Tempest Finance Audit Report

Version 2.0

Audited by:

xiaoming9090

SpicyMeatball

peakbolt

October 8, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Tempest Finance

Tempest Finance is an innovative ALM built on top of Ambient Finance, designed to simplify liquidity provision through two main strategies: the Symmetric Vault and the Arbitrage Vault.

Symmetric Vaults Tempest's Symmetric Vaults optimize user returns while mitigating impermanent loss. These vaults utilize a base symmetric order combined with range limit orders to address portfolio imbalances caused by price movements, avoiding the costly swap fees typical in active rebalancing strategies. This approach offers two key benefits:

Avoidance of Swap Fees: By not using swaps for rebalancing, the vault avoids paying transaction fees.

Passive Rebalancing with Price Movements: The vault rebalances as prices fluctuate within a predefined range (Limit Order Range), earning fees while rebalancing. Although it still faces some permanent loss, the fee income helps offset these losses. The strategy benefits from the fact that prices often do not move in straight lines, effectively betting that the fee income from the price volatility outweighs any permanent loss experienced.

Arbitrage Recapture Vault The Arbitrage Recapture Vault is the first of its kind to internalize MEV (Maximal Extractable Value) created when LSTs (Liquid Staking Tokens) depeg, which is typically the most volatile period for LST/LRT liquidity. It leverages knockout orders, which capture the delta created as LSTs deviate from their peg. This effectively forms a flexible buy wall that tracks the peg, capturing arbitrage opportunities that would otherwise be captured by external actors.

The Tempest Advantage Tempest Finance is redefining LST/LRT liquidity by recapturing profits for LPs and empowering Liquid Staking Protocols to offer new liquidity solutions that were previously only achievable by active arbitrageurs. All of this is achieved while maintaining the same swap functionality that traditional concentrated liquidity provides.

Tempest Finance is making LST/LRT liquidity great again, ensuring that liquidity providers can maximize their returns in a sustainable and efficient manner.

2.2 Overview

Project	Tempest Finance
Repository	tempest$_smart_contract$
Commit Hash	caf1103eceba...
Mitigation Hash	5d023ab3f64f...
Date	28 August 2024 - 13 September 2024

2.3 Issues Found

Severity	**Count**
High Risk	7

Medium Risk	12
Low Risk	5
Informational	6
Total Issues	**30**

3 Findings Summary

ID	Description	Status
H-1	DOS of core functionalities due to underflow when fee gets updated	Resolved
H-2	Incorrect fee accounting leads to protocol receiving lesser fees than expected	Resolved
H-3	Existing KO position might be overwritten during deposit	Resolved
H-4	`RswEthStrategy`'s total assets will be inflated	Resolved
H-5	Strategy vaults are vulnerable to share inflation attack	Resolved
H-6	`WstEthStrategy`'s total assets will be inflated	Resolved
H-7	Incorrect pivot time accounting can block asset withdrawals	Resolved
M-01	Fee can be applied retrospectively to a time period that has already passed	Resolved
M-02	Malicious users can force the strategy to invest all its assets bypassing the `investedPercentage` restriction	Resolved
M-03	Division by zero error if `totalUnderlying <= feeToClaim`	Resolved
M-04	`setFees()` fails to collect pending fees before updating `fee` percentage	Resolved
M-05	Incorrect rounding direction for `previewWithdraw()`	Resolved
M-06	`BaseAmbientStrategy` is incorrectly initialized with zero `liqSlippage`	Resolved
M-07	Lack of liquidity slippage check in `zapDeposit()`	Resolved
M-08	Liquidity might be provided to the incorrect KO positions	Resolved
M-09	There is no staleness check for the Chainlink price feed answer	Resolved
M-10	Vault's basic functions can be temporarily blocked	Resolved
M-11	An incorrect swap amount calculation leads to the underutilization of the user's provided liquidity	Resolved
M-12	Insufficient liquidity may be burned during redemption.	Resolved
L-1	Residual allowance granted to external protocol	Resolved
L-2	Lack of L2 Sequencer Uptime Check	Resolved
L-3	Overlapping of tick ranges can cause double counting of liquidity	Resolved
L-4	UsdcETHOracle has hardcoded Chainlink price feeds addresses	Resolved
L-5	Assets will be lost if a deposit is equal to or less than `padding`	Resolved
I-1	Explicit revert instead of if `amountUnwrappedLST` is less than `minWithdrawAmount`	Resolved
I-2	Unused Code	Resolved

ID	Description	Status
I-3	Redundant burnLiquidity cmd in `_collectFees()`	Resolved
I-4	`_burnAllLiquidities()` should skip burning of liquidity when `liqShares == 0`	Resolved
I-5	Tick range validation should reject `_upperTick == _lowerTick`	Resolved
I-6	Accumulated fees in KO liquidity will be forfeited if Merkle proof was not provided during withdrawal	Acknowledged

4 Findings

High Risk

[H-1] DOS of core functionalities due to underflow when fee gets updated

Context:

- WstEthStrategy.sol#L368
- RswEthStrategy.sol#L366

Description: At T0, assume the following state:

- totalAsset = 200
- invested = 100
- stratPnL = 0
- feeToClaim = 0
- feeClaimed = 0
- fee = 20%

When the `_updateStratPnLAndFees` function is executed:

```
stratPnL = 200 - 100 = 100

feeToClaim = (stratPnL * 20%) - feeClaimed
feeToClaim = (100 * 20%) - 0 = 20
```

At T1, the `_claimFees` function is executed:

```
feeToClaim: 20 -> 0
feeClaimed: 0 -> 20
```

At T2, the governance change the fee to 10%

From this point onwards, whenever the `_updateStratPnLAndFees` function is triggered during deposit, redemption, withdrawal, and rebalance, it will revert due to underflow.

```
stratPnL = 100 (unchange)

feeToClaim = (stratPnL * 10%) - feeClaimed
feeToClaim = (100 * 10%) - 20 = 10 - 20 (Revert)
```

Recommendation: Consider updating the fee claim logic to ensure that decreasing the fee does not lead to an underflow when computing the amount of the fee already claimed.

Client:

Instead of getting the underlying, feeRecipient receives shares of the vault : `https://github.com/Tempest-Finance/tempest_smart_contract/pull/110`

Renascence: Fixed. The original fee claim logic has been removed. `feeRecipient` now receives vault's shares as fee.

[H-2] Incorrect fee accounting leads to protocol receiving lesser fees than expected

Context:

- WstEthStrategy.sol#L432
- WstEthStrategy.sol#L365
- RswEthStrategy.sol#L526
- RswEthStrategy.sol#L363

Description:

```
File: WstEthStrategy.sol
365:   function _updateStratPnLAndFees() internal {
366:     uint256 _totalAsset = totalAssets();
367:     stratPnL = (_totalAsset > invested + stratPnL) ? (_totalAsset - invested) :
stratPnL;
368:     feeToClaim = ((stratPnL * fee) / BASE) - feeClaimed;
369:   }
```

Assume the fee is 10%.

At T0, Assume that the vault only has two users (Alice and Bob), each depositing 20 assets. At this point, the `totalAssets` and `invested` are 40 assets (20 * 2). Alice and Bob own an equal number of shares in the vault.

At T1, there is some profit and the `totalAssets` increased from 40 to 100. Assume that the `_updateStratPnLAndFees` function is triggered. The `stratPnL` is computed as follows, showing that there is a profit of 60 assets. The protocol is entitled to a fee of 6 assets.

```
stratPnL = (_totalAsset > invested + stratPnL) ? (_totalAsset - invested) : stratPnL;
stratPnL = (100 > 40 + 0) ? (_totalAsset - invested) : stratPnL;
stratPnL = True ? (_totalAsset - invested) : stratPnL;
stratPnL = _totalAsset - invested
stratPnL = 100 - 40 = 60

feeToClaim = stratPnL * 10% - feeClaimed
feeToClaim = 60 * 10% - feeClaimed = 6 - 0 = 6
```

Since Alice and Bob own equal numbers of shares, each is entitled to 50 assets (20 initial deposits + 30 earnings/profit).

