Teahouse

SMART CONTRACTS REVIEW

zokyo

September 26th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING TEAHOUSE

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 3 High issues: 3 resolved = 0 points deducted
- 10 Medium issues: 5 resolved and 5 acknowledged = - 7 points deducted
- 6 Low issues: 4 resolved and 2 acknowledged = - 2 points deducted
- 3 Informational issues: 1 resolved and 2 acknowledged = 0 points deducted

Thus, 100 - 7 - 2 = 91

TECHNICAL SUMMARY

This document outlines the overall security of the TEAHOUSE smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the TEAHOUSE smart contract/s codebase for quality, security, and correctness.

Contracts Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the TEAHOUSE team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the TEAHOUSE repository:
Repo: https://github.com/TeahouseFinance/TeaVaultAmbient/tree/main

Last commit - 1c664099e47669134ff5d72dafb54e9abcf93f95

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./SwapRelayer.sol
- ./TeaVaultAmbientFactory.sol
- ./library/VaultUtils.sol
- ./library/TokenUtils.sol
- ./TeaVaultAmbient.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of TEAHOUSE smart contracts. To do so, the code was reviewed line by line by our smart contract/s developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

TEAHOUSE LP Vaults is a multi strategy DeFi asset management platform which uses a distinctive approach to implementing dual asset pair tokens used to provide liquidity (also known as Permissionless Vaults) in order to achieve secure and flexible wealth management. This approach is implemented by collecting the best DeFi strategies to date and matching individuals with institutional investors to leverage high yield strategies curated by in-house and external strategy providers.

The vaults are deployed via the factory by the contract owner which allows the deployer to define various parameters such as the desired tokens, various fees capped at 30% and assign a pool manager who will manage the vault on behalf of the users. Once a vault is deployed, users are able to freely deposit into the vault. In exchange, they will receive a share token to represent their deposits. The pool manager then actively manages the various positions by allocating liquidity to the Ambient pool to collect swap fees. The pool manager also has the authority to make swaps on the users behalf (token0 → token1 or token1 → token0 as defined by the vault parameters via Ambient or otherwise) if they see an opportunity to extract value. Users are charged an entry fee, a management fee, a performance fee and an exit fee which is distributed to the treasury address.

Zokyo was tasked with conducting the security review of the smart contract component of TEAHOUSE's techstack. Overall, the contracts were well thought out and comply with existing standards such as ERC-1822 (Universal Upgradeable Proxy Standard), implemented modular programming where each functionality was separated into individual functionalities making maintenance easier for the developers and was well documented.

The issues discovered during the security review ranged from informational issues up to high severity issues (but not critical). These findings address errors around access controls violations through rug vectors and overpowered pool managers, errors around how fees were being charged to the users in addition to logic errors which were followed with recommendations on how to remediate these issues. A fix review was then undertaken to ensure that these recommendations were correctly implemented and did not introduce additional bugs into the codebase.

