



# Security Review For YieldFi



Collaborative Audit Prepared For:  
Lead Security Expert(s):

**YieldFi**  
**Drynooo**  
**montecristo**

Date Audited:

**December 29 - January 14, 2026**

# Introduction

YieldFi is the issuance, distribution, and intelligence layer for tokenized yield vaults—so curators can launch products, partners can distribute them, and allocators can track risk and performance in real time.

Yield opportunities across crypto and traditional markets are fragmented, opaque, and difficult to access. YieldFi makes yield:

- Accessible through tokenized vaults
- Distributable via SDK and partners
- Measurable with standardized NAV, APY, and risk analytics

## Scope

Repository: `YieldFiLabs/vaults`

Audited Commit: `19ceb0047fc10660e95519072c8760b3a8033bb1`

Final Commit: `4f5a5d0a2128be04d257b4d5c9d1fd0d9493ae26`

Files:

- `src/access/AccessControl.sol`
- `src/access/Administrator.sol`
- `src/core/interfaces/IEvents.sol`
- `src/core/interfaces/IFeature.sol`
- `src/core/interfaces/IManager.sol`
- `src/core/interfaces/IRegistry.sol`
- `src/core/interfaces/IVault.sol`
- `src/core/Manager.sol`
- `src/core/NAV.sol`
- `src/core/Registry.sol`
- `src/core/Vault.sol`
- `src/features/FeatureRegistry.sol`
- `src/features/greenlist/GreenlistFeature.sol`
- `src/features/greenlist/interfaces/IGreenlistFeature.sol`
- `src/features/interfaces/IFeatureRegistry.sol`
- `src/features/limits/interfaces/ILimitsFeature.sol`
- `src/features/limits/LimitsFeature.sol`

- src/infrastructure/Create2Factory.sol
- src/infrastructure/interfaces/IVaultFactory.sol
- src/infrastructure/VaultFactory.sol
- src/interfaces/IAccessControl.sol
- src/interfaces/IBlackList.sol
- src/interfaces/IPausable.sol
- src/interfaces/IRole.sol
- src/libraries/AccountingLib.sol
- src/libraries/AddressLib.sol
- src/libraries/Constants.sol
- src/libraries/Errors.sol
- src/libraries/MathLib.sol
- src/libraries/ProxyDeployer.sol
- src/libraries/Roles.sol
- src/libraries/SafeCastLib.sol
- src/libraries/SignerLib.sol
- src/oracles/adapters/ChainlinkOracleAdapter.sol
- src/oracles/interfaces/IOracleAdapter.sol
- src/oracles/interfaces/IPriceOracle.sol
- src/oracles/PriceOracle.sol

## Final Commit Hash

4f5a5d0a2128be04d257b4d5c9d1fd0d9493ae26

## 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
0	8	9

## Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

# Issue M-1: Cannot add a user to greenlist of the global vault [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/41>

## Summary

Setting `GLOBAL_VAULT` to hard-coded `address(0)` is a mistake because `GreenlistFeature::addToGreenlist` will fail due to zero-address validation.

## Vulnerability Detail

Global vault address is hard-coded to `address(0)`:

File: [vaults/src/features/greenlist/GreenlistFeature.sol](#)

```
34:    /// @notice Global vault address (address(0) used as default for global KYC)
35:    address public constant GLOBAL_VAULT = address(0);
```

However, `GreenlistFeature::addToGreenlist` has a validation that checks if passed vault is not `address(0)`:

File: [vaults/src/features/greenlist/GreenlistFeature.sol](#)

```
277:    function addToGreenlist(
278:        address vault,
279:        address user,
280:        uint256 tier
281:    ) external onlyOperator {
282:        _validateVaultAndUser(vault, user); // @audit won't work for
    ↪ GLOBAL_VAULT due to zero-address check
```

As a result, it's impossible to add a user to global vault's greenlist.

## Impact

Impossible to add a user to greenlist of the global vault.

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Either remove zero address check or set global vault's address to a non-zero address.

# Issue M-2: Incorrect redemption amount when MIN\_NAV feature is enabled [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/43>

## Summary

When calculating redemption amounts, `effectiveNav` is replaced by `currentNav` even when `minNavEnabled = true` and `effectiveNav < currentNav`. This is contradictory to MIN\_NAV feature.

## Vulnerability Detail

In `Manager::_calculateRedemptionAmounts`, `effectiveNav` is replaced by `currentNav` if `effectiveNav < currentNav`, even if `minNavEnabled = true`.