At T2, Alice decided to redeem all her shares, and she would receive 50 assets. The following code with the `_internalWithdraw` function will be executed to update the `invested` value.

```
File: WstEthStrategy.sol
    // Update invested amount
    invested = invested > assets ? invested - assets : 0;
```

```
invested = invested > assets ? invested - assets : 0;
invested = 40 > 50 ? invested - assets : 0;
invested = False ? invested - assets : 0;
invested = 0
```

The `invested` variable will be updated to zero after Alice has redeemed all her shares.

At this point, the state is as follows:

- `totalAssets` = 50
- `invested` = 0
- `stratPnL` = 60

Scenario 1

At T3, there is an earning/profit of 10 assets, and the `totalAssets` increases from 50 to 60. The `_updateStratPnLAndFees` is executed.

```
stratPnL = (_totalAsset > invested + stratPnL) ? (_totalAsset - invested) : stratPnL;
stratPnL = (60 > 0 + 60) ? (_totalAsset - invested) : stratPnL;
stratPnL = False ? (_totalAsset - invested) : stratPnL;
stratPnL = stratPnL
stratPnL = 60

feeToClaim = stratPnL * 10% - feeClaimed
feeToClaim = 60 * 10% - feeClaimed = 6 - 0 = 6
```

The `feeToClaim` remains at 6 assets, which is incorrect. The strategy has earned an additional 10 assets. Thus, the fee that the protocol is entitled to should be 7 (6 + 1).

Scenario 2

At T3, there is an earning/profit of 20 assets, and the `totalAssets` increases from 50 to 70. The `_updateStratPnLAndFees` is executed.

```
stratPnL = (_totalAsset > invested + stratPnL) ? (_totalAsset - invested) : stratPnL;
stratPnL = (70 > 0 + 60) ? (_totalAsset - invested) : stratPnL;
stratPnL = True ? (_totalAsset - invested) : stratPnL;
stratPnL = (_totalAsset - invested)
stratPnL = 70 - 0

feeToClaim = stratPnL * 10% - feeClaimed
feeToClaim = 70 * 10% - feeClaimed = 7 - 0 = 7
```

This is also incorrect because this time round the strategy/vault has earned 20 assets, and based on the 10% fee, the protocol should be entitled to an additional 2 assets. Thus, the final `feeToClaim` should be 8 (6 + 2)

Recommendation: Consider updating the fee accounting logic to ensure that user withdrawals do not impact the protocol's collected fees from the strategy's gains

Client:

Instead of getting the underlying, feeRecipient receives shares of the vault : `https://github.com/Tempest-Finance/tempest_smart_contract/pull/110`

Renascence: Fixed. The original fee claim logic has been removed. feeRecipient now receives vault's shares as fee.

[H-3] Existing KO position might be overwritten during deposit

Context:

- WstEthStrategy.sol#L141
- RswEthStrategy.sol#L144

Description: At T0, Alice deposits, and a new KO liquidity position called 100 is created where `tick = [a, b], pivot = T0, liquidity = 100`. This KO position is stored in `lstParams[0]`, and `lstParams[0].pivot = T0`.

At T1, this KO liquidity gets "knockout". On the Ambient side, once the KO liquidity is knocked out or crossed, the global KO pivot will be reset to zero, as shown here. Note that the pivot of individual KO liquidity position 100 created earlier still remains at T0 on Ambient storage, as this is used for tracking the position.

At T2, Bob deposit and a new KO liquidity position 101 with the same `tick = [a, b]` will be created on the Ambient side with a new pivot time because the earlier one has already crossed. Since it is a new KO liquidity position, the pivot will be set to the current time of T2, as shown here. Thus, `lstParams[0].pivot = T2` since the existing pivot (T0) is different from the current pivot (T2)

```
File: RswEthStrategy.sol
      (, uint32 pivotTime, ) = crocsQuery.queryKnockoutPivot(
        tokenAddresses[0],
        tokenAddresses[1],
        POOL_IDX,
        _isBid,
        _isBid ? lstParam.lowerTick : lstParam.upperTick
      );

      if (lstParam.pivot != pivotTime) lstParams[i].pivot = pivotTime;
```

As a result, the KO liquidity position 100 will be overwritten by KO liquidity position 101 since `lstParams[0]` can only store one KO liquidity position at any point in time. Thus, the KO liquidity position 100, which is worth 100 liquidity, will be lost.

Under normal circumstances, if a burn was performed at T2 in the earlier example, all liquidity in the crossed KO liquidity position 100 will be retrieved to the strategy, and no assets will be lost.

However, if a deposit were performed after the existing KO position had crossed, this issue would occur.

Recommendation: Consider checking if the existing KO position has already crossed during the deposit. If it has, claim the KO position before overwriting it.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/991e65c51b1979be0d6a195ea43c45f4592f5db4

Renascence: Fixed. The KO position will be recovered before the existing `lstParams.pivot` is overwritten.

[H-4] `RswEthStrategy`s total assets will be inflated

Context:

- RswEthStrategy.sol#L426
- RswEthStrategy.sol#L1121

Description:

```
File: RswEthStrategy.sol
423:       withdrawalDatas.push(
424:         WithdrawalData({
425:           requestId: tokenId + i + 1,
426:           amount: amounts[i].mulDiv(uint256(oracle.latestAnswer()), 10 **
oracle.decimals())
427:         })
428:       );
```

At T0, assume that the swell's `rswETHToETHRate` is 1.1. In this case, the `oracle.latestAnswer()` will return 1.1. Assume that the number of rswETH to withdraw is 1000. In this case, the `amount` will be set to 1100 ETH (`1000 rswETH * 1.1`)

When determining the total assets of the vault, the `getAmountInQueue` will be executed to compute the number of assets in the Swell's withdrawal queue.

```
File: RswEthStrategy.sol
1115:   /// @notice Returns the total amount in the withdrawal queue
1116:   /// @param _withdrawlDatas The array of withdrawal data
1117:   /// @return amount The total amount in the queue
1118:   function getAmountInQueue(WithdrawalData[] memory _withdrawlDatas) internal
pure returns (uint256 amount) {
1119:     uint lenWithdrawalDatas = _withdrawlDatas.length;
1120:     for (uint256 i; i < lenWithdrawalDatas; ++i) {
1121:       amount += _withdrawlDatas[i].amount;
1122:     }
1123:   }
```

The above code assumes that the amount of ETH received after the withdrawal request is completed is always equal to 1100 ETH (`1000 rswETH * 1.1`).

However, this assumption is incorrect as the following code shows that the rate used to compute the actual ETH received is the `min(processedRate, rateWhenCreated)`. If there is some mass-slashing event, the actual/final rate when processing the withdrawal request will be less than 1.1. In this case, the `processedRate` might be 1.05. Thus, the final amount of ETH received will be 1050 ETH instead of 1100 ETH.

https://etherscan.io/address/0xbd6a5ec8a78b57871ae17d22cd686a72ebe06479#code#F1#L138

https://etherscan.io/address/0xbd6a5ec8a78b57871ae17d22cd686a72ebe06479#code#F1#L275

```
File: RswEXIT.sol
250:   function finalizeWithdrawal(uint256 tokenId) external override {
..SNIP..
273:     uint256 rateWhenCreated = withdrawalRequests[tokenId].rateWhenCreated;
274:
275:     uint256 finalRate = processedRate > rateWhenCreated
276:       ? rateWhenCreated
277:       : processedRate;
278:
279:     uint256 ethClaim = wrap(withdrawalRequests[tokenId].amount)
280:       .mul(wrap(finalRate))
281:       .unwrap();
282:
283:     _burn(tokenId);
284:
285:     AddressUpgradeable.sendValue(payable(owner), ethClaim);
```

When such a scenario occurs, the `getAmountInQueue` function will return a value that is higher than expected, and the total assets of the vault will be inflated.

This could lead to significant issues, as many calculations (e.g., share calculations during deposit and redemption) depend on the accuracy of the vault's total asset value.

Recommendation: Consider fetching the market rate of the rswETH token when computing the total assets. Ensure that the market rate is obtained from oracles that cannot be manipulated (e.g., Chainlink).