As the majority of the protocol focuses on a reliable pool manager, it's recommended that TEAHOUSE thoroughly vet these pool managers by implementing some framework to become a pool manager which may include (but is not limited to) requesting relevant qualifications, perhaps undergo KYC (to deter rug vectors) or present a previous track record/portfolio of investments before allowing them to invest funds on users behalf. Whilst the codebase included some tests using the hardhat framework, we always recommend the developers to implement additional unit tests inspired by the issues found in the report to assert that such bugs are not present in the codebase should the protocol undergo an upgrade.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the TEAHOUSE team and the TEAHOUSE team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Arbitrary Address Supplied to executeSwap of the TeaVaultAmbient Contract Allows the Pool Manager to Drain the Pool Resulting in a Theft of Funds	High	Resolved
2	_fractionOfShares Function in TeaVaultAmbient Contract May Cause the Vault to Round in the Incorrect Direction During Mathematical Operations	High	Resolved
3	Management Fee For The First Time Will Be Way Larger Than It Should Be	High	Resolved
4	Expected price can get manipulated so that swap functionality is denied everytime	Medium	Acknowledged
5	Collecting management fees in setFeeConfig() can be exploited to dilute the shares of existing depositors	Medium	Resolved
6	No `deadline` + `limitPrice` set to maximum can lead to loss of funds	Medium	Resolved
7	Incorrect Order of Operations in TeaVaultAmbient Will Result In token1 to Never Be Deposited	Medium	Acknowledged
8	Incompatibility with Fee-on-Transfer Tokens	Medium	Acknowledged
9	The Manager Can Manipulate Position Liquidity at the Users Detriment	Medium	Acknowledged
10	Users May Accidentally Deposit into Malicious or Misconfigured Vaults	Medium	Resolved
11	The Decimal Multiplier Would Not Work For Tokens With > 18 Decimals	Medium	Acknowledged
12	Swap operation fails due to zero quantity in ambientSwap function	Medium	Resolved

#	Title	Risk	Status
13	Approval residual issue in ambientSwap function	Medium	Resolved
14	Unsanitized Token Symbol and Name May Allow an Attacker to Achieve Cross Site Scripting in the Front End	Low	Resolved
15	Missing __gap storage placeholder in upgradeable contracts	Low	Resolved
16	Direct Token Transfers to Vault Can Mislead Users About Share Value	Low	Resolved
17	Deployer Can Set Excessive Fees to Drain Funds From Unsuspecting users	Low	Acknowledged
18	Admin Fee Manipulation Combined with Front-Running Users' Transactions May Result in Overcharging of Fees	Low	Resolved
19	Manager Can Manipulate Swaps to Cause Losses to the Vault	Low	Acknowledged
20	Return values of _collectManagementFee() and _collectAllSwapFee() not consumed	Informational	Acknowledged
21	Missing initialization call for PausableUpgradeable in initialize function	Informational	Resolved
22	Insufficient Validation of Liquidity Parameters	Informational	Acknowledged

HIGH-1 | RESOLVED

Arbitrary Address Supplied to executeSwap of the TeaVaultAmbient Contract Allows the Pool Manager to Drain the Pool Resulting in a Theft of Funds

Location: TeaVaultAmbient.sol#executeSwap

Description:
The TeaVaultAmbient contract is a wrapper around the Ambient swap dex which allows funds to be deposited by users where these funds are managed by the pool manager actor who has the authority to add liquidity to a strategy and perform swaps on behalf of the user. The executeSwap function allows the pool manager to execute a swap from any router via the swap relayer. This function takes an address (swap router), however, the address supplied to the function can be a malicious contract which will allow the pool manager to extract funds from the contract by paying well under market rates bypassing the validation against baselineValue.

Consider the proof of concept secret gist below where a user deposits ETH into the contract and the pool manager proceeds to extract $23,577 USD (at the time of writing) worth of ETH in exchange for ~$5,000 DAI:

https://gist.github.com/chris-zokyo/0458e8bf019f1143211322b937a3aee6

A flashloan may allow the attacker to steal funds from vaults with significantly more value.

Recommendation:
Restrict the _swapRouter parameter to a whitelist of trusted and verified swap router addresses in the form of a mapping set by protocol admins. In addition to this, implement strict validation on the swap data and consider integrating with well-known and audited decentralized exchanges (DEXs) to prevent misuse.

_fractionOfShares Function in TeaVaultAmbient Contract May Cause the Vault to Round in the Incorrect Direction During Mathematical Operations

Location: TeaVaultAmbient.sol#_fractionOfShares

Description:
The `_fractionOfShares` function is used in the TeaVaultAbient contract to aid the distribution and pricing of vault shares. This function takes a boolean value where `true` indicates the intention to round up and `false` indicates the intention to round down. However, these operations are performed in an incorrect manner as `mulDivRoundingUp` is called when `_isRoundingUp` is `false`. This may cause the user to receive more and less shares than they deserve.

