 USDC -> 500 USDC lost in accounting, users can't claim full amounts.

(3) `executeRound` with empty `priceData` causes immediate revert (array access OOB), DoSing round execution. Wrong length causes zero prices for missing products -> incorrect TIE settlements.

(4) `setPriceInfo` mid-epoch changes price feed while round active. Orders submitted with old `priceId` settled with new `priceId` -> wrong strike calculations.

(5) `setLastFilledOrderId` backwards creates order ID collision. New order overwrites existing order data -> users lose filled order records, can't claim settlements. Combined impact: Operational instability, accounting errors, UX degradation, requires manual intervention to fix state.

Source:

- contracts/genesis-vault/facets/GenesisManagedVaultFacet.sol (`setDepositLimits`)
- contracts/genesis-vault/facets/GenesisVaultCoreFacet.sol (`onRoundSettled`)
- contracts/basevol/facets/RoundManagementFacet.sol (`executeRound`)
- contracts/basevol/facets/BaseVolAdminFacet.sol (`setPriceInfo` , `setLastFilledOrderId`)

Code:

```
// ISSUE 1: setDepositLimits (GenesisManagedVaultFacet.sol)
function setDepositLimits(uint256 _userLimit, uint256 _vaultLimit) external onlyOwner {
    // NO VALIDATION THAT userLimit <= vaultLimit
    s.userDepositLimit = _userLimit;
    s.vaultDepositLimit = _vaultLimit;

    // Logical impossibility: userLimit = 10M, vaultLimit = 5M
    // User can deposit 10M but vault max is 5M -> broken
}

// ISSUE 2: onRoundSettled (GenesisVaultCoreFacet.sol)
function onRoundSettled(
    uint256 epoch,
    uint256 depositAmount,
    uint256 redeemShares
) external onlyKeeper {
    // NO VALIDATION depositAmount matches pending
    // NO VALIDATION redeemShares matches pending

    RoundData storage round = s.roundData[epoch];
    // Keeper can submit arbitrary amounts -> accounting corruption
    round.processedDepositAssets = depositAmount;
    round.processedRedeemShares = redeemShares;
}

// ISSUE 3: executeRound (RoundManagementFacet.sol)
function executeRound(
    uint256 epoch,
    PriceData[] calldata priceData,
    ...
) external onlyOperator {
    // NO VALIDATION priceData.length == priceIdCount
    // NO VALIDATION priceData not empty

    Round storage round = bvs.rounds[epoch];

    // If empty -> DoS
    // If wrong length -> zero prices for some products
    for (uint i = 0; i < priceData.length; i++) {
        round.startPrice[i] = priceData[i].price;
    }
}

// ISSUE 4: setPriceInfo (BaseVolAdminFacet.sol)
function setPriceInfo(uint256 productId, bytes32 priceId, string symbol) external onlyAdmin {
    // NO VALIDATION epoch is not active
    // NO CHECK if round is ongoing

    bvs.priceInfos[productId] = PriceInfo(priceId, productId, symbol);
    // Changes price feed mid-epoch -> inconsistent settlements
}

// ISSUE 5: setLastFilledOrderId (BaseVolAdminFacet.sol)
function setLastFilledOrderId(uint256 _lastFilledOrderId) external onlyAdmin {
    // NO VALIDATION _lastFilledOrderId >= current
    bvs.lastFilledOrderId = _lastFilledOrderId;

    // Can set backwards: current = 100, set to 50
    // Next order gets ID 51 -> overwrites existing order 51
}
```

Proof of Concept:

Admin sets `userLimit=10M, vaultLimit=5M` (no validation). User tries to deposit 10M -> passes user check but fails vault check -> “My limit is 10M but I can’t deposit?” Similar issues: keeper submits wrong amounts causing accounting corruption, empty price arrays causing DoS.

Remediation:

```
// FIX 1: Validate setDepositLimits
function setDepositLimits(uint256 _userLimit, uint256 _vaultLimit) external onlyOwner {
    // Validate logical consistency
    require(_userLimit <= _vaultLimit, "User limit cannot exceed vault limit");

    s.userDepositLimit = _userLimit;
    s.vaultDepositLimit = _vaultLimit;

    emit DepositLimitsUpdated(_userLimit, _vaultLimit);
}

// FIX 2: Validate onRoundSettled amounts
function onRoundSettled(
    uint256 epoch,
    uint256 depositAmount,
    uint256 redeemShares
) external onlyKeeper {
    RoundData storage round = s.roundData[epoch];

    // Validate amounts match pending requests
    require(
        depositAmount == round.totalRequestedDepositAssets,
        "Deposit amount mismatch"
    );
    require(
        redeemShares == round.totalRequestedRedeemShares,
        "Redeem shares mismatch"
    );

    round.processedDepositAssets = depositAmount;
    round.processedRedeemShares = redeemShares;
    round.isSettled = true;
}

// FIX 3: Validate executeRound priceData
function executeRound(
    uint256 epoch,
    PriceData[] calldata priceData,
    ...
) external onlyOperator {
    // Validate array not empty
    require(priceData.length > 0, "Empty price data");

    // Validate array length matches expected
    require(
        priceData.length == bvs.priceIdCount,
        "Price data length mismatch"
    );

    Round storage round = bvs.rounds[epoch];
    for (uint i = 0; i < priceData.length; i++) {
        round.startPrice[i] = priceData[i].price;
    }
}

// FIX 4: Validate setPriceInfo not during active round
function setPriceInfo(
    uint256 productId,
    bytes32 priceId,
    string calldata symbol
) external onlyAdmin {
    uint256 currentEpoch = getCurrentEpoch();
    Round storage round = bvs.rounds[currentEpoch];

    // Prevent changes during active round
    require(
        round.startTimestamp == 0 || round.endTimestamp != 0,
        "Cannot change price info during active round"
    );

    bvs.priceInfos[productId] = PriceInfo(priceId, productId, symbol);
    emit PriceInfoUpdated(productId, priceId, symbol);
}

// FIX 5: Validate setLastFilledOrderId only increases
function setLastFilledOrderId(uint256 _lastFilledOrderId) external onlyAdmin {
    // Ensure ID only moves forward
    require(
        _lastFilledOrderId >= bvs.lastFilledOrderId,
        "Order ID can only increase"
    );

    bvs.lastFilledOrderId = _lastFilledOrderId;
    emit LastFilledOrderIdUpdated(_lastFilledOrderId);
}
```

Discussion:

> *Developer:*
>
> Fixed in commit a0a0eb40d5bbdd2b9425ad96438c2be9f9dff8de

> *Auditor:*
>
> **VERIFIED FIXED**(commit a0a0eb40d5bbdd2b9425ad96438c2be9f9dff8de).

Finding 21: Missing Oracle and Price Data Validation Across Settlement Flow

Severity: ! Medium

Status: Resolved

Description:

Three critical oracle validation gaps create risks of protocol DoS and incorrect settlements.

(1) **`setPriceInfo`** accepts arbitrary `priceId` without validating it exists in the oracle, causing future round executions to revert when oracle can't provide price for invalid `priceId`.

(2) **`executeRound`** doesn't validate that ALL products have corresponding prices in the `priceData` array, allowing settlements with zero prices for missing products leading to incorrect TIE outcomes.

(3) **`settleFilledOrders`** only validates `product[0]` prices exist but doesn't check prices for orders with `productId > 0`, allowing zero-price settlements for non-BTC products resulting in lost protocol fees. These gaps create operational DoS vectors and revenue loss from misconfigured or incomplete price data.

Impact:

MEDIUM - Protocol DoS + Revenue Loss + Incorrect Settlements.

Three distinct failure modes:

(1) **`setPriceInfo`** with invalid `priceId` causes permanent DoS. When operator calls `executeRound`, oracle query fails -> transaction reverts -> round cannot execute -> all settlements blocked until admin fixes `priceId` configuration. Recovery requires admin intervention, operational downtime.

(2) **`executeRound`** with incomplete `priceData` causes zero-price settlements. Example: 3 products configured (BTC, ETH, SOL) but only 2 prices provided. SOL orders settle with `startPrice=0`, `endPrice=0` -> interpreted as TIE -> users receive refunds instead of profit/loss settlements -> protocol loses all commission fees for SOL trades.

(3) **`settleFilledOrders`** missing per-product validation allows silent zero-price settlements for `productId > 0`. ETH/SOL orders can settle with zero prices without any validation error, resulting in incorrect TIE outcomes and lost fee revenue.

Combined impact: Operational instability (DoS from bad config), revenue loss (zero-price TIE settlements), incorrect user outcomes (wrong settlement calculations). Requires comprehensive validation at each step of price data flow.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (`setPriceInfo`)
- contracts/basevol/facets/RoundManagementFacet.sol (`executeRound`)
- contracts/basevol/facets/OrderProcessingFacet.sol (`settleFilledOrders`)

Code:

```
// ISSUE 1: setPriceInfo (BaseVolAdminFacet.sol)
function setPriceInfo(
    uint256 productId,
    bytes32 priceId,
    string calldata symbol
) external onlyAdmin {
    // NO VALIDATION priceId exists in oracle
    bvs.priceInfos[productId] = PriceInfo({
        priceId: priceId,  // Can be invalid/non-existent!
        productId: productId,
        symbol: symbol
    });

    // Later when executeRound queries oracle with this priceId -> REVERT
}

// ISSUE 2: executeRound (RoundManagementFacet.sol)
function executeRound(
    uint256 epoch,
    PriceData[] calldata priceData,
    ...
) external onlyOperator {
    Round storage round = bvs.rounds[epoch];

    // NO VALIDATION ALL products have prices
    // If priceData.length < priceIdCount -> missing products get zero price
    for (uint i = 0; i < priceData.length; i++) {
        round.startPrice[i] = priceData[i].price;
        round.endPrice[i] = priceData[i].price;
    }

    // Products beyond priceData.length have default value = 0
    // round.startPrice[2] = 0 if only 2 prices provided but 3 products exist
}

// ISSUE 3: settleFilledOrders (OrderProcessingFacet.sol)
function settleFilledOrders(...) internal {
    // ONLY CHECKS product[0] prices
    if (round.startPrice[0] == 0 || round.endPrice[0] == 0) {
        revert("Price not set");
    }

    // But orders can have ANY productId
    for (uint i = 0; i < orders.length; i++) {
        FilledOrder storage order = orders[i];
        uint256 productId = order.productId;  // Can be 1, 2, 3...

        // NO VALIDATION round.startPrice[productId] != 0
        uint256 startPrice = round.startPrice[productId];  // Can be 0!
        uint256 endPrice = round.endPrice[productId];  // Can be 0!

        // If both 0 -> startPrice == endPrice -> TIE outcome
        // Protocol charges 0 fee instead of correct settlement fee
    }
}
```

Proof of Concept:

Admin sets invalid priceId (no validation) -> `executeRound` queries oracle -> reverts “Price feed not found” -> all settlements blocked. Operator provides incomplete price array (2 prices for 3 products) -> missing product gets zero price -> incorrect TIE settlements -> lost fee revenue.

Remediation:

```
// FIX 1: Validate priceId exists in oracle (setPriceInfo)
function setPriceInfo(
    uint256 productId,
    bytes32 priceId,
    string calldata symbol
) external onlyAdmin {
    // Validate priceId exists by attempting to fetch price
    try bvs.oracle.getPrice(priceId) returns (PythStructs.Price memory) {
        // Price feed exists
    } catch {
        revert("Invalid priceId: feed not found in oracle");
    }

    bvs.priceInfos[productId] = PriceInfo({
        priceId: priceId,
        productId: productId,
        symbol: symbol
    });

    emit PriceInfoUpdated(productId, priceId, symbol);
}
```

```
// FIX 2: Validate priceData completeness (executeRound)
function executeRound(
    uint256 epoch,
    PriceData[] calldata priceData,
    ...
) external onlyOperator {
    // Validate ALL products have prices
    require(
        priceData.length == bvs.priceIdCount,
        "Price data incomplete: missing products"
    );

    Round storage round = bvs.rounds[epoch];

    for (uint i = 0; i < priceData.length; i++) {
        // Validate price is not zero
        require(priceData[i].price > 0, "Invalid price: cannot be zero");

        round.startPrice[i] = priceData[i].price;
        round.endPrice[i] = priceData[i].price;
    }

    // ... rest of round execution
}
```

```
// FIX 3: Validate per-product prices (settleFilledOrders)
function settleFilledOrders(...) internal {
    Round storage round = bvs.rounds[epoch];
    FilledOrder[] storage orders = bvs.filledOrders[epoch];

    for (uint i = 0; i < orders.length; i++) {
        FilledOrder storage order = orders[i];
        uint256 productId = order.productId;

        // Validate THIS product has valid prices
        uint256 startPrice = round.startPrice[productId];
        uint256 endPrice = round.endPrice[productId];

        require(
            startPrice > 0 && endPrice > 0,
            "Product prices not set"
        );

        // Now safe to proceed with settlement logic
        // ... settlement calculations
    }
}
```

```
// ALTERNATIVE: Add comprehensive validation helper
function validateProductPrices(uint256 epoch, uint256 productId) internal view {
    Round storage round = bvs.rounds[epoch];

    require(
        productId < bvs.priceIdCount,
        "Invalid productId"
    );
    require(
        round.startPrice[productId] > 0,
        "Start price not set"
    );
    require(
        round.endPrice[productId] > 0,
        "End price not set"
    );
}

// Use in settlement
function settleFilledOrders(...) internal {
    for (uint i = 0; i < orders.length; i++) {
        FilledOrder storage order = orders[i];
        validateProductPrices(epoch, order.productId);

        // Safe to proceed with settlement
    }
}
```

Discussion:

> *Developer:*
>
> Because the transaction will revert if the Oracle price data becomes abnormal, no settlement issues will occur.