File: `vaults/src/core/Manager.sol`

```
471:     function _calculateRedemptionAmounts(
472:         address vault,
473:         uint256 entryNavAtRequest,
474:         uint256 currentNav,
475:         uint256 netShares,
476:         address baseAsset
477:     ) internal view returns (uint256 effectiveNav, uint256 assetsToTransfer) {
478:         // Check if MIN_NAV feature is enabled
479:         bool minNavEnabled = contracts.featureRegistry.isFeatureEnabled(
480:             vault,
481:             Constants.MIN_NAV_FEATURE_ID
482:         );
483:
484:         // Calculate effective NAV (min of currentNAV and requestNAV)
485:         effectiveNav = minNavEnabled ? entryNavAtRequest : currentNav;
486:         if (effectiveNav > currentNav) { // This case happens when the NAV is
487:             ↪ decreased after the request
488:             effectiveNav = currentNav;
489:         }
490:
491:         // Calculate assets to transfer
492:         assetsToTransfer = _calculateRedemptionAssets(
493:             netShares,
494:             effectiveNav > currentNav ? effectiveNav : currentNav,
495:             baseAsset
496:         );
```

## Impact

Redeemers will redeem at current NAV (a better price) from a MIN\_NAV enabled vault when NAV at request (a fair price) is lower than current NAV.

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Remove inline condition in L493 and just use `effectiveNav`.

# Issue M-3: Deposits using lagging Vested NAV (currentNav) causes dilution of existing shareholders [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/49>

## Summary

The `Manager` contract uses the linearly vested `currentNav` instead of the actual target NAV (`endNav`) when calculating shares to mint for deposits. Due to the vesting mechanism, the update of `currentNav` lags behind the actual growth of asset value. This allows attackers to exploit profit realization events (such as large loan repayments) by minting excessive shares at an artificially low price, capturing previously generated yield and diluting existing shareholders.

## Vulnerability Detail

In `Manager.sol`, the deposit logic calls `_getAndValidateNav` to retrieve the `currentNav` for calculating the number of shares a user receives. The protocol employs a linear vesting mechanism to prevent front-running, causing the `currentNav` to slowly increase towards the actual target NAV over a period of time.

However, OTC lending operations often involve "Bullet Repayments", which can cause the NAV to jump significantly in a single update. Attackers can monitor the chain for such profit settlement events and deposit before the NAV is fully vested (vesting complete).

For example:

1. The real NAV jumps from 1.0 to 1.5 (`endNav` = 1.5) due to yield settlement.
2. Due to vesting, the `currentNav` reads only 1.1.
3. An attacker deposits 110 USDC.
  - **Expected Logic (Fair Value):** Calculated at 1.5 price -> Receives 73.3 shares.
  - **Actual Logic (Exploit):** Calculated at 1.1 price -> Receives 100 shares.
4. When vesting finishes and NAV hits 1.5, the attacker's 100 shares are worth 150 USDC, resulting in a profit of 40 USDC.

Although there may be a cooling period for withdrawals, this does not alter the fact that the attacker has "bought" more equity at a discount, causing dilution to other users.

## Impact

- Attackers can perform risk-free arbitrage by monitoring profit settlement events.

- Attackers can capture a portion of the yield generated by the protocol before their entry.
- The share value of existing long-term holders is diluted.

## Code Snippet

Manager.sol

```
depositValueInBase = MathLib.mulDiv18(amountIn18Decimals, depositPrice);

// Calculate total shares to mint:  $\Delta S = m / \text{NAV}$  (both in 18 decimals)
// Vulnerability: Using lagging currentNav for calculation
totalSharesToMint = AccountingLib.calculateSharesToMint(depositValueInBase,
↪ currentNav);
```

## Tool Used

Manual Review

## Recommendation

Modify the share calculation logic for deposits. To prevent dilution, the target NAV (endNAV) should be used instead of the lagging currentNav to calculate the number of shares minted, especially in cases where the NAV increases (profit).

## Discussion

nisuscold

Based on the internal conversation, we have enabled a feature flag for the request.