Recommendation:
It's recommended that this function is refactored to reflect the following to allow rounding in the correct direction:

```
function _fractionOfShares(
    uint256 _assetAmount,
    uint256 _shares,
    uint256 _totalShares,
    bool _isRoundingUp

) internal pure returns (
    uint256 amount
) {
    amount = _isRoundingUp ?
          _assetAmount.mulDivRoundingUp(_shares, _totalShares):
          _assetAmount.mulDiv(_shares, _totalShares);

}
```

Management Fee For The First Time Will Be Way Larger Than It Should Be

In TeaVaultAmbient.sol, the lastCollectedManagementFee should be initialized as block.timestamp , if not when the management fee is accrued for the first time it will be accrued as →

```
function _collectManagementFee() internal returns (uint256
collectedShares) {
        uint256 timeDiff = block.timestamp - lastCollectManagementFee;
```

```
        if (timeDiff > 0) {
            unchecked {
                uint256 feeTimesTimediff = feeConfig.managementFee *
timeDiff;//10000 * td
                uint256 denominator = (
                    FEE_MULTIPLIER * SECONDS_IN_A_YEAR > feeTimesTimediff?
                        FEE_MULTIPLIER * SECONDS_IN_A_YEAR -
feeTimesTimediff:
                        1
                );
                collectedShares =
totalSupply().mulDivRoundingUp(feeTimesTimediff, denominator);
```

As you can see since on declaration lastCollectedManagementFee would be 0 , the timeDiff for the first time would be block.timestamp - 0 which is block.timestamp meaning the fee would be way greater than what it should have been (should have been the timestamp difference when the vault was initialized to the current timestamp).

Recommendation:

Initialize the lastCollectedManagementFee to block.timestamp.

Expected price can get manipulated so that swap functionality is denied everytime

The function `executeSwap()` within the `TeamVaultAmbient.sol` contract, is used to swap tokens. One of the previous steps before `swap()`, is a call to `ambientImpact.calcImpact()`

```
(int128 token0Flow, int128 token1Flow, ) = ambientImpact.calcImpact(
_token0,
            _token1,
            poolIdx,
            _zeroForOne,
            _zeroForOne,
            _maxPaidAmount.toUint128(),
            0,
            _zeroForOne ? type(uint128).max : 0     // when buying, priceLimit is upper bound
        );
```

This `calcImpact` works like a swap simulation, but without actually executing the swap. `calcImpact` returns `token0Flow` and `token1Flow` which is actually used to get the amount of tokens that are supposed to be returned as a swap result.

This amount is later compared to the actual swapped amount. It is working like an slippage controller.

```
if (receivedAmount < baselineAmount) revert WorseRate(baselineAmount, receivedAmount);
```

1. `receivedAmount`: The actual amount returned as a swap result from any router.
2. `baselineAmount`: The amount expected if the swap was executed on Ambient Finance's pools.

This implementation can lead to the problem that if `receivedamount` is always less than `baselineAmount` then no swaps will be able to get executed. If would be a good idea if one of these was an input parameter and both were amounts from the same router, as it is done with `_minReceivedAmount`.

1. `baselineAmount` is the amount returned by `calcImpact` which is calculated on-chain, this means that it can get easily manipulated with a flashloan or having the enough funds to execute the manipulation. An attacker could front-run every swap transaction, manipulating this price in order to not allow any swap operation to be executed.
2. Even if it is not manipulated, if the pool from Ambient Finance becomes obsolete so that no funds are still on the pool, this price will always be lower than the actual one from other routers, this means that the swap will experience a denial of service.

Recommendation:

Do not rely on `calcImpact`. Better use an input variable the same way it has been done with `_minReceivedAmount`.
Client comment: The manager can use ambientSwap instead of executeSwap in this scenario. calcImpact is an approach to prevent minReceived from being set too low.

