// Security Assessment

02.28.2025 - 03.06.2025

Curve/Pendle Strategy

Blueprint Finance

HALBORN

Curve/Pendle Strategy - Blueprint Finance

Prepared by: HALBORN

Last Updated 03/24/2025

Date of Engagement: February 28th, 2025 - March 6th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
14	0	1	3	3	7

TABLE OF CONTENTS

8. Automated Testing

1. Introduction

Blueprint Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on February 28th, 2025 and ending on March 10th, 2025. The security assessment was scoped to the smart contracts provided in the Blueprint-Finance/sc_earn-v1 Github repository provided to Halborn. Further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 7 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 Blueprint Finance team. The main ones were the following:

- Accept a slight variation (slippage) on expected output when withdrawing from Pendle strategy.
- Replace the hard-coded 0 (zero) in the call to remove_liquidity_one_coin with a carefully computed minimum.
- When harvesting rewards, enforce safe transfer (passing true as argument to the parameter), so it would handle gracefully all non-standard ERC20 tokens.
- Remove the forced 3% slippage if the intent is to let users specify the full slippage themselves.
- Add an option for multi-hop swaps, using the ISwapRouter.exactInput(params) with an encoded path.
- In Pendle's strategy _protocolDeposit function, disable Keep YT mode, or directly use a function that does not mint YT at all.

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

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
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:

(b) Assessed Commit ID: f12bdbf

(c) Items in scope:

- src/strategies/PendleV2/PendleV2Strategy.sol
- src/strategies/CurveV2/CurveV2Strategy.sol
- src/strategies/StrategyBase.sol
- src/libraries/UniswapV3HelperV1.sol

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

REMEDIATION COMMIT ID:

- https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/1192fd367a597680ba665be88ed84ff0cf5bb867
- https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/9d8e3ac8bb8e9b181594851fd8e9e6138a29c582
- https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/be3f2dd8db2c57ab6bbc4701e17cb61b3a0f2ae6
- https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/9f2c0d62c77d157eee70ee61227cc23e774240c1
- https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/408b8144acc62d75c3d94bbcc5ff3954e426c419

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	3	3

INFORMATIONAL
7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INSUFFICIENT SLIPPAGE PROTECTION LEADS TO LOSS OF FUNDS	HIGH	SOLVED - 03/19/2025
REWARD ACCRUAL CALCULATION INACCURACY DUE TO FAILED TRANSFERS	MEDIUM	SOLVED - 03/21/2025
DOS FOR WITHDRAW OPERATIONS IN PENDLE STRATEGY	MEDIUM	SOLVED - 03/19/2025
BROKEN REWARD APPROVAL MECHANISM	MEDIUM	SOLVED - 03/19/2025
MISSING MULTI-HOP SUPPORT FOR UNISWAPV3	LOW	RISK ACCEPTED - 03/19/2025
FORCED SLIPPAGE INCLUSION	LOW	RISK ACCEPTED - 03/19/2025
SUBOPTIMAL DEPOSIT FLOW IN PENDLE STRATEGY	LOW	SOLVED - 03/19/2025
MISSING EVENT EMISSIONS FOR RELEVANT STATE UPDATES	INFORMATIONAL	ACKNOWLEDGED - 03/19/2025
DIRECT ERC20 TRANSFER TO VAULT (ERC4626) DOES NOT CREDIT SHARES	INFORMATIONAL	ACKNOWLEDGED - 03/19/2025
MISSING CHECK WHEN ASSIGNING ADDRESS TO STATE VARIABLE	INFORMATIONAL	SOLVED - 03/19/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT NATSPEC DOCUMENTATION	INFORMATIONAL	SOLVED - 03/19/2025
USE OF MAGIC NUMBERS	INFORMATIONAL	SOLVED - 03/19/2025
MISPLACED NONREENTRANT MODIFIER	INFORMATIONAL	SOLVED - 03/19/2025
PUSH0 AND NEWER OPCODES MIGHT NOT BE SUPPORTED IN ALL CHAINS	INFORMATIONAL	ACKNOWLEDGED - 03/19/2025

7. FINDINGS & TECH DETAILS

7.1 INSUFFICIENT SLIPPAGE PROTECTION LEADS TO LOSS OF FUNDS

// HIGH

Description

When adding or removing liquidity, the `CurveV2Stragegy` contract currently calculates `estimatedLp` (deposit side) or `calc_withdraw_one_coin` (withdrawal side) but does not consistently enforce a strict minimum amount out (beyond 0). This exposes the strategy to high slippage or price manipulation.

Specifically:

• **Deposit**:
The code calculates `estimatedLp` and sets `minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000)`. This is good for a baseline, but consider verifying the entire deposit flow to ensure `minLp` is being applied each time. If `minLp` is set to 0 (zero) or is not enforced in certain paths, attackers or front-runners could manipulate pool balances, causing the strategy to mint fewer LP tokens than expected.

```
    function _protocolDeposit(uint256 amount_, uint256 minLp) internal virtual override {
        uint256 estimatedLp;
        uint256 minLp;
        if (N_COINS == 2) {
            uint256[2] memory amounts;
            amounts[uint256(tokenIndex)] = amount_;
            estimatedLp = curvePool.calc_token_amount(amounts, true);
            minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);
            // @dev curvePool supports only fixedSize arrays, the interface is in vypers context
            curvePool.add_liquidity(amounts, minLp);
        } else if (N_COINS == 3) {
            uint256[3] memory amounts;
            amounts[uint256(tokenIndex)] = amount_;
            estimatedLp = curvePool.calc_token_amount(amounts, true);
            minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);
            curvePool.add_liquidity(amounts, minLp);
        } else if (N_COINS == 4) {
            uint256[4] memory amounts;
            amounts[uint256(tokenIndex)] = amount_;
            estimatedLp = curvePool.calc_token_amount(amounts, true);
            minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);
            curvePool.add_liquidity(amounts, minLp);
        }
        // Stake LP tokens into the gauge
        uint256 lpReceived = IERC20(lpToken).balanceOf(address(this));
        IERC20(lpToken).approve(address(curveGauge), lpReceived);
        ICurveGauge(curveGauge).deposit(lpReceived);
    }
```