> *Auditor:*
>
> We accept your “trust oracle validation” approach - no per-token bounds needed. However, please add simple zero-price validation: require(price > 0, “Invalid price”); after oracle calls. This is universal, works for all tokens, and catches obvious bugs.

> *Developer:*
>
> Fixed in commit cc36b3d646632ae0b0bba47626e2a48603ad4317

> *Auditor:*
>
> **VERIFIED FIXED**(commit cc36b3d646632ae0b0bba47626e2a48603ad4317).

Finding 22: Missing Zero-Address Validation Across Multiple Functions

Severity: ● Medium

Status: Resolved

Description:

Six distinct zero-address validation gaps exist across multiple critical functions. Admin functions accept `address(0)` without validation, leading to permanent fund loss or broken protocol state.

(1) **`setRedeemVault`** allows setting `redeemVault` to `address(0)`, causing all redemption funds to be sent to the burn address.

(2) **`redeemPairs`** doesn't validate `pair.user` addresses before processing, allowing permanent loss if redemption list contains `address(0)`.

(3) **`setOperator`** accepts `address(0)`, leaving no valid operator and breaking all operator functions.

(4) **`setFeeInfos`** allows `feeRecipient` to be zero, causing all collected fees to be permanently lost to burn address.

(5) GenesisVault **`initialize()`** doesn't validate `asset`, `feeRecipient`, or `strategy` addresses before initializing vault.

(6) **`transferFeesToRecipient`** can execute with zero `feeRecipient` from storage, burning fees. These validation gaps violate basic defensive programming principles and create multiple fund loss vectors.

Impact:

MEDIUM to HIGH - Permanent Fund Loss + Protocol Bricking. Multiple failure modes with varying severity:

(1) **`setRedeemVault(0)`** **-> ALL redemptions burn user funds permanently.** Estimated loss: Unlimited (all future redemptions).

(2) **`redeemPairs`** **with** **`address(0)`** **-> Direct fund burn on redemption execution.** Estimated loss: Per-transaction amounts.

(3) **`setOperator(0)`** **-> Complete operator DoS, all settlements blocked, protocol unusable.**

(4) **`setFeeInfos(0)`** **-> All collected fees burned forever, protocol loses revenue stream.**

(5) **`initialize()`** **with zeros -> Vault deployed in broken state, operations fail, requires redeployment.**

**(6) `transferFeesToRecipient` -> Silent fee burning if `feeRecipient` was set to zero.

** Combined impact: Critical business logic failure, permanent capital loss, operational DoS. While most require admin/privileged access, lack of validation violates defensive programming and creates unnecessary risk from typos, scripting errors, or UI bugs.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (`setRedeemVault` , `setOperator` , `setFeeInfos`)
- contracts/basevol/facets/RedemptionFacet.sol (`redeemPairs` , line 78)
- contracts/genesis-vault/facets/VaultCoreFacet.sol (`setStrategy` , `transferFeesToRecipient`)
- contracts/genesis-vault/facets/GenesisVaultInitializationFacet.sol (`initialize`)

Code:

```
// ISSUE 1: setRedeemVault (BaseVolAdminFacet.sol)
function setRedeemVault(address _redeemVault) external onlyAdmin {
    // NO VALIDATION
    bvs.redeemVault = _redeemVault;  // Can be address(0)!
    emit RedeemVaultSet(_redeemVault);
}
// Impact: All redemptions send funds to address(0) -> permanent loss

// ISSUE 2: redeemPairs (RedemptionFacet.sol line 78)
function redeemPairs(RedeemPair[] memory pairs) external {
    for (uint i = 0; i < pairs.length; i++) {
        address user = pairs[i].user;  // NO VALIDATION
        uint256 amount = pairs[i].amount;

        bvs.token.safeTransfer(user, amount);  // Sends to address(0)!
    }
}

// ISSUE 3: setOperator (BaseVolAdminFacet.sol)
function setOperator(address _operator) external onlyAdmin {
    // NO VALIDATION
    bvs.operatorAddress = _operator;  // Can be address(0)!
}
// Impact: All onlyOperator functions permanently broken

// ISSUE 4: setFeeInfos (VaultCoreFacet.sol)
function setFeeInfos(address _feeRecipient, ...) external onlyOwner {
    // NO VALIDATION
    s.feeRecipient = _feeRecipient;  // Can be address(0)!
}

// ISSUE 5: GenesisVault initialize (GenesisVaultInitializationFacet.sol)
function initialize(address asset_, address feeRecipient_, address strategy_) external {
    // NO VALIDATION FOR CRITICAL ADDRESSES
    s.asset = IERC20(asset_);  // Can be address(0)!
    s.feeRecipient = feeRecipient_;  // Can be address(0)!
    s.strategy = strategy_;  // Can be address(0)!
}

// ISSUE 6: transferFeesToRecipient (VaultCoreFacet.sol)
function transferFeesToRecipient(uint256 amount, string memory feeType) internal {
    address recipient = s.feeRecipient;  // NO VALIDATION
    IERC20(asset()).safeTransfer(recipient, amount);  // Burns if recipient = 0
}
```

Proof of Concept:

Admin script error calls `setRedeemVault(address(0))` (no validation). User redeems -> `safeTransfer(address(0), 1000e6)` -> 1000 USDC permanently burned. Same issue affects: operator setting, fee recipient, and batch redemptions with address(0) in array.

Remediation:

```
// FIX 1: Add validation to setRedeemVault
function setRedeemVault(address _redeemVault) external onlyAdmin {
    require(_redeemVault != address(0), "Invalid redeemVault address");
    bvs.redeemVault = _redeemVault;
    emit RedeemVaultSet(_redeemVault);
}

// FIX 2: Add validation to redeemPairs
function redeemPairs(RedeemPair[] memory pairs) external {
    for (uint i = 0; i < pairs.length; i++) {
        require(pairs[i].user != address(0), "Invalid user address");
        address user = pairs[i].user;
        uint256 amount = pairs[i].amount;

        bvs.token.safeTransfer(user, amount);
    }
}

// FIX 3: Add validation to setOperator
function setOperator(address _operator) external onlyAdmin {
    require(_operator != address(0), "Invalid operator address");
    bvs.operatorAddress = _operator;
    emit OperatorSet(_operator);
}

// FIX 4: Add validation to setFeeInfos
function setFeeInfos(address _feeRecipient, ...) external onlyOwner {
    require(_feeRecipient != address(0), "Invalid feeRecipient");
    s.feeRecipient = _feeRecipient;
    // ... rest of logic
}

// FIX 5: Add validation to GenesisVault initialize
function initialize(
    address asset_,
    address feeRecipient_,
    address strategy_,
    ...
) external initializer {
    // VALIDATE ALL CRITICAL ADDRESSES
    require(asset_ != address(0), "GenesisVault: invalid asset");
    require(feeRecipient_ != address(0), "GenesisVault: invalid feeRecipient");
    require(strategy_ != address(0), "GenesisVault: invalid strategy");

    s.asset = IERC20(asset_);
    s.feeRecipient = feeRecipient_;
    s.strategy = strategy_;
    // ... rest of initialization
}

// FIX 6: Add validation to transferFeesToRecipient
function transferFeesToRecipient(uint256 amount, string memory feeType) internal {
    address recipient = s.feeRecipient;
    require(recipient != address(0), "Fee recipient not set");
    IERC20(asset()).safeTransfer(recipient, amount);
    emit FeesTransferred(recipient, amount, feeType);
}

// BEST PRACTICE: Create reusable validation modifier
```

```
modifier validAddress(address _address) {
    require(_address != address(0), "Invalid address: zero address");
    _;
}

// Apply to all address-setting functions
function setRedeemVault(address _redeemVault) external onlyAdmin validAddress(_redeemVault) {
    bvs.redeemVault = _redeemVault;
    emit RedeemVaultSet(_redeemVault);
}
```

Discussion:

> *Developer:*
>
> This is fixed in latest commit.

> *Auditor:*
>
> **VERIFIED FIXED**(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Finding 23: No Maximum Fee Validation - DoS via Fee Misconfiguration

Severity: ● Medium

Status: Resolved

Description:

`setRedeemFee` function lacks maximum value validation, creating DoS risk from fee misconfiguration. Unlike `setCommissionfee` which has MAX_COMMISSION_FEE constant (50%), `setRedeemFee` accepts ANY value including `type(uint256).max`.

Two failure modes:

(1) Arithmetic Overflow: If redeemFee set to huge value, calculations in `redeemCoupons()` overflow causing ALL redemptions to revert.

(2) Excessive Fee Block: If redeemFee > coupon value, condition `totalCommission <= baseAmount` fails, blocking ALL redemptions. Common admin error: Setting fee in wrong units (wei instead of USDC) results in redeemFee = 1e18 instead of 1e6, instant DoS. This inconsistency violates defensive programming - all fee setters should have maximum validation.

Impact:

MEDIUM - Complete Redemption DoS + Extended Downtime. Admin sets `redeemFee` incorrectly -> ALL user redemptions fail.

Two scenarios:

(1) Wrong Units Error: Admin intends 0.1% fee (10 basis points) = 0.001 USDC per 1 USDC. Calculates: 0.001 x 1e6 = 1000. But accidentally uses wei: 0.001 x 1e18 = 1e15. Calls `setRedeemFee(1e15)`. Now every redemption: `totalCommission = (baseAmount x 1e15) / 10000` -> Massive overflow or exceeds baseAmount -> ALL redemptions revert.

(2) Max Value Error: Script error sets `setRedeemFee(type(uint256).max)`. Redemption calculation overflows -> ALL redemptions DoSed. **Impact:** All users unable to redeem coupons. Funds trapped until admin discovers issue and sets correct fee. No event emission makes debugging harder. Recovery requires upgrade if admin keys unavailable. Estimated downtime: Hours to days.

Source:

- contracts/basevol/facets/RedemptionFacet.sol (`setRedeemFee` , lines 210-213)

Code:

```
// RedemptionFacet.sol lines 210-213 - NO MAXIMUM VALIDATION
function setRedeemFee(uint256 _redeemFee) external onlyAdmin {
    // NO MAXIMUM CHECK
    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();
    bvs.redeemFee = _redeemFee;  // Accepts ANY value!
    // NO EVENT EMISSION
}

// Compare: BaseVolAdminFacet.sol - HAS maximum validation
uint256 private constant MAX_COMMISSION_FEE = 5000; // 50%

function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    // HAS MAXIMUM CHECK
    if (_commissionfee > MAX_COMMISSION_FEE)
        revert InvalidCommissionFee();

    bvs.commissionfee = _commissionfee;
    emit CommissionFeeUpdated(_commissionfee);  // HAS EVENT
}

// Impact on redemptions:
function redeemCoupons(uint256 productId, uint256 baseAmount) external {
    // ...
    uint256 totalCommission = (baseAmount * bvs.redeemFee) / 10000;

    // If redeemFee = type(uint256).max -> overflow
    // If redeemFee = 1e18 (wrong units) -> totalCommission >> baseAmount

    require(totalCommission <= baseAmount, "Fee exceeds amount");  // REVERTS
}
```

Proof of Concept:

Admin script uses wrong units: `setRedeemFee(5e15)` instead of `50` (no validation). User redeems 1000 USDC -> calculates fee: `(1000e6 x 5e15) / 10000 = 5e17` -> check fails: `5e17 <= 1000e6` -> reverts “Fee exceeds amount” -> all redemptions DoSed.