MEDIUM-2 | RESOLVED

Collecting management fees in setFeeConfig() can be exploited to dilute the shares of existing depositors

The admin can call the **setFeeConfig()** multiple times(even change it to max fees cap of 100%) and just keep accruing fees even when no new deposits happen. This is because on the **line: 168 _collectManagementFee()** is being called. This could be unfair for existing depositors as it would dilute their shares drastically.
It is also possible that the admin of the Vault can frontrun users and change the fee to 100% just before the user's transaction occurs and charge a high fee without user's notice.

Recommendation:

It is advised to rethink on the need to **_collectManagementFee()** in **setFeeConfig()** and consider removing it. Also it is advised to use a mutlisig wallet with at least 3/5 configuration to decentralize the access to the administrative functions of the Vault.

No `deadline` + `limitPrice` set to maximum can lead to loss of funds

The function `ambientSwap()` within the `TeamVaultAmbient.sol` contract, which is used to swap tokens using Ambient Finance, does not implement a `deadline` parameter and hardcodes `limitPrice` to its maximum value.

1. No deadline: although a `_maxPaidAmount` and a `_minReceivedAmount` are passed to ambient, if a deadline control is not used, the swap could result in a worse price due to front-running displacement attacks.
2. The `limitPrice` is set to `uint128.max` if `_zeroForOne` is true and to `0` if it is false. `limitPrice` means the worst price the user is willing to pay on the margin. So, no having a correct control of limitPrice could result in swapping with a worse price.

The `deposit` and `withdraw` function neither implement a `deadline` parameter and check.

Recommendation:

Both mentioned issues should be fixed:

1. Add a `deadline` parameter and revert the transaction if it is executed after the deadline.
2. Add a `limitPrice` parameter to the `ambientSwap` function, which should be later encoded.

Incorrect Order of Operations in TeaVaultAmbient Will Result In token1 to Never Be Deposited

Location: TeaVaultAmbient.sol#deposit

Description:
The deposit function of the TeaVaultAmbient allows a user to deposit into the protocol allowing their funds to be managed by a pool manager. This will let the pool manager deposit into strategies allowing the user to farm yield on their tokens after various fees are accounted for. The vault's first deposit will consider the amount sent of token0 in order to bootstrap the vault share's totalSupply minting token at a rate of 1:1 dividing by the decimals multiplier stipulated by the pool owner. During subsequent deposits, the vault will consider the balance of the contract for both tokens in order to determine how many vault shares to issue in the following lines:

```
 uint256 token0BalanceOfVault = _token0.getBalance(address(this)) -
msg.value;
 uint256 token1BalanceOfVault = _token1.getBalance(address(this));
 _charge(_token0, _amount0Max);
 _charge(_token1, _amount1Max);
```

The issue is that these orders of operations are incorrect as the balances for token1 will always be zero, as a result, the user will deposit token0 but never token1 as token1 will be entirely refunded during the refund process. This may cause certain functionalities of the vault to revert such as swapping token1 for token0 and providing liquidity to two token strategies significantly limiting what the pool manager can do.

Recommendation:
It's recommended that the vault reverses these orders of operations to first charge the user for token0 and token1 then obtain balances to distribute vault shares. The initial deposit logic will also need to be refactored in order to consider both `token0` and `token1`.

Client comment: There may be some token1 in LPs held by the vault, and deposit takes both tokens based on the portion of LPs, because it adds liquidity to all open liquidity positions proportionally.

Incompatibility with Fee-on-Transfer Tokens

Description:
The TeaVaultAmbient contract assumes that the amount of tokens transferred equals the amount specified in the transfer function calls. For tokens that implement a fee-on-transfer mechanism (i.e., deduct a fee during transfers), the actual amount received by the contract will be less than expected. This discrepancy can lead to incorrect accounting, miscalculations in share allocations, and potential losses for users.