• **Withdrawal**:
The function `_protocolWithdraw` calls `remove_liquidity_one_coin(lpAmountToWithdraw, int128(tokenIndex), 0)` with a hard-coded `minAmount=0` (zero). This means the strategy is vulnerable to receiving **far less underlying asset than expected if there's a significant pool imbalance or a front-running scenario**.

```
    function _protocolWithdraw(uint256 amount_, uint256) internal virtual override {
        uint256 lpBalance = IERC20(address(curveGauge)).balanceOf(address(this));
        uint256 assetAmount = curvePool.calc_withdraw_one_coin(lpBalance, int128(tokenIndex));
        uint256 lpAmountToWithdraw = amount_.mulDiv(lpBalance, assetAmount);
        // Unstake LP tokens from the gauge
        ICurveGauge(curveGauge).withdraw(lpAmountToWithdraw);
        // Withdraw single asset from Curve pool
        ICurvePool(curvePool).remove_liquidity_one_coin(lpAmountToWithdraw, int128(tokenIndex), 0);
    }
```

Proof of Concept

In order to reproduce this issue, use the following Foundry proof of concept. The code will illustrate that a lack of minimum out in `_protocolWithdraw` is an exploitable design.

- **PoC Code:**

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import "forge-std/Test.sol";
import {console2 as console} from "forge-std/console2.sol";
import {CurveV2StrategyTest} from "./CurveV2Strategy.t.sol";
import {ICurvePool, ICurveGauge} from "../../src/strategies/CurveV2/ICurveV2.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract CurveV2StrategyExploitMockCallTest is CurveV2StrategyTest {
    address internal attacker = address(0x9999);

    function setUp() public virtual override {
        super.setUp();
        // Provide the attacker with large capital
        deal(address(asset), attacker, 1_000_000e6);
        vm.label(attacker, "Attacker");
    }

    /**
     * @notice Demonstrates how the strategy's zero-minAmount slippage approach
     * allows an attacker to manipulate the pool, causing massive user loss.
     * We use vm.mockCall for simplicity.
     */
    function test_ExploitSlippageCurveStrategy_Mock() public {
        /*****************************************************
         * 1) Hazel Deposits into the Strategy
         *****************************************************/
        uint256 depositAmount = 10_000e6; // 10k USDC
        deal(address(asset), hazel, depositAmount);

        // The strategy calls:
        // calc_token_amount(uint256[3] memory, bool) => uint256
        // with deposit arrays for USDC-index=1 if nCoins=3
        {
            uint256[3] memory depositArray;
            depositArray[1] = depositAmount;
            bytes memory depositCalcData = abi.encodeWithSignature(
                "calc_token_amount(uint256[3],bool)",
                depositArray,
                true
            );
            // Pool returns 10,000 LP tokens for 10,000 USDC deposit
            bytes memory depositCalcReturn = abi.encode(10_000e18);

            vm.mockCall(address(usdcPool), depositCalcData, depositCalcReturn);

            // Also mock the actual add_liquidity function call if needed
            // for a 3-coin pool => add_liquidity(uint256[3], uint256)
            // We'll pass 10_000 LP as success, ignoring the returned value:
            bytes memory addLiquidityData = abi.encodeWithSignature(
                "add_liquidity(uint256[3],uint256)",
                depositArray,
                uint256(9000e18) // just must be <= the 10000 minted above after slippage
            );
            // Return empty to simulate success
            vm.mockCall(address(usdcPool), addLiquidityData, new bytes(0));
```

```
        }

        // user deposits
        vm.startPrank(hazel);
        IERC20(asset).approve(address(strategy), depositAmount);
        uint256 hazelShares = strategy.deposit(depositAmount, hazel);
        vm.stopPrank();

        console.log("Hazel deposit done. Received strategy shares:", hazelShares);


        // The strategy should now have staked ~10,000 LP in the gauge. Let's assume
        // gauge balanceOf(strategy) is 10,000e18.
        {
            bytes memory gaugeBalData = abi.encodeWithSignature(
                "balanceOf(address)",
                address(strategy)
            );
            // Return 10,000e18 as the gauge balance
            bytes memory gaugeBalReturn = abi.encode(10_000e18);
            vm.mockCall(address(usdcGauge), gaugeBalData, gaugeBalReturn);
        }

        /**************************************************
         * 2) Attacker Manipulates the Pool
         **************************************************/
        // We'll simulated that now the pool is drastically imbalanced.
        // This can be achieved on-chain, for example, through Flash loans.
        // The strategy calls:
        // calc_withdraw_one_coin(lpBalance, int128(tokenIndex)) => e.g. (10,000e18, 1)
        vm.startPrank(attacker);
        {
            bytes memory withdrawCalcData = abi.encodeWithSignature(
                "calc_withdraw_one_coin(uint256,int128)",
                uint256(10_000e18), // the staked LP tokens
                int128(1)          // USDC index
            );
            bytes memory withdrawCalcReturn = abi.encode(1_000e6);

            vm.mockCall(address(usdcPool), withdrawCalcData, withdrawCalcReturn);
        }

        {
            bytes memory removeLiquidityData = abi.encodeWithSignature(
                "remove_liquidity_one_coin(uint256,int128,uint256)",
                uint256(10_000e18),
                int128(1),
                uint256(0)
            );

            vm.mockCall(address(usdcPool), removeLiquidityData, new bytes(0));
        }
        vm.stopPrank();

        /**************************************************
         * 3) Hazel Withdraws => Massive Loss
         **************************************************/
        uint256 hazelBalBefore = IERC20(asset).balanceOf(hazel);
        console.log("Hazel USDC before withdraw:", hazelBalBefore);