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/9a0263adfdc17e275fd3768d9f4a350c08c54fa2

Renascence: Fixed. When computing the total assets, the current/process rate will be considered, and the minimal of the current/process rate and original rate will be used. If a mass-slashing event occurs, the reduced rate will be reflected.

[H-5] Strategy vaults are vulnerable to share inflation attack

Context:

- BaseAmbientStrategy.sol#L198-L214
- SymetricAmbientStrategy.sol#L214
- WstEthStrategy.sol#L141
- RswEthStrategy.sol#L144

Description: The strategy vaults are vulnerable to share inflation attack due to the use of `balanceOf()` and `.balance` to determine `totalAssets()`. This allows an attacker to steal from the vault by frontrunning the victim's `deposit()` to be the first depositor and inflating the share price via donation to increase `totalAssets()`, causing the victim's share amount to be rounded down.

```
function deposit(
  uint256 amount,
  address receiver
) external payable depositActive nonReentrant returns (uint256 shares) {
....
  // Capture totalAsset before user calls deposit
  uint256 _totalAssets = totalAssets(oracleLatestAnswer, oracleDecimals) - amount;

  // Swap and provide liquidity
  uint256 amountAfterSwap = _zapDeposit(
    ZapDepositParams({
      lpParams: _lpParams,
      assetIdx: _assetIdx,
      tokenAddresses: _tokenAddresses,
      amount: amount,
      oracleLatestAnswer: oracleLatestAnswer,
      oracleDecimals: oracleDecimals
    })
  );

  // Preview the number of shares to be minted for the deposit amount
  shares = _convertToShares(amountAfterSwap, _totalAssets);
  ...
  }

function totalAssets() public view returns (uint256) {
  return _underlyingBalance(assetIdx == 0, uint256(IOracle(oracle).latestAnswer()),
  IOracle(oracle).decimals());
}

function _underlyingBalance(
  bool _isToken0,
  uint256 oracleLatestAnswer,
  uint8 oracleDecimals
) internal view returns (uint256) {
  uint8 token1Decimals = IERC20Metadata(tokenAddresses[1]).decimals();
  uint8 token0Decimals = tokenAddresses[0] == address(0) ? 18 :
  IERC20Metadata(tokenAddresses[0]).decimals();

  uint256[2] memory tokenBal = _getBalances();
  (uint256 token0Bal, uint256 token1Bal) = (tokenBal[0], tokenBal[1]);
```

```
    uint256 token1InToken0 = (oracleLatestAnswer * 10 ** token0Decimals) / (10 **
    oracleDecimals);
    (, uint128 amount0, uint128 amount1) = getAllPositions(lpParams, tokenAddresses);

    uint256 curValBal = _isToken0 ? token1Bal : token0Bal;
    uint256 amount = _isToken0
      ? token1InToken0.mulDiv((uint256(amount1) + curValBal), 10 ** token1Decimals)
      : (uint256(amount0) + curValBal).mulDiv(10 ** token1Decimals, token1InToken0);
    uint256 curUnderlyingBal = _isToken0 ? token0Bal : token1Bal;
    return _isToken0 ? uint256(amount0) + amount + curUnderlyingBal : uint256(amount1)
    + amount + curUnderlyingBal;
  }

  function _getBalances() internal view returns (uint256[2] memory) {
    return [
      tokenAddresses[0] == address(0)
        ? address(this).balance
        : IERC20Metadata(tokenAddresses[0]).balanceOf(address(this)),
      IERC20Metadata(tokenAddresses[1]).balanceOf(address(this))
    ];
  }
```

Recommendation: Prevent first depositor attack that inflate share price by minting dead shares on deployment.

Client:

BaseAmbientStrategy

A `deposit()` will be performed during deployment of vault to ensure that the protocol is the 1st depositor. Along with that, `minDeposit` and `minShare` checks are in place. https://github.com/Tempest-Finance/tempest_smart_contract/pull/124 https://github.com/Tempest-Finance/tempest_smart_contract/pull/125

WstEthStrategy

We added minShare check here https://github.com/Tempest-Finance/tempest_smart_contract/pull/124/files, and when deploying the vault, we will broadcast the tx with a deposit() so we will be the 1st depositor.

After internal discussion, we - Tempest team aligned that we've had minDeposit already. So here is the case of inflation attack:

- Bob (attacker) deposits minDeposit as 1st depositor. Bob's shares = minDeposit, totalSupply= minDeposit, totalAssets = minDeposit. Bob donates donation => totalAssets = minDeposit + donation
- Alice (victim) deposits minDeposit.
- Condition for vault to be inflated: Alice's deposit * totalSupply / totalAssets < 1 => minDeposit * minDeposit / (minDeposit + donation) < 1 => donation > (minDeposit - 1) * minDeposit That means that the attacker have to donate (minDeposit - 1) * minDeposit to gain minDeposit from the victim. It's safe enough cuz when deploying the vault, we will broadcast the tx with a deposit() so we will be the 1st depositor.

We also checked and reverted tx if share = 0 in this PR https://github.com/Tempest-Finance/tempest_smart_contract/pull/125

Final fix with `DEAD_SHARES` to fix inflation attack https://github.com/Tempest-Finance/tempest_smart_contract/pull/139

Renascence: Resolved by minting dead shares to vault on first deposit by protocol,, along with additional safety mechanisms.

[H-6] `WstEthStrategy`s total assets will be inflated

Context:

- WstEthStrategy.sol#L516

Description:

```
File: WstEthStrategy.sol
514:     uint256[] memory requestIds = withdrawalQueue.requestWithdrawals(amounts,
address(this));
515:     for (uint256 i = 0; i < actualNumberOfSplit; ++i) {
516:       withdrawalDatas.push(WithdrawalData({ requestId: requestIds[i], amount:
amounts[i] }));
517:       emit RequestWithdrawal(amounts[i], requestIds[i]);
518:     }
```

The `_withdrawlDatas[i].amount` stores the number of stETH to be withdrawn, and that is currently in the withdrawal queue.

The code assumes that 1 stETH will always equal 1 ETH. While this is generally the case, however, in certain conditions (e.g., mass slashing and penalties), the final amount of claimable ETH can differ.

Reference: https://stake.lido.fi/withdrawals/request. The following screenshot is extracted from LIDO website:

Assuming a mass slashing event occurs, 1 stETH will be worth 0.8 ETH. However, the protocol still prices 1 stETH at 1 ETH, thus overinflating the total assets in the vault.

Recommendation: Consider using the Chainlink's stETH/ETH price feed.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/9a0263adfdc17e275fd3768d9f4a350c08c54fa2

Renascence: Fixed. The Chainlink's stETH/ETH price will be used when computing the total assets instead of assuming that 1 stETH is always worth 1 ETH.

[H-7] Incorrect pivot time accounting can block asset withdrawals

Context: WstEthStrategy.sol#L892-L899 RswEthStrategy.sol#L902-L909 Knockout-Counter.sol#L130 KnockoutCounter.sol#L355

Description: When a KO position is fully burned in Ambient Finance, it's pivot time is set to 0:

```
function recallPivot (bytes32 pool, KnockoutLiq.KnockoutPosLoc memory loc,
                      uint96 lots) private returns
    (uint32 pivotTime, bool killsPivot) {
    bytes32 lvlKey = KnockoutLiq.encodePivotKey(pool, loc.isBid_,
                                                 loc.knockoutTick());
    KnockoutLiq.KnockoutPivot storage pivot = knockoutPivots_[lvlKey];
    pivotTime = pivot.pivotTime_;
    require(lots <= pivot.lots_, "KB");
    killsPivot = (lots == pivot.lots_);

    if (killsPivot) {
        // Get the SSTORE refund when completely burning the level
        pivot.lots_ = 0;
        pivot.pivotTime_ = 0;
        pivot.rangeTicks_ = 0;
```

The Tempest vault is unaware of this change because it queries the pivot status before liquidity is burned from the position:

```
  function _burnLiquidity(
    bool knockedOut,
    bool isBid,
    uint256 indexLst,
    LstParam memory _lstParam,
    CrocsQuery _crocsQuery,
    uint128 quantity,
    bytes memory _merkleProofs
  ) internal returns (uint256 amount0, uint256 amount1) {
    bytes memory cmd;
    ---SNIP---
»  (, uint32 pivotTime, ) = _crocsQuery.queryKnockoutPivot(
      tokenAddresses[0],
      tokenAddresses[1],
      POOL_IDX,
      isBid,
      isBid ? _lstParam.lowerTick : _lstParam.upperTick
    );
    if (_lstParam.pivot != pivotTime) lstParams[indexLst].pivot = pivotTime;

    // Send the command to the crocsSwapDex contract
    // @audit Burn cmd
    crocsSwapDex.userCmd(7, cmd);
  }
```

As a result, the next time `_burnLiquidity` is called, it will attempt to burn a non-existent position, causing the entire withdraw/redeem operation to revert.