Recommendation:
Modify the contract to handle tokens with fee-on-transfer behavior by implementing one of the following:

- Asserting the actual amount of tokens received after each transfer and adjusting calculations accordingly.
- Alternatively, enforce a restriction that only allows tokens without fee-on-transfer mechanisms by adding a validation step during initialization that rejects incompatible tokens.

Client comment: We choose to not support fee-on-transfer token. There does not seem to be a standard method to verify if an ERC20 token does this or not, so we will have to verify manually when creating a vault.

The Manager Can Manipulate Position Liquidity at the Users Detriment

Description:
The manager has full control over adding and removing liquidity positions through the addLiquidity() and removeLiquidity() functions. The manager could manipulate these positions in ways that negatively impact users, such as:

- Adding liquidity to illiquid or high-risk positions that are unlikely to generate returns.
- Removing profitable liquidity positions before performance fees are calculated, reducing the vault's apparent gains.
- Timing the addition or removal of liquidity to coincide with user deposits or withdrawals, affecting the share value calculations to the detriment of users.

Recommendation:
Implement policies and mechanisms to align the manager's incentives with those of the users, such as:

- Setting guidelines or limits on the types of positions the manager can enter.
- Requiring a time delay or community approval for significant changes in liquidity positions.
- Implementing performance metrics that reward the manager for positive outcomes and penalize negative ones.

Client comment: Acknowledged.This vault will be used by in-house strategy team. Time delay may not be practical because the market sometimes may change very rapidly. There is a performance fee for the manager. A vault is underperforming the TVL will be very low.

Users May Accidentally Deposit into Malicious or Misconfigured Vaults

Description:
Since anyone can deploy a vault via the factory contract and assign themselves as the owner and manager, users might unknowingly deposit funds into vaults controlled by malicious actors. These actors could set high fees, manipulate liquidity positions, or perform other actions that result in losses for users.

Recommendation:
Establish a registry of verified and audited vaults that users can refer to when choosing where to deposit funds. Educate users on the risks of depositing into unverified vaults and encourage them to perform due diligence. Additionally, implement access controls or verification steps in the factory contract to limit the deployment of new vaults to trusted parties.

The Decimal Multiplier Would Not Work For Tokens With > 18 Decimals

The DECIMAL_MULTIPLIER has been used for tokens such as USDC with 6 decimals (non 18 decimal tokens) , the multiplier would be applied here →

```
if (totalShares == 0) {
            // vault is empty, default to 1:1 share to token0 ratio
(offseted by _decimalOffset)
            depositedAmount0 = _shares / DECIMALS_MULTIPLIER;
```

So if token was USDC , DECIMAL_MULTIPLIER woud be 1e12 to normalize the deposited amount to 6 decimals . But if the token has say 22 decimals then the following would be incorrect cause in that case we would need to multiply by the DECIMAL_MULTIPLIER , for example we would have DECIMAL_MULTIPLIER as 1e4 and multiply shares with DECIMAL_MULTIPLIER to provide us with the correct value.

Recommendation:

Handle the case where tokens have > 18 decimals .

Client comment: There's no hard constraints on the decimals of vaults, it's controlled by decimals offset when initializing. If token0 has more than 18 decimal digits we can still use 0 or more for offset.

Swap operation fails due to zero quantity in ambientSwap function

Location: TeaVaultAmbient.sol

In the `ambientSwap` function of the `TeaVaultAmbient.sol` contract, the `_maxPaidAmount` is set to zero in cases involving non-native token transfers to ensure the value passed is zero. However, this also sets the `qty` parameter of the `userCmd` function to zero, which results in a no-operation swap (outcome is (0, 0)). This essentially makes the swap ineffective.

```
        // swap using Ambient pool
         bytes memory results =
ambientSwapDex.userCmd{value:_maxPaidAmount}(
            paramsConfig.swapCallPath,
            abi.encode(
                token0,
                token1,
                poolIdx,
                _zeroForOne,
                _zeroForOne,
                _maxPaidAmount,    // @param qty
                0,
                _zeroForOne ? type(uint128).max : 0,      // when buying,
priceLimit is upper bound
                _minReceivedAmount,
                0
            )
        );
```