New code looks like this

```
/**
 * @notice Calculate total shares to mint for deposit (combines base asset,
↪ price, value, and shares calculation)
 * @param vault Vault address
 * @param asset Asset address being deposited
 * @param amount Amount of assets to deposit
 * @param currentNav Current NAV (18 decimals)
 * @return totalSharesToMint Total shares to mint before fees (18 decimals)
 * @return depositValueInBase Deposit value in base asset (18 decimals)
 */
function _calculateDepositSharesFromAsset(
    address vault,
    address asset,
```

```

        uint256 amount,
        uint256 currentNav
    ) internal view returns (uint256 totalSharesToMint, uint256 depositValueInBase)
    ↪ {
        // Get and validate base asset
        address baseAsset = contracts.registry.baseAssets(vault);
        baseAsset.requireNonZero();

        uint256 depositPrice;
        bool priceValid;

        if (asset == baseAsset) {
            depositPrice = Constants.PRECISION;
            priceValid = true;
        }

        if (asset != baseAsset) {
            (depositPrice, priceValid) = contracts.priceOracle
                .getDepositPrice(asset, baseAsset);
            if (!priceValid || depositPrice == 0) {
                revert Errors.InvalidOraclePrice(asset, baseAsset, "INVALID_PRICE");
            }
        }

        // Calculate deposit value in base asset (18 decimals):  $m = q \times$ 
        ↪  $P_{\text{asset, deposit}}$ 
        // Work directly in 18 decimals to avoid unnecessary conversions
        uint8 assetDecimals = IERC20Metadata(asset).decimals(); // asset decimals

        // Scale amount to 18 decimals, then multiply by depositPrice (already 18
        ↪ decimals)
        uint256 amountIn18Decimals = MathLib.scaleDecimals(amount, assetDecimals,
            ↪ 18);
        depositValueInBase = MathLib.mulDiv18(amountIn18Decimals, depositPrice);

        uint256 effectiveNav = _getMaxNavIfEnabled(vault, currentNav);

        // Calculate total shares to mint:  $\Delta S = m / \text{NAV}$  (both in 18 decimals)
        totalSharesToMint = AccountingLib.calculateSharesToMint(depositValueInBase,
            ↪ effectiveNav);
    }

/**
 * @notice Get effective NAV (max nav if enabled)
 * @param vault @param vault Vault address
 * @param currentNav Current NAV (18 decimals)
 */
function _getMaxNavIfEnabled(address vault, uint256 currentNav) internal view
    ↪ returns (uint256 maxNav) {

```

```

bool maxNavEnabled = contracts.featureRegistry.isFeatureEnabled(
    vault,
    Constants.MAX_NAV_FEATURE_ID
);

// If max nav feature is not enabled, return current nav - validation is
↳ already done in _getAndValidateNav
if (!maxNavEnabled) {
    return currentNav;
}

// Get max nav from NAV contract
maxNav = contracts.navContract.getStoredNav(vault);

// If max nav is not initialized, return current nav -- since in the
↳ initializeNav function, we set the max nav to the current nav this
↳ should never happen, where as this is protection for any unforeseen
↳ issues
if (maxNav == 0) {
    revert Errors.NavNotInitialized(vault);
}

// Max function to return the higher of the two navs
return maxNav > currentNav ? maxNav : currentNav;
}

```

# Issue M-4: deposit function in Manager contract lacks slippage protection [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/50>

## Summary

The `deposit` function in `Manager.sol` lacks a minimum output share parameter (e.g., `minShares`), preventing users from protecting against slippage risks caused by price volatility.

## Vulnerability Detail

The `deposit` function in `Manager.sol` allows users to deposit whitelisted assets and calculates the number of shares to be minted based on oracle prices and the current NAV.

However, the function signature does not include `minShares` or a similar slippage protection parameter. If oracle prices fluctuate between the time the user submits the transaction and when it is confirmed, the number of shares the user ultimately receives may be significantly lower than expected.

## Impact

Users may suffer financial loss due to oracle price fluctuations or on-chain state changes, minting fewer shares than anticipated.

## Code Snippet

`src/core/Manager.sol`

```
function deposit(
    address vault,
    address asset,
    uint256 amount,
    address receiver,
    bytes32 referralCode
) external override whenNotPaused nonReentrant returns (uint256 shares) {
    // Check vault-level pause
    if (contracts.registry.vaultPaused(vault)) {
        revert Errors.VaultPaused(vault);
    }

    // Validate deposit inputs
    _validateDeposit(vault, asset, amount, receiver);
```

## Tool Used

Manual Review

## Recommendation

It is recommended to modify the `deposit` function to add a `minShares` parameter. Verify the number of shares received by the user.

# Issue M-5: Flawed on-chain performance fee check in updateNav causes transaction reversion [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/51>

## Summary

The `_validateNavUpdateGuardrails` function in `Manager.sol` contains a logical error when verifying the performance fee cap. The check uses the diluted Net NAV increase to calculate the maximum allowed fee, whereas the actual performance fee is calculated based on the Gross Profit before fee deduction. This mathematical mismatch causes legitimate `updateNav` transactions to revert due to being incorrectly flagged as excessive.

