Ginger Security

Smart Contract Security Report

for

Teahouse Finance

Prepared By: Jonathan S & Ron S

Ginger Security

07 March, 2024

Table of Contents

General Information

Title	Cairo Smart Contract Security Report
Client	Teahouse Finance
Website	https://teahouse.finance/
Platform	Cairo, Starknet
Product	Teahouse Starknet LP Vault
Authors	Jonathan S, Ron S
Auditors	Jonathan S, Ron S
Audit Start Data	19.02.2024
Audit End Date	23.02.2024
Repository	TeaVaultJediV2
Commit Hash	b40d307929f82805311e2ccf2204085db8f1ef4c
Final Commit Hash (Post Mitigation)	31f664cd74124b1279f2dda9bde87e4df0e144ea
Documentation	https://docs.teahouse.finance
Final Report Date (Post Mitigation)	07.03.2024
Classification	Public

Contact

For more information please contact Ginger Security Inc.: hello@gingersec.xyz

About Teahouse

Teahouse Finance operates as a **DeFi asset management platform**, offering secure and flexible solutions for wealth management. Through the integration of cutting-edge tools and smart contracts, Teahouse simplifies the complexities associated with liquidity and crypto asset management. It provides users with a diverse array of investment options tailored to varying risk tolerances, empowering them to generate passive income by entrusting their assets to Teahouse.

Teahouse LP Vaults, also known as Permissionless Vaults, play a pivotal role in facilitating liquidity provision within the DeFi ecosystem. These Vaults employ dual-asset token pairs, allowing users to deposit tokens into strategies managed by Teahouse or its partners. This setup enables users to seamlessly enter and exit positions as needed. LP Vaults are managed by professional Strategy Managers who dynamically respond to market signals, ensuring transparency and flexibility for users in managing their liquidity positions.

Teahouse's decision to deploy their LP Vaults on the Starknet blockchain atop JediSwapV2, a Uniswap V3 Fork on Starknet, underscores their commitment to innovation within the DeFi space. However, this move required a significant transition, as smart contracts on Starknet are written in Cairo rather than Solidity. As a result, Teahouse rewrote their audited vault smart contracts from Solidity to Cairo. Understanding the critical importance of security in DeFi protocols, Teahouse proactively engaged Gigner Security to conduct a comprehensive review of the newly implemented Cairo contracts. This partnership reflects Teahouse's dedication to maintaining the highest standards of security and reliability in its platform, ensuring that users can engage with confidence in the Teahouse ecosystem on Starknet.

About Ginger Security

At Ginger Security, we are dedicated to providing the highest level of protection for your blockchain assets and applications. Our team of expert security professionals, including former blackhat hackers, have the knowledge and experience to ensure the integrity and safety of your smart contracts and other web3 assets.

We also have remarkable specialization in Layer 2 solutions, particularly in the Starknet ecosystem and programming languages like Cairo. With deep expertise in these areas, we offer comprehensive auditing and protection services tailored to the intricacies of Layer 2 environments. Our proficiency extends to breaking down the complexities of Cairo smart contracts, ensuring that projects deploying on Starknet benefit from robust security measures.

In addition to smart contract auditing, we offer full-stack protection for your blockchain projects. This includes expertise in protocol design, tokenomics, web2, and anti-phishing. Contact us today to learn more about how Ginger Security can help protect your assets and reputation on the blockchain.

Methodology

We take a holistic approach to smart contract auditing and use a combination of manual analysis and cutting-edge tools to thoroughly test your contracts and provide you with a comprehensive security assessment.

Our team of experts begins by conducting a manual review of your contract, using our extensive knowledge and experience to identify potential vulnerabilities or weaknesses. We also use advanced static analysis tools to scan your code for common security issues automatically.

Once we have identified potential risks, we conduct dynamic analysis to test your contract in a safe and controlled environment. This involves executing the contract with a variety of input scenarios and monitoring its behavior to uncover hidden vulnerabilities. We also use symbolic execution and formal verification techniques to rigorously test the behavior of your contract and ensure its correctness.

Throughout the auditing process, we provide regular updates and progress reports to keep you informed of our findings and recommendations. Our goal is to provide you with a detailed and actionable security assessment that can help you improve the security and reliability of your contract

Severity Definition

In Ginger Security, we use a system of severity levels to classify the vulnerabilities that we find during our security audits. Our severity levels include High, Medium, and Low, and are determined based on the potential impact of the vulnerability on the security and functionality of the contract. **High-severity** vulnerabilities pose a significant risk to the contract and should be addressed as soon as possible. **Medium-severity** vulnerabilities are less severe, but still require attention and should be addressed promptly. **Low-severity** vulnerabilities are the least severe, but may still need to be addressed depending on the specific circumstances of the contract.

Disclaimers

Please note that a security audit is not a guarantee of 100% safety. It is important for users and clients to always do their due diligence and not rely solely on the results of an audit. The auditing process is designed to identify potential vulnerabilities and weaknesses, but it is not foolproof. There may be risks that are not uncovered during an audit, and it is the responsibility of the user or client to take appropriate measures to protect their assets and mitigate those risks. Ginger Security cannot be held liable for any losses or damages resulting from the use of our services. We strongly recommend that users and clients carefully evaluate all risks and take appropriate measures to protect their assets.

Executive Summary

In February 2024, Ginger Security undertook a comprehensive security assessment of Teahouse Finance's Cairo smart contracts that will be deployed on the Starknet blockchain for their LP Vaults. The audit carefully looked at how the Teahouse Vault smart contracts work. It checked them thoroughly using automated tests and by going through each part manually to make sure they were trustworthy and worked well.

Throughout the audit process, our team spotted several potential vulnerabilities that necessitate immediate attention to fortify the security posture of the Teahouse Vault smart contracts. We have furnished Teahouse Finance with detailed recommendations tailored to mitigate these vulnerabilities and enhance the overall robustness of their smart contracts.

Even though Ginger Security found some problems in the Teahouse Vault smart contracts, we still believe they're mostly secure. We're committed to helping Teahouse Finance make their products as safe as possible. Our goal is to always give top-notch security help to our clients, keeping their assets and reputations safe in the blockchain world.

Issues Identified during the Audit:

Medium Severity - 7

Low Severity - 15

Scope

The scope of the project is the following smart contracts:

Filename	nCLOC (Cairo Lines of Code)
src/tea_vault_jedi_v2.cairo	1043
src/libraries/vault_utils.cairo	139
Total	1182

System Overview

The Tea Vault Jedi V2 enables users to engage in the JediSwap V2 concentrated liquidity (a fork of Uniswap V3) yield strategy. Each vault is associated with a **specific Liquidity Pool** on JediSwap V2, containing 2 tokens (`token0` and `token1`). It's crucial to note that the vault contract can be deployed multiple times to access yields from multiple liquidity pools. However, each vault contract is exclusively tied to a single liquidity pool.

Managed Assets

A vault functions as a container and manager for various assets:

1. **`token0`:** The liquidity pool's `token0`.
2. **`token1`:** The liquidity pool's `token1`.
3. **Liquidity positions:** These are JediSwapV2 liquidity positions..

Users can acquire shares (ERC20 vault tokens) that represent a portion of the overall assets under the vault's management.

Participating in the Vault

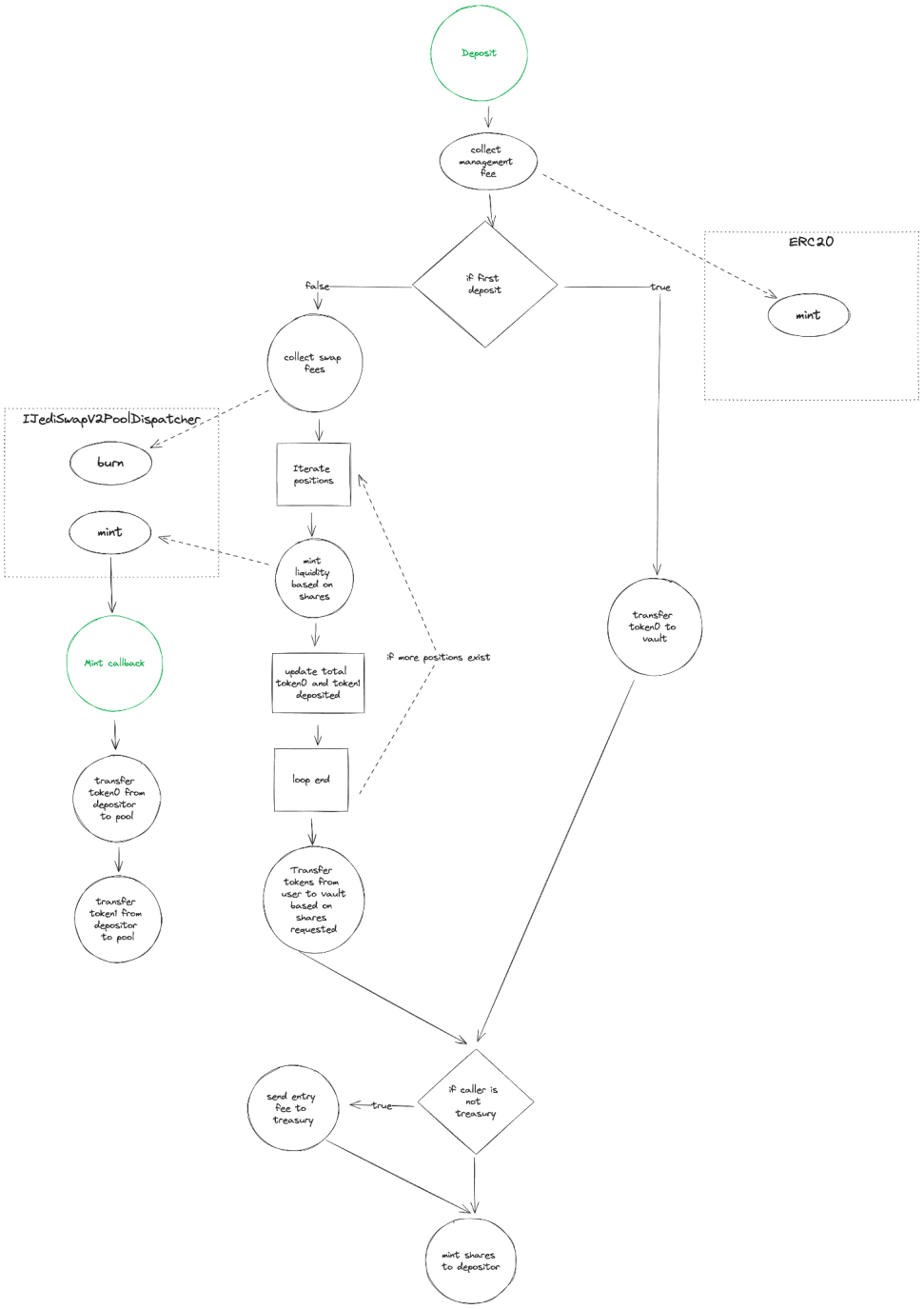
Users can participate in the vault by using the external deposit and withdraw functions.

Deposit

The deposit function enables users to join the vault and acquire shares (ERC20 tokens) representing their ownership fraction of the vault-managed assets. When users invoke the deposit function, they specify the desired amount of shares through the `shares` parameter. Additionally, users can opt for slippage protection by including the `amount0_max` and `amount1_max` parameters.

Before calling the deposit function, users must grant approval to the vault to spend `token0` and `token1`. The vault utilizes this spending approval to transfer tokens to the pool while adding liquidity to all open positions. Moreover, considering the current balances of `token0` and `token1`, the vault calculates the required amounts of both tokens that the user needs to provide and attempts to transfer them from the user's account.

Below is a diagram illustrating the high-level logic of the `deposit` function:

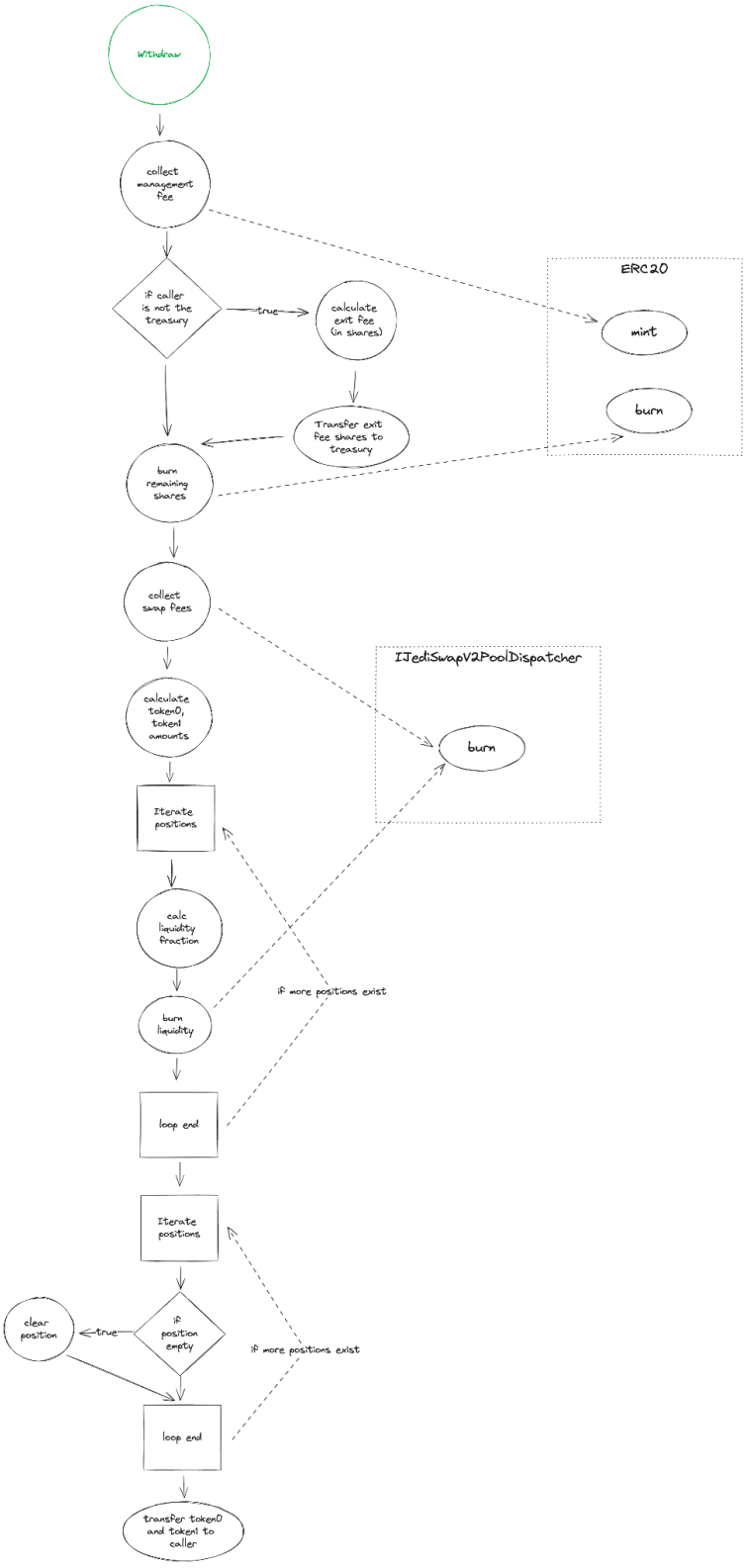


WIthdraw

The withdraw function permits users to exit the vault, burn their shares, and receive their proportional ownership in the form of `token0` and `token1`. Upon invocation, the function calculates the relative ownership of the vault assets and initiates the removal of liquidity from open positions accordingly. If liquidity is entirely withdrawn from a position, the position is removed from the `positions` state variable. The vault disburses the proportional amounts of

`token0` and `token1` to the user. Additionally, the function enables users to specify the `amount0_min` and `amount1_min` slippage protection parameters.