Recommendation: Consider querying pivot time after burn `cmd` is executed.

Client: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/126/files`

Renascence: The issue has been resolved as per recommendation.

Medium Risk

[M-01] Fee can be applied retrospectively to a time period that has already passed

Context:

- WstEthStrategy.sol#L368
- RswEthStrategy.sol#L366

Description: At T0, assume the following state:

- totalAsset = 200
- invested = 100
- stratPnL = 0
- feeToClaim = 0
- feeClaimed = 0
- fee = 20%

When the `_updateStratPnLAndFees` function is executed:

```
stratPnL = 200 - 100 = 100

feeToClaim = (stratPnL * 20%) - feeClaimed
feeToClaim = (100 * 20%) - 0 = 20
```

At T1, the `_claimFees` function is executed:

```
feeToClaim: 20 -> 0
feeClaimed: 0 -> 20
```

At T2, the governance change the fee to 50%

The `_updateStratPnLAndFees` function is triggered when an action is executed, leading to the following state changes, which indicates that an additional 30 can be claimed.

```
stratPnL = 100 (unchange)

feeToClaim = (stratPnL * 50%) - feeClaimed
feeToClaim = (100 * 50%) - 20 = 50 - 20 = 30
```

Next, the `_claimFees` function is triggered:

```
feeToClaim: 30 -> 0
feeClaimed: 20 -> 50
```

From T0 to (T2 - 1 second), the fee is 20%. Thus, the fee that can be claimed is 20% of the PnL (100), which is 20.

From T2 onwards, the fee is 50%. This means that from T2 onwards, a fee of 50% will be charged against the PnL.

However, over here, the governance can apply the increased fee (20% → 50%) retrospectively to the time period (T0 to (T2 - 1 second)) that has already passed, which is incorrect.

Recommendation: The updated fee should only apply from the moment it is updated and should not be applied retroactively to any previous time period.

Client:

Instead of getting the underlying, feeRecipient receives shares of the vault taking into account the fees set at the moment it's computed : `https://github.com/Tempest-Finance/tempest_smart_contract/pull/110`

Renascence: Fixed. The original fee claim logic has been removed. feeRecipient now receives vault's shares as fee.

[M-02] Malicious users can force the strategy to invest all its assets bypassing the `investedPercentage` restriction

Context:

- WstEthStrategy.sol#L609
- RswEthStrategy.sol#L619

Description: Assume a scenario where providing liquidity in the current market condition might result in a negative expected value/return. In this case, the `investedPercentage` might be set at 50% to reduce the exposure to the external protocol.

Assuming that the `investedPercentage` is 50%.

Alice deposits 1000. Since the `investedPercentage` is 50%, 500 will remain in the vault while the remaining 500 will be deposited to external protocol (providing liquidity).

```
vault = 500, extProtocol = 500
```

Subsequently, a malicious user called Bob deposits the smallest amount of assets accepted by the vault (e.g., 1 wei) to trigger the execution of `_investBalance` function. The `assetBalance` below represents the current balance of the assets on the vault, and the value will be 500. 250 will be deposited to external protocol.

```
investedAmount = assetBalance.mulDiv(investedPercentage, BASE);
investedAmount = 500 * 50% = 250

vault = 250, extProtocol = 500 + 250 = 750
```

Bob could repeat the transaction, and the state changed to as follows:

```
investedAmount = 250 * 50% = 125

vault = 125, extProtocol = 500 + 250 + 125 = 875
```

Bob can repeat the same transaction until the number of assets remaining in the vault is close to zero, which means the vault has close to 100% exposure to the external protocol even though the current market condition might result in a negative expected value/return.

Note: The attack cost is minimal as the gas fee on L2 is very low.

Recommendation: Consider implementing measures to ensure that the strategy does not invest additional assets to the external protocol once the `investedPercentage` threshold is reached.

Client:

added small amount check : Tempest-Finance/tempest_smart_contract#97

Tempest-Finance/tempest_smart_contract#116

1. Instead of using `balance * investedPercentage`, we used the `input amount * investedPercentage`
2. Everytime a `InvestedPercentageSet` happened, we also do a rebalance to respect the settings

This applies also to the RswETHStrategy

Renascence: Fixed. The `investedAmount` is based on the user's deposit amount instead of the contract's balance in the new implementation.

[M-03] Division by zero error if `totalUnderlying <= feeToClaim`

Context:

- WstEthStrategy.sol#L1061
- WstEthStrategy.sol#L1027
- RswEthStrategy.sol#L1071
- RswEthStrategy.sol#L1037

Description:

```
File: WstEthStrategy.sol
1042:   function _underlyingBalance(bool _isToken0) internal view returns (uint256
totalUnderlying) {
..SNIP..
1057:     totalUnderlying = _isToken0
1058:       ? uint256(amount0) + amount + curUnderlyingBal
1059:       : uint256(amount1) + amount + curUnderlyingBal;
1060:     totalUnderlying += getAmountInQueue(withdrawalDatas);
1061:     totalUnderlying = totalUnderlying > feeToClaim ? totalUnderlying -
feeToClaim : 0;
1062:   }
```

If `totalUnderlying <= feeToClaim`, then `totalUnderlying` will be zero. This also means that `totalAssets()` will be zero.

In this case, the `_convertToShares` function will always revert due to division by zero error in Line 1027 below. Any feature that internally call `_convertToShares` will be broken.

```
File: WstEthStrategy.sol
1151:   /// @notice Returns the total assets held by the contract
1152:   /// @return The total assets
1153:   function totalAssets() public view returns (uint256) {
1154:     return _underlyingBalance(assetIdx == 0);
1155:   }
..SNIP..
1019:   /// @notice Converts an amount to shares
1020:   /// @param amount The amount to convert
1021:   /// @return The number of shares
1022:   function _convertToShares(uint256 amount) internal view returns (uint256) {
1023:     uint256 supply = totalSupply();
1024:     if (supply == 0) return amount;
1025:     uint256 _totalAsset = totalAssets();
1026:     if (tokenAddresses[assetIdx] == address(0) && isDepositing) return
amount.mulDiv(supply, _totalAsset - amount);
1027:     return amount.mulDiv(supply, _totalAsset);
1028:   }
```

Recommendation: Consider implementing additional logic to ensure that the `feeToClaim` does not lead to a division by zero error.

Client:

Instead of getting the underlying, feeRecipient receives shares of the vault taking into account the fees set at the moment it's computed : `https://github.com/Tempest-Finance/tempest_smart_contract/pull/110`

Renascence: Renascence: Fixed. The original fee claim logic has been removed. `feeRecipient` now receives vault's shares as fee.

[M-04] `setFees()` fails to collect pending fees before updating `fee` percentage

Context:

- BaseAmbientStrategy.sol#L1302-L1308
- SymetricAmbientStrategy.sol#L1401-L1407

Description: Both `BaseAmbientStrategy` and `SymetricAmbientStrategy` will collect protocol fee based on a percentage of the earned LP fee. The protocol `fee` percentage can be changed by governance via `setFees()`.