Recommendation:

To resolve this issue, separate the value passed to `userCmd` from the quantity of tokens being swapped. This can be done by properly setting the `qty` parameter even when `_maxPaidAmount` is zero. Ensure that `_maxPaidAmount` correctly represents the value passed for the transaction, while qty should represent the actual amount of tokens intended to be swapped.

Approval residual issue in ambientSwap function

Location: TeaVaultAmbient.sol

The `ambientSwap` function in TeaVaultAmbient.sol approves a token amount of `_maxPaidAmount` to the ambientSwapDex but does not clear the residual allowance post swapping. This behavior can result in future calls to `_src.approve(address(ambientSwapDex), _maxPaidAmount);` failing for certain tokens, such as USDT, that do not allow non-zero approvals unless cleared. This can lead to unintended approval states and hinder the functionality of the swap mechanism.

Recommendation:

It is recommended to set the approval to zero after the swap is executed to ensure that no residual approval amounts persist. This can be done by adding `_src.approve(address(ambientSwapDex), 0);` right after the swap is called. This ensures compatibility with tokens (e.g., USDT) that enforce zero approval before setting a new approval amount.

LOW-1 | RESOLVED

Unsanitized Token Symbol and Name May Allow an Attacker to Achieve Cross Site Scripting in the Front End

Location: TeaVaultAmbientFactory.sol#createVault

Description:
The TeaVaultAmbientFactory allows a pool owner to create a new vault taking two tokens allowing users to deposit into their vault to take advantage of the strategies offered. The token name and symbol for the vault are displayed in the frontend website, this could allow an attacker to inject javascript code to potentially achieve Cross Site Scripting (XSS). Arbitrary JavaScript code may result in the siphoning of user funds through the signing of malicious transactions and aid in phishing attempts.

Recommendation:
It's recommended that the developers implement dompurify in their frontend to escape special characters which may be attempting to achieve Cross Site Scripting.

Missing __gap storage placeholder in upgradeable contracts

Location: TeaVaultAmbientFactory.sol, TeaVaultAmbient.sol

The contracts `TeaVaultAmbientFactory` and `TeaVaultAmbient` are upgradeable but lack the `__gap` storage variable, which is crucial for accommodating future storage changes without causing conflicts. This omission can lead to serious storage layout issues when upgrading the contracts, jeopardizing their functionality and security.

Recommendation:
To mitigate potential storage conflicts in future upgrades, it is recommended to include a `__gap` storage variable in both `TeaVaultAmbientFactory` and `TeaVaultAmbient`. This variable should be an array of a fixed size, typically `50` slots, reserved for future storage variables. This allocation ensures that any new storage variables can be added in later versions of the contracts without disrupting the existing storage layout.

Direct Token Transfers to Vault Can Mislead Users About Share Value

Description:
The TeaVaultAmbient contract allows users to deposit tokens in exchange for shares representing their stake in the vault. However, users can directly transfer tokens to the vault contract without going through the deposit function. These direct transfers increase the vault's total assets without minting additional shares, artificially inflating the share price. This can mislead other users and create an illusion of higher vault performance or value.

Scenario:
1. **Initial Deposit:** A user deposits $1 worth of tokens into the vault via the deposit function and receives shares proportional to their deposit.
2. **Direct Transfer:** The same user then directly transfers an additional $1,000 worth of tokens to the vault contract without using the deposit function. This increases the vault's total assets but does not mint additional shares.
3. **Inflated Share Price:** The share price effectively doubles due to the increased assets without an increase in total shares. Users viewing the vault's statistics might see a high share price or performance, potentially misleading them into thinking the vault is more profitable than it is.
4. **Withdrawal:** When the user withdraws, they can only redeem assets proportional to their shares. The extra assets remain in the vault, benefiting all shareholders equally.

