// Security Assessment

01.02.2025 - 01.24.2025

Harvest Strategy - Base

Harvest Finance

HALBORN

Harvest Strategy - Base - Harvest Finance

Prepared by: HALBORN

Last Updated 02/25/2025

Date of Engagement by: January 2nd, 2025 - January 24th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	0	2	3	1

TABLE OF CONTENTS

1. Introduction

Harvest Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on January 2nd, 2025 and ending on January 27th, 2025. The security assessment was scoped to the smart contracts provided in the private repository provided to Halborn. Further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 16 days for the engagement, and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the Harvest Finance team:

- Use a single consistent precision factor (e.g., 1e18) for accumulating rewards across all tokens (Solved).
- Replace amountOutMin = 1 with a slippage-aware calculation (Not Solved - Risk Accepted).
- Implement a _claimRewards() function that retrieves external rewards from the lending protocol (Solved).
- Consider adding a nonReentrant modifier (from OpenZeppelin's ReentrancyGuard) in all functions across the code-base performing external calls to potentially untrusted contracts (Solved).
- Add a storage gap in upgradeable contracts that are parent contracts (Solved).
- Upgrade the Solidity version to 0.8.x (Solved).

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (slither).
- Testnet deployment (Foundry).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: harvest-strategy-base

(b) Assessed Commit ID: dfc630c

(c) Items in scope:

- base/interface/IDex.sol
- base/interface/IUpgradableStrategy.sol
- base/interface/IVault.sol
- base/interface/IUpgradeSource.sol
- base/interface/extrafi/ILendingPool.sol
- base/interface/extrafi/DataTypes.sol
- base/interface/extrafi/IStakingRewards.sol
- base/interface/aerodrome/IPool.sol
- base/interface/aerodrome/IRouter.sol
- base/interface/aerodrome/IGauge.sol
- base/interface/IStrategy.sol
- base/interface/IUniversalLiquidatorRegistry.sol
- base/interface/IRewardForwarder.sol
- base/interface/balancer/IBVault.sol
- base/interface/IUniversalLiquidator.sol
- base/interface/IERC4626.sol
- base/interface/compound/IComet.sol
- base/interface/compound/ICometRewards.sol
- base/interface/weth/IWETH.sol
- base/interface/moonwell/EIP20NonStandardInterface.sol
- base/interface/moonwell/ComptrollerInterface.sol
- base/interface/moonwell/ErrorReporter.sol
- base/interface/moonwell/IRModels
- base/interface/moonwell/IRModels/InterestRateModel.sol
- base/interface/moonwell/MTokenInterfaces.sol
- base/interface/IProfitSharingReceiver.sol
- base/interface/IPotPool.sol
- base/interface/IController.sol
- base/Controller.sol
- base/inheritance/ControllableInit.sol
- base/inheritance/Governable.sol
- base/inheritance/Storage.sol
- base/inheritance/GovernableInit.sol
- base/inheritance/Controllable.sol
- base/RewardForwarder.sol
- base/PotPool.sol
- base/VaultV1.sol
- base/VaultV2.sol
- base/upgradability/BaseUpgradeabilityProxy.sol
- base/upgradability/BaseUpgradeableStrategyStorage.sol
- base/upgradability/BaseUpgradeableStrategy.sol
- base/upgradability/ReentrancyGuardUpgradeable.sol
- base/upgradability/StrategyProxy.sol
- base/VaultProxy.sol
- base/VaultStorage.sol
- strategies/extrafi/ExtraFiLendStrategyMainnet_AERO.sol
- strategies/extrafi/ExtraFiLendStrategy.sol
- strategies/aerodrome/AerodromeStableStrategyMainnet_jEUR_EURC.sol

- strategies/aerodrome/AerodromeStableStrategy.sol
- strategies/aerodrome/AerodromeVolatileStrategyMainnet_AERO_USDC.sol
- strategies/aerodrome/AerodromeVolatileStrategy.sol
- strategies/fluid/FluidLendStrategy.sol
- strategies/fluid/FluidLendStrategyMainnet_USDC.sol
- strategies/compound/CompoundStrategyMainnet_USDbC.sol
- strategies/compound/CompoundStrategy.sol
- strategies/moonwell/MoonwellFoldStrategyV2.sol
- strategies/moonwell/MoonwellFoldStrategyV2Mainnet_WETH.sol
- strategies/moonwell/MoonwellFoldStrategyV2Mainnet_USDC.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- aec5437
- 79868b1
- 19d0668
- 805b649
- 6234bd9

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	3	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DECIMAL MISMATCH IN REWARD CALCULATIONS	MEDIUM	SOLVED - 02/03/2025
MISSING SLIPPAGE CHECKS IN MULTIPLE LOCATIONS	MEDIUM	RISK ACCEPTED - 02/03/2025
POTENTIAL REENTRANCY	LOW	SOLVED - 02/03/2025
VAULTV1 UPGRADEABLE CONTRACT IS MISSING STORAGE GAP	LOW	SOLVED - 02/03/2025
OUTDATED SOLIDITY VERSION	LOW	SOLVED - 02/03/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MINIMAL EVENT EMISSIONS FOR KEY STATE CHANGES	INFORMATIONAL	SOLVED - 02/03/2025

7. FINDINGS & TECH DETAILS

7.1 DECIMAL MISMATCH IN REWARD CALCULATIONS

// MEDIUM

Description

When computing rewards, the `PotPool` contract uses the pool's **LP token decimals** (set via `_setupDecimals`) in `rewardPerToken(...)` and then divides by `10^(rewardToken.decimals())` in the `earned(...)` function. In other words:

- `rewardPerToken(rt)` multiplies by `10^(poolDecimals)`
- `earned(rt, account)` divides by `10^(rewardToken.decimals())`

– contracts/base/PotPool.sol

```
233     function rewardPerToken(address rt) public view returns (uint256) {
234         if (totalSupply() == 0) {
235             return rewardPerTokenStoredForToken[rt];
236         }
237         return
238             rewardPerTokenStoredForToken[rt].add(
239                 lastTimeRewardApplicable(rt)
240                     .sub(lastUpdateTimeForToken[rt])
241                     .mul(rewardRateForToken[rt])
242                     .mul(10**uint256(decimals()))
243                     .div(totalSupply())
244             );
245     }
```

```
255     function earned(address rt, address account) public view returns (uint256) {
256         return
257             stakedBalanceOf[account]
258                 .mul(rewardPerToken(rt).sub(userRewardPerTokenPaidForToken[rt][account]
259                 .div(10**uint256(ERC20(rt).decimals()))
260                 .add(rewardsForToken[rt][account]);
261     }
```

If the reward token's decimals differ from the LP token's decimals, the final reward amounts could be scaled incorrectly or lead to unexpected rounding.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:M/R:N/S:C (6.3)

Recommendation

It is recommended to:

- Use a single consistent precision factor (e.g., `1e18`) for accumulating rewards across all tokens.
- Alternatively, ensure that the reward token's decimals always match the LP token's decimals if that is the intended design.
- Clearly document any difference between the pool's decimal usage and the reward token's decimal usage so that integrators are aware of potential rounding effects.

Remediation Comment

SOLVED: The `Harvest Finance` **team** has solved this issue as recommended. The commit hash for reference is `aec543797fc0688445504b727e443cbc0cacca85`.

Remediation Hash

https://github.com/harvestfi/harvest-strategy-base/commit/aec543797fc0688445504b727e443cbc0cacca85

7.2 MISSING SLIPPAGE CHECKS IN MULTIPLE LOCATIONS

// MEDIUM

Description

In multiple strategies, when converting reward tokens back to the underlying asset, the code commonly passes a minimum output of `1`:

```
IUniversalLiquidator(...).swap(rewardToken, underlying, balance, 1, address(this));
```

In the `RewardForwarder` contract, when calling `IUniversalLiquidator(liquidator).swap(...)`, the code sets `amountOutMin = 1`. This effectively means the contract wants to "accept any non-zero swap output," which can lead to highly unfavorable slippage if the market shifts or if the liquidator is front-run

```
59      function _notifyFee(
60          address _token,
61          uint256 _profitSharingFee,
62          uint256 _strategistFee,
63          uint256 _platformFee
64      ) internal {
65          address _controller = controller();
66          address liquidator = IController(_controller).universalLiquidator();
67
68          uint totalTransferAmount = _profitSharingFee.add(_strategistFee).add(_platform
69          require(totalTransferAmount > 0, "totalTransferAmount should not be 0");
70
71          IERC20(_token).safeTransferFrom(msg.sender, address(this), totalTransferAmount
72
73          address _targetToken = IController(_controller).targetToken();
74
75          if (_token != _targetToken) {
76              IERC20(_token).safeApprove(liquidator, 0);
77              IERC20(_token).safeApprove(liquidator, _platformFee);
78
79              uint amountOutMin = 1;
80
81              if (_platformFee > 0) {
82                  IUniversalLiquidator(liquidator).swap(
83                      _token,
84                      _targetToken,
85                      _platformFee,
86                      amountOutMin,
87                      IController(_controller).protocolFeeReceiver()
88                  );
89              }
90          } else {
91              IERC20(_targetToken).safeTransfer(IController(_controller).protocolFeeRece
92          }
93
94          if (_token != iFARM) {
95              IERC20(_token).safeApprove(liquidator, 0);
96              IERC20(_token).safeApprove(liquidator, _profitSharingFee.add(_strategistFe
97
98              uint amountOutMin = 1;
99
100             if (_profitSharingFee > 0) {
```

```
                IUniversalLiquidator(liquidator).swap(
                    _token,
                    iFARM,
                    _profitSharingFee,
                    amountOutMin,
                    IController(_controller).profitSharingReceiver()
                );
            }
            if (_strategistFee > 0) {
                IUniversalLiquidator(liquidator).swap(
                    _token,
                    iFARM,
                    _strategistFee,
                    amountOutMin,
                    IStrategy(msg.sender).strategist()
                );
            }
        } else {
            if (_strategistFee > 0) {
                IERC20(iFARM).safeTransfer(IStrategy(msg.sender).strategist(), _strate
            }
            IERC20(iFARM).safeTransfer(IController(_controller).profitSharingReceiver(
        }
    }
```

Within the `_liquidateReward()` function, the code calls the Universal Liquidator with `minAmountOut` set to `1`:

```
IUniversalLiquidator(_universalLiquidator).swap(
  _rewardToken,
  _underlying,
  remainingRewardBalance,
  1,
  address(this)
);
```

A `minAmountOut` of `1` leaves no meaningful on-chain protection against large price slippage or sandwich attacks. In a hostile environment, a manipulated or malicious swap could cause the strategy to receive far fewer underlying tokens than expected, resulting in lost yield for strategy users.

Seen In:

- **CompoundStrategy** (in `_liquidateReward()`)
- **AerodromeStableStrategy / AerodromeVolatileStrategy** (in `_liquidateReward()`)
- **ExtraFiLendStrategy** (in `_liquidateRewards()`)
- **FluidLendStrategy** (in `_liquidateRewards()`)
- **MoonwellFoldStrategyV2** (in `_liquidateRewards()`)

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:N/R:N/S:C (6.3)

Recommendation

To solve issues related to missing slippage checks, it is recommended to:

- Replace `amountOutMin = 1` with a slippage-aware calculation. For example, pass in a minimum expected output from off-chain price checks or keep track of a tolerance limit.
- If dynamic slippage is not practical, at least allow the caller (strategy or governance) to specify a suitable `amountOutMin` to protect against front-running or illiquid markets.
- Alternatively, implement off-chain mechanisms for slippage prevention.

Remediation Comment

RISK ACCEPTED: The `Harvest Finance` **team** has accepted the risk related to this finding and did not perform any code modification regarding this issue. Additionally, the `Harvest Finance` team has informed that off-chain mechanisms are in-place to prevent slippage issues.

7.3 POTENTIAL REENTRANCY

// LOW

Description

In the `PotPool` contract, `pushAllRewards(address recipient)` calls `updateRewards(recipient)` and transfers rewards to `recipient`. Unlike `getAllRewards` or `withdraw`, `pushAllRewards` does **not** have the `nonReentrant` or `defense` modifiers. Although it is restricted to governance only, if `recipient` is a malicious contract, it could potentially re-enter via token callbacks (depending on the token used):

```
300     function pushAllRewards(address recipient) external updateRewards(recipient) onlyG
301       bool rewardPayout = (!smartContractStakers[recipient] || !IController(controller
302       for(uint256 i = 0 ; i < rewardTokens.length; i++ ){
303         uint256 reward = earned(rewardTokens[i], recipient);
304         if (reward > 0) {
305             rewardsForToken[rewardTokens[i]][recipient] = 0;
306             if (rewardPayout) {
307                 IERC20(rewardTokens[i]).safeTransfer(recipient, reward);
308                 emit RewardPaid(recipient, rewardTokens[i], reward);
309             } else {
310                 emit RewardDenied(recipient, rewardTokens[i], reward);
311             }
312         }
313       }
```

Moving forward, most strategies do not implement a `nonReentrant` guard. Although many DeFi protocols do not expose arbitrary callbacks, if a reward token or external lending/pool contract allowed reentrant calls, the strategy's state-changing logic could be re-invoked unexpectedly.

Seen in:

- **CompoundStrategy** (internal calls to `IComet`, no `nonReentrant`)
- **Aerodrome*Strategy** (calls `IGauge`, `IPool`, no `nonReentrant`)
- **ExtraFiLendStrategy**, **FluidLendStrategy** (calls external vaults)
- **MoonwellFoldStrategyV2** (flash loan from Balancer `bVault`, no standard reentrancy guard)

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:L/R:N/S:C (3.9)

Recommendation

In order to solve this issue, it is recommended to:

- If `liquidator` is trusted and never calls back into `RewardForwarder`, re-entrancy is likely not an issue. However, if future changes might allow untrusted or user-supplied liquidators, consider adding a `nonReentrant` modifier (from OpenZeppelin's `ReentrancyGuard`) in all functions across the code-base performing external calls to potentially untrusted contracts.
- Keep documentation indicating that only a trusted liquidator is intended.

If external protocols are trusted and have no callbacks, the risk is low. Otherwise, consider adding `ReentrancyGuard` to state-changing functions.
Keep track of potential tokens or new protocol features that could introduce callback functions in the future.

Remediation Comment

SOLVED: The `Harvest Finance` **team** has solved this issue as recommended. The commit hash for reference is `79868b1da1323db8eee0f593ef0ba0b950afe161`.

Remediation Hash

https://github.com/harvestfi/harvest-strategy-base/commit/79868b1da1323db8eee0f593ef0ba0b950afe161

7.4 VAULTV1 UPGRADEABLE CONTRACT IS MISSING STORAGE GAP

// LOW

Description

The `VaultV1` contract, which inherits or acts as an upgradeable parent contract, does not reserve a storage gap at the end of its contract variables. In the context of upgradeable contracts, a storage gap is critical to preventing storage layout conflicts when future state variables are added in subsequent contract versions. Without a storage gap, any future modifications to the contract's storage layout could overwrite existing state variables, leading to unpredictable behavior or loss of state.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:C (3.9)

Recommendation

In order to resolve this issue, it is recommended to:

- Add a reserved storage gap array (e.g., `uint256[50] private __gap;`) at the end of the contract to preserve the current storage layout.
- Ensure all future variables are added above this gap array, maintaining the existing layout for previously deployed contract instances.

Remediation Comment

SOLVED: The `Harvest Finance` **team** has solved this issue as recommended. The commit hash for reference is `19d066836ae71ef0719a0c434d536d07d2770494`.

Remediation Hash

https://github.com/harvestfi/harvest-strategy-base/commit/19d066836ae71ef0719a0c434d536d07d2770494

7.5 OUTDATED SOLIDITY VERSION

// LOW

Description

Several contracts utilize the older Solidity 0.6.x compiler version, which lacks the newest language features, improvements in error handling, and security enhancements introduced in later releases. Continuing to use a legacy compiler version may expose these contracts to known bugs and limit compatibility with modern tooling and best practices.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

To solve this issue, it is recommended to:

- Migrate the codebase to a more recent Solidity version (e.g., 0.8.x), after confirming no breaking changes in dependencies.
- Leverage updated compiler checks, safe math built-ins (`unchecked` blocks in Solidity 0.8+), and other improvements to enhance the contracts' reliability and maintainability.

Remediation Comment

SOLVED: The `Harvest Finance` **team** has solved the issue as recommended. The commit hash for reference is `805b64915d6fcd5903639ea8de06235d9c795de9`.

Remediation Hash

https://github.com/harvestfi/harvest-strategy-base/commit/805b64915d6fcd5903639ea8de06235d9c795de9

7.6 MINIMAL EVENT EMISSIONS FOR KEY STATE CHANGES

// INFORMATIONAL

Description

Several strategies do not emit events when critical parameters change (e.g., changing gauges, adding reward tokens, toggling folding, or pausing investments). This hinders off-chain analytics and transparency.

Seen In:

- **CompoundStrategy** (no events for certain changes like `market()` updates)
- **AerodromeStable/VolatileStrategy** (no events for `setGauge`, `addRewardToken`, etc.)
- **ExtraFiLendStrategy** (no events for adjusting fees or `market` updates)
- **FluidLendStrategy** (no events for `addRewardToken`, emergency exit, etc.)
- **MoonwellFoldStrategyV2** (no events on factor updates, toggling folding)

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

To solve this issue, it is recommended to:

- Emit events (e.g., `GaugeChanged`, `RewardTokenAdded`, `FoldingToggled`, etc.) with relevant parameters (old/new values).
- Include `indexed` parameters to facilitate off-chain querying.

Remediation Comment

SOLVED: The `Harvest Finance` **team** has solved this issue as recommended. The commit hash for reference is `6234bd9b9e93a8c1b5a756aa9056fa798f2b455c`.

Remediation Hash

https://github.com/harvestfi/harvest-strategy-base/commit/6234bd9b9e93a8c1b5a756aa9056fa798f2b455c

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team conducted a comprehensive review of all findings generated by the Slither static analysis tool.

After careful examination and consideration of the flagged issues, it was determined that within the project's specific context and scope, all were false positives or irrelevant.