## Vulnerability Detail

In the `updateNav` function, the Operator submits a `newNav` (which has been diluted off-chain) and the `performanceFee` shares to be minted.

However, the on-chain guardrail logic verifies the reasonableness of the performance fee using the following steps:

1. Calculates the NAV increase: `newNav - highWaterMark`. Since `newNav` is the net value after fees are deducted, this increase represents the **Net Profit attributable to users**.
2. Calculates total asset appreciation: `navIncrease * totalSupply`.
3. Calculates the maximum allowed fee: `assetValueIncrease * performanceFeeRate`.

**The issue is:** Performance fees are charged based on **Gross Profit**, while the on-chain check calculates the cap based on **Net Profit**. Since `Net Profit = Gross Profit - Fee`, the Net Profit is inherently smaller than the Gross Profit. Consequently, the "maximum allowed fee" derived from the Net Profit will be significantly lower than the actual fee intended to be charged, causing the check to fail.

## Impact

The core functionality `updateNav` will suffer from a Denial of Service (DoS). The Operator cannot update the vault NAV, yield cannot be distributed, and the protocol is unable to collect legitimate management and performance fees.

## Code Snippet

```
// src/core/Manager.sol
```

```

// 1. Calculate NAV Increase (NewNav - HWM)
// newNav reflects the value AFTER fees are deducted
uint256 navIncreaseAboveHwm = newNav > highWaterMark ? MathLib.safeSub(newNav,
    ↪ highWaterMark) : 0;

if (navIncreaseAboveHwm > 0) {
    // 2. Calculate Total Asset Value Increase (Based on current supply)
    // This represents the Net Profit belonging to users, NOT the Gross Profit used
    ↪ for fee calculation
    uint256 assetValueIncrease = MathLib.mulDiv18(navIncreaseAboveHwm, totalSupply);

    // 3. Calculate Maximum Allowable Performance Fee Value
    // The check applies the rate to the Net Profit, resulting in a much lower
    ↪ limit than expected
    uint256 maxPerformanceFeeInAssets = MathLib.percentageOf(assetValueIncrease,
        ↪ performanceFeeRate);

    // ...

    // 5. Check: Does the submitted fee exceed the calculated limit
    if (performanceFeeShares > maxPerformanceFeeInShares) {
        revert Errors.ExcessiveDeviation(Constants.OP_PERFORMANCE_FEE,
            ↪ performanceFeeShares, maxPerformanceFeeInShares);
    }
}

```

## Tool Used

Manual Review

## Recommendation

Correct the calculation formula in `_validateNavUpdateGuardrails` to derive the Gross Profit from the Net Profit, or adjust the fee rate cap logic to account for the fact that the input is Net Profit.

# Issue M-6: Decimal Mismatch in LimitsFeature Causes Deposit Limit Bypass [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/52>

## Summary

LimitsFeature.sol fails to scale the current deposit amount during limit checks, leading to a magnitude mismatch with the native-decimal balances and configuration parameters.

## Vulnerability Detail

In the LimitsFeature contract, the validateDeposit function is used to ensure that deposits do not exceed the cap.

1. `_getUserDeposit` and `_getTotalDeposit` return values in the base asset's native decimals (e.g., 6 decimals for USDC).
2. The deposit amount is passed in 18-decimal precision and is directly added to the native-decimal balance before being compared with the limit configuration.
3. This discrepancy in magnitude (18 decimals vs. 6 decimals) causes the limit check to be easily bypassed for low-precision tokens.

## Impact

For tokens with fewer than 18 decimals, the deposit limit becomes ineffective. Users can deposit amounts far exceeding the intended cap because 18-decimal values are treated as extremely large numbers within the 6-decimal logic.

## Code Snippet

```
// LimitsFeature.sol

// _getUserDeposit returns values in native decimals (e.g., 10^6 for USDC)
uint256 currentUserDeposit = _getUserDeposit(vault, user);

// amount is passed in 18-decimal precision (10^18)
// Direct addition of different scales causes magnitude error
uint256 newUserDeposit = currentUserDeposit + amount;

if (newUserDeposit > limits.maxDepositPerUser) {
    revert Errors.ExceedsUserDepositLimit(
        newUserDeposit,
```

```
        limits.maxDepositPerUser  
    );  
}
```

## Tool Used

Manual Review

## Recommendation

Ensure that all values are scaled to the same decimal precision before performing the comparison.