Recommendation:
Prevent Direct Transfers: Implement a mechanism to reject direct token transfers to the vault contract. This can be achieved by adding a fallback function that reverts any direct transfers by selfdestruct functions.

- **Adjust Share Price Calculations:** Modify the share price calculation to exclude tokens received via direct transfers or to account for them properly.
- **UI Warnings:** Update the user interface to warn users when the vault has a low TVL but a high share price, indicating that the share price may not be indicative of actual performance.
- **Mint Initial Token Shares To Zero Address:** When the `TeaVaultAmbient` is initialized, consider minting a small amount of tokens to the zero address. This will bootstrap the totalSupply and make inflationary attacks extremely expensive to trigger.
- **Use of Virtual Accounting:** The Aave v3.1 upgrade introduced virtual accounting to decrease the attack surface of their contracts which were using `balanceOf`. This ensures that such externally manipulatable functions no longer pose a risk to the contracts.

Deployer Can Set Excessive Fees to Drain Funds From Unsuspecting users

Description:
In the initialize() function of the TeaVaultAmbient contract, the deployer can set the FEE_CAP parameter to a value just below FEE_MULTIPLIER (which is set to 1,000,000). This allows the owner to configure the feeConfig with excessively high fee percentages, up to 99.9999%. Specifically, the owner can set:

- entryFee and exitFee such that their sum is less than or equal to FEE_CAP.
- performanceFee up to FEE_CAP.
- managementFee up to FEE_CAP.

By setting these fees to extremely high values, the owner can effectively drain funds from the vault through fees charged on deposits, withdrawals, performance gains, and management over time.

Recommendation:
Implement strict upper limits on all fee percentages that are hard-coded into the contract and cannot be exceeded, regardless of the FEE_CAP value. For example, set maximum allowable percentages for entryFee, exitFee, performanceFee, and managementFee (e.g., no more than 5%). This prevents the owner from setting excessive fees that could harm users.

Client comment: FEE_CAP is the cap of the various fees and it can't be changed once initialized. Its purpose is to provide lesser flexibility on fees. A user may opt to avoid a vault with FEE_CAP higher than comfortable. We generally set FEE_CAP to 30%.

Admin Fee Manipulation Combined with Front-Running Users' Transactions May Result in Overcharging of Fees

Description:
The admin can exploit the setFeeConfig() function to manipulate fees right before users' transactions, effectively front-running users. By monitoring pending user transactions, the admin can increase fees, such as entry or exit fees, just before those transactions are confirmed. This combination of front-running and arbitrary fee increases can result in significantly higher costs for users or a reduction in the value of their withdrawals, making the system highly unfair and potentially harmful for users.

Scenario:

- A user submits a transaction to deposit or withdraw funds from the vault.
- The admin monitors the pending transaction and quickly increases the entry or exit fees by calling setFeeConfig() before the transaction is confirmed.
- The user's transaction is processed with the new, higher fees, resulting in a significantly reduced deposit or withdrawal value.
- The admin benefits from the increased fees, while users suffer from unexpected, higher costs without any warning.

Also the **_collectManagementFee()** is exhorbitantly high if deposit is made after large periods of time such as a month or over a month. This can misalign the economic incentives as for some reason if the **Vault** is uninteracted with for a long time, then a depositor would be discouraged from depositing first due to a high management fee. For example: Let's say that the **totalsupply()** of shares is **1000*10**18**. Now let's say if the Vault is uninteracted with for 6 months(taking a longer time to illustrate impact but this can also be considered as 1 month), then let's assume

managementFee = 1,000,000
timeDiff = SECONDS_IN_A_YEAR / 2 = 15,768,000
Then,
FEE_MULTIPLIER * SECONDS_IN_A_YEAR = 1,000,000 * 31,536,000 = 31,536,000,000,000
denominator = 31,536,000,000,000 - 15,768,000,000,000 = 15,768,000,000,000
collectedShares = totalSupply().mulDivRoundingUp(feeTimesTimediff, denominator)
= 1000*1018**
Also note that for 1 month time difference the collected shares turns out to be approx 89*10**18 which is still significantly high.