        // Even trying to withdraw only 20% of its shares, Hazel gets a revert.
        // ERC4626ExceededMaxWithdraw()
        vm.startPrank(hazel);
        vm.expectRevert();
        strategy.withdraw(hazelShares * 20 / 100, hazel, hazel);
        vm.stopPrank();
    }
}
```

- **Logs**

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:N/I:L/D:M/Y:C (8.0)

Recommendation

To address this issue, it is recommended to:

- Continue using `calc_token_amount` to estimate the LP received and set `minLp = estimatedLp * (100% - slippageTolerance)` in all deposit paths. Choose a tight but feasible slippage tolerance (e.g., 0.5–1%), which is commonly used by strategies integrators.
- Replace the hard-coded 0 (zero) in the call to `remove_liquidity_one_coin` with a carefully computed minimum. For example, use `uint256 expected = curvePool.calc_withdraw_one_coin(lpAmountToWithdraw, int128(tokenIndex)); uint256 minOut = expected.mulDiv(10000 - slippageBasisPoints, 10000);` to revert if actual output is below the threshold.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/1192fd367a597680ba665be88ed84ff0cf5bb867

7.2 REWARD ACCRUAL CALCULATION INACCURACY DUE TO FAILED TRANSFERS

// MEDIUM

Description

The `harvestRewards` function of the `StrategyBase` contract currently trusts the success indicator (boolean return value) of `TokenHelper.attemptSafeTransfer` without verifying the actual transferred amount.

Tokens that do not strictly adhere to the ERC20 standard or unexpectedly implemented tokens may return a success status (true) despite transferring fewer tokens than intended or no tokens at all.

```
function harvestRewards()

{ ... }

                uint256 collectedFee = claimedBalance.mulDiv(rewardTokens[i].fee, 10000, Math.Rounding.Ceil);
                if (TokenHelper.attemptSafeTransfer(address(rewardAddress), feeRecipient, collectedFee, false)) {
                    rewardTokens[i].accumulatedFeeAccounted += collectedFee;
                    netReward = claimedBalance - collectedFee;
                    emit Harvested(_vault, netReward);
                }
```

As a result, the contract incorrectly updates `accumulatedFeeAccounted`, causing discrepancies between internal accounting and actual token balances. Over time, this could lead to significant unnoticed financial damage, loss of user funds or leakage of rewards. An attacker or faulty token implementation could exploit this vulnerability, causing the strategy to miscalculate its total value locked (TVL), distribute rewards incorrectly or silently lose tokens.

BVSS

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

Recommendation

It is recommended to **enforce safe transfer** (passing `true` as argument to the parameter), so it would handle gracefully all non-standard ERC20 tokens.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/9d8e3ac8bb8e9b181594851fd8e9e6138a29c582

7.3 DOS FOR WITHDRAW OPERATIONS IN PENDLE STRATEGY

// MEDIUM

Description

In the `PendleV2Strategy` contract, when removing liquidity in `_protocolWithdraw`, the code currently does:

```
            TokenOutput memory output = createTokenOutputSimple(asset(), amount_);
            router.removeLiquiditySingleToken(
                address(this), address(market), netTokenIn, output, createEmptyLimitOrderData()
            );
```

Where `createTokenOutputSimple()` is:

```
    function createTokenOutputSimple(address tokenOut, uint256 minTokenOut)
        internal
        pure
        returns (TokenOutput memory)
    {
        return TokenOutput({
            tokenOut: tokenOut,
            minTokenOut: minTokenOut,
            tokenRedeemSy: tokenOut,
            pendleSwap: address(0),
            swapData: createSwapTypeNoAggregator()
        });
    }
```

In other words, **this creates a scenario of strict equality on the slippage parameter**, where `minTokenOut == amount_`. If there are any negative price impact, protocol fees, or rounding inside the router, the `amount_` out will be slightly less, and this will cause consistent reverts.

In a real-world scenario, this will effectively lead the withdrawals to a Denial of Service state.

Proof of Concept

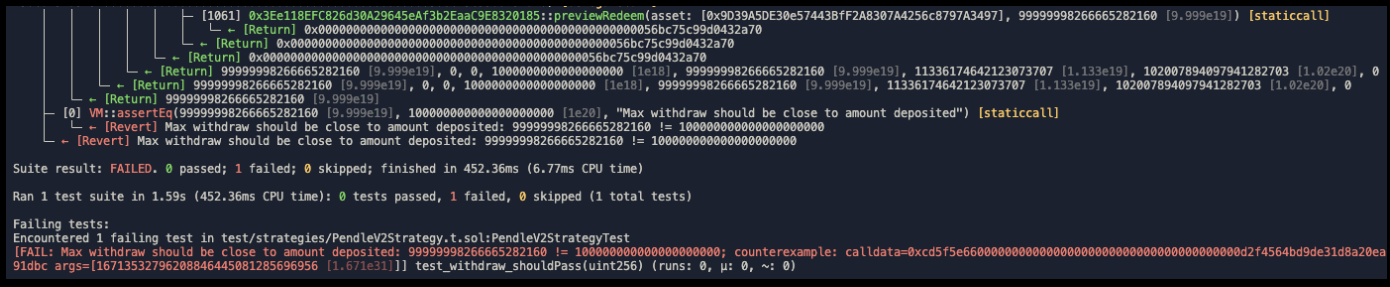
In order to reproduce this issue, consider using the following Foundry proof of concept.

- **PoC Code:**

```
    function test_withdraw_shouldPass(uint256 amount_) public {
        vm.startPrank(hazel);
        uint256 amount = 100 ether;
        _mintAsset(amount, hazel);

        asset.approve(address(strategy), amount);
        uint256 shares = strategy.deposit(amount, hazel);

        uint256 hazelBalBeforeWwl = asset.balanceOf(hazel);
        assertGe(asset.balanceOf(hazel), 0, "Hazel asset balance");
        assertEq(shares, amount, "Hazel share balance == amount");

        uint256 expireTime = IPMarket(wstETHMarket).expiry();
        vm.warp(expireTime + 100);
        assertEq(
            strategy.maxWithdraw(hazel), amount, "Max withdraw should be close to amount deposited"
        );
        strategy.withdraw(strategy.maxWithdraw(hazel), hazel, hazel);
```