# Issue M-7: L2 Sequencer Status Check Logic Inversion Leading to Denial of Service (DoS) [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/54>

## Summary

The L2 sequencer status check in `ChainlinkOracleAdapter.sol` contains a logic inversion error. The contract triggers a revert when the sequencer is in its normal active state (returning 0), causing all related transactions to fail when the sequencer is healthy.

## Vulnerability Detail

In the `_checkSequencerUptime` function of `ChainlinkOracleAdapter.sol`, the contract checks the L2 status using the Chainlink Sequencer Uptime Feed.

According to Chainlink official documentation, a return value of 0 for `answer` indicates that the sequencer is active (Up), while a return value of 1 indicates the sequencer is down (Down). However, the contract's current logic executes a revert when `answer == 0`. Consequently, when the sequencer is in a healthy state, the contract incorrectly identifies it as faulty and halts services.

## Impact

This leads to a Denial of Service (DoS) during normal protocol operations. All functionalities relying on price fetching (such as deposits, redemptions, and lending LTV checks) will be unusable on L2 chains due to the incorrect logic evaluation.

## Code Snippet

```
// The contract incorrectly identifies a healthy state (answer == 0) as a failure
↳ and triggers a revert.
if (answer == 0 || updatedAt == 0) {
    revert Errors.SequencerDown();
}
```

## Tool Used

Manual Review

## Recommendation

Correct the sequencer status check logic to trigger a revert when `answer == 1`.

# Issue M-8: Improper L2 Sequencer Check Implementation in Chainlink Oracle Adapter [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/55>

## Summary

The `_checkSequencerUptime` function in `ChainlinkOracleAdapter` incorrectly enforces a staleness check on the Sequencer status feed and lacks a Grace Period verification after the Sequencer recovers. This results in a Denial of Service (DoS) during normal operation and exposes the protocol to risks from outdated price data immediately after the Sequencer comes back online.

## Vulnerability Detail

The vulnerability consists of two main issues:

1. **Incorrect Staleness Check:** In `_checkSequencerUptime`, the code checks if the Sequencer feed has been updated within `MIN_SEQUENCER_STALENESS` (1 hour) using `block.timestamp - updatedAt`. However, Chainlink's L2 Sequencer Uptime Feed only triggers an update when the status changes (e.g., from up to down or vice versa). If the Sequencer remains healthy for an extended period, the `updatedAt` timestamp may be much older than 1 hour. The current mandatory check causes the protocol to revert and become unusable even when the Sequencer is functioning correctly, leading to a DoS.
2. **Missing Grace Period Verification:** When a Sequencer comes back online after downtime, the price data in oracle contracts may still be stale and requires time to be updated for all assets. Currently, the code only checks if the Sequencer is up (`answer == 0`) but does not verify if a sufficient grace period has passed since it resumed operation. This could allow the protocol to perform critical actions (such as liquidations or deposit valuations) using incorrect prices in the moments immediately following a Sequencer recovery.

## Impact

- **Denial of Service (DoS):** The protocol becomes unusable during normal operation because the Sequencer feed does not update frequently enough to satisfy the staleness check.
- **Price Manipulation or Incorrect Pricing:** Immediately after the Sequencer recovers, the protocol may use stale prices, leading to potential user fund losses or exposing the protocol to arbitrage attacks.

## Code Snippet

```
// Check if sequencer feed data is stale (should be very recent)
uint256 sequencerAge = block.timestamp - updatedAt;
if (sequencerAge > Constants.MIN_SEQUENCER_STALENESS) {
    revert Errors.StaleData(sequencerAge);
}
```

## Tool Used

Manual Review

## Recommendation

1. Remove the hardcoded staleness check on the Sequencer feed's updatedAt timestamp.
2. After checking that the Sequencer is up (`answer == 0`), add a Grace Period validation. Ensure that `block.timestamp - updatedAt` is **greater** than a configured Grace Period threshold before allowing the oracle to return prices.

# Issue L-1: User KYC date can be updated to future timestamp [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/42>

## Summary

A user's KYC date can be updated to a future timestamp. In this case, `GreenlistFeature::isUserGreenlisted` will underflow, which will bring a general DoS to the affected user.

## Vulnerability Detail

`GreenlistFeature::updateUserKyc` does not check if `date <= block.timestamp`:

File: `vaults/src/features/greenlist/GreenlistFeature.sol`

```
322:     function updateUserKyc(  
323:         address vault,  
324:         address user,  
325:         uint256 tier,  
326:         uint256 date  
327:     ) external onlyOperator {  
328:         _validateVaultAndUser(vault, user);  
329:         UserKycData storage kycData = _greenlist[vault][user];  
330:         if (!kycData.status) {  
331:             revert Errors.NotGreenlisted(vault, user);  
332:         }  
333:  
334:         if (tier > 0) {  
335:             kycData.tier = tier;  
336:         }  
337:@>         if (date > 0) {  
338:             kycData.date = date;  
339:         }  
340:  
341:         emit UserGreenlisted(vault, user, msg.sender);  
342:     }
```

If the user's `kycData.date` is set to future, `GreenlistFeature::isUserGreenlisted` will underflow until that time:

File: `vaults/src/features/greenlist/GreenlistFeature.sol`

```
189:     function isUserGreenlisted(  
190:         address vault,  
191:         address user  
192:     ) external view returns (bool isGreenlisted) {
```

```
...
210:         // Check staleness if configured
211:         if (config.maxStaleness > 0) {
212:@>             uint256 age = block.timestamp - kycData.date;
213:             if (age > config.maxStaleness) {
214:                 return false;
215:             }
216:         }
217:
218:         return true;
```

As a result, the affected user cannot use vaults until `kycData.date`.

## Impact

Affected user will not be able to interact with vaults.

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Either add a date validation in `updateUserKyc`, or handle underflow in `isUserGreenlisted` function.

## Discussion

**nisuscold**

@g-lightspeed

This is not an issue right? The user anyways not intended to do any operation on the vault till he is whitelisted.

The DoS is not for any other user here.

**g-lightspeed**

OK, will downgrade it to Low

## Issue L-2: Incorrect constant percents [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/44>

### Summary

Some constant percents are incorrect

### Vulnerability Detail

File: vaults/src/libraries/Constants.sol

```
18:      /// @notice 100% in precision format
19:      uint256 public constant HUNDRED_PERCENT = 100e18;
20:
21:      /// @notice 1% in precision format
22:      uint256 public constant ONE_PERCENT = 1e18;
```

Correct values are 1e18 and 1e16 respectively.

### Impact

Naming and actual values differ.

### Code Snippet

### Tool Used

Manual Review

### Recommendation

## Issue L-3: Fee receiver cannot fully redeem shares [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/45>

### Summary

When a redemption request is processed, a fee share is minted to fee receiver. This can be a problem when the fee receiver requests a redemption, as redemption fee is applied in this case too.

As a result, fee receiver cannot fully redeem shares to assets.

## Vulnerability Detail

When a redemption request is processed, redemption fee share is transferred from redeemer to fee receiver.

File: vaults/src/core/Manager.sol

```
571:         if (feeShares > 0) {
572:@>             IVault(vault).burn(owner, feeShares);
573:         }
574:
575:         // Mint fee shares directly to feeReceiver
576:         if (feeShares > 0) {
577:             address feeReceiver = contracts.registry.feeReceivers(vault);
578:             feeReceiver.requireNonZero();
579:@>             IVault(vault).mint(feeReceiver, feeShares);
580:             emit FeeTransferred(vault, feeReceiver, feeShares,
↪ FeeType.RedemptionFee, baseAsset);
581:         }
```

In order to realize fee, the fee receiver will eventually request redemption. In this case, redemption fee logic will be triggered as well, so the fee receiver cannot fully redeem assets.

## Impact

Protocol fee cannot be fully redeemed.

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Set feeShare to 0 for fee receiver's redemption process.

# Issue L-4: Vault fee guardrail bypass when feeRate = 0 or totalSupply = 0 [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/46>

## Summary

Vault's fee guardrail is applied only when `feeRate != 0` and `totalSupply != 0`. This means management fee / performance fee can be charged during vault's initial stage (i.e. `totalSupply == 0`) or if fee rate is set to 0.

## Vulnerability Detail

Vault's fee guardrail is applied only when `feeRate != 0` and `totalSupply != 0`:

File: `vaults/src/core/Manager.sol`

```
1043:@>         if (managementFeeRate != 0 && totalSupply != 0) { // For gas
    ↪ optimization, avoiding unnecessary 1060:
    ...
    if (managementFeeShares > maxManagementFeeShares) {
1061:             revert Errors.ExcessiveDeviation(Constants.OP_MANAGEMENT_FEE,
    ↪ managementFeeShares, maxManagementFeeShares);
1062:         }
1063:     }
```

This means that if `totalSupply = 0` or `managementFeeRate = 0`, `managementFeeShares` can be an arbitrary value.

A similar issue exists for `performanceFee`.

## Impact

Vault's fee guardrail can be bypassed if `feeRate == 0` or `totalSupply == 0`.