Recommendation:
Implement a cooldown period between the time a fee change is requested and when it becomes effective. This prevents admins from making last-minute fee changes to front-run user transactions. Also consider lowering the cap of the management Fee in order to prevent the admin from setting Management fees as 100%

LOW-6 | ACKNOWLEDGED

Manager Can Manipulate Swaps to Cause Losses to the Vault

Description:
The manager has the ability to perform token swaps on behalf of the vault using the ambientSwap() and executeSwap() functions. There are insufficient checks to ensure that swaps are executed at fair market rates. The manager could intentionally execute swaps at unfavorable rates, causing significant losses to the vault. For instance, the manager might swap large amounts of tokens in illiquid pools or manipulate price slippage parameters to execute trades that are detrimental to the vault's value.

Recommendation:
It's recommended that pool managers have a limited number of operations that they can perform per day in order to reduce risky transactions and promote thoughtfulness when performing such transactions.

Return values of _collectManagementFee() and _collectAllSwapFee() not consumed

Description: TeaVaultAmbient.sol

In the functions `setFeeConfig(...)`, `deposit(...)`, and `withdraw(...)` within TeaVaultAmbient.sol, the return values of the functions `_collectManagementFee()` and `_collectAllSwapFee()` are not being utilized or checked. This may result in overlooked fee collection statuses.

Recommendation:
Evaluate the return values of `_collectManagementFee()` and `_collectAllSwapFee()` within the `setFeeConfig(...)`, `deposit(...)`, and `withdraw(...)` functions. Ensure that the results are properly checked and handled to confirm proper execution and to appropriately address potential unexpected values.

Client comment: Returned values are included in events.

Missing initialization call for PausableUpgradeable in initialize function

Location: TeaVaultAmbientFactory.sol

The `initialize(...)` function in `TeaVaultAmbientFactory.sol` does not invoke the initialization function of the `PausableUpgradeable` contract. Given that `TeaVaultAmbientFactory` inherits from `PausableUpgradeable`, omitting this call may leave the `PausableUpgradeable` contract improperly initialized, potentially leading to unexpected behavior when attempting to pause or unpause the contract.

Recommendation:
To resolve this issue, ensure that the `initialize` function calls the initializer function of the `PausableUpgradeable` contract. You can do this by adding `__Pausable_init();` within the `initialize` function. This ensures that the inherited `PausableUpgradeable` contract is properly initialized.

Insufficient Sanity Checks of Liquidity Parameters

Description:
The addLiquidity() function relies on the manager to provide appropriate values for _tickLower, _tickUpper, and _liquidity. If the manager provides invalid or extreme values, it could result in failed transactions or unintended behavior. Additionally, there's a risk of rounding errors due to the adjustments made to the liquidity amount (roundUpX12Liquidity).

Recommendation:
Implement validation checks on the liquidity parameters provided by the manager. This could include:

- Ensuring that _tickLower is less than _tickUpper.
- Verifying that the _liquidity amount is within acceptable bounds.
- Handling rounding adjustments carefully to prevent significant discrepancies.

Client comment: We left the checking of the parameters to Ambient's contracts. They should revert if the parameters are incorrect.

	./SwapRelayer.sol ./TeaVaultAmbientFactory.sol ./library/VaultUtils.sol ./library/TokenUtils.sol ./TeaVaultAmbient.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the TEAHOUSE team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the TEAHOUSE team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo