Security Review For Aleph Finance

Collaborative Audit Prepared For:	**Aleph Finance**
Lead Security Expert(s):	**0x52**
	Drynooo
Date Audited:	**September 22 - October 6, 2025**

Introduction

Aleph brings institutional funds into open finance. We build a secure, transparent, and efficient infrastructure that enables allocators to instantly access institutional money managers with diversified yield strategies, while maintaining the key controls expected in traditional finance. Designed for asset managers, digital asset funds, and accredited allocators, Aleph's blockchain-based architecture streamlines complex on-chain and off-chain tasks, eliminates bureaucratic frictions, reduces operational costs, and enhances capital formation.

Scope

Repository: AlephFi/smart-contracts

Audited Commit: 1812325a6df0787b27086d3a1e4c1d8458c01216

Final Commit: c7504956d58eb763c6f5e040acf0cc954404cccc

Files:

- src/Accountant.sol
- src/AccountantStorage.sol
- src/AlephPausable.sol
- src/AlephPausableStorage.sol
- src/AlephVaultBase.sol
- src/AlephVault.sol
- src/AlephVaultStorage.sol
- src/factory/AlephVaultFactory.sol
- src/factory/AlephVaultFactoryStorage.sol
- src/interfaces/IAccountant.sol
- src/interfaces/IAlephPausable.sol
- src/interfaces/IAlephVaultDeposit.sol
- src/interfaces/IAlephVaultFactory.sol
- src/interfaces/IAlephVaultFull.sol
- src/interfaces/IAlephVaultRedeem.sol
- src/interfaces/IAlephVaultSettlement.sol
- src/interfaces/IAlephVault.sol
- src/interfaces/IFeeManager.sol
- src/interfaces/IMigrationManager.sol

- src/libraries/AuthLibrary.sol
- src/libraries/ERC4626Math.sol
- src/libraries/ModulesLibrary.sol
- src/libraries/PausableFlows.sol
- src/libraries/RolesLibrary.sol
- src/libraries/SeriesAccounting.sol
- src/libraries/TimelockRegistry.sol
- src/modules/AlephVaultDeposit.sol
- src/modules/AlephVaultRedeem.sol
- src/modules/AlephVaultSettlement.sol
- src/modules/FeeManager.sol
- src/modules/MigrationManager.sol

Final Commit Hash

c7504956d58eb763c6f5e040acf0cc954404cccc

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
2	4	5

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: `AlephVaultFactory#deployVault` signature fails to verify custodian and treasury [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/37

Summary

The `deployVault` function verifies signatures containing manager, name, and configId but does not include custodian or vaultTreasury addresses in the signature. This allows anyone to frontrun a pending deployment transaction with compromised custodian or treasury addresses while using a valid signature.

Vulnerability Detail

The signature verification at AlephVaultFactory.sol#L207-L214 calls `verifyVaultDeploymentAuthSignature` with only manager, name, and configId:

```
AuthLibrary.verifyVaultDeploymentAuthSignature(
    _userInitializationParams.manager,
    address(this),
    _userInitializationParams.name,
    _userInitializationParams.configId,
    _sd.authSigner,
    _userInitializationParams.authSignature
);
```

The signature hash calculation at AuthLibrary.sol#L83-L84 confirms this:

```
bytes32 _hash =
    keccak256(abi.encode(_manager, _vaultFactory, _name, _configId, block.chainid,
    ↪   _authSignature.expiryBlock));
```

However, `UserInitializationParams` also contains custodian and vaultTreasury addresses that are not verified by the signature. An attacker can observe a pending transaction in the mempool, extract the valid signature, and submit their own transaction with the same manager/name/configId but substitute malicious custodian or treasury addresses.

Attack scenario:

1. Legitimate user requests signature from authSigner for vault deployment with trusted custodian and treasury
2. User submits transaction with valid signature
3. Attacker observes pending transaction in mempool

4. Attacker submits identical transaction with higher gas, same signature, but substitutes their own addresses for custodian/vaultTreasury
5. Attacker's transaction executes first due to higher gas
6. Vault deploys with attacker-controlled custodian and treasury addresses

The function also does not verify msg.sender, so any address can use a signature meant for someone else as long as the manager/name/configId match.

Impact

Attackers can frontrun vault deployments to deploy vaults with compromised custodian or treasury addresses while using legitimate signatures. This gives attackers control over critical vault roles and fund destinations.

The custodian address receives delegated custody of vault assets, and the vaultTreasury address is set as the treasury for fee collection via the Accountant contract at line 241. Compromising either address allows theft of vault assets or fees.

Since the signature does not bind to a specific deployer address, legitimate signatures can be reused by unauthorized parties to deploy vaults with malicious parameters.

