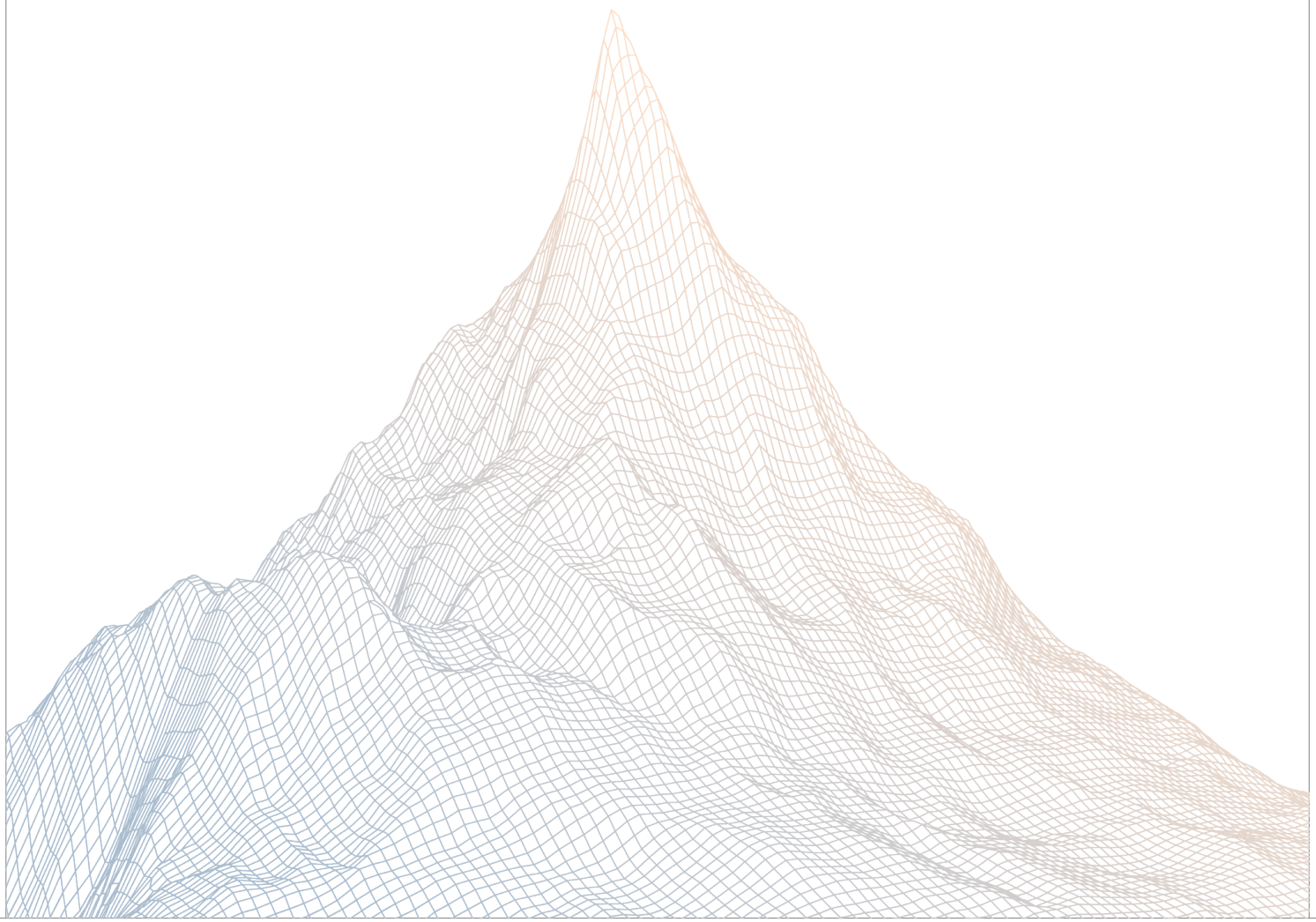
Zenith

Tempest Finance

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 7th to March 11th, 2025

AUDITED BY: peakbolt
spicymeatball

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

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

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Tempest Finance

Tempest is the AI-enabled liquidity network for the DeFi Renaissance. Tempest is backed by Robot Ventures and built on top of Ambient Finance, the industry's most versatile, flexible, and powerful DEX. Tempest's Vaults automatically manage user liquidity in a non-custodial fashion while introducing ERC4626 Compliance to Ambient Finance. This will revolutionize the ability for Liquidity on Ambient to be built on-top of, offering devs a familiar ERC-20 interface.

2.2 Scope

The engagement involved a review of the following targets:

Target	tempest_smart_contract
Repository	https://github.com/Tempest-Finance/tempest_smart_contract/pull/206
Commit Hash	`5d5dd30c5b33e533f95c42aa421a136f0e6e320f`
Files	`Changes in the PR-206`

2.3 Audit Timeline

March 7, 2025	Audit start
March 11, 2025	Audit end
March 20, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	3
Informational	0
Total Issues	**5**

3

Findings Summary

ID	Description	Status
M-1	Imcompatibility with Royco IAM	Resolved
M-2	Idle deposits allow depletion of LP pool tokens	Resolved
L-1	Risk of pool manipulation to profit from depositIdle()	Resolved
L-2	Unnecessary skipCheckStartedAt check for plume deployments	Resolved
L-3	Idle deposits are slightly more profitable than regular deposits	Resolved

4

Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Imcompatibility with Royco IAM

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AmbientUser.sol
- AmbientInterface.sol

Description:

The Ambient strategy vaults that will be integrated with the Royco IAM have a few compatibility as follows.

1. Lack of view functions e.g. previewRedeem(), maxRedeem(),previewMint(), maxMint(), maxDeposit(), maxWithdraw(), asset()
2. Lack of mint() function
3. ERC4626 non-compliance for deposit(), redeem(), withdraw() functions
4. Fails to account for slippage tolerance, swap fees, LP fees collection in the preview functions and conversion function()

Recommendations:

This can be resolved by adding/update the functions to be ERC-4626 compliant and account for the fee/slippage in the share/asset computation.

Tempest: Fixed in baa165eef141 and @473b6ece6588....

Zenith: Verified. Resolved with implementation to support Royco with `redeem()` and `deposit()` and associated view functions, while `withdraw()` and `mint()` will revert to indicate no support for them. It is noted that the `previewDeposit()` will not return the same result as `deposit()` due to fee collection and swap fess, and this is compatible with Royco vault IAM as it does not enforce it unlike `withdraw()`.

[M-2] Idle deposits allow depletion of LP pool tokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AmbientUser.sol#L173-L214
- AmbientUser.sol#L611

Description:

The `depositIdle` function allows users to deposit assets without immediately providing liquidity to the Ambient pool:

```
function depositIdle(uint8 assetIdx_, uint256 amount, address receiver)
  external payable returns (uint256 shares) {
  --- SNIP ---

  // calculate equivalent asset amount
  uint256 assetAmount = amount;
  if (assetIdx_ ≠ _assetIdx) {
    assetAmount = _toAssetAmount(
      _assetIdx,
      _tokenAddresses,
      assetIdx_ == 0 ? amount : 0,
      assetIdx_ == 0 ? 0 : amount,
      oraclePrice.latestAnswer,
      oraclePrice.decimals
    );
  }
  if (assetAmount < minimumDeposit) revert(SMALL_AMOUNT);

  // Collect fees from the specified tick range
  _collectAllFees(lpParams, _tokenAddresses, feeRecipient, fee);

  // Transfer tokens from the user to the contract
  VaultLibrary._transferTokenFromUser(_tokenAddresses[assetIdx_], amount);

  // mint shares
```

```
    uint256 totalAssets = _totalAssets(_assetIdx, _tokenAddresses,
    oraclePrice.latestAnswer, oraclePrice.decimals) -
      assetAmount;
>>  shares = _mintShares(receiver, assetAmount, totalAssets, totalSupply());
  }
```

Even though liquidity isn't immediately provided to the pool, shares are minted at the time of the transaction. This allows users to withdraw assets right after completing a `depositIdle` transaction.

Since the strategy contract distributes both positioned and idle assets based on the user's share, a malicious actor could exploit this by repeatedly depositing and withdrawing assets, effectively draining liquidity from the Ambient pool:

```
  function _withdrawLiquidity(
    LpParam[] memory _lpParams,
    uint256 _shares,
    address[2] memory _tokenAddresses,
    bool checkSlippage,
    uint128 sqrtOraclePrice
  ) internal returns (uint256 totalAmountToSend) {
    // Get the current token balances
    uint256[2] memory balBefore
    = VaultLibrary._getBalances(_tokenAddresses);
    (uint256 bal0Before, uint256 bal1Before) = (balBefore[0], balBefore[1]);

    // Calculate the token balances for the given shares
    uint256 token0FromBal = bal0Before.mulDiv(_shares, totalSupply());
    uint256 token1FromBal = bal1Before.mulDiv(_shares, totalSupply());

    // Burn all liquidities for the given shares
>>  _burnAllLiquidities(_shares, _lpParams, _tokenAddresses, false,
    checkSlippage, sqrtOraclePrice);
```

Since the only source of profit for strategy depositors comes from Ambient LP fees, this exploit could reduce the amount of collected tokens. Operator intervention would then be required to rebalance idle assets, potentially disrupting the strategy's efficiency.

Recommendations:

One possible solution is to prioritize sending idle assets to the withdrawer first without affecting the Ambient position. For example, if there are 1,000 USDC in idle assets and a user withdraws 800 USDC, it would be more efficient to distribute the idle assets first instead of proportionally withdrawing both idle and positioned assets based on their share. The downside of this approach is that it would require a significant architectural change to

the contract.

Another approach to address this issue is to separate idle deposits from regular deposits. For example, during the early stages of liquidity acquisition, it may be more reasonable to allow only idle deposits. However, once the pool has sufficient liquidity, the strategy should restrict deposits to those that also provide liquidity.

Tempest: Fixed in PR-210 & commit @473b6ece658...

Zenith: Verified. Idle deposits now incur fees proportional to the deposit amount. Additionally, the operator can restrict idle deposits when they are not needed.

4.2 Low Risk

A total of 3 low risk findings were identified.

[L-1] Risk of pool manipulation to profit from `depositIdle()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- AmbientUser.sol#L173-L214)

Description:

`depositIdle()` introduces a new function allow users/contracts to deposit funds into the strategy vault without triggering a liquidity provision into Ambient pools. The deposited funds will remain idle till the next rebalance, allowing deposit of large singled-sided liquidity.

The shares that will be minted for `depositIdle()` is then calculated based on the total value of the liquidity positions and idle funds in the vault.

As the vault's liquidity positions depends on the Ambient pool's reserves ratio, it is technically possible to manipulate it via pool swaps. That is one of the reason for the slippage check in `depositIdle()`, to ensure that it will reject any deposits when the pool deviates from the oracle price, implying pool manipulation.

The risk is mitigated to a certain extend, but there is still a possibility of stealing from the vault within the slippage tolerance, if `liqSlippage` is set to a wide range, or when `checkSlippageDepositIdle` is disabled.

In the first case, the attacker can perform swaps before and after `depositIdle()` to manipulate the pool balance and lower the total asset value, to mint more shares than expected. Note that, this is only profitable if the gain exceeds the swap fees.

To make it unfeasible for such pool manipulation attacks, it is recommended to use a tighter slippage value than the current `liqSlippage` and keep the `checkSlippageDepositIdle` always enabled.

Also, the slippage check in `depositIdle()`, performs the check with respect to the max and min of oracle price/pool price, which means the reference price for the check could be oracle price or pool price. This could allow the deposits to continue even when it is above

the slippage tolerance. It should instead check the slippage value against the deviation of the pool price from the reference oracle price.

```
function depositIdle(uint8 assetIdx_, uint256 amount, address receiver)
    external payable returns (uint256 shares) {
  if (assetIdx_ > 1) revert(ASSET_IDX_INVALID);

  (uint8 _assetIdx, address[2] memory _tokenAddresses) = (assetIdx,
    tokenAddresses);
  OraclePrice memory oraclePrice = _calOraclePrice(_tokenAddresses);

  // check slippage
  if (checkSlippageDepositIdle) {
    (, uint8 token1Decimals)
    = VaultLibrary._getTokenDecimals(_tokenAddresses);
    uint128 sqrtPrice = crocsQuery.queryCurve(_tokenAddresses[0],
    _tokenAddresses[1], POOL_IDX).priceRoot_;
    uint256 token1InToken0PoolPrice = (uint256(sqrtPrice)
    * uint256(sqrtPrice) * 10 ** token1Decimals) >> 128;
    if (

    oraclePrice.priceInToken0Decimals.max(token1InToken0PoolPrice).mulDiv(BASE
    - liqSlippage, BASE) >
      oraclePrice.priceInToken0Decimals.min(token1InToken0PoolPrice)
    ) revert(HIGH_SLIPPAGE);
  }

  // calculate equivalent asset amount
  uint256 assetAmount = amount;
  if (assetIdx_ != _assetIdx) {
    assetAmount = _toAssetAmount(
      _assetIdx,
      _tokenAddresses,
      assetIdx_ == 0 ? amount : 0,
      assetIdx_ == 0 ? 0 : amount,
      oraclePrice.latestAnswer,
      oraclePrice.decimals
    );
  }
  if (assetAmount < minimumDeposit) revert(SMALL_AMOUNT);

  // Collect fees from the specified tick range
  _collectAllFees(lpParams, _tokenAddresses, feeRecipient, fee);

  // Transfer tokens from the user to the contract
  VaultLibrary._transferTokenFromUser(_tokenAddresses[assetIdx_], amount);
```

```
    // mint shares
    uint256 totalAssets = _totalAssets(_assetIdx, _tokenAddresses,
      oraclePrice.latestAnswer, oraclePrice.decimals) -
      assetAmount;
    shares = _mintShares(receiver, assetAmount, totalAssets, totalSupply());
  }
```

Recommendations:

Consider the following changes,

1. Create new variable to allow the slippage for `depositIdle()` to be set with a tighter interval.
2. keep the `checkSlippageDepositIdle` always enabled.
3. Check the slippage based on the deviation of the pool price from the oracle price.

Tempest: Fixed in @2e7f5bc1294...

Zenith: Resolved with a new `depositIdleSlippage` to allow a slippage to be set independently from `liqSlippage`. `checkSlippageDepositIdle` is removed to enforce slippage check at all times and adjusted based on `checkSlippageDepositIdle`. `_checkDepositIdleSlippage()` now checks the slippage based on deviation of the pool price from the oracle price.

[L-2] Unnecessary `skipCheckStartedAt` check for plume deployments

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BridgeOracle.sol#L169

Description:

A new flag `skipCheckStartedAt` is added to `BridgeOracle` to skip the validation of `startedAt` in `_checkSequencerDowntime()`. That is because the deployment in Plume will use Chronicle's oracle, which will always returns `startedAt = 0`, unlike Chainlink.

However, for Plume, the sequencer uptime in `BridgeOracle` will be updated manually via `setIsSequencerDown()` by the protocol's off chain keeper.

In that scenario `sequencerUptimeOracle == address(0)` and `_checkSequencerDowntime()` will revert instead of calling `latestRoundData()`. That means `skipCheckStartedAt` flag is redundant as it does not utilized `startedAt` for Plume deployments.

In fact, if a grace period is desired after the sequencer recovers, it will be a good idea to update `startedAt` via `setIsSequencerDown()` for Plume deployments.

```
  function _checkSequencerDowntime() internal view {
    if (!isL2) return;

    // there's no sequencer uptime oracle contract
    if (sequencerUptimeOracle == address(0)) {
      if (isSequencerDown) revert(SEQUENCER_DOWN);
    } else {
      // check sequencer downtime via sequencer uptime oracle contract
      (, int256 answer, uint256 startedAt, , )
    = IOracle(sequencerUptimeOracle).latestRoundData();

      // ref: https://codehawks.cyfrin.io/c/2024-07-zaros/s/189 for checking
    startedAt ≠ 0
>>>   bool isSequencerUp = answer == 0 && (skipCheckStartedAt || startedAt
    ≠ 0);
```

```
      if (!isSequencerUp) {
        revert(SEQUENCER_DOWN);
      }

      // Make sure the grace period has passed after the
      // sequencer is back up.
      uint256 timeSinceUp = block.timestamp - startedAt;
      if (timeSinceUp <= sequencerDowntimeLimit) {
        revert(SEQUENCER_DOWN);
      }
    }
  }
```

Recommendations:

Consider adding a `startedAt` state that will be updated via `setIsSequencerDown()` for Plume deployments, to impose a grace period after recovery of sequencer, similiar to deployment in other chains.

Tempest: Fixed with PR-211

Zenith: Verified. The operator can now prevent the vault from accepting idle deposits. Additionally, idle deposits now incur fees.

[L-3] Idle deposits are slightly more profitable than regular deposits

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- AmbientUser.sol#L110
- AmbientUser.sol#L213

Description:

Users who deposit and provide liquidity in the same transaction may be at a disadvantage because they have to pay swap fees, resulting in fewer shares compared to users who deposit using `depositIdle`:

```
  function deposit(uint256 amount, address receiver, bool checkSlippage)
    external payable returns (uint256 shares) {
      --- SNIP ---

    // Swap and provide liquidity
>>  uint256 amountAfterSwap = _zapDeposit(
      ZapDepositParams({
        lpParams: _lpParams,
        assetIdx: _assetIdx,
        tokenAddresses: _tokenAddresses,
        amount: amount,
        oracleLatestAnswer: oraclePrice.latestAnswer,
        oracleDecimals: oraclePrice.decimals,
        checkSlippage: checkSlippage,
        sqrtOraclePrice: oraclePrice.sqrtPrice
      })
    );

    // mint shares
>>  shares = _mintShares(receiver, amountAfterSwap, totalAssets,
    totalSupply());
  }
```

As a result, users may avoid depositing and providing liquidity, leaving the operator responsible for manually converting idle assets into liquidity.

Recommendations:

Since idle deposits are used in strategies where the associated Ambient pools have low liquidity, it is recommended to separate regular deposit functions from `depositIdle`. For example, at early stages, only `depositIdle` should be allowed. Once sufficient liquidity is available in the Ambient pool, only regular deposits should be permitted.

Tempest: Fixed with PR-210

Zenith: Verified. The operator can now prevent the vault from accepting idle deposits. Additionally, idle deposits now incur fees.