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Apply fee guardrail in all cases.

# Issue L-5: Greenlist/blacklist check can be bypassed by depositing through a vault [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/47>

## Summary

Manager contract requires `msg.sender` and `receiver` to be non-blacklisted and greenlisted during deposit. However, such check can be bypassed by depositing through vault because `msg.sender = vault` in this case.

## Vulnerability Detail

There are two ways to deposit to Manager contract: via `Manager.deposit` and `Vault.deposit`.

Manager contract requires `msg.sender` to be non-blacklisted and be greenlisted (only if green list feature is enabled):

File: `vaults/src/core/Manager.sol`

```
243:     function _validateDeposit(  
244:         address vault,  
245:         address asset,  
246:         uint256 amount,  
247:         address receiver  
248:     ) internal view {  
...  
263:         // Blacklist checks  
264:@>         _notBlacklisted(msg.sender);  
265:@>         _notBlacklisted(receiver);  
266:  
267:         // Validate greenlist for sender and receiver  
268:@>         _validateGreenlist(vault, msg.sender, receiver);
```

However, when base asset is deposited through vault, `msg.sender` will be vault address and will pass all those checks even if the transaction originator is blacklisted:

File: `vaults/src/core/Vault.sol`

```
275:     function deposit(  
276:         uint256 assets,  
277:         address receiver  
278:     ) external returns (uint256 shares) {  
279:         IManager managerContract = IManager(manager);  
280:         address baseAsset = managerContract.getVaultAsset(address(this));  
281:
```

```
282:         // Transfer assets to the vault from the sender
283:         IERC20(baseAsset).safeTransferFrom(msg.sender, address(this), assets);
284:
285:         // Deposit assets into the vault (manager already has infinite
↳ allowance)
286:         return
287: @>         managerContract.deposit(
288:             address(this),
289:             baseAsset,
290:             assets,
291:             receiver,
292:             bytes32(0)
293:         );
```

## Impact

Blacklist and greenlist check can be bypassed by depositing through vault

## Code Snippet

## Tool Used

Manual Review

## Recommendation

Expose only a single entrypoint to deposit assets.

# Issue L-6: Cannot cancel a redemption with black-listed receiver [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/48>

## Summary

When cancelling a redemption, there is a over-restrictive check that receiver should not be blacklisted. This might block redemption cancelling due to receiver being blacklisted while waiting for the redemption process.

## Vulnerability Detail

When canceling a redemption, the entry is validated:

File: vaults/src/core/Manager.sol

```
899:     function cancelRedemption(  
900:         address vault,  
901:         uint256 queueIndex  
902:     ) external override {  
903:  
904:         // Validate and get redemption entry  
905:@>         RedemptionQueueEntry memory entry =  
↪         _validateAndGetRedemptionEntry(vault, queueIndex);
```

It requires that receiver should not be blacklisted.

File: vaults/src/core/Manager.sol

```
374:     function _validateAndGetRedemptionEntry(  
375:         address vault,  
376:         uint256 queueIndex  
377:     ) internal view returns (RedemptionQueueEntry memory entry) {  
...  
396:         _notBlacklisted(entry.owner);  
397:@>         _notBlacklisted(entry.receiver);  
398:     }
```

Now consider the following scenario:

- A redemption request is created with different owner and receiver
- Later, receiver gets blacklisted
- Owner wants to cancel the redemption but he cannot due to the check in cancellation

## **Impact**

Redemption request cannot be cancelled and will be stuck at pending status.

## **Code Snippet**

## **Tool Used**

Manual Review

## **Recommendation**

Skip blacklist checking during redemption cancellation.

# Issue L-7: ChainlinkOracleAdapter Initialization Bypasses Parent Initialization Method [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/56>

## Summary

The `initialize` function in `ChainlinkOracleAdapter.sol` directly assigns the `administrator` state variable instead of calling the parent contract's initialization function `__AccessControl_init`.

## Vulnerability Detail

The `ChainlinkOracleAdapter` contract inherits from `AccessControl`. During initialization, its `initialize` function performs a direct assignment: `administrator = administrator_;`.

This approach bypasses the standard `__AccessControl_init` initialization path of the parent contract, leading to several issues:

1. It skips critical checks defined in the parent contract (such as the non-zero address validation performed by `AddressLib.requireNonZeroContract(_administrator)`).
2. It violates OpenZeppelin upgradeable contract design standards, which require child contracts to invoke the initializers of all inherited parent contracts.
3. It may leave the parent contract's internal state uninitialized, introducing potential compatibility issues or logic risks during future version upgrades or functional expansions.

## Impact

Critical inheritance checks (such as address validation) are bypassed, and standard upgradeable contract practices are violated. This could result in an incomplete parent contract state or introduce security risks during subsequent contract upgrades.

## Code Snippet

```
function initialize(address administrator_) external initializer {
    administrator = administrator_;
    emit AdministratorSet(address(0), administrator_);
}
```

## Tool Used

Manual Review

## Recommendation

Modify the `initialize` function to explicitly call `__AccessControl_init(administrator_)` to ensure the parent contract is correctly initialized, and remove the manual direct assignment.

# Issue L-8: Performance fee guardrail calculation uses real-time total supply leading to incorrect cap [RE-SOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/57>

## Summary

`Manager.sol` uses the real-time `totalSupply` when validating the performance fee cap. Because total supply fluctuates with user deposits and withdrawals, funds deposited right before an `updateNav` call will inflate the calculated cap, rendering the guardrail ineffective in preventing excessive fee extraction.

## Vulnerability Detail

In the `_validateNavUpdateGuardrails` function in `Manager.sol`, the contract calculates a maximum allowable performance fee based on NAV growth and the current `totalSupply` to prevent the Operator from extracting excessive fees.

The calculation is performed as follows:

```
uint256 assetValueIncrease = MathLib.mulDiv18(navIncreaseAboveHwm, totalSupply);
uint256 maxPerformanceFeeInAssets = MathLib.percentageOf(assetValueIncrease,
↳ performanceFeeRate);
```

The vulnerability lies in the fact that `totalSupply` is a dynamic variable. Performance fees are typically charged on yield generated by managed assets over a past period (i.e., the interval between the last NAV update and the current one). If a significant amount of capital is deposited right before the Operator calls `updateNav`, the `totalSupply` increases immediately. These new funds did not participate in the profit-generating activities of the previous cycle, yet they are included in the base for the guardrail calculation, which artificially inflates the `assetValueIncrease`. This allows an Operator to submit a performance fee much higher than the actual proportional profit and still pass the guardrail check.

## Impact

The performance fee guardrail cannot accurately limit fee extraction. In scenarios with large capital fluctuations, the fees collected by the Operator may exceed the intended rate cap set by the protocol. This leads to excessive dilution of user assets within the Vault and creates an unfair distribution of value between new and existing users.

## Code Snippet

```
if (navIncreaseAboveHwm > 0) {
    // Calculate asset value increase above HWM: (navIncreaseAboveHwm *
    ↪ totalSupply) / PRECISION
    // This gives us the asset value increase in asset terms (18 decimals)
    // Use MathLib.mulDiv18 for precision math
    uint256 assetValueIncrease = MathLib.mulDiv18(navIncreaseAboveHwm, totalSupply);

    // Calculate expected performance fee in asset terms: (assetValueIncrease *
    ↪ performanceFeeRate) / PRECISION
```

## Tool Used

Manual Review

## Recommendation

It is recommended to introduce a snapshot mechanism. Record the totalSupply at the end of each updateNav call and use this snapshotted value as the base for calculating the performance fee cap during the next update.

# Issue L-9: Use of Deprecated answeredInRound Check [RESOLVED]

Source: <https://github.com/sherlock-audit/2025-12-yield-finance-dec-29th/issues/58>

## Summary

The ChainlinkOracleAdapter contract uses the `answeredInRound` field when validating oracle responses. However, Chainlink has deprecated this field, and relying on it for validity checks may lead to integration risks.

## Vulnerability Detail

In the `_validateChainlinkResponse` function, the code checks the timeliness of price data by comparing `answeredInRound` with `roundId`. According to Chainlink's official documentation and best practices, the `answeredInRound` field is no longer recommended for use, as its behavior may not meet expectations on certain chains or with upgraded oracle versions. The currently recognized standard for checking data freshness is verifying whether the `updatedAt` timestamp falls within a predefined threshold (Heartbeat).

## Impact

Using a deprecated field may result in false positives or negatives regarding data validity, increasing the unreliability of the protocol integration.

## Code Snippet

```
if (updatedAt == 0) {
    revert Errors.InvalidTimestamp(updatedAt);
}
if (answeredInRound < roundId) {
    revert Errors.StalePrice(asset, block.timestamp - updatedAt);
}
```

## Tool Used

Manual Review

## Recommendation

It is recommended to remove the logic check for `answeredInRound` and rely solely on the `updatedAt` timestamp combined with a heartbeat threshold to verify data freshness.

# 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.