Code Snippet

AlephVaultFactory.sol#L200-L248

Tool Used

Manual Review

Recommendation

Include custodian and vaultTreasury addresses in the signature hash to prevent parameter substitution:

```
function verifyVaultDeploymentAuthSignature(
    address _manager,
    address _vaultFactory,
    string memory _name,
    string memory _configId,
+   address _custodian,
+   address _vaultTreasury,
    address _authSigner,
    AuthSignature memory _authSignature
) internal view {
-   bytes32 _hash =
-       keccak256(abi.encode(_manager, _vaultFactory, _name, _configId,
 ↪  block.chainid, _authSignature.expiryBlock));
+   bytes32 _hash = keccak256(
+       abi.encode(
+           _manager,
+           _vaultFactory,
+           _name,
+           _configId,
+           _custodian,
+           _vaultTreasury,
+           block.chainid,
+           _authSignature.expiryBlock
+       )
+   );
    _verifyAuthSignature(_hash, _authSigner, _authSignature);
}
```

Alternatively, include msg.sender in the signature to bind the signature to a specific deployer address, preventing unauthorized use of valid signatures.

Issue H-2: Iterator in `for` loop is incorrectly incremented repeatedly [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/49

Summary

The logic in several core functions for iterating through Share Series contains a flaw. The loop variable `_seriesId` is incorrectly incremented by `_lastConsolidatedSeriesId` in each iteration, instead of only once to skip over consolidated series. This causes the loop to skip active share series, leading to incorrect calculations for total assets, user assets, redemption settlements, and fee collection.

Vulnerability Detail

The issue exists in multiple `for` loops across the protocol that iterate through share series using `_seriesId`. The loop's intent is to bypass old, consolidated series and begin calculations from the first active, unconsolidated series.

The faulty implementation pattern is as follows:

```
// AlephVaultBase.sol#L101-L108
for (uint8 _seriesId; _seriesId <= _shareClass.shareSeriesId; _seriesId++) {
    if (_seriesId > SeriesAccounting.LEAD_SERIES_ID) {
        _seriesId += _lastConsolidatedSeriesId;
    }
    // ... operations on _seriesId ...
}
```

The problem lies in the line `_seriesId += _lastConsolidatedSeriesId;`. When `_seriesId` is first greater than 0 (i.e., `_seriesId` = 1), it correctly jumps to the first unconsolidated series. However, in every subsequent iteration where `_seriesId` > 0, `_lastConsolidatedSeriesId` is **repeatedly added** to `_seriesId`.

Example: Assume `_lastConsolidatedSeriesId` = 5 and `_shareClass.shareSeriesId` = 8.

- **Expected Behavior:** The loop should iterate through `_seriesId` = 0, 6, 7, 8.
- **Actual Behavior:**
    1. `_seriesId` = 0: Loop body executes.
    2. `_seriesId` = 1: The `if` condition is met, `_seriesId` becomes `1 + 5 = 6`. The loop body executes for `_seriesId` = 6.
    3. `_seriesId` = 7: The `if` condition is met, `_seriesId` becomes `7 + 5 = 12`.

4. `_seriesId` is now 12, which is greater than `_shareClass.shareSeriesId` (8), and the loop terminates.

As a result, the valid share series with `_seriesId` 7 and 8 are completely skipped, leading to incomplete calculations.

This flawed pattern is present in the following critical locations:

- `AlephVaultBase#_totalAssetsPerClass`
- `AlephVaultBase#_assetsPerClassOf`
- `SeriesAccounting#settleRedeemForUser`
- `FeeManager#_collectFees`

Impact

Since this loop logic is used in several core financial calculation functions, the impact is widespread and severe:

1. **Incorrect Asset Calculation:** `totalAssetsPerClass` and `assetsOf` will under-report assets by omitting those in the skipped share series. This leads to incorrect reporting of the vault's total assets and the value of user holdings.
2. **Failed or Incomplete Redemptions:** During redemption settlement, `settleRedeemForUser` will fail to access user shares in the skipped series, causing the redeemed amount to be much lower than expected or even fail entirely despite the user having sufficient assets.
3. **Incomplete Fee Collection:** `collectFees` will under-calculate the accrued fees because fee shares in the skipped series will not be included, resulting in a loss of revenue for both the protocol and the fund manager.

Code Snippet

AlephVaultBase.sol#L101-L108

AlephVaultBase.sol#L191-L198

SeriesAccounting.sol#L159-L170

FeeManager.sol#L349-L369

Tool Used

Manual Review

Recommendation

The loop logic should be modified to ensure that `_seriesId` is advanced to skip consolidated series only once, with subsequent iterations proceeding normally.

Issue M-1: `MigrationManager#migrateGuardian` fails to transfer `WITHDRAW_FLOW` role [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/32

Summary

The `migrateGuardian` function fails to transfer the `WITHDRAW_FLOW` pausable role from the old guardian to the new guardian, resulting in incomplete role migration where the old guardian retains withdrawal flow pause/unpause control while the new guardian lacks this permission.

Vulnerability Detail

When migrating the guardian address through `migrateGuardian()`, the function transfers 5 pausable flow roles (DEPOSIT_REQUEST_FLOW, SETTLE_DEPOSIT_FLOW, REDEEM_REQUEST_FLOW, SETTLE_REDEEM_FLOW) but omits the WITHDRAW_FLOW role.

During initialization, the guardian is granted 6 roles total including `WITHDRAW_FLOW` as shown in AlephPausable.sol#L146-L150:

```
_grantRole(PausableFlows.DEPOSIT_REQUEST_FLOW, _guardian);
_grantRole(PausableFlows.SETTLE_DEPOSIT_FLOW, _guardian);
_grantRole(PausableFlows.REDEEM_REQUEST_FLOW, _guardian);
_grantRole(PausableFlows.SETTLE_REDEEM_FLOW, _guardian);
_grantRole(PausableFlows.WITHDRAW_FLOW, _guardian); // This role controls
 ↪  withdrawRedeemableAmount
```

However, the migration function only revokes and grants 5 of these roles, omitting both the revocation from the old guardian and the grant to the new guardian for `WITHDRAW_FLOW`.

The WITHDRAW_FLOW role gates the critical `withdrawRedeemableAmount()` function at AlephVault.sol#L771-L773:

```
function withdrawRedeemableAmount() external
 ↪  whenFlowNotPaused(PausableFlows.WITHDRAW_FLOW) {
    _delegate(ModulesLibrary.ALEPH_VAULT_REDEEM);
}
```

Impact

Broken Access Control: After guardian migration, the old guardian retains the ability to pause/unpause the withdrawal flow while the new guardian cannot. This violates the principle of complete role transfer during migration.

DoS Risk: If the guardian migration was triggered due to key loss or compromise:

- The old guardian can still call `pauseAll()` (which pauses WITHDRAW_FLOW among others) at AlephPausable.sol#L73-L75
- The new guardian cannot unpause WITHDRAW_FLOW to restore functionality
- Users cannot withdraw their redeemed funds via `withdrawRedeemableAmount()`
- Contract upgrade would be required to fix the paused state, causing significant operational disruption

Split Responsibilities: The system now has two addresses with partial guardian powers:

- Old guardian: Can pause/unpause WITHDRAW_FLOW
- New guardian: Can pause/unpause all other flows but not WITHDRAW_FLOW
- This creates confusion and security risks in emergency response scenarios

Migration Intent Violation: Guardian migration typically occurs when the old key is compromised or lost. In these scenarios, the old guardian retaining any control defeats the purpose of migration.

Code Snippet

MigrationManager.sol#L77-L95

Tool Used

Manual Review

Recommendation

Add the missing WITHDRAW_FLOW role revocation and grant to complete the guardian migration:

```
function migrateGuardian(address _newGuardian) external {
    if (_newGuardian == address(0)) {
        revert InvalidGuardianAddress();
    }
    AlephVaultStorageData storage _sd = _getStorage();
    address _guardian = _sd.guardian;
    _sd.guardian = _newGuardian;
    _revokeRole(RolesLibrary.GUARDIAN, _guardian);
    _revokeRole(PausableFlows.DEPOSIT_REQUEST_FLOW, _guardian);
    _revokeRole(PausableFlows.SETTLE_DEPOSIT_FLOW, _guardian);
    _revokeRole(PausableFlows.REDEEM_REQUEST_FLOW, _guardian);
    _revokeRole(PausableFlows.SETTLE_REDEEM_FLOW, _guardian);
+   _revokeRole(PausableFlows.WITHDRAW_FLOW, _guardian);
    _grantRole(RolesLibrary.GUARDIAN, _newGuardian);
    _grantRole(PausableFlows.DEPOSIT_REQUEST_FLOW, _newGuardian);
    _grantRole(PausableFlows.SETTLE_DEPOSIT_FLOW, _newGuardian);
    _grantRole(PausableFlows.REDEEM_REQUEST_FLOW, _newGuardian);
    _grantRole(PausableFlows.SETTLE_REDEEM_FLOW, _newGuardian);
+   _grantRole(PausableFlows.WITHDRAW_FLOW, _newGuardian);
    emit GuardianMigrated(_newGuardian);
}
```

This ensures complete role transfer and maintains the security invariant that only the current guardian has pause/unpause authority over all flows.

Issue M-2: `FeeManager#_calculatePerformanceFeeAmount` uses incorrect formula inflating fees [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/33

Summary

The `_calculatePerformanceFeeAmount` function uses an incorrect formula with `BPS_DENOMINATOR - _performanceFee` as the denominator, which is designed for share-based calculations. Since performance fees are calculated in assets, this inflates the fee amount above the intended rate.

Vulnerability Detail

The performance fee calculation at FeeManager.sol#L332-L333 uses the formula:

```
_performanceFeeAmount = _profit.mulDiv(
    uint256(_performanceFee),
    uint256(BPS_DENOMINATOR - _performanceFee),
    Math.Rounding.Ceil
);
```

This formula `profit * fee / (BPS_DENOMINATOR - fee)` is the correct formula when calculating shares to mint (where you need to account for dilution), but incorrect when calculating an asset amount.

For asset-based fees, the correct formula is simply `profit * fee / BPS_DENOMINATOR`, matching the pattern used in `_calculateManagementFeeAmount` at FeeManager.sol#L303-L304:

```
_annualFees = _newTotalAssets.mulDiv(
    uint256(_managementFee),
    uint256(BPS_DENOMINATOR),
    Math.Rounding.Ceil
);
```

Example with 20% performance fee (2000 bps) on 1000 profit:

- Current incorrect formula: `1000 * 2000 / (10000 - 2000) = 1000 * 2000 / 8000 = 250` assets
- Correct formula: `1000 * 2000 / 10000 = 200` assets
- Overcharge: 25% more fees than intended

The overcharge percentage increases non-linearly with the fee rate:

- 10% fee (1000 bps): 11.11% overcharge
- 20% fee (2000 bps): 25% overcharge
- 30% fee (3000 bps): 42.86% overcharge

Impact

Users pay significantly more performance fees than the configured rate. The overcharge is proportional to the fee rate itself, meaning higher performance fees result in disproportionately larger overcharges. This constitutes an unintended wealth transfer from vault users to the fee recipient (accountant).

The formula mismatch indicates a conceptual error where asset-denominated fees were calculated using a share-dilution formula, likely due to confusion between the two fee calculation methodologies.

Code Snippet

FeeManager.sol#L317-L335

Tool Used

Manual Review

Recommendation

Change the denominator to `BPS_DENOMINATOR` to match the management fee calculation pattern:

```
function _calculatePerformanceFeeAmount(
    uint32 _performanceFee,
    uint256 _newTotalAssets,
    uint256 _totalShares,
    uint256 _highWaterMark
) internal pure returns (uint256 _performanceFeeAmount) {
    uint256 _pricePerShare = _getPricePerShare(_newTotalAssets, _totalShares);
    if (_pricePerShare > _highWaterMark) {
        uint256 _profitPerShare = _pricePerShare - _highWaterMark;
        uint256 _profit =
            _profitPerShare.mulDiv(_totalShares,
            ↪  SeriesAccounting.PRICE_DENOMINATOR, Math.Rounding.Ceil);
        _performanceFeeAmount =
-           _profit.mulDiv(uint256(_performanceFee), uint256(BPS_DENOMINATOR -
 ↪  _performanceFee), Math.Rounding.Ceil);
+           _profit.mulDiv(uint256(_performanceFee), uint256(BPS_DENOMINATOR),
 ↪  Math.Rounding.Ceil);
```

```
    }
}
```

Issue M-3: `FeeManager#_accumulateFees` uses incorrect totalAssets for share minting [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/34

Summary

The `_accumulateFees` function uses `_newTotalAssets` when calculating shares to mint for fees, but should use `_newTotalAssets - feeAmount` to account for the fee being extracted. This causes fee recipients to receive fewer shares than entitled, creating a shortfall that increases with fee magnitude.

Vulnerability Detail

When minting fee shares at FeeManager.sol#L257-L261, the function uses `_newTotalAssets` as the total assets parameter:

```
_feesAccumulatedParams.managementFeeSharesToMint =
    ERC4626Math.previewDeposit(_feesAccumulatedParams.managementFeeAmount,
    ↪   _totalShares, _newTotalAssets);
_feesAccumulatedParams.performanceFeeSharesToMint =
    ERC4626Math.previewDeposit(_feesAccumulatedParams.performanceFeeAmount,
    ↪   _totalShares, _newTotalAssets);
```

The ERC4626 `previewDeposit` formula is: `shares = assets * (totalShares + 1) / (totalAssets + 1)`

However, `_newTotalAssets` includes the fee amount that is being minted as shares. The correct totalAssets for the calculation should exclude the fee being extracted, since those assets are being converted to shares for the fee recipient.

Example with 1,000,000 totalAssets, 1,000,000 totalShares, and 20,000 management fee:

Current incorrect calculation:

```
shares = 20,000 * (1,000,000 + 1) / (1,020,000 + 1)
       = 20,000 * 1,000,001 / 1,020,001
       = 19,607.83 shares
```

If fee recipient immediately redeems:

```
assets = 19,607.83 * (1,020,000 + 1) / (1,019,607.83 + 1)
       = 19,607.84 assets
```

Shortfall: 20,000 - 19,607.84 = 392.16 assets (1.96% loss)

Correct calculation using `_newTotalAssets - feeAmount`:

```
shares = 20,000 * (1,000,000 + 1) / (1,000,000 + 1)
       = 20,000 shares (exactly)
```

The shortfall percentage increases with the fee amount: a 2% fee results in ~1.96% shortfall, a 5% fee results in ~4.76% shortfall, and a 10% fee results in ~9.09% shortfall.

Impact

Fee recipients (management and performance fee addresses) receive fewer shares than they are entitled to based on the fee amounts calculated. The shortfall represents value that remains with existing vault shareholders rather than being properly transferred to fee recipients.

This creates an accounting mismatch where the fee amount calculated does not correspond to the actual value received when shares are redeemed. The issue compounds with both management and performance fees, as both use the incorrect totalAssets value.

The protocol's fee structure becomes unpredictable and unfair to fee recipients, who effectively receive discounted fees below the configured rates.

Code Snippet

FeeManager.sol#L234-L287

Tool Used

Manual Review

Recommendation

Subtract the fee amount from `_newTotalAssets` when calculating shares to mint:

```
function _accumulateFees(
    IAlephVault.ShareClass storage _shareClass,
    uint256 _newTotalAssets,
    uint256 _totalShares,
    uint48 _currentBatchId,
    uint48 _lastFeePaidId,
    uint8 _classId,
    uint8 _seriesId
) internal returns (uint256) {
    FeesAccumulatedParams memory _feesAccumulatedParams;
    IAlephVault.ShareClassParams memory _shareClassParams =
     ↪  _shareClass.shareClassParams;
    _feesAccumulatedParams.managementFeeAmount = _calculateManagementFeeAmount(
        _newTotalAssets, _currentBatchId - _lastFeePaidId,
        ↪  _shareClassParams.managementFee
    );
    _feesAccumulatedParams.performanceFeeAmount = _calculatePerformanceFeeAmount(
        _shareClassParams.performanceFee,
        _newTotalAssets,
        _totalShares,
        _shareClass.shareSeries[_seriesId].highWaterMark
    );

    _feesAccumulatedParams.managementFeeSharesToMint =
-       ERC4626Math.previewDeposit(_feesAccumulatedParams.managementFeeAmount,
↪  _totalShares, _newTotalAssets);
+       ERC4626Math.previewDeposit(
+           _feesAccumulatedParams.managementFeeAmount,
+           _totalShares,
+           _newTotalAssets - _feesAccumulatedParams.managementFeeAmount
+       );

    _feesAccumulatedParams.performanceFeeSharesToMint =
-       ERC4626Math.previewDeposit(_feesAccumulatedParams.performanceFeeAmount,
↪  _totalShares, _newTotalAssets);
+       ERC4626Math.previewDeposit(
+           _feesAccumulatedParams.performanceFeeAmount,
+           _totalShares,
+           _newTotalAssets - _feesAccumulatedParams.performanceFeeAmount
+       );

    // rest of function...
}
```

Issue M-4: `FeeManager#_accumulateFees` calculates performance fees before management dilution [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/35

Summary

The `_accumulateFees` function calculates performance fees using the pre-management-fee share count, causing performance fees to be charged on assets that should be attributed to management fees. This results in double-charging on the management fee portion.

Vulnerability Detail

The function calculates both fee amounts and mints shares in the wrong order at FeeManager.sol#L246-L261:

```
// calculate management fee amount
_feesAccumulatedParams.managementFeeAmount = _calculateManagementFeeAmount(
    _newTotalAssets, _currentBatchId - _lastFeePaidId,
     ↪  _shareClassParams.managementFee
);
// calculate performance fee amount
_feesAccumulatedParams.performanceFeeAmount = _calculatePerformanceFeeAmount(
    _shareClassParams.performanceFee,
    _newTotalAssets,
    _totalShares,  // Uses original totalShares before management dilution
    _shareClass.shareSeries[_seriesId].highWaterMark
);
// calculate management fee shares to mint
_feesAccumulatedParams.managementFeeSharesToMint =
    ERC4626Math.previewDeposit(_feesAccumulatedParams.managementFeeAmount,
     ↪  _totalShares, _newTotalAssets);
// calculate performance fee shares to mint
_feesAccumulatedParams.performanceFeeSharesToMint =
    ERC4626Math.previewDeposit(_feesAccumulatedParams.performanceFeeAmount,
     ↪  _totalShares, _newTotalAssets);
```

The issue is that `_calculatePerformanceFeeAmount` uses `_totalShares` (the share count before management fees are applied) to calculate price per share. This means the performance fee is calculated on profit that includes the portion going to management fees.

Example scenario with 1000 shares, 1200 assets (20% gain), 2% management fee, 20% performance fee:

Current incorrect order:

1. Price per share = 1200 / 1000 = 1.2
2. Profit per share = 1.2 - 1.0 = 0.2
3. Total profit = 0.2 * 1000 = 200
4. Management fee = 1200 * 2% = 24 assets
5. Performance fee = 200 * 20% / 80% = 50 assets (charged on full 200 profit)

The performance fee is charged on 200 assets of profit, but 24 of those 200 assets are already claimed by management fees.

Correct order:

1. Management fee = 24 assets, mint 24 shares
2. New state: 1024 shares, 1200 assets
3. Price per share = 1200 / 1024 = 1.171875
4. Profit per share = 1.171875 - 1.0 = 0.171875
5. Total profit for original holders = 0.171875 * 1000 = 171.875
6. Performance fee = 171.875 * 20% / 80% = 42.97 assets (charged on remaining 176 profit)

Difference: 50 - 42.97 = 7.03 assets overcharge (14% excess performance fee)

Impact

Performance fee recipients receive fees calculated on profit that includes management fee amounts, effectively double-charging users on the management fee portion. The overcharge increases with both the management fee rate and the performance fee rate.

Original vault shareholders are diluted more than intended, as they bear performance fees on assets that should only be subject to management fees. The excess performance fees come at the expense of original shareholders rather than representing actual performance gains attributable to them.

The compounding effect means higher management fees amplify the performance fee overcharge, creating an unfair fee structure where the two fee types interact incorrectly.

Code Snippet

FeeManager.sol#L234-L287

Tool Used

Manual Review

Recommendation

Restructure the function to follow the correct order of operations:

1. Calculate management fee amount
2. Calculate and mint management fee shares
3. Update totalShares to include the minted management shares
4. Calculate performance fee amount using the updated totalShares (which now includes management dilution)
5. Calculate and mint performance fee shares using the updated totalShares

This ensures that performance fees are only charged on profit attributable to original shareholders after management fees have been accounted for, preventing double-charging on the management fee portion.

Issue L-1: `AlephVault#migrateAccountant` lacks implementation in MigrationManager [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/38

Summary

The `migrateAccountant` function in AlephVault delegates to MigrationManager, but MigrationManager does not implement this function. Additionally, AlephVaultFactory lacks a corresponding setter function to trigger accountant migration across all vaults, rendering accountant migration completely broken.

Vulnerability Detail

The AlephVault contract declares a `migrateAccountant` function at AlephVault.sol#L867-L869 that delegates to the MigrationManager module:

```
function migrateAccountant(address _newAccountant) external
 ↪  onlyRole(RolesLibrary.VAULT_FACTORY) {
    _delegate(ModulesLibrary.MIGRATION_MANAGER);
}
```

However, MigrationManager does not implement this function. When AlephVault delegates to MigrationManager for `migrateAccountant`, the call will reach the fallback function but find no matching function selector, causing the transaction to revert.

Furthermore, AlephVaultFactory lacks a `setAccountant` function to update the accountant address in factory storage and propagate the change to all deployed vaults. The factory implements setter functions for all other global parameters (operationsMultisig, oracle, guardian, authSigner, moduleImplementations) that follow the pattern of updating factory storage and calling the corresponding migrate function on all vaults, but accountant migration is entirely missing from this infrastructure.

The factory stores accountant address at AlephVaultFactoryStorage.sol#L39 and uses it during vault deployment, but provides no mechanism to update it across existing vaults.

Impact

The accountant address cannot be changed after vault deployment. If the accountant contract needs to be upgraded, replaced due to a security issue, or changed for any operational reason, there is no functional way to migrate to a new accountant.

This creates a permanent dependency on the initially deployed accountant contract. If that contract becomes compromised, deprecated, or requires replacement, all deployed vaults remain locked to the original accountant with no migration path.

The broken migration path contradicts the protocol's design intent, which includes migration infrastructure for all other critical protocol components. This asymmetry suggests the accountant migration was planned but incompletely implemented.

Code Snippet

AlephVault.sol#L867-L869

Tool Used

Manual Review

Recommendation

Implement the missing `migrateAccountant` function in MigrationManager following the pattern of other migration functions:

```
+   function migrateAccountant(address _newAccountant) external {
+       if (msg.sender != address(this)) {
+           revert Unauthorized();
+       }
+       if (_newAccountant == address(0)) {
+           revert InvalidParam();
+       }
+       AlephVaultStorageData storage _sd = _getStorage();
+       _sd.accountant = _newAccountant;
+       emit AccountantMigrated(_newAccountant);
+   }
```

Add a corresponding setter function in AlephVaultFactory:

```
+   function setAccountant(address _accountant) external
 ↪  onlyRole(RolesLibrary.OPERATIONS_MULTISIG) {
+       if (_accountant == address(0)) {
+           revert InvalidParam();
+       }
+       AlephVaultFactoryStorageData storage _sd = _getStorage();
+       _sd.accountant = _accountant;
+       uint256 _len = _sd.vaults.length();
+       for (uint256 i = 0; i < _len; i++) {
+           address _vault = _sd.vaults.at(i);
+           IMigrationManager(_vault).migrateAccountant(_accountant);
+       }
+       emit AccountantSet(_accountant);
+   }
```

Issue L-2: `AlephVaultFactory#setIsAuthEnabled` allows vault spam DoS [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/39

Summary

Disabling authentication via `setIsAuthEnabled` allows attackers to deploy unlimited fake vaults, which can brick migration functions that loop through all vaults.

Impact

Migration functions like `setOperationsMultisig`, `setOracle`, `setGuardian`, `setAuthSigner`, and `setModuleImplementation` iterate through all registered vaults. An attacker can deploy thousands of fake vaults when authentication is disabled, causing these migration functions to exceed block gas limits and permanently fail.

Code Snippet

AlephVaultFactory.sol#L107-L110

Tool Used

Manual Review

Recommendation

Never disable authentication. Keep `isAuthEnabled` set to `true` at all times to prevent unauthorized vault deployments from creating a denial of service condition on migration functions.

Issue L-3: `AlephVaultFactory` contains unused fee maximum constants [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/40

Summary

The `MAX_MANAGEMENT_FEE` and `MAX_PERFORMANCE_FEE` constants in AlephVaultFactory are never used. These duplicate the `MAXIMUM_MANAGEMENT_FEE` and `MAXIMUM_PERFORMANCE_FEE` constants defined in AlephVaultBase.

Impact

The unused constants create code clutter and potential confusion about which fee limits are actually enforced. The effective fee limits are enforced by AlephVaultBase, not the factory.

Code Snippet

AlephVaultFactory.sol#L42-L43

Tool Used

Manual Review

Recommendation

Remove the unused constants:

```
contract AlephVaultFactory is IAlephVaultFactory, AccessControlUpgradeable {
    using EnumerableSet for EnumerableSet.AddressSet;

-   uint32 public constant MAX_MANAGEMENT_FEE = 1000; // 10%
-   uint32 public constant MAX_PERFORMANCE_FEE = 5000; // 50%

    /*//////////////////////////////////////////////////////////////
                            INITIALIZER
    //////////////////////////////////////////////////////////////*/
```

Issue L-4: `collectFees` may prioritize fee payments over user funds by not checking vault balance [RE-SOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/43

Summary

The `FeeManager#_collectFees` function, when transferring fee assets from the vault, does not verify if the vault's balance is sufficient to cover pending user deposits and settled redeemable amounts. This can cause fee collection to take precedence over user withdrawals and deposit settlements, depleting funds that belong to users and ultimately causing their `withdrawRedeemableAmount` calls to fail due to insufficient funds.

Vulnerability Detail

The `_collectFees` function in the `FeeManager` module is responsible for converting accrued fee shares into the underlying asset and transferring them from the vault contract to the `Accountant` contract.

However, this process does not account for user funds held within the vault contract. The vault acts as a temporary escrow for pending user deposits (`totalAmountToDeposit`) and as a pool for settled withdrawals awaiting claim (`totalAmountToWithdraw`). The `_collectFees` function directly transfers fees without first checking if the remaining balance is sufficient to meet these user liabilities.

The relevant code snippet is shown below:

```
function _collectFees(AlephVaultStorageData storage _sd)
    internal
    returns (uint256 _managementFeesToCollect, uint256 _performanceFeesToCollect)
{
    // ... (fee calculation logic) ...

    IERC20(_sd.underlyingToken).safeTransfer(_sd.accountant,
    ↪  _managementFeesToCollect + _performanceFeesToCollect);
    emit FeesCollected(_currentBatch(_sd), _managementFeesToCollect,
    ↪  _performanceFeesToCollect);
}
```

As shown, the code directly executes `safeTransfer` without any prior balance checks on the vault. In contrast, the `withdrawExcessAssets` function in `AlephVaultRedeem.sol` correctly includes a check to ensure the vault retains enough funds to cover user deposits and withdrawals.

Attack Scenario Example:

1. The vault holds 1000 `USDC` in pending user deposits (`totalAmountToDeposit`) and 500 `USDC` in settled, redeemable funds (`totalAmountToWithdraw`). The vault contract's current balance is 1500 `USDC`.
2. Simultaneously, the protocol has accrued 200 `USDC` in fees.
3. A manager calls `collectFees`. The function executes successfully, transferring 200 `USDC` from the vault to the accountant contract.
4. The vault's balance is now only 1300 `USDC`, but its liabilities to users remain 1500 `USDC`.
5. When a user subsequently calls `withdrawRedeemableAmount` to claim their 500 `USDC`, the transaction may fail due to insufficient funds.

Impact

This vulnerability breaks the priority of user funds. Fee collection can deplete assets that should be reserved for returning to users or for settling deposits, causing user withdrawal functions (`withdrawRedeemableAmount`) to fail. This can trap user funds until a manager replenishes the vault's liquidity, damaging user trust and fund safety.

Code Snippet

smart-contracts/src/modules/FeeManager.sol#L340

Tool Used

Manual Review

Recommendation

Before executing the `safeTransfer` in the `_collectFees` function, a check should be added to ensure that the vault's balance, after paying fees, remains sufficient to cover the sum of `totalAmountToDeposit` and `totalAmountToWithdraw`.

Issue L-5: Double Ceiling Rounding in `_consolidateUserShares` Leads to Fund Insufficiency and Unfair Share Dilution [RESOLVED]

Source:
https://github.com/sherlock-audit/2025-09-aleph-finance-sept-22nd/issues/48

Summary

In the `_consolidateUserShares` function within the `SeriesAccounting.sol` library, the series consolidation process uses ceiling rounding twice in succession. This approach can slightly inflate the asset value converted from an old series to the lead series, leading to a cumulative fund insufficiency in the old series and causing unfair share dilution for existing investors in the lead series.

Vulnerability Detail

During the series consolidation process, the code first converts a user's shares from an old series (`_shareSeries`) into its corresponding asset value, and then converts that asset value into shares in the lead series (`_leadSeries`). Both of these calculations use ceiling rounding.

The code is located at lines 210-215 of `SeriesAccounting.sol`:

```
// calculate amount to transfer from outstanding series to lead series
_userConsolidationDetails.amountToTransfer = ERC4626Math.previewMint(
    _userConsolidationDetails.shares, _shareSeries.totalAssets,
     ↪   _shareSeries.totalShares
);
// calculate corresponding shares to deposit in lead series
_userConsolidationDetails.sharesToTransfer = ERC4626Math.previewWithdraw(
    _userConsolidationDetails.amountToTransfer, _leadSeries.totalShares,
     ↪   _leadSeries.totalAssets
);
```

1. `ERC4626Math.previewMint` internally calls `convertToAssets`, which uses `Math.Rounding.Ceil`. This means the calculated `amountToTransfer` may be **slightly higher** than the precise asset value represented by the user's shares.
2. `ERC4626Math.previewWithdraw` internally calls `convertToShares`, which also uses `Math.Rounding.Ceil`. This means that when calculating new shares from a potentially already inflated asset value, the result is **inflated again**.

The cumulative effect of this double ceiling rounding leads to:

- The total asset value (`totalAmountToTransfer`) calculated for withdrawal from the old series could eventually **exceed** the actual total assets held by that series.
- The shares minted for the user in the lead series (`sharesToTransfer`) will be **slightly more** than what they are entitled to, thereby diluting the equity of other existing investors in the lead series.

Impact

This vulnerability has two primary impacts:

1. **Risk of Protocol Fund Insufficiency:** Although a single rounding error is minuscule, the cumulative effect over many users and multiple series consolidations could leave an old series with insufficient assets to cover the calculated `totalAmountToTransfer`, creating a fund deficit within the protocol.
2. **Unfairness to Existing Investors:** Existing investors in the lead series will be slightly diluted because newly consolidated users receive more shares than their assets are worth. This violates the principle of fair value distribution.

Code Snippet

smart-contracts/src/libraries/SeriesAccounting.sol#L210-L215

Tool Used

Manual Review

Recommendation

To protect the protocol and its existing investors, it is recommended to consistently use a rounding direction that favors the protocol. The specific changes should be:

1. When converting shares from an old series to assets (`previewMint`), use **floor rounding** to ensure the withdrawn asset value does not exceed the user's actual entitlement.
2. When converting assets into shares in the lead series (`previewWithdraw`), also use **floor rounding** to ensure that no excess shares are minted for the user.

By standardizing the rounding direction to floor rounding, the risk of fund insufficiency is eliminated, and fairness is guaranteed for all investors. This provides a more robust fundamental layer of protection for the protocol.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.