Remediation:

```
// FIX: Add maximum validation to setRedeemFee
uint256 public constant MAX_REDEEM_FEE = 1000;  // Max 10% (1000 basis points)

function setRedeemFee(uint256 _redeemFee) external onlyAdmin {
    // Add maximum validation
    require(_redeemFee <= MAX_REDEEM_FEE, "Redeem fee exceeds maximum");

    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();

    uint256 oldFee = bvs.redeemFee;
    bvs.redeemFee = _redeemFee;

    // Add event emission
    emit RedeemFeeUpdated(oldFee, _redeemFee, block.timestamp);
}

// ALTERNATIVE: More comprehensive validation
function setRedeemFee(uint256 _redeemFee) external onlyAdmin {
    require(_redeemFee <= MAX_REDEEM_FEE, "Redeem fee exceeds maximum");
    require(_redeemFee >= 0, "Redeem fee cannot be negative");  // Redundant but defensive

    LibBaseVolStrike.DiamondStorage storage bvs = LibBaseVolStrike.diamondStorage();
    bvs.redeemFee = _redeemFee;

    emit RedeemFeeUpdated(bvs.redeemFee, _redeemFee, block.timestamp);
}

// BEST PRACTICE: Consistent validation across all fee setters
uint256 public constant MAX_COMMISSION_FEE = 5000;   // 50%
uint256 public constant MAX_REDEEM_FEE = 1000;       // 10%
uint256 public constant MAX_WITHDRAWAL_FEE = 100;    // 1%

function setCommissionfee(uint256 _fee) external onlyAdmin {
    require(_fee <= MAX_COMMISSION_FEE, "Fee exceeds max");
    bvs.commissionfee = _fee;
    emit CommissionFeeUpdated(_fee);
}

function setRedeemFee(uint256 _fee) external onlyAdmin {
    require(_fee <= MAX_REDEEM_FEE, "Fee exceeds max");
    bvs.redeemFee = _fee;
    emit RedeemFeeUpdated(_fee);
}

function setWithdrawalFee(uint256 _fee) external onlyAdmin {
    require(_fee <= MAX_WITHDRAWAL_FEE, "Fee exceeds max");
    $.withdrawalFee = _fee;
    emit WithdrawalFeeUpdated(_fee);
}
```

Discussion:

Developer:

Fixed in commit e610de6b672052e42c2e7c20e434337c7bfe433f

Auditor:

VERIFIED FIXED(commit e610de6b672052e42c2e7c20e434337c7bfe433f).

Finding 24: Single Operator Centralization Without Safeguards

Severity: ! Medium

Status: Resolved

Description:

Single operator address controls ALL critical protocol operations without multi-signature requirements, timelocks, or secondary oversight mechanisms. Creates single point of failure where operator key compromise grants attacker complete protocol control.

Seven critical powers without safeguards:

(1) Arbitrary Price Setting: Can set any prices via `setManualRoundEndPrices` with ZERO bounds validation (can set BTC=$1 or $1M).

(2) Round Timing Control: Complete control over when rounds start/end, enabling timing attacks.

(3) Indefinite Settlement Delay: Can skip settlements forever, locking all user funds indefinitely.

(4) Stale Price Replay: Can reuse old oracle prices from any timestamp, manipulating outcomes.

(5) No Price Sanity Checks: Manual prices have no validation against oracle prices (no +-50% bounds).

(6) No Secondary Approval: All operations execute immediately without timelock or guardian review.

(7) No Emergency Pause: Only operator can pause - if operator compromised, no one can stop malicious actions. This centralization violates DeFi security best practices and creates catastrophic risk from single key compromise.

Impact:

MEDIUM to HIGH - Single Key Compromise = Total Protocol Control. Seven attack vectors from operator key compromise:

(1) Price Manipulation: Compromised operator sets BTC endPrice = $1 when actual = $50k. All OVER positions win maximum, all UNDER positions lose everything. Estimated theft: Unlimited (can drain all escrowed funds per epoch).

(2) Timing Attacks: Operator observes mempool, times round execution to benefit confederate's positions. Estimated profit: 5-10% per round through strategic timing.

(3) Settlement Delay Attack: Operator stops executing settlements. All user funds locked in escrow indefinitely. Users cannot withdraw until settlements complete. No timeout mechanism.

(4) Stale Price Replay: Operator uses 24-hour-old oracle prices during high volatility, settling with prices that benefit attacker's positions.

(5) Selective Settlement: Operator settles profitable rounds immediately, delays unprofitable rounds, optimizing theft.

(6) No Recovery Path: If operator key lost/compromised, protocol has no guardian role to pause or no multi-sig to revoke access.

(7) Regulatory Risk: Single point of control makes protocol appear centralized, creating regulatory vulnerability. **Real-world comparable:** Similar to early DeFi protocols (e.g., Harvest Finance) where admin keys were compromised leading to >$30M theft. Current design has NO defense against compromised operator.

Source:

- contracts/basevol/facets/RoundManagementFacet.sol (`setManualRoundEndPrices`, `executeRound`, `all` onlyOperator‘ functions)

Code:

```
// ISSUE 1: Arbitrary prices without bounds (RoundManagementFacet.sol)
function setManualRoundEndPrices(
    uint256 epoch,
    uint256[] calldata endPrices
) external onlyOperator {  // single operator, no multi-sig
    Round storage round = bvs.rounds[epoch];

    // NO VALIDATION AGAINST ORACLE PRICES
    // NO BOUNDS CHECKING (can set any value)
    // NO TIMELOCK
    // NO SECONDARY APPROVAL
    for (uint i = 0; i < endPrices.length; i++) {
        round.endPrice[i] = endPrices[i];  // Operator sets arbitrary prices
    }

    // Operator can set: BTC = $1 or $1,000,000
    // -> Manipulate all settlements
}

// ISSUE 2: Round timing control (RoundManagementFacet.sol)
function executeRound(...) external onlyOperator {
    // Only operator can execute rounds
    // Can delay indefinitely
    // Can time rounds to manipulate outcomes
}

// ISSUE 3: Settlement control
function settleOrders(...) external onlyOperator {
    // Only operator can settle
    // Can skip settlements forever
    // Funds locked until operator acts
}

```

```
// ISSUE 4: No emergency pause by others
function pause() external onlyOperator {
    // ONLY operator can pause
    // If operator compromised -> no one can stop attack
}

// ISSUE 5: Single point of failure
address public operatorAddress;  // Single address

modifier onlyOperator() {
    require(msg.sender == operatorAddress, "Not operator");
    _;
    // No multi-sig
    // No role separation
    // No backup operators
}

// ISSUE 6: Stale price replay
function executeRound(
    uint256 epoch,
    PriceData[] calldata priceData,  // Any timestamp
    bytes[] calldata priceUpdateData
) external onlyOperator {
    // NO CHECK: priceData must be recent
    // Operator can use 1-day-old prices
}
```

Proof of Concept:

Operator key compromised -> attacker calls `setManualRoundEndPrices(epoch, [1])` setting BTC=$1 (no validation, no timelock) -> settles orders -> UNDER positions win when OVER should win -> 2M USDC stolen. Attack repeatable every round. No multi-sig, no bounds checks, no recovery mechanism.

Remediation:

```
// FIX 1: Implement Multi-Sig for Critical Operations
// Use Gnosis Safe 3-of-5 multi-sig
address public operatorMultiSig;  // Gnosis Safe address

modifier onlyOperatorMultiSig() {
    require(msg.sender == operatorMultiSig, "Not operator multi-sig");
    _;
}

function setManualRoundEndPrices(
    uint256 epoch,
    uint256[] calldata endPrices
) external onlyOperatorMultiSig {  // Requires 3-of-5 signatures
    // ... validation and execution
}

// FIX 2: Add Bounds Validation for Manual Prices
function setManualRoundEndPrices(
    uint256 epoch,
    uint256[] calldata endPrices
) external onlyOperatorMultiSig {
    Round storage round = bvs.rounds[epoch];

    for (uint i = 0; i < endPrices.length; i++) {
        uint256 oraclePrice = _getLatestOraclePrice(i);
        uint256 manualPrice = endPrices[i];

        // Validate within +-50% of oracle price
        uint256 maxDeviation = oraclePrice / 2;  // 50%
        require(
            manualPrice >= oraclePrice - maxDeviation &&
            manualPrice <= oraclePrice + maxDeviation,
            "Price deviates too much from oracle"
        );

        round.endPrice[i] = manualPrice;
        emit ManualPriceSet(epoch, i, manualPrice, oraclePrice);
    }
}

// FIX 3: Implement Timelock for Manual Prices
uint256 public constant MANUAL_PRICE_TIMELOCK = 1 hours;

mapping(uint256 => ManualPriceProposal) public manualPriceProposals;

struct ManualPriceProposal {
    uint256[] prices;
    uint256 proposedAt;
    bool executed;
}

function proposeManualPrices(
    uint256 epoch,
    uint256[] calldata endPrices
) external onlyOperatorMultiSig {
    // Propose prices with timelock
    manualPriceProposals[epoch] = ManualPriceProposal({
        prices: endPrices,
        proposedAt: block.timestamp,
        executed: false
    });

    emit ManualPricesProposed(epoch, endPrices, block.timestamp + MANUAL_PRICE_TIMELOCK);
}

function executeManualPrices(uint256 epoch) external onlyOperatorMultiSig {
    ManualPriceProposal storage proposal = manualPriceProposals[epoch];

    // Enforce timelock
    require(
        block.timestamp >= proposal.proposedAt + MANUAL_PRICE_TIMELOCK,
        "Timelock not expired"
    );
    require(!proposal.executed, "Already executed");

    // Apply prices after timelock
    Round storage round = bvs.rounds[epoch];
    for (uint i = 0; i < proposal.prices.length; i++) {
        round.endPrice[i] = proposal.prices[i];
    }

    proposal.executed = true;
}

// FIX 4: Separate Roles for Different Operations
address public priceOperator;      // Can only set prices (with timelock)
address public settlementOperator; // Can only execute settlements
address public pauseGuardian;      // Can pause but not unpause

modifier onlyPriceOperator() {
    require(msg.sender == priceOperator, "Not price operator");
    _;
}

modifier onlySettlementOperator() {
    require(msg.sender == settlementOperator, "Not settlement operator");
    _;
}

// FIX 5: Add Guardian Role for Emergency Pause
```

```
modifier onlyGuardian() {
    require(msg.sender == pauseGuardian, "Not guardian");
    _;
}

function emergencyPause() external onlyGuardian {
    // Guardian can pause even if operator compromised
    _pause();
    emit EmergencyPause(msg.sender, block.timestamp);
}

// FIX 6: Add Settlement Timeout Mechanism
uint256 public constant MAX_SETTLEMENT_DELAY = 24 hours;

function executeRound(uint256 epoch, ...) external onlySettlementOperator {
    Round storage round = bvs.rounds[epoch];

    // Validate not delayed too long
    require(
        block.timestamp <= round.endTimestamp + MAX_SETTLEMENT_DELAY,
        "Settlement timeout - use fallback mechanism"
    );

    // ... settlement logic
}

// Fallback if operator doesn't settle within 24 hours
function emergencySettle(uint256 epoch) external {
    Round storage round = bvs.rounds[epoch];

    // Anyone can settle if operator delays > 24 hours
    require(
        block.timestamp > round.endTimestamp + MAX_SETTLEMENT_DELAY,
        "Operator timeout not reached"
    );

    // Use oracle prices only (no manual override)
    // ... automatic settlement with oracle data
}

// FIX 7: Add Monitoring and Alerts
event ManualPriceDeviation(
    uint256 indexed epoch,
    uint256 indexed productId,
    uint256 manualPrice,
    uint256 oraclePrice,
    uint256 deviationPercent
);

function setManualRoundEndPrices(...) external {
    // ... after setting prices ...

    // Emit detailed events for monitoring
    for (uint i = 0; i < endPrices.length; i++) {
        uint256 oracle = _getLatestOraclePrice(i);
        uint256 deviation = _calculateDeviation(endPrices[i], oracle);

        emit ManualPriceDeviation(epoch, i, endPrices[i], oracle, deviation);
    }
}
```

Discussion:

> *Developer:*
>
> This is fixed in latest commit.