However, when `setFees()` is called, fails to collect pending fee based on previous `fee` percentage. This issue will cause the protocol to incorrectly apply the new `fee` percentage to fees that were not yet collected (before the new fee percentage takes effect).

```
function setFees(uint16 _fee) external onlyRole(GOVERNANCE_ROLE) {
  if (_fee > BASE) {
    revert BadSetup();
  }
  fee = _fee;
  emit FeesSet(_fee);
}
```

Recommendation: Collect pending fees based on the existing fee percentage before updating it.

```
  function setFees(uint16 _fee) external onlyRole(GOVERNANCE_ROLE) {
    if (_fee > BASE) {
      revert BadSetup();
    }
+    _collectAllFees(lpParams, tokenAddresses, feeRecipient, fee);
    fee = _fee;
    emit FeesSet(_fee);
  }
```

Client: Fixed at https://github.com/Tempest-Finance/tempest_smart_contract/commit/fd9e385e6f9975a595ab9c2d74125d0243ffa09e

Renascence: The issue has been resolved as per recommendation.

[M-05] Incorrect rounding direction for `previewWithdraw()`

Context:

- BaseAmbientStrategy.sol#L1150-L1152
- SymetricAmbientStrategy.sol#L1247
- RswEthStrategy.sol#L1137
- WstEthStrategy.sol#L1127

Description: For a secure implementation of a ERC4626 vault, it should always perform rounding in the direction that favors the vault.

However, `previewWithdraw(uint256 assets)` incorrectly rounds down the required `shares` to withdraw `assets`, which favors the user instead. The correct implementation for `previewWithdraw()` is to round up so that it favors the vault to prevent any shortfall in fulfilling withdrawals.

```
function previewWithdraw(uint256 assets) public view returns (uint256) {
  return _convertToShares(assets, totalAssets());
}

function _convertToShares(uint256 amount, uint256 _totalAssets) internal view returns
(uint256) {
  uint256 supply = totalSupply();
  if (supply == 0) return amount;
  return amount.mulDiv(supply, _totalAssets);
}
```

Recommendation: Ensure that `previewWithdraw()` rounds up to favor the vault.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/95

Renascence: The issue has been resolved as per recommendation.

[M-06] `BaseAmbientStrategy` is incorrectly initialized with zero `liqSlippage`

Context:

- BaseAmbientStrategy.sol#L158

Description: In the constructor of `BaseAmbientStrategy`, only `swapSlippage` is set and not `liqSlippage`, even though both are part of `StrategyParameters`.

That means `liqSlippage` will be initialized to a zero value. That will cause functions that provides liquidity to always revert as the slippage check enforces a zero slippage for liquidity provisioning.

```
constructor(
  int24[] memory _upperTicks,
  int24[] memory _lowerTicks,
  StrategyParameters memory sParams
) ERC20(sParams.name, sParams.symbol) {
  ...
  swapSlippage = sParams.swapSlippage;
  //@audit missing initialization of liqSlippage
  ...
}
```

Recommendation:

```
  constructor(
    int24[] memory _upperTicks,
    int24[] memory _lowerTicks,
    StrategyParameters memory sParams
  ) ERC20(sParams.name, sParams.symbol) {
    ...
    swapSlippage = sParams.swapSlippage;
+   liqSlippage = sParams.liqSlippage;
    ...
  }
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/101

Renascence: Resolved as per recommendation.

[M-07] Lack of liquidity slippage check in `zapDeposit()`

Context:

- BaseAmbientStrategy.sol#L778-L787
- SymetricAmbientStrategy.sol#L877-L886

Description: `_zapDeposit()` will directly provide liquidity for the first tick range with the user's deposited asset. However, it is performed with `checkSlippage = false`. That could possibly allow someone to sandwich the liquidity provisioning and manipulate the current tick, to cause an incorrect amount of tokens to be paid for minting the liquidity.

```
function _zapDeposit(ZapDepositParams memory params) internal returns (uint256) {
  // Swap tokens and get the amounts received and spent
  (uint256 received, uint256 spent, uint256 amountAfterSwap) = _swapTokensForLP(
    params.lpParams[0].upperTick,
    params.lpParams[0].lowerTick,
    params.assetIdx,
    params.oracleLatestAnswer,
    params.oracleDecimals,
    params.amount
  );

  ...

  if (liq != 0) {
    // Provide liquidity based on the calculated liquidity and current ticks
    _provideLiquidity(
      ProvideLiqParams({
        upperTick: params.lpParams[0].upperTick,
        lowerTick: params.lpParams[0].lowerTick,
        liq: liq,
        ethVal: params.tokenAddresses[0] == address(0) ? (assetIdx == 0 ?
        params.amount - spent : received) : 0,
        checkSlippage: false,
        sqrtOraclePrice: 0
      })
    );
  }

  return amountAfterSwap;
}
```

Recommendation: Perform slippage check for minting of liquidity during `zapDeposit()`.

Client: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/113`

Renascence: The issue has been resolved as per recommendation.

[M-08] Liquidity might be provided to the incorrect KO positions

Context:

- WstEthStrategy.sol#L637
- RswEthStrategy.sol#L647

Description: Per Line 685 below, liquidity will only be provided to KO positions that meet the `((isBid && _currentTick > _lstParams[i].upperTick) || (!isBid && _currentTick < _lstParams[i].lowerTick))` condition.

```
File: RswEthStrategy.sol
675:   function _rebaseWeights(
676:     LstParam[] memory _lstParams,
677:     uint16[] memory _lpWeights,
678:     int24 _currentTick,
679:     bool isBid
680:   ) internal pure returns (uint256 normalizedLiquidity) {
681:     uint256 lenLstWeight = _lstParams.length;
682:     normalizedLiquidity = 0;
683:     // Adjust the normalized liquidity based on the current tick and bid status
684:     for (uint256 i; i < lenLstWeight; ++i) {
685:       if ((isBid && _currentTick > _lstParams[i].upperTick) || (!isBid &&
_currentTick < _lstParams[i].lowerTick))
686:         normalizedLiquidity += _lpWeights[i];
687:     }
688:   }
```

When providing liquidity, Line 647 will skip KO positions that do not meet the conditions mentioned earlier.

However, it was observed that the condition `((!(_isBid && _currentTick > lstParam.upperTick) || (!_isBid && _currentTick < lstParam.lowerTick)))` at Line 647 is not the negate condition of "((isBid && _currentTick > _lstParams[i].upperTick) || (!isBid && _currentTick < _lstParams[i].lowerTick)) .

As a result, liquidity may be added to KO positions when it should not be or withheld when it should be provided.

```
File: RswEthStrategy.sol
  function _provideLiquidity(
    uint16[] memory _lpWeights,
    LstParam[] memory _lstParams,
    uint256 normalizedLiquidity,
    uint256 investedAmount,
    int24 _currentTick,
    bool _isBid
  ) internal {
    uint256 lenLstWeight = _lstParams.length;
    for (uint256 i; i < lenLstWeight; ++i) {
      LstParam memory lstParam = _lstParams[i];
      if (!(_isBid && _currentTick > lstParam.upperTick) || (!_isBid && _currentTick < lstParam.lowerTick)) continue;

      _provideKOLiquidity(
        _isBid,
        tokenAddresses,
        lstParam.upperTick,
        lstParam.lowerTick,
        _uint128Safe((investedAmount * _lpWeights[i]) / normalizedLiquidity)
      );
```

Recommendation: Consider making the following changes:

```diff
- if (!(_isBid && _currentTick > lstParam.upperTick) || (!_isBid && _currentTick <
lstParam.lowerTick)) continue;
+ if (!((_isBid && _currentTick > lstParam.upperTick) || (!_isBid && _currentTick <
lstParam.lowerTick))) continue;
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/102

Renascence: Fixed as per recommendation.

[M-09] There is no staleness check for the Chainlink price feed answer

Context: UsdcETHOracle.Sol#L13

Description: The `UsdcETHOracle.Sol` contract uses the deprecated `latestAnswer` function to fetch token prices from the Chainlink price feed:

https://docs.chain.link/data-feeds/api-reference#latestanswer

Additionally, the oracle doesn't check if the retrieved price is up-to-date:

```
function latestAnswer() external view returns (int256) {
  return (IOracle(UsdcUsdOracle).latestAnswer() * int256(10 ** decimals)) /
  IOracle(EthUsdOracle).latestAnswer();
}
```

Since `UsdcETHOracle.Sol` plays a critical role in both `BaseAmbientStrategy.sol` and `SymetricAmbientStrategy.sol`, where it used to estimate the total value locked in the vault, relying on stale prices could lead to financial losses for the protocol or its users.

Recommendation: It is advised to use the `latestRoundData` function, along with a time limit check, to ensure the price returned from the feed is current:

```
(, int256 price, , uint256 updateAt, ) = IOracle(Oracle).latestRoundData();
require(price > 0 && updateAt + timelimit > block.timestamp, 'Price Feed: invalid
price');
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/6ea5aa14dc269642654260c7b95f42f7bc8f1e34

Renascence: The issue has been resolved as per recommendation.

[M-10] Vaults basic functions can be temporarily blocked

Context: BaseAmbientStrategy.sol#L175 SymetricAmbientStrategy.sol#L214 RswEthStrategy.sol#L144 WstEthStrategy.sol#L141 PoolRegistry.sol#L307

Description: Ambient Finance implements JIT (Just-In-Time) threshold security checks, which prevent users from interacting with a particular position for a set period after minting liquidity. For instance, in the ETH/USDC pool, `jitThresh = 3`, meaning users can mint or burn liquidity in a position after 30 seconds (3 * 10 seconds):

```
    function assertJitSafe (uint32 posTime, bytes32 poolIdx) internal view {
        uint32 JIT_UNIT_SECONDS = 10;
        uint32 elapsedSecs = SafeCast.timeUint32() - posTime;
        uint32 elapsedUnits = elapsedSecs / JIT_UNIT_SECONDS;
»       if (elapsedUnits <= type(uint8).max) {
            require(elapsedUnits >= pools_[poolIdx].jitThresh_, "J");
        }
    }
```

However, since the vault owns the position and mints liquidity within a specific range (`lpParams[0].upperTick, lpParams[0].lowerTick`), a malicious depositor could disrupt basic vault functionality by making a minimal deposit, effectively blocking the vault for 30 seconds. This attack could be extended indefinitely with additional small deposits and is particularly effective on blockchains with low gas fees.

Recommendation: To prevent such exploits, consider implementing a minimum deposit amount requirement to discourage malicious actors from taking advantage of this vulnerability.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/8d574083a1a18926f0e71c66d4f0f1205ed9f6f6#diff-9dc0165546fe17ee4192e2ade7323f1867f22c4ab3616d7d93dc75

Renascence: The issue has been resolved as per recommendation.

[M-11] An incorrect swap amount calculation leads to the underutilization of the users provided liquidity

Context: BaseAmbientStrategy.sol#L545-L575 SymetricAmbientStrategy.sol#L575-L605

Description: When a user deposits assets into the vault, the protocol splits the deposited amount between token0 and token1 by swapping a portion of the deposited assets into the other token in the pool:

```
    function _swapTokensForLP(
      int24 _upperTick,
      int24 _lowerTick,
      uint8 _assetIdx,
      uint256 oracleLatestAnswer,
      uint8 oracleDecimals,
      uint256 amount
    ) internal returns (uint256 received, uint256 spent, uint256 amountAfterSwap) {
      // Swap tokens and get the amounts received and spent
      (received, spent) = _swapTokens(
        SwapParams({
          tokenIn: tokenAddresses[_assetIdx],
          tokenOut: tokenAddresses[1 - _assetIdx],
»        amountIn: _calcAmountToSwap(_assetIdx, currentTick(), _upperTick, _lowerTick,
amount, tokenAddresses)
        })
      );
```

Afterwards, liquidity is minted with the following amounts (ignoring padding for simplicity):

- `asset = depositedAmount - swappedAmount`
- `non-asset = amountReceivedFromSwap`

Unfortunately, there is an error in the swap amount calculation, which results in non-optimal tokens amounts, and not all liquidity being utilized by the pool. For example:

- A user deposits 1 ETH into the vault
- The amount of ETH to swap is calculated as 0.343 ETH
- The resulting balances before position minting will are 0.656 ETH and 1049.78 USDC
- After minting the position, the vault is left with a surplus of 0.47 ETH, meaning this excess ETH is not utilized to collect yield in the pool

Recommendation: Consider adjusting the swap amount formula in `_calcAmountToSwap` for an optimal one sided supply.

Client: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/93/files`

Renascence: Fixed. The _calcAmountToSwap algorithm has been refactored to optimize liquidity provision to the Ambient pools.

[M-12] Insufficient liquidity may be burned during redemption.

Context: WstEthStrategy.sol#L739 RswEthStrategy.sol#L749

Description: In the `_burnForRedeem` function, padding is added, multiplied by the number of ranges, to `amountAssetRetrieved`:

```
  function _burnForRedeem(
    uint256 _amounttoBurn,
    LstParam[] memory _lstParams,
    address[2] memory _tokenAddresses,
    uint8 _assetIdx,
    bytes memory _merkleProofs
  ) internal {
    uint256 lstParamsLength = _lstParams.length;
    uint256 amountAssetRetrieved;
    CrocsQuery _crocsQuery = crocsQuery;

    for (uint i; i < lstParamsLength; ++i) {
»    if (amountAssetRetrieved + padding * lstParamsLength >= _amounttoBurn) break;
```

However, not all ranges may be activated during the redemption process. As a result, the retrieved amount could be slightly inflated, leading to a revert in `_internalWithdraw` due to an insufficient amount of retrieved tokens. Consider the following scenario:

- A user has 100 shares, which correspond to `2000000000000000011` tokens.
- There are three positions: Pos1 and Pos2 each have `1999999999999999996` tokens, while Pos3 is empty.
- Liquidity is burned from the positions sequentially. From Pos1, `1999999999999999995` tokens are retrieved.
- Since `1999999999999999995 + 5*3 = 2000000000000000011`, Pos2 is not touched.
- `_redeemAssets` will return `min(1999999999999999995, 2000000000000000011)`.
- Finally, in `_internalWithdraw`, the transaction reverts with an "insufficient liquidity" error because `1999999999999999995 + 15 < 2000000000000000011`.

```
if (assetsAfterRounding + padding * _lstParams.length < assets) revert
NotEnoughLiquidFund();
```

Recommendation: Consider adding padding only if the position was actually touched, ignoring empty or unused positions:

```
+  uint256 num;
    for (uint i; i < lstParamsLength; ++i) {
+    if (amountAssetRetrieved + padding * num >= _amounttoBurn) break;
     if (_lstParams[i].pivot == 0) continue;
+    ++num;
      (, uint128 curAmount0, uint128 curAmount1, bool knockedOut) = _getPosition(
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/134

Renascence: Resolved. The protocol has abandoned calculating padding based on the number of positions because `_getBurnAmounts()` returns the token amounts without padding.

Low Risk

[L-1] Residual allowance granted to external protocol

Context:

- WstEthStrategy.sol#L477

Description: The actual amount of stETH pulls from LIDO is not always equal to amountUnwrappedLST.

If (numberOfSplit > actualNumberOfSplit, a portion of the amountUnwrappedLST will be left for the next rebalance. As a result, it will result in a residual allowance granted to LIDO (external protocol). If LIDO is compromised, assets within Tempest's strategy contracts could be transferred out to the malicious actor's wallet.

Recommendation: Consider only granting _totalWithdrawAmount amount of allowance to LIDO.

```
 function _claimFromLido(uint256 amountUnwrappedLST) internal {
-   IERC20(unWrappedToken).safeIncreaseAllowance(address(withdrawalQueue),
amountUnwrappedLST);

    uint256 maxWithdrawAmount = withdrawalQueue.MAX_STETH_WITHDRAWAL_AMOUNT();
    uint256 minWithdrawAmount = withdrawalQueue.MIN_STETH_WITHDRAWAL_AMOUNT();

    ..SNIP..
    uint256[] memory amounts = new uint256[](actualNumberOfSplit);
    uint256 _totalWithdrawAmount = 0;
    for (uint256 i = 0; i < numberOfSplit; ++i) {
        ..SNIP..
    }

+   IERC20(unWrappedToken).safeIncreaseAllowance(address(withdrawalQueue),
_totalWithdrawAmount);
    uint256[] memory requestIds = withdrawalQueue.requestWithdrawals(amounts,
    address(this));
    for (uint256 i = 0; i < actualNumberOfSplit; ++i) {
      withdrawalDatas.push(WithdrawalData({ requestId: requestIds[i], amount:
      amounts[i] }));
      emit RequestWithdrawal(amounts[i], requestIds[i]);
    }
  }
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/106

Renascence: Fixed as per recommendation.

[L-2] Lack of L2 Sequencer Uptime Check

Context:

- UsdcETHOracle.Sol#L13

Description: The in-scope chains will be `Ethereum`, `Scroll`, `Blast`, `Canto`.

Assuming that Chainlink price feed is used here. It was found that the oracle does not rely on the L2 Sequencer Uptime Feed. Thus, outdated prices might be consumed by the protocol instead of reverting in the event the sequencer is down.

Recommendation: Consider checking the sequencer's status within the oracle.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/5d4e98ed729de081b5a00f937d45bd7dbb4d0b4c

Renascence: Fixed as per recommendation.

[L-3] Overlapping of tick ranges can cause double counting of liquidity

Context: Description:

Both AmbientStrategies can be initialized with multiple tick ranges, to allow provision of liquidity with as much asset tokens as possible.

However, if the vaults are incorrect intialized with overlapping tick ranges, it would cause `getAllPosition()` to double-count the provided liquidity amount, causing the `totalAssets()` to be incorrect.

```
function getAllPositions(
  LpParam[] memory _lpParams,
  address[2] memory _tokenAddresses
) private view returns (uint128 liquidity, uint128 amount0, uint128 amount1) {
  // Get the length of the LP parameters array
  uint256 lpParamsLength = _lpParams.length;

  // Iterate over all LP parameters
  for (uint i; i < lpParamsLength; ++i) {
    uint128 curAmount0;
    uint128 curAmount1;
    uint128 curLiquidity;

    // Get the position for the current tick ranges and token addresses
    (curLiquidity, curAmount0, curAmount1) = _getPosition(
      _lpParams[i].upperTick,
      _lpParams[i].lowerTick,
      _tokenAddresses
    );

    // Sum up the liquidity, token0 amount, and token1 amount
    liquidity += curLiquidity;
    amount0 += curAmount0;
    amount1 += curAmount1;
  }
}
```

Recommendation: Perform a validation check in `checkLiqParams()` to prevent overlapping tick ranges.

Client: Fixed at https://github.com/Tempest-Finance/tempest_smart_contract/pull/96/files

Renascence: The issue has been resolved as per recommendation.

[L-4] UsdcETHOracle has hardcoded Chainlink price feeds addresses

Context: UsdcETHOracle.Sol#L8-L9

Description: The `UsdcETHOracle.sol` contract is used to fetch the ETH/USDC price via Chainlink price feeds. The protocol currently uses hardcoded addresses for the feeds on the Scroll mainnet:

```
contract UsdcETHOracle is IOracle {
  address public constant EthUsdOracle = 0x6bF14CB0A831078629D993FDeBcB182b21A8774C;
  address public constant UsdcUsdOracle = 0x43d12Fb3AfCAd5347fA764EeAB105478337b7200;
  uint8 public constant decimals = 18;
```

However, the team has stated that the protocol will be deployed across multiple chains, where these addresses may vary.

Recommendation: It is recommended to specify the price feed addresses in the oracle's constructor rather than hardcoding them.

Client: https://github.com/Tempest-Finance/tempest_smart_contract/commit/6ea5aa14dc269642654260c7b95f42f7bc8f1e34

Renascence: Fixed. The Oracle contract has been refactored to allow customization of price feed addresses and parameters during deployment.

[L-5] Assets will be lost if a deposit is equal to or less than `padding`

Context:

- BaseAmbientStrategy.sol#L175
- SymetricAmbientStrategy.sol#L214
- WstEthStrategy.sol#L141
- RswEthStrategy.sol#L144

Description: If a user deposits a number of assets that is equal to or less than `padding`, they will lose all assets due to rounding errors on the Ambient side.

However, the existing codebase does not explicitly prevent users from depositing `padding` amounts of assets into a strategy.

Recommendation: Consider implementing a minimum deposit feature that will revert if users deposit an amount that is equal to or less than `padding`

Client:

Introduction of minDeposit : https://github.com/Tempest-Finance/tempest_smart_contract/pull/97/files#diff-9dc0165546fe17ee4192e2ade7323f1867f22c4ab3616d7d93dc757d8fd14818
check that minDeposit>padding : https://github.com/Tempest-Finance/tempest_smart_contract/pull/138/files

Renascence: Fixed. The `minimumDeposit` has to be larger than the `padding`, and the user's deposit must not be smaller than the `minimumDeposit`. Thus, this means that the user's deposit will always be larger than `padding`.

```
amount >= minimumDeposit > padding
```

Informational

[I-1] Explicit revert instead of if `amountUnwrappedLST` is less than `minWithdrawAmount`

Context:

- WstEthStrategy.sol#L480

Description: Explicitly revert if `amountUnwrappedLST` is less than `minWithdrawAmount` here instead of continuing with the execution and waiting for a revert to occur within the for-loop.

```
+ require(amountUnwrappedLST >= minWithdrawAmount, ”Does not meet the minimum
amount”);
```

The internal `_claimFromLido` is called by two functions (`WstEthStrategy.claimFromLido` and `WstEthStrategy._unwrapAndClaimFromLido`).

The `WstEthStrategy._unwrapAndClaimFromLido` will check if the amount is less than `MIN_STETH_-WITHDRAWAL_AMOUNT`, and return if it is not.

However, the `WstEthStrategy.claimFromLido` function does not perform any validation check against the amount.

Recommendation: As per the above description

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/108

Renascence: Fixed. A minimum withdrawal amount check is implemented within the `WstEthStrat-egy.claimFromLido` function.

[I-2] Unused Code

Context: Refer to the description section.

Description: Instance 1

The `amount0` and `amount1` return values are not initialized within the `_burnLiquidity` function. Thus, they will always return a value of zero.

- WstEthStrategy.sol#L854
- RswEthStrategy.sol#L864

```
function _burnLiquidity(
  bool knockedOut,
  bool isBid,
  uint256 indexLst,
  LstParam memory _lstParam,
  CrocsQuery _crocsQuery,
  uint128 quantity,
  bytes memory _merkleProofs
) internal returns (uint256 amount0, uint256 amount1) {
```

Instance 2

The `_uint32Safe` function is not used anywhere in the codebase. Consider removing it if it is not required.

- WstEthStrategy.sol#L716
- RswEthStrategy.sol#L726

Instance 3

The following functions within the library are not used anywhere in the code.

- `LiquidityAmounts.getAmountsForLiquidity`
- `LiquidityAmounts.getAmount0ForLiquidity`
- `LiquidityAmounts.getAmount1ForLiquidity`
- `LiquidityAmountsNative.getAmountsForLiquidity`
- `LiquidityAmountsNative.getAmount0ForLiquidity`
- `LiquidityAmountsNative.getAmount1ForLiquidity`

Recommendation: Consider removing the unused code if they are not necessary.

Client:

Instance 1: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/107`

Instance 2: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/103`

Instance 3: `https://github.com/Tempest-Finance/tempest_smart_contract/pull/120`

Renascence: Instance 1: Fixed in `https://github.com/Tempest-Finance/tempest_smart_contract/pull/107`. Unnecessary return is removed. Instance 2: Fixed in `https://github.com/Tempest-Finance/tempest_smart_contract/pull/103`. Unused `_uint32Safe` has been removed. Instance 3: Fixed. `LiquidityAmounts.getAmountsForLiquidity`, `LiquidityAmounts.getAmount0ForLiquidity`, `LiquidityAmounts.getAmount1ForLiquidity` have been removed in `https://github.com/Tempest-Finance/tempest_smart_contract/pull/120`. `LiquidityAmountsNative.getAmountsForLiquidity`, `LiquidityAmountsNative.getAmount0ForLiquidity`, `LiquidityAmountsNative.getAmount1ForLiquidity` have not been removed. However, per the protocol team's response, it was noted that these functions will be required for upcoming dual-sided deposits. Thus, these functions should not be removed.

[I-3] Redundant burnLiquidity cmd in `_collectFees()`

Context:

- BaseAmbientStrategy.sol#L460-L475
- SymetricAmbientStrategy.sol#L490-L505

Description: The function `_collectFees()` will perform a user command type 2 (burnLiquidity) on crocsSwapDex before the command for harvesting fee (command type 5). This is redundant as the harvesting of fee can be performed without burning of liquidity.

```
function _collectFees(
  int24 _upperTick,
  int24 _lowerTick,
  address[2] memory _tokenAddresses
) internal returns (uint128 liq, int128 owed0, int128 owed1) {
  // Get the current position for the specified tick ranges and token addresses
  (liq, , ) = _getPosition(_upperTick, _lowerTick, _tokenAddresses);
  if (liq == 0) return (0, 0, 0);

  // Encode the command for collecting fees
  bytes memory cmd = abi.encode(
    2, // Command type for collecting fees
    _tokenAddresses[0],
    _tokenAddresses[1],
    POOL_IDX,
    _lowerTick,
    _upperTick,
    0,
    0,
    type(uint128).max,
    0,
    address(0)
  );
  // Send the command to the crocsSwapDex contract
  crocsSwapDex.userCmd(cmdId, cmd);
  ...
```

Recommendation: Remove the command type 2.

```
  function _collectFees(
    int24 _upperTick,
    int24 _lowerTick,
    address[2] memory _tokenAddresses
  ) internal returns (uint128 liq, int128 owed0, int128 owed1) {
    // Get the current position for the specified tick ranges and token addresses
    (liq, , ) = _getPosition(_upperTick, _lowerTick, _tokenAddresses);
    if (liq == 0) return (0, 0, 0);

-    // Encode the command for collecting fees
-    bytes memory cmd = abi.encode(
-      2, // Command type for collecting fees
-      _tokenAddresses[0],
-      _tokenAddresses[1],
-      POOL_IDX,
-      _lowerTick,
-      _upperTick,
-      0,
-      0,
-      type(uint128).max,
-      0,
-      address(0)
-    );
-    // Send the command to the crocsSwapDex contract
-    crocsSwapDex.userCmd(cmdId, cmd);
    ...
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/123

Renascence: The issue has been resolved as per recommendation.

[I-4] `_burnAllLiquidities()` should skip burning of liquidity when `liqShares == 0`

Context: BaseAmbientStrategy.sol#L923-L934

Description: `_burnAllLiquidities()` will burn all the liquidities for the specified tick ranges. The `liqShares` amount to burn is determined either based on the existing position or the given `_shares` amount.

It is possible for `liqShares` to be zero for only one of the tick range (e.g. limit), if the `_shares` amount is too small or there are no current position for that specific tick range.

When that occur, it will be more optimal to skip burning of liquidity and only proceeds with the burning of the other tick ranges.

```
function _burnAllLiquidities(
  uint256 _shares,
  LpParam[] memory _lpParams,
  address[2] memory _tokenAddresses,
  bool isRebalance,
  bool checkSlippage,
  uint128 sqrtOraclePrice
) internal {
  // Get the length of the LP parameters array
  uint256 lpParamsLength = _lpParams.length;
```

```
    // Iterate over all LP parameters
    for (uint i; i < lpParamsLength; ++i) {
      uint128 liqShares;

      // Determine the liquidity to burn based on whether this is a rebalance operation
      if (isRebalance) {
        // Get the current position for the specified tick ranges
        (liqShares, , ) = _getPosition(_lpParams[i].upperTick, _lpParams[i].lowerTick,
        _tokenAddresses);
      } else {
        // Calculate the liquidity for the given shares and tick ranges
        liqShares = _liquidityForShares(_lpParams[i].upperTick, _lpParams[i].lowerTick,
        _shares, _tokenAddresses);
      }

      // Burn the liquidity for the specified tick ranges
      _burnLiquidity(
        BurnLiqParams({
          upperTick: _lpParams[i].upperTick,
          lowerTick: _lpParams[i].lowerTick,
          liq: liqShares,
          tokenAddresses: _tokenAddresses,
          checkSlippage: checkSlippage,
          sqrtOraclePrice: sqrtOraclePrice
        })
      );
    }
  }
```

Recommendation: Skip burning of liquidity when `liqShares == 0`.

```
    function _burnAllLiquidities(
      uint256 _shares,
      LpParam[] memory _lpParams,
      address[2] memory _tokenAddresses,
      bool isRebalance,
      bool checkSlippage,
      uint128 sqrtOraclePrice
    ) internal {
      // Get the length of the LP parameters array
      uint256 lpParamsLength = _lpParams.length;

      // Iterate over all LP parameters
      for (uint i; i < lpParamsLength; ++i) {
        uint128 liqShares;

        // Determine the liquidity to burn based on whether this is a rebalance
        operation
        if (isRebalance) {
          // Get the current position for the specified tick ranges
          (liqShares, , ) = _getPosition(_lpParams[i].upperTick, _lpParams[i].lowerTick,
          _tokenAddresses);
        } else {
          // Calculate the liquidity for the given shares and tick ranges
          liqShares = _liquidityForShares(_lpParams[i].upperTick,
          _lpParams[i].lowerTick, _shares, _tokenAddresses);
        }


  +      if (liqShares == 0) continue;

        // Burn the liquidity for the specified tick ranges
        _burnLiquidity(
          BurnLiqParams({
            upperTick: _lpParams[i].upperTick,
            lowerTick: _lpParams[i].lowerTick,
            liq: liqShares,
            tokenAddresses: _tokenAddresses,
            checkSlippage: checkSlippage,
            sqrtOraclePrice: sqrtOraclePrice
          })
        );
      }
    }
```

Client: Fixed at https://github.com/Tempest-Finance/tempest_smart_contract/commit/8d574083a1a18926f0e71c66d4f0f1205ed9f6f6

Renascence: The issue has been resolved as per recommendation.

[I-5] Tick range validation should reject `_upperTick == _lowerTick`

Context: VaultLibrary.sol#L20-L25

Description: `VaultLibrary._checkTicks()` should also reject invalid tick range where `_upperTick == _lowerTick`.

```
library VaultLibrary {
  function _checkTicks(int24 _upperTick, int24 _lowerTick, int24 tickSize) internal
  pure {
    if (_upperTick < _lowerTick || _upperTick % tickSize != 0 || _lowerTick %
    tickSize != 0) {
      revert BadRange();
    }
  }
```

Recommendation: Reject `_upperTick == _lowerTick` as well, to ensure it is a valid tick range.

```
- if (_upperTick < _lowerTick || _upperTick % tickSize != 0 || _lowerTick % tickSize
!= 0) {
+ if (_upperTick <= _lowerTick || _upperTick % tickSize != 0 || _lowerTick % tickSize
!= 0) {
```

Client: https://github.com/Tempest-Finance/tempest_smart_contract/pull/109

Renascence: The issue has been resolved as per recommendation.

[I-6] Accumulated fees in KO liquidity will be forfeited if Merkle proof was not provided during withdrawal

Context:

- WstEthStrategy.sol#L196
- RswEthStrategy.sol#L198

Description: Per the Ambient documentation:

> Recover a fully filled knockout liquidity position. (Forfeits accumulated fees but requires no Merkle proof)

Users have the option to call the withdraw function with or without Merkle proof. When users attempt to withdraw without Merkle proof, the ”Recover Call” will be executed internally. In this case, all fees accumulated by the KO liquidity to be burned, regardless of the size of the fee accumulated, will be lost.

Recommendation: Consider claiming the accumulated fees before burning the KO positions.