```
        assertGt(asset.balanceOf(hazel), hazelBalBeforeWwl, "Hazel new asset balance");
        assertEq(asset.balanceOf(hazel), hazelBalBeforeWwl + amount, "Hazel new asset balance");
        assertEq(strategy.balanceOf(hazel), 0, "Hazel new share balance");
        vm.stopPrank();
    }
```

- Logs:



BVSS

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

Recommendation

It is recommended to accept a slight variation (slippage) on expected output. Usually, the calculation should look something like the following:

```
minTokenOut = amount_.mulDiv(100_00 - someAllowedSlippage, 100_00);
```

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/be3f2dd8db2c57ab6bbc4701e17cb61b3a0f2ae6

7.4 BROKEN REWARD APPROVAL MECHANISM

// MEDIUM

Description

The approvals across the `StrategyBase` contract, mainly in the initializer `__StrategyBase_init`, but also in the `addRewardToken` function have several issues that can lead to denial of service, degraded user experience and inability to properly distribute rewards.The main reasons are as follows:

1. Using unsafe `ERC20.approve()` method, instead of `SafeERC20.forceApprove()`: As already mentioned, some ERC20 tokens are non-standard and require the allowances to be zeroed-out before modified. USDT, for example, a widely used stablecoin, will revert unsafe calls to `approve()` if the current allowance of the caller is greater than zero. If used as a reward token and the approver has non-zero allowance, the contract initialization will effectively revert.

2. While newly added reward tokens are pushed into the `rewardTokens` array and marked as approved, the actual call to `token.approve` (or even better, `token.forceApprove`) never happens. During the initialization, the contract explicitly approves the targeted ERC20 with `type(uint256).max`, but fails to do so in the `addRewardToken` function, despite updating the state to `rewardTokenApproved`. This can cause later transfers or fee handling of newly added token rewards to fail if the contract attempts to move them.

3. Some ERC20 tokens, such as $COMP and $UNI, will revert calls to `approve()` if the amount passed as parameter is greater than `type(uint96).max`, also leading to a DoS state.

```
function __StrategyBase_init(
    IERC20 baseAsset_,
    string memory shareName_,
    string memory shareSymbol_,
    address feeRecipient_,
    uint256 depositLimit_,
    address owner_,
    RewardToken[] memory rewardTokens_,
    address vault_
) internal initializer nonReentrant {
    // Initialize inherited contracts
    __ERC4626_init(IERC20Metadata(address(baseAsset_)));
    __ERC20_init(shareName_, shareSymbol_);
    __Ownable_init(owner_);

    // Iterate through the provided reward tokens to set them up
    if (rewardTokens_.length != 0) {
        for (uint256 i; i < rewardTokens_.length;) {
            // Validate reward token address, current fee accounted, and high watermark
            if (address(rewardTokens_[i].token) == address(0)) {
                revert InvalidRewardTokenAddress();
            }
            if (rewardTokens_[i].accumulatedFeeAccounted != 0) {
                revert AccumulatedFeeAccountedMustBeZero();
            }

            // Approve the strategy to spend the reward token without limit
            if (!rewardTokens_[i].token.approve(address(this), type(uint256).max)) revert ERC20ApproveFail();
            // Add the reward token to the strategy's list and mark it as approved
            rewardTokens.push(rewardTokens_[i]);
            rewardTokenApproved[address(rewardTokens_[i].token)] = true;
            unchecked {
                i++;
```

```
            }
        }
    }

    // Validate and set the fee recipient address
    if (feeRecipient_ == address(0)) revert InvalidFeeRecipient();
    feeRecipient = feeRecipient_;

    // Set the deposit limit for the strategy
    if (depositLimit_ == 0) revert InvalidDepositLimit();

    depositLimit = depositLimit_;
    // Calculate and set the decimals for the strategy's shares based on the base asset's decimals
    _decimals = IERC20Metadata(address(baseAsset_)).decimals() + DECIMAL_OFFSET;
    _vault = vault_;
}
```

```
function addRewardToken(RewardToken calldata rewardToken_) external onlyOwner nonReentrant {
    // Ensure the reward token address is not zero, not already approved, and its parameters are correctly initialized.
    if (address(rewardToken_.token) == address(0)) {
        revert InvalidRewardTokenAddress();
    }
    if (rewardTokenApproved[address(rewardToken_.token)]) {
        revert RewardTokenAlreadyApproved();
    }
    if (rewardToken_.accumulatedFeeAccounted != 0) {
        revert AccumulatedFeeAccountedMustBeZero();
    }

    // Add the reward token to the list and approve it for unlimited spending by the strategy.
    rewardTokens.push(rewardToken_);
    rewardTokenApproved[address(rewardToken_.token)] = true;
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

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

- Replace unsafe `ERC20.approve()` methods for `SafeERC20.forceApprove()`, which will handle edge-cases gracefully.
- Modify the `addRewardToken` function so it actually performs the intended approval on the targeted token.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/9f2c0d62c77d157eee70ee61227cc23e774240c1

7.5 MISSING MULTI-HOP SUPPORT FOR UNISWAPV3

// LOW

Description

All swap logic in the the `UniswapV3HelperV1` library uses `exactInputSingle`, which supports only one pool (single-hop). If the best route for a token pair requires multiple hops (e.g., no direct pool or better pricing through an intermediate token), the library cannot handle it.