Auditor:

VERIFIED FIXED(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Finding 25: Strategy Rebalancing Transfer Failure - Missing Self-Allowance (Simple 1-Line Bug)

Severity: Medium

Status: Resolved

Description:

The `morphoWithdrawCompletedCallback` function uses `safeTransferFrom(address(this), baseVolManager, amount)` which requires the contract to have approved itself to spend its own tokens. This self-allowance was never granted and cannot exist. When strategy attempts to rebalance funds from Morpho protocol back to BaseVol, the transfer ALWAYS REVERTS with "ERC20: insufficient allowance". This is a simple 1-line bug where `safeTransferFrom` should be `safeTransfer`. When a contract transfers its own tokens, it should use `transfer()` or `safeTransfer()`, NOT `transferFrom()` which requires allowance. This bug completely breaks Morpho->BaseVol rebalancing, leaving strategy stuck in REBALANCING status and unable to move capital efficiently between protocols.

```
// GenesisStrategy.sol line 1086
function morphoWithdrawCompletedCallback(uint256 amount, bool success)
    external authCaller(morphoVaultManager()) {

    GenesisStrategyStorage.Layout storage $ = GenesisStrategyStorage.layout();

    if (success) {
        // Funds withdrawn from Morpho, now transfer to BaseVolManager
        IERC20 _asset = IERC20(asset());

        // BUG: Uses transferFrom for contract's own tokens
        _asset.safeTransferFrom(
            address(this),         // From: this contract
            address(baseVolManager()),  // To: BaseVolManager
            amount
        );

        // This requires: allowance[address(this)][address(this)] >= amount
        // But: Contract never approved itself!
        // Result: ALWAYS REVERTS "ERC20: insufficient allowance"

        $.rebalancing.status = RebalanceStatus.IDLE;
    }
}

// Correct pattern (for comparison):
function correctTransfer(address recipient, uint256 amount) internal {
    IERC20 _asset = IERC20(asset());

    // CORRECT: Use safeTransfer for own tokens
    _asset.safeTransfer(recipient, amount);

    // transferFrom is for spending someone else's tokens (requires allowance)
    // transfer is for sending own tokens (no allowance needed)
}
```

Impact:

MEDIUM - Complete DoS of Morpho->BaseVol Rebalancing + Capital Inefficiency. Strategy cannot rebalance funds from Morpho back to BaseVol, creating multiple operational failures:

(1) Rebalancing DoS: Every attempt to move funds Morpho->BaseVol fails. Keeper calls `keeperRebalance()` -> Strategy calculates need to withdraw from Morpho -> Initiates withdrawal -> Callback executes -> Transfer reverts -> Rebalancing fails.

(2) Strategy Stuck: After failed rebalancing, strategy remains in REBALANCING status. Further rebalancing attempts blocked until status reset.

(3) Capital Trapped: Funds deposited to Morpho can never be moved back to BaseVol for utilization. Strategy loses ability to respond to vault redemption needs.

(4) Liquidity Management Broken: Vault requests liquidity -> Strategy cannot provide from Morpho -> Users experience withdrawal delays.

(5) Capital Efficiency Degraded: Cannot optimize capital allocation between Morpho (yield) and BaseVol (options trading). Stuck with initial allocation.

(6) Recovery Requires Upgrade: Bug in callback function requires contract upgrade to fix. No workaround available. Estimated downtime: Days/weeks for upgrade deployment. During this time, all Morpho deposits are effectively locked from rebalancing.

Source:

- contracts/core/vault/GenesisStrategy.sol (morphoWithdrawCompletedCallback, line 1086)

Code:

```
// GenesisStrategy.sol line 1086
function morphoWithdrawCompletedCallback(uint256 amount, bool success)
    external authCaller(morphoVaultManager()) {

    GenesisStrategyStorage.Layout storage $ = GenesisStrategyStorage.layout();

    if (success) {
        // Funds withdrawn from Morpho, now transfer to BaseVolManager
        IERC20 _asset = IERC20(asset());

        // BUG: Uses transferFrom for contract's own tokens
        _asset.safeTransferFrom(
            address(this),          // From: this contract
            address(baseVolManager()),  // To: BaseVolManager
            amount
        );

        // This requires: allowance[address(this)][address(this)] >= amount
        // But: Contract never approved itself!
        // Result: ALWAYS REVERTS "ERC20: insufficient allowance"

        $.rebalancing.status = RebalanceStatus.IDLE;
    }
}

// Correct pattern (for comparison):
function correctTransfer(address recipient, uint256 amount) internal {
    IERC20 _asset = IERC20(asset());

    // CORRECT: Use safeTransfer for own tokens
    _asset.safeTransfer(recipient, amount);

    // transferFrom is for spending someone else's tokens (requires allowance)
    // transfer is for sending own tokens (no allowance needed)
}
```

Proof of Concept:

Keeper calls `keeperRebalance()` -> Strategy withdraws 200K from Morpho -> Callback executes `safeTransferFrom(address(this), baseVolManager, 200K)` -> Checks allowance[strategy][strategy] = 0 -> reverts "Insufficient allowance" -> funds stuck in strategy contract -> all Morpho->BaseVol rebalancing broken.

Remediation:

```
// FIX: Change line 1086 from safeTransferFrom to safeTransfer (1-line fix)

function morphoWithdrawCompletedCallback(uint256 amount, bool success)
    external authCaller(morphoVaultManager()) {

    GenesisStrategyStorage.Layout storage $ = GenesisStrategyStorage.layout();

    if (success) {
        IERC20 _asset = IERC20(asset());

        // FIX: Use safeTransfer instead of safeTransferFrom
        _asset.safeTransfer(
            address(baseVolManager()),  // To: BaseVolManager
            amount
        );

        // safeTransfer doesn't require allowance
        // Contract can always transfer its own tokens

        $.rebalancing.status = RebalanceStatus.IDLE;
    }
}

// EXPLANATION:
// transferFrom(from, to, amount) - Requires: allowance[from][msg.sender] >= amount
// transfer(to, amount)           - Requires: balance[msg.sender] >= amount

// When contract sends its own tokens:
// CORRECT: asset.safeTransfer(recipient, amount)
// WRONG:   asset.safeTransferFrom(address(this), recipient, amount)

// KEY PRINCIPLE:
// - Use transfer() to send YOUR OWN tokens
// - Use transferFrom() to send SOMEONE ELSE'S tokens (requires their approval)
```

Discussion:

> *Developer:*
>
> Fixed

> *Auditor:*
>
> **VERIFIED FIXED**(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Finding 26: Systematic Input Validation Missing in Initialization Functions

Severity: ● Medium

Status: Resolved

Description:

Seven initialization functions across the codebase lack comprehensive input validation, creating deployment-time risks from human error. While these contracts use proper re-initialization protection (OpenZeppelin `initializer` modifier or custom checks), they fail to validate critical parameters.

This systematic pattern creates multiple failure modes:

(1) Division-by-Zero Risk (3 contracts): InitializationFacet (both versions) and BaseVolStrike accept `_intervalSeconds` without validation. If `_intervalSeconds = 0`, causes division-by-zero in `getCurrentEpoch()` calculation, completely DoSing the protocol.

(2) Zero-Address Silent Failures (6 contracts): BaseVolOneMin, ClearingHouse, BaseVolManager, PythLazer, GenesisVault lack zero-address checks for critical addresses. Deployment succeeds but operations fail silently.

(3) Critical Dependency Missing (1 contract): GenesisVault doesn't validate `baseVolContract` parameter. If `address(0)`, causes complete vault DoS on `getCurrentEpoch` calls.

Affected Contracts:

1. InitializationFacet.sol (facets/ and basevol/facets/) - DIVISION-BY-ZERO RISK

```
// Line 26-42: Has re-init protection but NO input validation
function initialize(..., uint256 _intervalSeconds) public {
    require(ds.contractOwner == address(0), "Already initialized");  // ✅ Re-init protected

    // ❌ MISSING: _intervalSeconds validation
    // ❌ MISSING: Zero-address checks for 5 addresses
    bvs.intervalSeconds = _intervalSeconds;  // Can be 0 → division-by-zero!
}

// Impact: If intervalSeconds = 0 → Complete protocol DoS
// Result: getCurrentEpoch() reverts → ALL user functions unusable
```

2. BaseVolStrike.sol - DIVISION-BY-ZERO RISK

```
// Line 82: Has OZ initializer but NO input validation
function initialize(...) public initializer {
    __UUPSUpgradeable_init();  // ✅ Re-init protected

    // ❌ MISSING: intervalSeconds validation (can be 0 → DoS)
    // ❌ MISSING: Zero-address checks for addresses
}
```

3. BaseVolOneMin.sol - NO INPUT VALIDATION

```
// Line 88: Has OZ initializer but NO validation at all
function initialize(
    address _usdcAddress,
    address _adminAddress,
    address _operatorAddress,
    address _clearingHouseAddress
) public initializer {
    __UUPSUpgradeable_init();  // ⊠ Re-init protected

    // ⊠ NO VALIDATION FOR ANY PARAMETER
    $.token = IERC20(_usdcAddress);  // Can be address(0)!
    $.adminAddress = _adminAddress;  // Can be address(0)!
    // ... all parameters unchecked
}
```

4. GenesisVault.sol - CRITICAL DEPENDENCY VALIDATION MISSING

```
// Line 79: Has OZ initializer, validates keeper but NOT baseVolContract
function initialize(
    address baseVolContract_,  // ⊠ CRITICAL: Not validated!
    address asset_,
    address initialKeeper_,
    ...
) external initializer {
    __GenesisManagedVault_init(msg.sender, msg.sender, asset_, name_, symbol_);
    require(initialKeeper_ != address(0), "GenesisVault: invalid initial keeper");

    // ⊠ MISSING: baseVolContract_ validation
    $.baseVolContract = baseVolContract_;  // If address(0) → complete vault DoS

    // If baseVolContract = address(0):
    // → LibGenesisVault.getCurrentEpoch() reverts
    // → ALL vault operations DoS
}
```

5. ClearingHouse.sol - MISSING OPERATOR VAULT VALIDATION

```
// Line 133: Has OZ initializer, validates 4 addresses but NOT operatorVaultAddress
function initialize(..., address _operatorVaultAddress) public initializer {
    __UUPSUpgradeable_init();

    // Lines 139-142: ⊠ Validates 4 addresses
    if (_usdcAddress == address(0)) revert InvalidAddress();
    if (_adminAddress == address(0)) revert InvalidAddress();

    // ⊠ MISSING: _operatorVaultAddress validation
    $.operatorVaultAddress = _operatorVaultAddress;  // Can be address(0)
}
```

6. BaseVolManager.sol - MISSING STRATEGY VALIDATION

```
// Line 74: Has OZ initializer, validates clearingHouse but NOT strategy
function initialize(address _clearingHouse, address _strategy) external initializer {
    __UUPSUpgradeable_init();

    require(_clearingHouse != address(0), "Invalid ClearingHouse address");  // ⊠

    // ⊠ MISSING: _strategy validation
    address _asset = IGenesisStrategy(_strategy).asset();  // Will revert if strategy = 0!
}
```

7. PythLazer.sol - MISSING TOP AUTHORITY VALIDATION

```
// Line 22: Has OZ initializer but NO input validation
function initialize(address _topAuthority) public initializer {
    __Ownable_init(_topAuthority);  // ⊠ No validation before passing
    __UUPSUpgradeable_init();

    // If _topAuthority = address(0):
    // → OZ Ownable sets owner = address(0)
    // → All onlyOwner functions permanently inaccessible
}
```

Impact:

MEDIUM - Deployment Bricking + Operational Failures. Severity varies by contract:

(1) InitializationFacet/BaseVolStrike - `intervalSeconds=0` → Complete protocol DoS, requires redeployment.

(2) GenesisVault - `baseVolContract=0` → Vault completely unusable, all operations revert on `getCurrentEpoch()`.

(3) BaseVolOneMin/ClearingHouse/BaseVolManager/PythLazer - Zero addresses → Silent failures during operations, confusing deployment errors, poor UX. All issues preventable with input validation, typically caught in testing but lack of defensive programming creates unnecessary risk from typos, script errors, or wrong parameter order.

Source:

- Multiple initialization functions lacking proper input validation
- Scope: 7 contracts affected
- Affected files:
  - InitializationFacet.sol (facets/ and basevol/facets/)
  - BaseVolStrike.sol
  - BaseVolOneMin.sol
  - GenesisVault.sol
  - ClearingHouse.sol
  - BaseVolManager.sol
  - PythLazer.sol

Proof of Concept:

Deploy script error: `baseVolContract_ = address(0)` (no validation). Vault deploys successfully. User calls `requestDeposit()` -> internally calls `getCurrentEpoch()` -> tries `IBaseVol(address(0)).currentEpoch()` -> reverts “Call to address(0)” -> all vault operations DoSed -> requires redeployment.

Remediation:

```solidity
// STANDARD VALIDATION TEMPLATE (apply to all 7 contracts)

// FOR CONTRACTS WITH intervalSeconds (InitializationFacet, BaseVolStrike):
function initialize(..., uint256 _intervalSeconds) public initializer {
    // ⊠ Existing re-init protection
    __UUPSUpgradeable_init();

    // ⊠ ADD: Time parameter validation
    require(_intervalSeconds >= 60 && _intervalSeconds <= 7 days, "Invalid interval");
    require(_startTimestamp >= block.timestamp - 7 days, "Invalid start time");

    // ⊠ ADD: Address validation
    require(_usdcAddress != address(0), "Invalid USDC address");
    require(_oracleAddress != address(0), "Invalid oracle address");
    require(_adminAddress != address(0), "Invalid admin address");
    require(_operatorAddress != address(0), "Invalid operator address");
    require(_clearingHouseAddress != address(0), "Invalid clearing house address");

    // ... rest of initialization
}

// FOR GenesisVault.sol (CRITICAL - prevents vault DoS):
function initialize(
    address baseVolContract_,
    address asset_,
    address initialKeeper_,
    ...
) external initializer {
    // ⊠ ADD: Validate BEFORE parent init
    require(asset_ != address(0), "GenesisVault: invalid asset");
    require(baseVolContract_ != address(0), "GenesisVault: invalid baseVol contract");
    require(initialKeeper_ != address(0), "GenesisVault: invalid initial keeper");

    // ⊠ ADD: Verify baseVolContract has code (is deployed contract)
    require(baseVolContract_.code.length > 0, "GenesisVault: baseVol must be contract");

    __GenesisManagedVault_init(msg.sender, msg.sender, asset_, name_, symbol_);
    // ... rest of initialization
}

// FOR ALL OTHER CONTRACTS: Add zero-address checks for ALL address parameters
function initialize(...) external initializer {
    __UUPSUpgradeable_init();

    // ⊠ Validate ALL addresses
    require(_param1 != address(0), "Invalid address");
    require(_param2 != address(0), "Invalid address");

    // ... rest of initialization
}

// BEST PRACTICE: Follow validation order
// 1. Call parent initializers (__UUPSUpgradeable_init, etc.)
// 2. Validate ALL addresses (zero-address checks)
// 3. Validate numeric parameters (bounds checks)
// 4. Validate time parameters (reasonable ranges)
// 5. Initialize storage
```

Discussion:

Developer:

This is fixed in latest commit.

Auditor:

VERIFIED FIXED(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Finding 27: Unbounded Loops and O(n) Operations Cause DoS at Scale

Severity: ! Medium

Status: Resolved

Description:

Seven distinct unbounded loop and O(n) operation issues create DoS vectors at scale.

(1) `couponBalanceOf` PUBLIC View DoS [CRITICAL]: PUBLIC view function iterates over ALL user coupons. Anyone can call it. Users with 1000+ coupons cause out-of-gas reverts, permanently blocking frontends, analytics tools, and integrations from displaying balances.

(2) `calculateMaxDepositRequest`: Iterates up to 50 user epochs (~105k gas), causing expensive deposits and potential failures at scale.

(3) `totalAssets` Memory Overhead: Uses memory arrays with 68x overhead vs direct storage iteration, causing unnecessary gas consumption.

(4) `executeForceWithdraw` Linear Search: O(n) array search for withdrawal requests, becoming expensive with many requests.

(5) Balance Operations: Various O(n) balance calculations without optimization.

(6) Signer Array Validation: O(n) iteration over signers array without bounds.

(7) Keeper Array Unbounded: No `MAX_KEEPERS` limit allows compromised owner to add thousands of keepers causing permanent DoS on keeper operations.

Impact:

MEDIUM to HIGH - PUBLIC Function DoS + Operational Degradation. Seven failure modes with varying severity:

(1) `couponBalanceOf` [CRITICAL PUBLIC DoS]: Users with 1000+ coupons cannot be queried by frontends (out-of-gas). Analytics dashboards broken. DEX aggregators fail to display balances. Portfolio trackers unusable. This is the ONLY externally exploitable issue - any attacker can force users to accumulate coupons making their balances unqueryable forever. **Estimated impact:** 20-30% of active users eventually accumulate 1000+ coupons through normal usage -> their balances become invisible to all integrations.

(2) `calculateMaxDepositRequest`: 50 epochs x 2.1k gas = 105k gas overhead on deposits. Poor UX, users pay excessive gas. At scale (100+ epochs) may hit gas limits.

(3) **`totalAssets`**: 68x memory overhead wastes gas on every view call. Not critical but poor optimization.

(4) **`executeForceWithdraw`**: Linear search creates O(n) withdrawal costs. With 10k requests, finding user's request costs 200k+ gas.

(5) Balance operations: Various O(n) ops degrade performance at scale.

(6) Signer validation: O(n) on every price update. Scales poorly with multiple signers.

(7) Keeper array: Compromised owner adding 10k keepers creates permanent DoS on all keeper operations (settlement, price updates). **Combined impact:** #1 is production-blocking (external DoS), others create operational friction and scaling issues.

Source:

- contracts/core/ClearingHouse.sol (`couponBalanceOf` - PUBLIC function)
- contracts/genesis-vault/libraries/LibGenesisVault.sol (`calculateMaxDepositRequest`, `totalAssets`)
- contracts/core/ClearingHouse.sol (`executeForceWithdraw`)
- contracts/core/vault/VaultManager.sol (balance operations)
- contracts/libraries/PythLazer.sol (signer validation)
- contracts/genesis-vault/facets/KeeperFacet.sol (keeper management)

Code:

```
// ISSUE 1: couponBalanceOf PUBLIC view (ClearingHouse.sol) - CRITICAL
function couponBalanceOf(address user, uint256 productId)
    public view returns (uint256) {  // PUBLIC - anyone can call

    uint256[] memory userCoupons = $.userCoupons[user];
    uint256 balance = 0;

    // UNBOUNDED LOOP - iterates over ALL coupons
    for (uint256 i = 0; i < userCoupons.length; i++) {
        uint256 couponId = userCoupons[i];
        Coupon storage coupon = $.coupons[couponId];

        if (coupon.productId == productId) {
            balance += coupon.amount;
        }
    }

    return balance;

    // User with 1000 coupons:
    // - Gas: 1000 x ~20k = 20M gas
    // - Block limit: 30M gas
    // - Frontends calling this -> REVERT "Out of Gas"
    // - Analytics tools -> BROKEN
    // - Integrations -> FAIL
}

// ISSUE 2: calculateMaxDepositRequest (LibGenesisVault.sol)
```

```
function _calculateMaxDepositRequest(address receiver) internal view returns (uint256) {
    uint256[] memory userEpochs = $.userDepositEpochs[receiver];
    uint256 userPendingAssets = 0;

    // Up to 50 iterations
    uint256 epochsToCheck = userEpochs.length > 50 ? 50 : userEpochs.length;
    for (uint256 i = 0; i < epochsToCheck; i++) {
        uint256 epoch = userEpochs[userEpochs.length - 1 - i];
        if (!$.roundData[epoch].isSettled) {
            userPendingAssets += $.userEpochDepositAssets[receiver][epoch];
        }
    }

    // Gas: 50 x ~2.1k = 105k gas
    // Called on every deposit -> expensive UX
}

// ISSUE 3: totalAssets with memory overhead (LibGenesisVault.sol)
function totalAssets() public view returns (uint256) {
    // Creates memory array (expensive)
    uint256[] memory assets = new uint256[](vaultCount);

    for (uint256 i = 0; i < vaultCount; i++) {
        assets[i] = vault[i].balance;  // Memory write
    }

    uint256 total = 0;
    for (uint256 i = 0; i < assets.length; i++) {
        total += assets[i];  // Memory read
    }

    // Memory array: 68x more expensive than direct storage iteration
}

// ISSUE 4: executeForceWithdraw linear search (ClearingHouse.sol)
function executeForceWithdraw(uint256 amount) external {
    WithdrawalRequest[] storage requests = $.withdrawalRequests;

    // O(n) linear search
    for (uint256 i = 0; i < requests.length; i++) {
        if (requests[i].user == msg.sender && !requests[i].processed) {
            // Process request
            break;
        }
    }

    // With 10,000 requests: 10k iterations to find user's request
}

// ISSUE 5: Signer array O(n) (PythLazer.sol)
function isValidSigner(address signer) public view returns (bool) {
    address[] storage signers = trustedSigners;

    // O(n) iteration
    for (uint256 i = 0; i < signers.length; i++) {
        if (signers[i] == signer) {
            return block.timestamp < expiresAt[signer];
        }
    }
    return false;
}

// ISSUE 6: Keeper array unbounded (KeeperFacet.sol)
function addKeeper(address keeper) external onlyAdmin {
    // NO MAX_KEEPERS LIMIT
    $.keepers.push(keeper);

    // Compromised admin adds 10,000 keepers
    // -> All keeper operations iterate 10k times
    // -> Permanent DoS
}

modifier onlyKeeper() {
    bool isKeeper = false;
    // O(n) check on EVERY keeper call
    for (uint i = 0; i < $.keepers.length; i++) {
        if (msg.sender == $.keepers[i]) {
            isKeeper = true;
            break;
        }
    }
    require(isKeeper, "Not a keeper");
    _;
}
```

Proof of Concept:

Active user accumulates 1800 coupons over 6 months. Frontend calls `couponBalanceOf(user)` -> iterates 1800 times -> requires 36M gas -> exceeds 30M block limit -> reverts “Out of Gas” -> user balance unqueryable. Compromised owner adds 10k keepers -> `onlyKeeper` modifier loops 10k times -> 200M gas -> exceeds block limit -> all keeper operations DoSed.

Remediation:

```
// FIX 1: Add pagination to couponBalanceOf (CRITICAL)
function couponBalanceOf(
    address user,
    uint256 productId,
    uint256 startIndex,  // Add pagination
    uint256 maxResults
) public view returns (uint256 balance, uint256 nextIndex) {
    uint256[] memory userCoupons = $.userCoupons[user];
    uint256 endIndex = Math.min(startIndex + maxResults, userCoupons.length);

    // Limited iteration
    for (uint256 i = startIndex; i < endIndex; i++) {
        uint256 couponId = userCoupons[i];
        Coupon storage coupon = $.coupons[couponId];

        if (coupon.productId == productId) {
            balance += coupon.amount;
        }
    }

    nextIndex = endIndex < userCoupons.length ? endIndex : 0;
    return (balance, nextIndex);
}

// Frontend usage:
uint256 totalBalance = 0;
uint256 nextIndex = 0;
do {
    (uint256 pageBalance, uint256 next) = couponBalanceOf(user, productId, nextIndex, 100);
    totalBalance += pageBalance;
    nextIndex = next;
} while (nextIndex > 0);

// FIX 2: Cache pending assets in storage
mapping(address => uint256) public cachedPendingAssets;

function requestDeposit(...) external {
    // ... deposit logic ...
    cachedPendingAssets[msg.sender] += netAssets;  // Update cache
}
```

```

function _calculateMaxDepositRequest(address receiver) internal view returns (uint256) {
    // O(1) lookup instead of O(n) loop
    return cachedPendingAssets[receiver];
}

// FIX 3: Direct storage iteration for totalAssets
function totalAssets() public view returns (uint256 total) {
    // Direct storage iteration (no memory array)
    for (uint256 i = 0; i < vaultCount; i++) {
        total += vault[i].balance;
    }
}

// FIX 4: Use mapping for O(1) withdrawal lookup
mapping(address => uint256[]) public userWithdrawalRequests;

function requestWithdrawal(uint256 amount) external {
    uint256 idx = $.withdrawalRequests.length;
    $.withdrawalRequests.push(...);
    userWithdrawalRequests[msg.sender].push(idx);  // Track user's requests
}

function executeForceWithdraw(uint256 requestIdx) external {
    // O(1) access instead of O(n) search
    WithdrawalRequest storage request = $.withdrawalRequests[requestIdx];
    require(request.user == msg.sender, "Not your request");
    // ... process withdrawal
}

// FIX 5: Use mapping for O(1) signer validation
mapping(address => bool) public isValidSignerMapping;

function addSigner(address signer, uint256 expiresAt) external onlyOwner {
    isValidSignerMapping[signer] = true;
    signerExpiry[signer] = expiresAt;
}

function isValidSigner(address signer) public view returns (bool) {
    // O(1) lookup
    return isValidSignerMapping[signer] &&
           block.timestamp < signerExpiry[signer];
}

// FIX 6: Add MAX_KEEPERS limit
uint256 public constant MAX_KEEPERS = 50;

function addKeeper(address keeper) external onlyAdmin {
    require($.keepers.length < MAX_KEEPERS, "Max keepers reached");
    $.keepers.push(keeper);
}

// Use mapping for O(1) keeper check
mapping(address => bool) public isKeeper;

modifier onlyKeeper() {
    require(isKeeper[msg.sender], "Not a keeper");  // O(1)
    _;
}

// FIX 7: Add batch keeper removal
function removeKeepersBatch(address[] calldata keepers) external onlyOwner {
    for (uint256 i = 0; i < keepers.length; i++) {
        isKeeper[keepers[i]] = false;
    }
}
```

Discussion:

> *Developer:*
>
> Fixed in commit 9874d6e4bbdcde40d1cf0c41c7b98c487c6a7def

> *Auditor:*
>
> **VERIFIED FIXED**(commit 9874d6e4bbdcde40d1cf0c41c7b98c487c6a7def).

Finding 28: Vault Shares Transferable When Paused - Design Ambiguity

Severity: ● Medium

Status: Resolved

Description:

ERC20 `transfer()` and `transferFrom()` functions for vault shares have NO pause check, allowing shares to be freely traded even when vault is paused or shutdown. This creates design ambiguity: Is it **intentional** (allowing secondary market trading during emergencies) or **oversight** (should block all activity)? Other vault functions correctly implement pause checks: `requestDeposit` reverts when paused, `requestRedeem` reverts when paused, but transfer functions bypass pause entirely.

If unintentional, creates security risk during emergency response. If intentional, lacks documentation and granular control (cannot pause transfers separately from deposits/withdrawals). This ambiguity creates either a security vulnerability OR a missing feature depending on intended design.

Impact:

MEDIUM - Emergency Control Bypass OR Missing Feature. Two interpretations with different impacts:

If UNINTENTIONAL (Security Issue):

(1) Emergency Control Bypassed: Admin pauses vault during incident (e.g., oracle failure) to prevent new activity. Users can still transfer shares -> circumvents emergency controls.

(2) Front-Running During Response: Insider learns of vulnerability before public pause. Transfers shares to accomplice before pause announced. Pause blocks deposits/redeems but not transfers -> accomplice can sell shares OTC.

(3) Price Manipulation During Pause: Whale transfers large share amounts to multiple wallets during pause, creating artificial distribution for governance or other purposes.

(4) Incomplete Emergency Freeze: Pause intended to freeze ALL activity but transfers continue -> partial effectiveness.

If INTENTIONAL (Missing Documentation):

(1) Undocumented Design: Secondary market trading allowed during pause but not documented -> users/auditors confused.

(2) No Granular Control: Cannot pause ONLY transfers while allowing redeems, or vice versa. Single pause flag affects only deposits/redeems, not transfers.

(3) Inconsistent Behavior: Other vault operations respect pause, transfers don't -> confusing UX.

Source:

- contracts/genesis-vault/facets/ERC20Facet.sol (`transfer` line 21, `transferFrom` line 46)

Code:

```
// ERC20Facet.sol line 21 - NO PAUSE CHECK
function transfer(address to, uint256 amount)
    public override returns (bool) {

    // NO PAUSE CHECK
    // NO SHUTDOWN CHECK

    address owner = _msgSender();
    _transfer(owner, to, amount);  // Executes even when paused
    return true;
}

// ERC20Facet.sol line 46 - NO PAUSE CHECK
function transferFrom(address from, address to, uint256 amount)
    public override returns (bool) {

    // NO PAUSE CHECK
    // NO SHUTDOWN CHECK

    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);  // Executes even when paused
    return true;
}

// Compare: VaultCoreFacet.sol - HAS PAUSE CHECK
function requestDeposit(...) external {
    require(!paused() && !isShutdown(), "Vault not active");  // Checks pause
    // ... deposit logic
}

function requestRedeem(...) external {
    require(!paused() && !isShutdown(), "Vault not active");  // Checks pause
    // ... redeem logic
}
```

Proof of Concept:

Admin pauses vault for emergency. User Alice: `requestDeposit()` -> reverts "Vault not active". Insider Charlie: `transfer(accomplice, 1000 shares)` -> succeeds (no pause check) -> insider circumvents emergency controls -> accomplice can sell OTC.

Remediation:

```
// OPTION 1: If Full Freeze Intended (Security-First Approach)
function transfer(address to, uint256 amount)
    public override returns (bool) {
```

```

    // Add pause check
    require(!paused() && !isShutdown(), "Vault not active");

    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}

function transferFrom(address from, address to, uint256 amount)
    public override returns (bool) {

    // Add pause check
    require(!paused() && !isShutdown(), "Vault not active");

    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, amount);
    return true;
}

// OPTION 2: If Transfers Should Be Allowed (Granular Control)
bool public transfersPaused;  // Separate flag for transfer pause

function setTransfersPaused(bool _paused) external onlyOwner {
    transfersPaused = _paused;
    emit TransfersPausedUpdated(_paused);
}

function transfer(address to, uint256 amount)
    public override returns (bool) {

    // Check separate transfersPaused flag
    require(!transfersPaused, "Transfers paused");
    // Note: Does NOT check general pause
    // Allows transfers during vault pause if transfersPaused = false

    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}

// OPTION 3: Most Flexible (Recommended)
bool public depositsPaused;
bool public redeemsPaused;
bool public transfersPaused;

function pause() external onlyOwner {
    // Pause all activities
    depositsPaused = true;
    redeemsPaused = true;
    transfersPaused = true;  // Now also pauses transfers
    emit VaultPaused(block.timestamp);
}

function pauseSelective(bool _deposits, bool _redeems, bool _transfers)
    external onlyOwner {
    // Granular control: pause specific activities
    depositsPaused = _deposits;
    redeemsPaused = _redeems;
    transfersPaused = _transfers;
    emit SelectivePause(_deposits, _redeems, _transfers);
}

function transfer(address to, uint256 amount)
    public override returns (bool) {

    require(!transfersPaused, "Transfers paused");

    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}

// RECOMMENDED: Clarify design intent in documentation
/**

 * @notice Transfer vault shares to another address
 * @dev DESIGN DECISION: Transfers are NOT blocked by general pause.
 *      This allows secondary market trading during vault maintenance.
 *      Use `setTransfersPaused(true)` to specifically block transfers.
 * @param to Recipient address
 * @param amount Amount of shares to transfer
 */
function transfer(address to, uint256 amount) public override returns (bool) {
    // Intentionally no pause check - design allows transfers during pause
    // See documentation for rationale

    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}
```

Discussion:

> *Developer:*
>
> Fixed in commit d7b28240986eed0157dd5b011c1371e5349e7354

> *Auditor:*
>
> **VERIFIED FIXED**(commit d7b28240986eed0157dd5b011c1371e5349e7354).

Finding 29: ETH Transfers Use Deprecated transfer() with 2300 Gas Stipend

Severity: Low

Status: Resolved

Description:

Two functions use `address.transfer()` for ETH transfers, which has a hardcoded 2300 gas stipend. This is a deprecated pattern that can fail when recipient is a contract with non-trivial `receive()` or `fallback()` functions.

(1) retrieveMisplacedETH (BaseVolAdminFacet): Uses `payable(bvs.adminAddress).transfer(address(this).balanc` to recover misplaced ETH. If admin address is a multisig or contract with expensive fallback, transfer fails and ETH remains stuck. Additionally, function doesn't check if balance > 0 before attempting transfer, wasting gas on zero-balance calls.

(2) PythLazer Oracle Refunds: Uses `payable(msg.sender).transfer(excessAmount)` to refund excess oracle verification fees. If caller is a contract with expensive receive() (common for proxy patterns, multisigs, or contracts with logging), refund fails and user loses gas refund. Modern Solidity best practice is to use `call()` which forwards all available gas and doesn't have the 2300 gas limit. The 2300 stipend was originally designed to prevent reentrancy, but now we have explicit reentrancy guards which are more robust.

Impact:

LOW - Transfer Failures to Contract Recipients.

(1) retrieveMisplacedETH: If admin is multisig or contract, ETH recovery fails. ETH remains stuck until admin changed or contract upgraded. Not urgent since ETH in BaseVol contracts is unexpected.

(2) PythLazer Refunds: Users calling from contracts (common with account abstraction, proxy patterns) lose gas refunds. Main transaction succeeds but refund fails silently. User experience degradation. Not a security vulnerability since main oracle update succeeds, only refund affected. Modern best practice is to use `call()` which works with all recipient types.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (retrieveMisplacedETH, line 52-55)
- contracts/libraries/PythLazer.sol (verifyUpdate, line 73)

Code:

```
// (1) BaseVolAdminFacet.sol line 52-55 - retrieveMisplacedETH
function retrieveMisplacedETH() external onlyAdmin {
    payable(bvs.adminAddress).transfer(address(this).balance);
    // Uses transfer() with 2300 gas limit
    // Fails if adminAddress is multisig/contract with expensive fallback
    // No check if balance > 0 (gas waste)
}

// (2) PythLazer.sol line 73 - Oracle fee refund
function verifyUpdate(bytes calldata updateData) internal returns (...) {
    // ... verify oracle update ...

    uint256 fee = IPyth(pyth).getUpdateFee(priceIds);
    require(msg.value >= fee, "Insufficient fee");

    uint256 excessAmount = msg.value - fee;
    if (excessAmount > 0) {
        payable(msg.sender).transfer(excessAmount);
        // Uses transfer() with 2300 gas limit
        // Fails if msg.sender is contract with expensive receive()
        // User loses gas refund
    }
}
```

Proof of Concept:

Admin is Gnosis Safe multisig -> Safe's fallback costs >2300 gas -> admin calls `retrieveMisplacedETH()` -> `transfer()` limited to 2300 gas -> reverts "Failed to send ETH" -> ETH stuck. Similarly: user calls oracle via proxy -> proxy's receive() costs >2300 gas -> refund transfer fails -> user loses gas refund (oracle update succeeds but refund lost).

Remediation:

```
// (1) Fix retrieveMisplacedETH - Use call() with balance check
function retrieveMisplacedETH() external onlyAdmin {
    uint256 balance = address(this).balance;
    require(balance > 0, "No ETH to retrieve");  // Gas savings

    (bool success, ) = payable(bvs.adminAddress).call{value: balance}("");
    require(success, "ETH transfer failed");

    emit ETHRetrieved(bvs.adminAddress, balance);  // Also add event
}

// (2) Fix PythLazer refunds - Use call() with success check
function verifyUpdate(bytes calldata updateData) internal returns (...) {
    // ... verify oracle update ...

    uint256 fee = IPyth(pyth).getUpdateFee(priceIds);
    require(msg.value >= fee, "Insufficient fee");

    uint256 excessAmount = msg.value - fee;
    if (excessAmount > 0) {
        (bool success, ) = payable(msg.sender).call{value: excessAmount}("");
        // Don't revert if refund fails - main update should succeed
        if (!success) {
            // Optional: Store failed refund for later claim
            emit RefundFailed(msg.sender, excessAmount);
        }
    }
}

// ALTERNATIVE: Store failed refunds for manual claim
```

```
mapping(address => uint256) public pendingRefunds;

function verifyUpdate(...) internal returns (...) {
    // ...
    if (excessAmount > 0) {
        (bool success, ) = payable(msg.sender).call{value: excessAmount}("");
        if (!success) {
            pendingRefunds[msg.sender] += excessAmount;
            emit RefundPending(msg.sender, excessAmount);
        }
    }
}

function claimPendingRefund() external {
    uint256 amount = pendingRefunds[msg.sender];
    require(amount > 0, "No pending refund");
    pendingRefunds[msg.sender] = 0;
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Refund claim failed");
}

// BEST PRACTICE: Always use call() for ETH transfers
// - Forwards all available gas
// - Works with all recipient types (EOAs, contracts, multisigs)
// - Check return value and handle failures appropriately
// - Consider storing failed transfers for later claim
```

Discussion:

Developer:

Fixed in 371ced2e6eaf4ac38d80b3b86782dda8b901f4ce

Auditor:

VERIFIED FIXED(commit 371ced2e6eaf4ac38d80b3b86782dda8b901f4ce).

Finding 30: Missing Input Validation for Critical Admin Parameters Across Multiple Functions

Severity: ● Low

Status: Resolved

Description:

Multiple critical admin functions across the protocol lack proper input validation, creating operational risks from human error. This comprehensive finding covers five distinct validation gaps:

(1) Owner Zero-Address (Deployment): Diamond constructor accepts `_contractOwner` without checking for `address(0)`, risking permanent loss of owner control if deployed incorrectly.

(2) Zero-Unit Validation (Critical DoS): `setUnit()` allows admin to set `unit = 0`, causing immediate division-by-zero in ALL settlement calculations, breaking the entire protocol until fixed.

(3) Order Price Sum Validation: `submitFilledOrders()` doesn't validate that `overPrice + underPrice` equals expected total, allowing invalid accounting data to be processed.

(4) Constructor Parameters: LibDiamond functions don't validate critical address parameters for zero-address during diamond upgrades.

(5) Signer Expiration: `updateTrustedSigner()` doesn't validate that `expiresAt > block.timestamp`, allowing admin to accidentally set expiration in the past, immediately invalidating signers. While admin/owner are trusted roles, lack of validation creates unnecessary operational risk from typos, copy-paste errors, or script bugs.

Impact:

LOW - Multiple Admin Misconfiguration Risks. Impact varies by validation type:

(1) Owner Zero-Address: Deployment-only risk, would be caught in testing before mainnet.

(2) Zero-Unit DoS: MOST CRITICAL - Breaks ALL settlements immediately, protocol-wide DoS until admin fixes. Recovery requires emergency admin action.

(3) Price Sum: Invalid accounting data processed, gas waste, confusion in settlement records.

(4) Constructor Parameters: Diamond deployment failure, caught in testing.

(5) Signer Expiration: Oracle signers instantly invalid, protocol cannot update prices until fixed. All issues require admin/owner mistakes to trigger, but lack of validation means mistakes aren't caught at transaction time.

Source:

- contracts/Diamond.sol (constructor, lines 8-21)
- contracts/basevol/facets/BaseVolAdminFacet.sol (setUnit function)
- contracts/basevol/facets/OrderProcessingFacet.sol (submitFilledOrders)
- contracts/diamond-common/libraries/LibDiamond.sol (setContractOwner, lines 34-38)
- contracts/libraries/PythLazer.sol (updateTrustedSigner, lines 31-61)

Code:

```
// (1) Diamond.sol - No owner zero-address check
constructor(address _contractOwner, address _diamondCutFacet) payable {
    LibDiamond.setContractOwner(_contractOwner);  // No zero check
}

// (2) BaseVolAdminFacet.sol - CRITICAL: No unit zero check (DoS risk)
function setUnit(uint256 _unit) external onlyAdmin {
    bvs.unit = _unit;  // If unit = 0, ALL settlements revert on division
    // Settlement math: amount / unit -> Division by zero!
}

// (3) OrderProcessingFacet.sol - No price sum validation
function submitFilledOrders(...) {
    // Process order with overPrice and underPrice
    // No check: require(overPrice + underPrice == expectedTotal)
}

// (4) LibDiamond.sol - No parameter validation
function setContractOwner(address _newOwner) internal {
    ds.contractOwner = _newOwner;  // No zero check
}

// (5) PythLazer.sol - No expiration validation
function updateTrustedSigner(address signer, uint256 expiresAt, ...) {
    // No check: require(expiresAt > block.timestamp)
    // Admin could set expiration in past, instantly invalidating signer
}
```

Proof of Concept:

Admin calls `setUnit(0)` by mistake (no validation) -> transaction succeeds -> next settlement: `amount / 0` -> reverts -> ALL settlements broken. Similarly: deploy with `owner = address(0)` -> all owner functions inaccessible forever. Set signer expiration in past -> price updates rejected immediately.

Remediation:

```
// (1) Add owner zero-address validation
constructor(address _contractOwner, address _diamondCutFacet) payable {
    require(_contractOwner != address(0), "Owner cannot be zero address");
    LibDiamond.setContractOwner(_contractOwner);
}

// (2) Add zero-unit validation with reasonable bounds (CRITICAL)
function setUnit(uint256 _unit) external onlyAdmin {
    require(_unit > 0, "Unit cannot be zero");
    require(_unit >= 1e6 && _unit <= 1e18, "Unit out of bounds");
    uint256 oldUnit = bvs.unit;
    bvs.unit = _unit;
    emit UnitUpdated(oldUnit, _unit, block.timestamp);  // Also add event
}
```

```

// (3) Add price sum validation
function submitFilledOrders(...) {
    // For each order:
    require(
        order.overPrice + order.underPrice == expectedTotal,
        "Price sum mismatch"
    );
}

// (4) Add parameter validation in LibDiamond
function setContractOwner(address _newOwner) internal {
    require(_newOwner != address(0), "Owner cannot be zero address");
    ds.contractOwner = _newOwner;
}

// (5) Add expiration validation
function updateTrustedSigner(address signer, uint256 expiresAt, ...) {
    require(expiresAt > block.timestamp, "Expiration must be in future");
    require(expiresAt <= block.timestamp + 365 days, "Expiration too far");
    // ... rest of logic
}

// GENERAL PATTERN: Add validation to ALL admin functions
// - Zero-address checks for address parameters
// - Non-zero checks for division operands
// - Range checks for bounded values
// - Timestamp checks for expiration times
// - Sum/consistency checks for related values
```

Discussion:

> *Developer:*
>
> Fixed in 92c690c2ac9e109fc6314322188ee564306dbfdf

> *Auditor:*
>
> **VERIFIED FIXED**(commit 92c690c2ac9e109fc6314322188ee564306dbfdf).

Finding 31: Missing or Improperly Timed Event Emissions for Critical Admin Operations

Severity: Low

Status: Resolved

Description:

Critical admin operations and state changes lack proper event emissions, creating significant observability gaps. This comprehensive finding covers four distinct event emission issues:

(1) Missing Events for Admin State Changes: Functions like `setCommissionfee()`, `setLastFilledOrderId()`, `setUnit()`, `setOracle()` modify critical protocol parameters but don't emit events. Off-chain systems cannot track configuration changes without expensive storage polling.

(2) Missing Event Differentiation: `setPriceInfo()` emits the same `PriceIdAdded` event for both adding new priceIds AND updating existing ones. Off-chain indexers cannot distinguish between additions and updates without tracking state.

(3) Missing Initialization Events: `GenesisVaultInitializationFacet.initialize()` completes initialization without emitting an event. In multi-step deployment processes, difficult to track when vault becomes operational.

(4) Event Emitted Before Initialization: LibDiamond emits `DiamondCut` event BEFORE calling `_init.delegatecall()`. If initialization reverts, event was already emitted, creating inconsistency where off-chain systems index a successful cut that actually failed. Events are critical for monitoring, auditing, debugging, and building reliable off-chain indexers.

```
// (1) BaseVolAdminFacet.sol - NO events for state changes
function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    bvs.commissionfee = _commissionfee;
    // No event emission - off-chain can't track changes
}

function setUnit(uint256 _unit) external onlyAdmin {
    bvs.unit = _unit;
    // No event emission - critical parameter change invisible
}

function setOracle(address _oracle) external onlyAdmin {
    bvs.oracle = IPyth(_oracle);
    // No event emission - oracle changes untracked
}

function setLastFilledOrderId(uint256 _lastFilledOrderId) external onlyAdmin {
    bvs.lastFilledOrderId = _lastFilledOrderId;
    // No event emission - order ID changes invisible
}

// (2) BaseVolAdminFacet.sol - Same event for different operations
function setPriceInfo(...) external onlyAdmin {
    if (bvs.productAddresses[_productId] == address(0)) {
        // NEW price ID
        _addPriceId(_productId, _productAddress, _priceFeedId);
        emit PriceIdAdded(_productId, _productAddress, _priceFeedId);
    } else {
        // UPDATING existing price ID
        bvs.productAddresses[_productId] = _productAddress;
        bvs.priceFeedIds[_productId] = _priceFeedId;
        emit PriceIdAdded(_productId, _productAddress, _priceFeedId);
        // Same event for update - indexers can't distinguish
    }
}

// (3) GenesisVaultInitializationFacet.sol - No initialization event
function initialize(...) external {
    // ... initialization logic ...
    // No event emission - can't track when vault is initialized
}

// (4) LibDiamond.sol - Event emitted BEFORE initialization
function diamondCut(...) internal {
    // ... add/replace/remove facets ...

    emit DiamondCut(_diamondCut, _init, _calldata);  // Line 77: Event FIRST
    initializeDiamondCut(_init, _calldata);          // Line 78: Init SECOND

    // If initializeDiamondCut reverts, event was already emitted
    // Off-chain systems see successful cut but transaction actually failed
}
```

Impact:

LOW - Observability and Monitoring Degradation. No security vulnerabilities but significant operational impact:

(1) Missing Admin Events: Off-chain monitoring systems cannot track critical parameter changes. Requires expensive storage polling to detect configuration changes. Security auditors cannot review parameter history. Difficult to debug configuration-related issues.

(2) Missing Event Differentiation: Indexers must maintain state to distinguish adds from updates. More complex and error-prone off-chain logic. Poor developer experience for integrators.

(3) Missing Initialization Events: Multi-step deployments harder to orchestrate. Cannot reliably detect when vault is ready for use. Deployment scripts must poll storage. **(4) Event Timing Issues:** Off-chain inconsistency if initia

lization fails. Events log successful operations that actually reverted. Indexer data diverges from actual chain state. All issues affect monitoring, debugging, and off-chain integration quality but don't create security vulnerabilities.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (multiple admin functions)
- contracts/basevol/facets/BaseVolAdminFacet.sol (setPriceInfo, lines 107-130)

- contracts/genesis-vault/facets/GenesisVaultInitializationFacet.sol (initialize function)
- contracts/diamond-common/libraries/LibDiamond.sol (diamondCut, lines 77-78)

Code:

```
// (1) BaseVolAdminFacet.sol - NO events for state changes
function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    bvs.commissionfee = _commissionfee;
    // No event emission - off-chain can't track changes
}

function setUnit(uint256 _unit) external onlyAdmin {
    bvs.unit = _unit;
    // No event emission - critical parameter change invisible
}

function setOracle(address _oracle) external onlyAdmin {
    bvs.oracle = IPyth(_oracle);
    // No event emission - oracle changes untracked
}

function setLastFilledOrderId(uint256 _lastFilledOrderId) external onlyAdmin {
    bvs.lastFilledOrderId = _lastFilledOrderId;
    // No event emission - order ID changes invisible
}

// (2) BaseVolAdminFacet.sol - Same event for different operations
function setPriceInfo(...) external onlyAdmin {
    if (bvs.productAddresses[_productId] == address(0)) {
        // NEW price ID
        _addPriceId(_productId, _productAddress, _priceFeedId);
        emit PriceIdAdded(_productId, _productAddress, _priceFeedId);
    } else {
        // UPDATING existing price ID
        bvs.productAddresses[_productId] = _productAddress;
        bvs.priceFeedIds[_productId] = _priceFeedId;
        emit PriceIdAdded(_productId, _productAddress, _priceFeedId);
        // Same event for update - indexers can't distinguish
    }
}

// (3) GenesisVaultInitializationFacet.sol - No initialization event
function initialize(...) external {
    // ... initialization logic ...
    // No event emission - can't track when vault is initialized
}

// (4) LibDiamond.sol - Event emitted BEFORE initialization
function diamondCut(...) internal {
    // ... add/replace/remove facets ...

    emit DiamondCut(_diamondCut, _init, _calldata);  // Line 77: Event FIRST
    initializeDiamondCut(_init, _calldata);          // Line 78: Init SECOND

    // If initializeDiamondCut reverts, event was already emitted
    // Off-chain systems see successful cut but transaction actually failed
}
```

Remediation:

```
// (1) Add events for ALL admin state changes
event CommissionFeeUpdated(uint256 oldFee, uint256 newFee, uint256 timestamp);
event UnitUpdated(uint256 oldUnit, uint256 newUnit, uint256 timestamp);
event OracleUpdated(address oldOracle, address newOracle, uint256 timestamp);
event LastFilledOrderIdUpdated(uint256 oldId, uint256 newId, uint256 timestamp);

```

```
function setCommissionfee(uint256 _commissionfee) external onlyAdmin {
    uint256 oldFee = bvs.commissionfee;
    bvs.commissionfee = _commissionfee;
    emit CommissionFeeUpdated(oldFee, _commissionfee, block.timestamp);
}

function setUnit(uint256 _unit) external onlyAdmin {
    uint256 oldUnit = bvs.unit;
    bvs.unit = _unit;
    emit UnitUpdated(oldUnit, _unit, block.timestamp);
}

function setOracle(address _oracle) external onlyAdmin {
    address oldOracle = address(bvs.oracle);
    bvs.oracle = IPyth(_oracle);
    emit OracleUpdated(oldOracle, _oracle, block.timestamp);
}

// (2) Differentiate events for add vs update
event PriceIdAdded(uint32 indexed productId, address productAddress, bytes32 priceFeedId);
event PriceIdUpdated(
    uint32 indexed productId,
    address oldProductAddress,
    address newProductAddress,
    bytes32 oldPriceFeedId,
    bytes32 newPriceFeedId
);

function setPriceInfo(...) external onlyAdmin {
    if (bvs.productAddresses[_productId] == address(0)) {
        // NEW
        _addPriceId(_productId, _productAddress, _priceFeedId);
        emit PriceIdAdded(_productId, _productAddress, _priceFeedId);
    } else {
        // UPDATE
        address oldAddress = bvs.productAddresses[_productId];
        bytes32 oldFeedId = bvs.priceFeedIds[_productId];

        bvs.productAddresses[_productId] = _productAddress;
        bvs.priceFeedIds[_productId] = _priceFeedId;

        emit PriceIdUpdated(_productId, oldAddress, _productAddress, oldFeedId, _priceFeedId);
    }
}

// (3) Add initialization event
event GenesisVaultInitialized(
    address indexed vault,
    address indexed asset,
    string name,
    string symbol,
    uint256 timestamp
);

function initialize(...) external {
    // ... initialization logic ...
    emit GenesisVaultInitialized(address(this), asset, name, symbol, block.timestamp);
}

// (4) Move event emission AFTER initialization
function diamondCut(...) internal {
    // ... add/replace/remove facets ...

    initializeDiamondCut(_init, _calldata);           // Initialize FIRST
    emit DiamondCut(_diamondCut, _init, _calldata);  // Event AFTER success

    // OR emit separate event for initialization:
    // emit DiamondCut(_diamondCut, _init, _calldata);
    // initializeDiamondCut(_init, _calldata);
    // emit DiamondInitializationComplete(_init);
}

// BEST PRACTICE: Emit events for ALL state changes
// - Include old and new values
// - Include indexed parameters for filtering
// - Include timestamp for historical tracking
// - Emit AFTER state changes succeed
```

Discussion:

> *Developer:*
>
> Fixed in 0d16bb3f3775ff14f5711970fcf3d7b758f22121

> *Auditor:*
>
> **VERIFIED FIXED**(commit 0d16bb3f3775ff14f5711970fcf3d7b758f22121).

Finding 32: View Functions Iterate Unbounded Arrays Without Pagination

Severity: ● Low

Status: Resolved

Description:

Multiple view functions iterate over unbounded arrays without pagination support, creating scalability issues as the protocol grows. This affects two main areas:

(1) BaseVol View Functions: Functions like `userFilledOrders()`, `getAllPriceIds()`, `getAllOrders()` iterate entire arrays regardless of size. As protocol scales to 100+ products or users accumulate 1000+ orders, these view calls become extremely expensive or exceed block gas limit and revert. Frontend integration breaks at scale.

(2) DiamondLoupe View Functions: Functions like `facets()` and `facetFunctionSelectors()` iterate over all facets and perform multiple storage reads per iteration. As diamond grows to 10+ facets with 100+ functions total, view calls become expensive. While this is primarily a UX issue (view functions don't consume user gas in most contexts), it breaks frontend integrations and off-chain systems that rely on these functions. Users cannot view their order history, protocol cannot display product lists, and diamond introspection becomes impossible at scale.

Impact:

LOW - Scalability and Frontend Integration Issues.

(1) User Experience Degradation: As protocol grows, view functions time out or revert. Users cannot view their order history (1000+ orders). Frontend cannot display product lists (100+ products). Dashboard pages break or load extremely slowly.

(2) Off-Chain System Impact: Indexers and monitoring systems cannot call view functions to fetch data. Must resort to event parsing or direct storage access (more complex). Third-party integrations (analytics, aggregators) cannot fetch protocol data.

(3) Diamond Introspection: Cannot inspect diamond structure at scale. Upgrade tools and verification scripts may fail.

(4) Not Urgent: For initial deployment with few users and products, functions work fine. Only becomes critical as protocol scales to production volumes. Frontends can work around with direct storage access, events, or subgraphs, but this is more complex.

Source:

- contracts/basevol/facets/BaseVolViewFacet.sol (`userFilledOrders`, `getAllPriceIds`, `getAllOrders`, etc.)
- contracts/diamond-common/facets/DiamondLoupeFacet.sol (`facets`, `facetFunctionSelectors`, etc.)

Code:

```
// BaseVolViewFacet.sol lines 87-111 - userFilledOrders
function userFilledOrders(address user) external view returns (FilledOrder[] memory) {
    BaseVolStrikeStorage.Layout storage bvs = BaseVolStrikeStorage.layout();
    uint256[] memory orderIds = bvs.userOrders[user];
    FilledOrder[] memory orders = new FilledOrder[](orderIds.length);

    // Iterates ENTIRE array - no pagination
    for (uint256 i = 0; i < orderIds.length; i++) {
        orders[i] = bvs.filledOrders[orderIds[i]];
    }
    return orders;
    // If user has 1000+ orders, this becomes extremely expensive or reverts
}

// getAllPriceIds - iterates all products
function getAllPriceIds() external view returns (...) {
    // Iterates entire product list - no pagination
    for (uint32 i = 0; i < bvs.priceIdCount; i++) {
        // ... return all products
    }
    // If protocol has 100+ products, expensive
}

// DiamondLoupeFacet.sol - facets() and facetFunctionSelectors()
function facets() external view returns (Facet[] memory facets_) {
    LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
    uint256 numFacets = ds.facetAddresses.length;
    facets_ = new Facet[](numFacets);

    // Iterates ALL facets with multiple storage reads per facet
    for (uint256 i; i < numFacets; i++) {
        address facetAddress_ = ds.facetAddresses[i];
        facets_[i].facetAddress = facetAddress_;
        facets_[i].functionSelectors = ds.facetFunctionSelectors[facetAddress_].functionSelectors;
    }
    return facets_;
    // If diamond has 10+ facets with 100+ functions, expensive
}
```

Remediation:

```
// OPTION 1: Add paginated versions of view functions (RECOMMENDED)

// Add paginated userFilledOrders
function userFilledOrders(
    address user,
    uint256 startIdx,
    uint256 count
) external view returns (FilledOrder[] memory, uint256 total) {
    BaseVolStrikeStorage.Layout storage bvs = BaseVolStrikeStorage.layout();
    uint256[] memory orderIds = bvs.userOrders[user];
    uint256 total = orderIds.length;

    // Calculate actual count to return
    uint256 end = startIdx + count;
    if (end > total) end = total;
    uint256 actualCount = end - startIdx;

    FilledOrder[] memory orders = new FilledOrder[](actualCount);
    for (uint256 i = 0; i < actualCount; i++) {
        orders[i] = bvs.filledOrders[orderIds[startIdx + i]];
    }

    return (orders, total);
}

// Keep original for backward compatibility
function userFilledOrders(address user) external view returns (FilledOrder[] memory) {
    // Just call paginated version with max reasonable size
    (FilledOrder[] memory orders, ) = this.userFilledOrders(user, 0, 100);
    return orders;
}

// Add paginated getAllPriceIds
function getAllPriceIds(
    uint256 startIdx,
    uint256 count
) external view returns (PriceInfo[] memory, uint256 total) {
    // ... paginated implementation
}

// OPTION 2: Add DiamondLoupe pagination
function facets(
    uint256 startIdx,
    uint256 count
) external view returns (Facet[] memory facets_, uint256 total) {
    LibDiamond.DiamondStorage storage ds = LibDiamond.diamondStorage();
    uint256 numFacets = ds.facetAddresses.length;
    total = numFacets;

    uint256 end = startIdx + count;
    if (end > total) end = total;
    uint256 actualCount = end - startIdx;

    facets_ = new Facet[](actualCount);
    for (uint256 i = 0; i < actualCount; i++) {
        address facetAddress_ = ds.facetAddresses[startIdx + i];
        facets_[i].facetAddress = facetAddress_;
        facets_[i].functionSelectors = ds.facetFunctionSelectors[facetAddress_].functionSelectors;
    }
    return (facets_, total);
}

// OPTION 3: Add individual getter functions
function getUserOrder(address user, uint256 index) external view returns (FilledOrder memory) {
    // Single order lookup - always efficient
}

function getUserOrderCount(address user) external view returns (uint256) {
    // Just return count - no iteration
}

// BEST PRACTICE:
// - Add paginated versions of all array-returning view functions
// - Keep original functions for backward compatibility (with reasonable limits)
// - Return total count so callers know how many pages exist
// - Document recommended page sizes (e.g., 50-100 items)
// - Consider implementing in phases (add pagination before scaling becomes issue)
```

Discussion:

> *Developer:*
>
> Fixed in 85f690cf105f778deb6c91ebc40e50e552024945

> *Auditor:*
>
> **VERIFIED FIXED**(commit 85f690cf105f778deb6c91ebc40e50e552024945).

Finding 33: Critical Admin Operations Lack Timelock Protection for Both Parameter Changes and Upgrades

Severity: ● Info

Status: Resolved

Description:

Critical admin operations execute immediately without timelock delays, covering two main categories:

(1) Parameter Changes: Functions like `setOracle`, `setCommissionfee`, `setUnit` have immediate effect with only `onlyAdmin` modifier.

(2) Contract Upgrades: DiamondCut upgrades take effect immediately via `LibDiamond.diamondCut` with no delay. Industry best practice for DeFi protocols is to implement timelocks (24-48 hours) on critical operations to give users time to exit if they disagree with changes. While admin/owner is trusted, timelocks provide transparency and user protection against sudden changes, compromised keys, or malicious upgrades.

Impact:

INFO - Best practice recommendation. No immediate security risk since admin/owner is trusted. Improves protocol transparency, user trust, and governance. Gives users exit opportunity before critical changes take effect.

Source:

- contracts/basevol/facets/BaseVolAdminFacet.sol (`setOracle` line 70, `setCommissionfee` line 76, etc.)
- contracts/diamond-common/facets/DiamondCutFacet.sol (`diamondCut` function)

Remediation:

- Implement Timelock contract (e.g., `OpenZeppelin TimelockController`) for both admin parameter changes and diamond upgrades
- Set appropriate delay periods (24-48 hours recommended for production)
- Emit proposal events when timelock operations are scheduled
- Use multi-sig for admin/owner roles
- Document all admin operations and upgrade procedures clearly
- Standard pattern in mature DeFi protocols (Compound, Aave, Uniswap governance)

Discussion:

> *Developer:*
>
> In the beginning, can we set it to 1–2 hours, and then increase it to 24–48 hours once things stabilize? And do we only need to apply this to setOracle, setCommissionFee, and setUnit?

> *Auditor:*
>
> YES to your phased approach, with adjustments: Launch (0-6 months): 6-hour timelock (not 1-2 hours - users need time to react) After stabilization: Increase to 24-48 hours This matches industry standards (Compound, Aave, Uniswap). Functions Requiring Timelock Your list (Oracle, CommissionFee, Unit) is a good start but incomplete. missed to mention it's for all admin functions, 26 functions across all contracts can directly impact user funds or protocol security: Critical Functions (6-hour → 48-hour timelock): BaseVol: setOracle, setCommissionfee, setPythLazer, setOperator, setAdmin, setToken Genesis Vault: setStrategy, setFeeInfos (managementFee, performanceFee), setAdmin, setBaseVolContract, setClearingHouse Other Contracts: setRedeemFee, setWithdrawalFee, setVaultManager, setGenesisVault, setBaseVolManager, setMorphoVaultManager, setTargetAllocations Less Critical (2-hour → 12-hour timelock): setDepositLimits, setEntryAndExitCost, setRebalanceThreshold, setConfig No Timelock Needed: pause() / unpause() (emergency functions must be instant) Implementation Use OpenZeppelin's TimelockController with two delay periods: 6 hours for fund-impacting functions 2 hours for UX-impacting functions Gradually increase both after 6-12 months of stable operation.

> *Developer:*
>
> applied through commit 5b36b2b6ab183962774890dfb4656cf2fc91273b

> *Auditor:*
>
> **VERIFIED FIXED**(commit 5b36b2b6ab183962774890dfb4656cf2fc91273b).

Finding 34: Empty Order Array Causes Array Out-of-Bounds Error

Severity: Info

Status: Resolved

Description:

If `submitFilledOrders` is called with empty `transactions` array, the function attempts to access `transactions[0]` which causes out-of-bounds error. While operator controls this input, adding a simple check improves code robustness and error messages.

Impact:

INFO - Edge case. Operator would never intentionally pass empty array. Better error handling for robustness.

Source:

contracts/facets/OrderProcessingFacet.sol (`submitFilledOrders` , line 25)

Code:

```
function submitFilledOrders(FilledOrder[] calldata transactions) external {
    // Line 25: Accesses transactions[0] without checking length
    if (bvs.lastFilledOrderId + 1 > transactions[0].idx) revert;
    // If transactions.length == 0, this reverts with unhelpful error
}
```

Remediation:

Add `require(transactions.length > 0, "Empty transactions array")` at function start. Provides clear error message.

Discussion:

> *Developer:*
>
> Fixed in 257534c83748f8ea14ae4423de6a18bcae1a5263

> *Auditor:*
>
> **VERIFIED FIXED**(commit 257534c83748f8ea14ae4423de6a18bcae1a5263).

Finding 35: Missing Input Validation on View Function Parameters

Severity: ● Info

Status: Resolved

Description:

View functions don't validate input parameters (e.g., checking if epoch/product ID exists before querying). While view functions don't modify state, missing validation can cause confusing behavior or reverts when invalid parameters are passed, providing poor developer experience.

Impact:

INFO - UX issue for integrators. View functions may revert unexpectedly with unhelpful error messages when passed invalid parameters.

Source:

- contracts/basevol/facets/BaseVolViewFacet.sol (various view functions)

Remediation:

Add input validation to view functions with clear error messages. Example:

```
require(epoch < currentEpoch, "Invalid epoch");
```

Improves developer experience.

Discussion:

> *Developer:*
>
> b311dc10368a7e53bd2717a3631028a534038a0d

> *Auditor:*
>
> **VERIFIED FIXED**(commit b311dc10368a7e53bd2717a3631028a534038a0d).

Disclaimer

This security report ("Report") is provided by FailSafe ("Tester") for the exclusive use of the client ("Client"). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe's prior written consent in each instance.

This Report is not, nor should it be considered, an "endorsement" or "disapproval" of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology's 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 it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe's position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe's goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are 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. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED “AS IS” AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.