Although many pairs have direct liquidity, a missing multi-hop path can limit price efficiency or break entirely if no direct pool exists for determined pair.

```
function swapExactInputSingle(ISwapRouter swapRouter, SwapParams memory params)
    public
    returns (uint256 swappedAmountOut)
{
    ISwapRouter.ExactInputSingleParams memory exactInputSingleParams = ISwapRouter.ExactInputSingleParams({
        tokenIn: params.tokenIn,
        tokenOut: params.tokenOut,
        fee: params.poolFee,
        recipient: params.recipient,
        deadline: block.timestamp,
        amountIn: params.amountIn,
        amountOutMinimum: params.minAmountOut,
        sqrtPriceLimitX96: 0
    });
    swappedAmountOut = swapRouter.exactInputSingle(exactInputSingleParams);
}
```

BVSS

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

Recommendation

To solve this issue, it is recommended to add an option for multi-hop swaps, using the `ISwapRouter.exactInput(params)` with an encoded path. Alternatively, in the case of supporting single-hop swaps only, clearly document that the library does not support multi-hop swaps.

Remediation Comment

RISK ACCEPTED: The **Blueprint Finance team** has accepted the risk related to this finding.

7.6 FORCED SLIPPAGE INCLUSION

// LOW

Description

In the `UniswapV3HelperV1` library, the `swapToToken` function takes an user-defined `minAmountOut` but then applies an additional 3% slippage.

```
        // calculate minAmountOut with max slippage
        minAmountOut = minAmountOut.mulDiv(100_00 - UNISWAPV3_MAX_SLIPPAGE, 100_00, Math.Rounding.Floor);
```

If the user already included slippage in their `minAmountOut`, this code further enforces another 3% slippage. This can be surprising or undesirable if the user expects `minAmountOut` to be exactly as passed in the function parameter.

BVSS

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

Recommendation

To solve this issue, it is recommended to:

- Clarify in the documentation that `minAmountOut` will be subjected to another 3% slippage factor.
- Rename the parameter, for example, to `expectedAmountOut` and calculate a final `minAmountOut` in the function implementation.
- Remove the forced 3% slippage if the intent is to let users specify the full slippage themselves.

Remediation Comment

RISK ACCEPTED: The **Blueprint Finance team** has accepted the risk related to this finding.

7.7 SUBOPTIMAL DEPOSIT FLOW IN PENDLE STRATEGY

// LOW

Description

When depositing into the Pendle strategy, the code explicitly calls Pendle's `addLiquiditySingleTokenKeepYt()` methods, which results in both LP and YT tokens being minted. Immediately afterward, the strategy swaps the newly minted YT back into the underlying asset and sends it back to the vault:

```
    function _protocolDeposit(uint256 amount_, uint256) internal virtual override {
        if (market.isExpired()) revert MarketExpired();

        (uint256 netLpOut, uint256 netYtOut,,) =
            routerStatic.addLiquiditySingleTokenKeepYtStatic(address(market), address(asset()), amount_);
        uint256 minLpOut = netLpOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
        uint256 minYtOut = netYtOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
        TokenInput memory input = createTokenInputSimple(address(asset()), amount_);
        router.addLiquiditySingleTokenKeepYt(address(this), address(market), minLpOut, minYtOut, input);

        // @dev: we swap all the YT to asset
        uint256 ytBalance = IERC20(_YT).balanceOf(address(this));
        if (ytBalance > 0) {
            (uint256 estimatedTokenOut,,,,,,) =
                routerStatic.swapExactYtForTokenStatic(address(market), ytBalance, asset());
            uint256 minTokenOut = estimatedTokenOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
            LimitOrderData memory limitOrderData = createEmptyLimitOrderData();
            TokenOutput memory tokenOutput = createTokenOutputSimple(address(asset()), minTokenOut);

            (uint256 netTokenOut,,) =
                router.swapExactYtForToken(address(this), address(market), ytBalance, tokenOutput, limitOrderData);
            // @dev: transfer the assets to the vault, else it can mess up the _protocolWithdraw function
            IERC20(asset()).safeTransfer(_vault, netTokenOut);
        }
    }
```

This flow can cause extra slippage, gas costs and confusion for depositors because part of the deposit is effectively "refunded" to the vault instead of remaining in the strategy. Per standard Pendle best practices, a yield strategy that does not intend to hold YT (short yield) usually calls the variant of Pendle's add-liquidity zap that omits Keep YT mode, thus consolidating the deposit into LP or PT tokens in a single atomic transaction. That approach reduces complexity, gas overhead and potential slippage issues.

Alternatively, if the strategy aims to be "long yield", it should intentionally retain YT and handle its claimable yield properly, rather than selling it immediately.

In the current implementation, the strategy performs unnecessary mint-and-sell steps, increasing gas usage and slippage costs. A portion of the user's deposit is returned to the vault soon after deposit, which may not match the user's expectation that 100% of the deposit is invested. If the design is to remain "short yield", the simpler route is to avoid YT creation altogether, which leads to more predictable user outcomes.

BVSS

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

Recommendation

If the strategy intends to be "short yield" (no YT exposure), disable Keep YT mode, or directly use a function that does not mint YT at all. Pendle natively supports an atomic deposit flow that converts the underlying asset to LP or PT in a single step. This ensures the user's deposit goes directly into the final intended tokens (PT or LP), avoiding an unnecessary second step of selling YT and transferring proceeds out.

If the contract is never supposed to hold YT, do not mint YT in the first place. Instead, deposit the underlying into the Pendle Router in a single call that produces only the desired tokens (PT or LP).

In any case, clarify the deposit flow through NatSpec for users, integrators and auditors.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/be3f2dd8db2c57ab6bbc4701e17cb61b3a0f2ae6

7.8 MISSING EVENT EMISSIONS FOR RELEVANT STATE UPDATES

// INFORMATIONAL

Description

Throughout the scope, including `PendleV2Stragegy`, `CurveV2Strategy`, `UniswapV3HelperV1` and `StrategyBase`, the only declared event is in the `StragegyBase` contract, as follows:

```
event Harvested(address indexed harvester, uint256 tvl);
```

Effectively, there are no other events emitted upon significant state transitions or user actions. Typical operations such as deposits, withdrawals, reward claims and token swaps do not produce any on-chain logs besides the raw ERC20 transfer events.

Events provide several benefits, such as:

- **User and Developer transparency:** Allows off-chain clients (block explorers, UI dashboards, analytics services) to detect and parse events for real-time updates or historical queries.
- **Debug and Auditing:** Simplifies debugging transactions and verifying the correctness of operations.
- **Monitoring and Alerts:** Third-party watchers rely on event emission to trigger notifications, logs or further automated actions.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

For each relevant state update or user-driven action, emit an event that includes relevant details - e.g. sender, amounts, timestamps and references to input and output tokens, when applicable. For example:

```
event Withdraw(address indexed caller, uint256 indexed amountIn, uint256 indexed amountOut, uint64 timestamp);
```

Although slightly less gas-efficient, `indexed` event fields make the data more quickly accessible to off-chain tools that parse events, and add them to a special data structure known as "topics" instead of the data part of the log. In Solidity, each event declaration is allowed to carry up to 3 (three) indexed fields.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** has acknowledged the finding.

7.9 DIRECT ERC20 TRANSFER TO VAULT (ERC4626) DOES NOT CREDIT SHARES

// INFORMATIONAL

Description

In `_protocolDeposit` of the `PendleV2Stragegy` contract, the strategy swaps YT for the underlying asset and transfer it directly to the vault via `IERC20(asset()).safeTransfer(_vault, netTokenOut);`.

```
    function _protocolDeposit(uint256 amount_, uint256) internal virtual override {
        if (market.isExpired()) revert MarketExpired();

        (uint256 netLpOut, uint256 netYtOut,,) =
            routerStatic.addLiquiditySingleTokenKeepYtStatic(address(market), address(asset()), amount_);
        uint256 minLpOut = netLpOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
        uint256 minYtOut = netYtOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
        TokenInput memory input = createTokenInputSimple(address(asset()), amount_);
        router.addLiquiditySingleTokenKeepYt(address(this), address(market), minLpOut, minYtOut, input);

        // @dev: we swap all the YT to asset
        uint256 ytBalance = IERC20(_YT).balanceOf(address(this));
        if (ytBalance > 0) {
            (uint256 estimatedTokenOut,,,,,,) =
                routerStatic.swapExactYtForTokenStatic(address(market), ytBalance, asset());
            uint256 minTokenOut = estimatedTokenOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
            LimitOrderData memory limitOrderData = createEmptyLimitOrderData();
            TokenOutput memory tokenOutput = createTokenOutputSimple(address(asset()), minTokenOut);

            (uint256 netTokenOut,,) =
                router.swapExactYtForToken(address(this), address(market), ytBalance, tokenOutput, limitOrderData);
            // @dev: transfer the assets to the vault, else it can mess up the _protocolWithdraw function
            IERC20(asset()).safeTransfer(_vault, netTokenOut);
        }
    }
```

In a standard ERC4626 vault, direct ERC20 transfers do not credit anyone with new shares. The tokens end up in the vault contract address, but the vault has no record of who should own those assets, effectively resulting in loss of funds for the depositor. There's an immediate risk of funds being permanently loss.

BVSS

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

Recommendation

Do **not** perform direct transfers of underlying tokens to the vault address. Instead, call proper ERC4626 methods such as `deposit()` or `mint()` so it properly updates internal share balances for the intended recipient. If the vault operates under non-standard patterns (i.e. different from ERC4626), confirm that direct ERC20 deposits are recognized.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** has acknowledged the finding.

7.10 MISSING CHECK WHEN ASSIGNING ADDRESS TO STATE VARIABLE

// INFORMATIONAL

Description

In the `StrategyBase` contract, during contract's initialization through the `__StrategyBase_init` function, a value for the `_vault` address state variable is attributed. However, the function is missing a check to revert in cases the value passed as parameter is the `address(0)` - zero address, what could lead to ultimate inefficiency.

```
    function __StrategyBase_init(
        IERC20 baseAsset_,
        string memory shareName_,
        string memory shareSymbol_,
        address feeRecipient_,
        uint256 depositLimit_,
        address owner_,
        RewardToken[] memory rewardTokens_,
        address vault_
    ) internal initializer nonReentrant {
        // Initialize inherited contracts
        __ERC4626_init(IERC20Metadata(address(baseAsset_)));
        __ERC20_init(shareName_, shareSymbol_);
        __Ownable_init(owner_);

        // Iterate through the provided reward tokens to set them up
        if (rewardTokens_.length != 0) {
            for (uint256 i; i < rewardTokens_.length;) {
                // Validate reward token address, current fee accounted, and high watermark
                if (address(rewardTokens_[i].token) == address(0)) {
                    revert InvalidRewardTokenAddress();
                }
                if (rewardTokens_[i].accumulatedFeeAccounted != 0) {
                    revert AccumulatedFeeAccountedMustBeZero();
                }

                // Approve the strategy to spend the reward token without limit
                if (!rewardTokens_[i].token.approve(address(this), type(uint256).max)) revert ERC20ApproveFail();
                // Add the reward token to the strategy's list and mark it as approved
                rewardTokens.push(rewardTokens_[i]);
                rewardTokenApproved[address(rewardTokens_[i].token)] = true;
                unchecked {
                    i++;
                }
            }
        }

        // Validate and set the fee recipient address
        if (feeRecipient_ == address(0)) revert InvalidFeeRecipient();
        feeRecipient = feeRecipient_;

        // Set the deposit limit for the strategy
        if (depositLimit_ == 0) revert InvalidDepositLimit();

        depositLimit = depositLimit_;
        // Calculate and set the decimals for the strategy's shares based on the base asset's decimals
        _decimals = IERC20Metadata(address(baseAsset_)).decimals() + DECIMAL_OFFSET;
        _vault = vault_;
    }
```

BVSS

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

Recommendation

Include a control-flow verification (`if` statement with `revert()` - following current style-guide), and revert the call in case the value passed to the function parameter `address vault_` is the `address(0)`.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/be3f2dd8db2c57ab6bbc4701e17cb61b3a0f2ae6

7.11 INCORRECT NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

In both `PendleV2Strategy` and `CurveV2Stragegy` contracts, in the `_protocolWithdraw` function, the NatSpec documentation incorrectly mentions the Aave protocol, as follows:

```
    /**
     * @dev Withdraws assets from the Aave protocol.
     * @param amount_ The amount of assets to withdraw.
     */
```

BVSS

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

Recommendation

Update the NatSpec documentation to reflect the actual callee protocol.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/408b8144acc62d75c3d94bbcc5ff3954e426c419

7.12 USE OF MAGIC NUMBERS

// INFORMATIONAL

Description

In multiple instances, the usage of magic numbers was observed. If the same constant (literal number) is used multiple times across the contract or code-base, it should be defined using a constant of appropriate type.

```
- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 146]

                minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 149]

            } else if (N_COINS == 3) {

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 150]

                uint256[3] memory amounts;

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 153]

                minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 155]

            } else if (N_COINS == 4) {

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 156]

                uint256[4] memory amounts;

- As seen in src/strategies/CurveV2/CurveV2Strategy.sol [Line: 159]

                minLp = estimatedLp.mulDiv(10_000 - MAX_SLIPPAGE, 10_000);

- As seen in src/strategies/CurveV2/ICurveV2.sol [Line: 15]

        function calc_token_amount(uint256[3] memory amounts, bool isDeposit) external view returns (uint256);

- As seen in src/strategies/CurveV2/ICurveV2.sol [Line: 16]

        function calc_token_amount(uint256[4] memory amounts, bool isDeposit) external view returns (uint256);

- As seen in src/strategies/CurveV2/ICurveV2.sol [Line: 19]

        function add_liquidity(uint256[3] memory amounts, uint256 minLP) external;

- As seen in src/strategies/CurveV2/ICurveV2.sol [Line: 20]

        function add_liquidity(uint256[4] memory amounts, uint256 minLP) external;
```

```
- As seen in src/strategies/PendleV2/PendleV2Strategy.sol [Line: 42]

            if (MAX_SLIPPAGE > 100_00) revert InvalidSlippage();

- As seen in src/strategies/PendleV2/PendleV2Strategy.sol [Line: 127]

            uint256 minLpOut = netLpOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);

- As seen in src/strategies/PendleV2/PendleV2Strategy.sol [Line: 128]

            uint256 minYtOut = netYtOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);

- As seen in src/strategies/PendleV2/PendleV2Strategy.sol [Line: 137]

                uint256 minTokenOut = estimatedTokenOut.mulDiv(100_00 - MAX_SLIPPAGE, 100_00);
```

BVSS

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

Recommendation

Replace magic numbers across the code-based for well-defined constants with the appropriate type.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/1192fd367a597680ba665be88ed84ff0cf5bb867

7.13 MISPLACED NONREENTRANT MODIFIER

// INFORMATIONAL

Description

Some contracts in-scope are utilizing the `ReentrancyGuard` from OpenZeppelin for reentrancy locks (mutex), which is best practice. However, the `nonReentrant` modifier should be placed **before all other modifiers**, to effectively prevent reentrancy on the function's modifiers.

```
    function __StrategyBase_init(
        IERC20 baseAsset_,
        string memory shareName_,
        string memory shareSymbol_,
        address feeRecipient_,
        uint256 depositLimit_,
        address owner_,
        RewardToken[] memory rewardTokens_,
        address vault_
    ) internal initializer nonReentrant {
```

```
    function addRewardToken(RewardToken calldata rewardToken_) external onlyOwner nonReentrant {
```

BVSS

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

Recommendation

It is recommended to place the `nonReentrant` modifier before all other modifiers.

Remediation Comment

SOLVED: The **Blueprint Finance team** has solved the issue as recommended.

Remediation Hash

https://github.com/Blueprint-Finance/sc_earn-v1/pull/152/commits/408b8144acc62d75c3d94bbcc5ff3954e426c419

7.14 PUSH0 AND NEWER OPCODES MIGHT NOT BE SUPPORTED IN ALL CHAINS

// INFORMATIONAL

Description

Solc compiler `version 0.8.20` switches the default target EVM version to Shanghai. The generated bytecode will include **PUSH0** opcodes. The recently released Solc compiler `version 0.8.25` switches the default target EVM version to Cancun, so it is also important to note that it also adds-up new opcodes such as **TSTORE**, **TLOAD** and **MCOPY**.

Be sure to select the appropriate EVM version in case you intend to deploy on a chain apart from Ethereum mainnet, like L2 chains that may not support **PUSH0**, **TSTORE**, **TLOAD** and/or **MCOPY**, otherwise deployment of your contracts will fail.

BVSS

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

Recommendation

It is important to consider the targeted deployment chains before writing immutable contracts because, in the future, there might exist a need for deploying the contracts in a network that could not support new opcodes from Shanghai or Cancun EVM versions.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** has acknowledged the finding.

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.

```
ERC4626Upgradeable._deposit(address,address,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol#261-273) uses arbitrary from in transferFrom: SafeERC20.safeTransferFrom(IERC20(asset()),caller,address(this),assets) (li
b/openzeppelin-contracts-upgradeable/contracts/token/ERC20/extensions/ERC4626Upgradeable.sol#269)
ERC4626._deposit(address,address,uint256,uint256) (lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol#239-251) uses arbitrary from in transferFrom: SafeERC20.safeTransferFrom(IERC20(asset()),caller,address(this),assets) (lib/openzeppelin-contracts/contracts
/token/ERC20/extensions/ERC4626.sol#247)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#257)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
MockERC4626Protect.lendFunds(uint256) (src/utils/mocks/MockERC4626Protect.sol#83-86) ignores return value by IERC20(asset()).transfer(msg.sender,amount_) (src/utils/mocks/MockERC4626Protect.sol#85)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
MockBeraOracle.values (src/utils/mocks/MockBeraOracle.sol#10) is never initialized. It is used in:
        - MockBeraOracle.getPrice(string) (src/utils/mocks/MockBeraOracle.sol#30-32)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#257)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#261)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#262)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#263)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#264)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#265)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#242)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#266)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#204-275) performs a multiplication on the result of a division:
        - low = low / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#246)
        - result = low * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#272)
Math.invMod(uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#315-361) performs a multiplication on the result of a division:
        - quotient = gcd / remainder (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#337)
        - (gcd,remainder) = (remainder,gcd - remainder * quotient) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#339-346)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
CurveV2Strategy._totalAssets() (src/strategies/CurveV2/CurveV2Strategy.sol#92-96) uses a dangerous strict equality:
        - lpBalance == 0 (src/strategies/CurveV2/CurveV2Strategy.sol#94)
CurveV2Strategy.getAvailableAssetsForWithdrawal() (src/strategies/CurveV2/CurveV2Strategy.sol#80-86) uses a dangerous strict equality:
        - lpBalance == 0 (src/strategies/CurveV2/CurveV2Strategy.sol#82)
PendleV2Strategy._totalAssets() (src/strategies/PendleV2/PendleV2Strategy.sol#102-108) uses a dangerous strict equality:
        - lpBalance == 0 (src/strategies/PendleV2/PendleV2Strategy.sol#104)
PendleV2Strategy.getAvailableAssetsForWithdrawal() (src/strategies/PendleV2/PendleV2Strategy.sol#83-96) uses a dangerous strict equality:
        - lpBalance == 0 (src/strategies/PendleV2/PendleV2Strategy.sol#88)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
StrategyBase.harvestRewards(bytes).remainingRewards (src/strategies/StrategyBase.sol#361) is a local variable never initialized
StrategyBase.harvestRewards(bytes).rewardsToVault (src/strategies/StrategyBase.sol#359) is a local variable never initialized
StrategyBase.harvestRewards(bytes).effectiveRewardsToVault (src/strategies/StrategyBase.sol#360) is a local variable never initialized
CurveV2Strategy.getRewardTokenAddresses().isExtraRewardToken (src/strategies/CurveV2/CurveV2Strategy.sol#105) is a local variable never initialized
CurveV2Strategy.getRewardTokenAddresses().extraRewardToken (src/strategies/CurveV2/CurveV2Strategy.sol#104) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
StrategyBase.harvestRewards(bytes) (src/strategies/StrategyBase.sol#345-416) ignores return value by TokenHelper.attemptSafeTransfer(address(rewardAddress),_vault,rewardsToVault,false) (src/strategies/StrategyBase.sol#374)
CurveV2Strategy._protocolDeposit(uint256,uint256) (src/strategies/CurveV2/CurveV2Strategy.sol#139-166) ignores return value by IERC20(lpToken).approve(address(curveGauge),lpReceived) (src/strategies/CurveV2/CurveV2Strategy.sol#164)
PendleV2Strategy.getAvailableAssetsForWithdrawal() (src/strategies/PendleV2/PendleV2Strategy.sol#83-96) ignores return value by (netTokenOut) = routerStatic.removeLiquiditySingleTokenStatic(address(market),lpBalance,asset()) (src/strategies/PendleV2/PendleV2Strategy.sol#89-95)
PendleV2Strategy._totalAssets() (src/strategies/PendleV2/PendleV2Strategy.sol#102-108) ignores return value by (netTokenOut,None,None,None,None,None,None,None) = routerStatic.removeLiquiditySingleTokenStatic(address(market),lpBalance,asset()) (src/strategies/PendleV2/PendleV2St
rategy.sol#105-106)
PendleV2Strategy._protocolDeposit(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#122-146) ignores return value by (netLpOut,netYtOut,None,None) = routerStatic.addLiquiditySingleTokenKeepYtStatic(address(market),address(asset()),amount_) (src/strategies/PendleV2/
PendleV2Strategy.sol#125-126)
PendleV2Strategy._protocolDeposit(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#122-146) ignores return value by router.addLiquiditySingleTokenKeepYt(address(this),address(market),minLpOut,minYtOut,input) (src/strategies/PendleV2/PendleV2Strategy.sol#130)
PendleV2Strategy._protocolDeposit(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#122-146) ignores return value by (estimatedTokenOut,None,None,None,None,None,None) = routerStatic.swapExactYtForTokenStatic(address(market),ytBalance,asset()) (src/strategies/P
endleV2/PendleV2Strategy.sol#135-136)
PendleV2Strategy._protocolDeposit(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#122-146) ignores return value by (netTokenOut,None,None) = router.swapExactYtForToken(address(this),address(market),ytBalance,tokenOutput,limitOrderData) (src/strategies/PendleV2/Pe
ndleV2Strategy.sol#141-142)
PendleV2Strategy._protocolWithdraw(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#152-164) ignores return value by (netTokenOut,None,None,None,None,None,None,None) = routerStatic.removeLiquiditySingleTokenStatic(address(market),lpBalance,asset()) (src/strategies
/PendleV2/PendleV2Strategy.sol#156-157)
PendleV2Strategy._protocolWithdraw(uint256,uint256) (src/strategies/PendleV2/PendleV2Strategy.sol#152-164) ignores return value by router.removeLiquiditySingleToken(address(this),address(market),netTokenIn,output,createEmptyLimitOrderData()) (src/strategies/PendleV2/PendleV2Str
ategy.sol#160-162)
PendleV2Strategy._getRewardsToStrategy(bytes) (src/strategies/PendleV2/PendleV2Strategy.sol#172-174) ignores return value by market.redeemRewards(address(this)) (src/strategies/PendleV2/PendleV2Strategy.sol#173)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Slither:. analyzed (65 contracts with 58 detectors), 34 result(s) found
```

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.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.