Below is a diagram illustrating the high-level logic of the `withdraw` function:



Liquidity Management

The Vault's liquidity is managed by the manager role (refer to the access control section for more details) and provides various functions to facilitate liquidity management and yield optimization. The manager can access restricted functions enabling token swapping, liquidity addition, liquidity removal, and the collection of swap fees from open positions.

Fees

The type of fees are:

1. **Entry Fee**: Collected from users every deposit in the deposit function, transferred directly to the vault treasury in the form of `token0` and `token1`
2. **Exit_Fee**: Collected on any redemption of shares In the `withdraw` function. The fee is collected as a fraction of the redeemed shares transferred directly to the treasury.
3. **Performance fee**: Collected as a fraction of the tokens received on swap fee collection and transferred directly to the treasury.
4. **Management fee**: Collected on every block by minting new shares to the treasury.

Access Control and Centralization Risks

The Teahouse Vault has three main roles with different access levels:

The Owner Role

The owner of the vault contract is designated during deployment through the constructor function, utilizing the OpenZeppelin Ownable Cairo library. This role holds **significant authority** within the vault and can execute the following actions:

1. **Upgrading Smart Contract Logic:** The owner can upgrade the entire smart contract logic, essentially exerting complete control over the vault's functionality.
2. **Managing Fee Configuration:** The owner can set and update the `fee_config`, which includes adjusting entry fees, exit fees, performance fees, and management fees associated with the vault, as well as the address of the treasury. However, it's important to note that a “Fee Cap” hardcoded into the smart contract limits the owner's authority over fee configuration.
3. **Assigning Manager Role:** Additionally, the owner can designate and update the address associated with the manager role.

Given the critical nature of the owner role, it is vital to ensure that the corresponding account is a multisig wallet account. As Starknet inherently supports and implements account abstraction, a straightforward solution may involve leveraging products like Argent or Braavos Multisig wallet for enhanced security and control.

The Manager Role

The manager role within the vault holds lesser authority than the owner role, as it lacks control over contract assets and fee adjustments. Moreover, the owner retains the ability to replace the manager at will.

While the manager can not withdraw assets from the contract, it can effectively manage the assets and positions within the vault while engaging with JediSwapV2 AMM.

The Manager role encompasses the following key actions:

1. **Management Fee Collection:** The manager can trigger the external `collect_management_fee` function to collect management fees, directing them to the vault treasury configured in `fee_config.treasury`.

2. **Position Yield Collection:** The manager can collect specific position swap fees (position yields) through either the `collect_position_swap_fee` or `collect_all_swap_fee` external functions.
3. **Liquidity Management:** The manager can add liquidity to existing positions or create new ones (up to 5 positions) by invoking the external `add_liquidity` function. Additionally, liquidity can be removed from positions through the external `remove_liquidity` function, with the closure of a position occurring upon full liquidity removal.

4. **Token Swapping:** The manager can facilitate token swaps between `token0` and `token1` managed by the vault, and vice versa, utilizing the external `swap_input_single` and `swap_output_single` functions.

It is essential to acknowledge that while the manager lacks direct control over fund destinations, it can effectively manage vault assets. The manager cannot withdraw or misappropriate funds from the vault but may impact its performance by executing actions that disrupt liquidity provision or result in unfavorable swaps, potentially leading to denial-of-service (DOS) attack scenarios. In addition, the manager has the ability to drain the entirety of the fund's assets as described in issue M-01.

While the manager role poses a lesser risk of fund misappropriation than the owner role, we recommend using a multi-sig wallet for the manager account for added security measures.

Regular Users (Vault Investors)

Regular users function as vault investors, without any specific privileged roles, with their functionality limited to triggering the external `deposit` and `withdraw` functions. Regular users can execute two primary actions within the vault:

1. **Deposit Assets:** Users can deposit assets into the vault in the form of `token0` and `token1`, receiving newly minted vault shares in return. This process is facilitated by triggering the external `deposit` function.
2. **Withdraw Assets:** Users can withdraw assets from the vault by triggering the external withdraw function. This action entails burning their shares and receiving their relative funds in the form of `token0` and `token1`.

The limited functionalities available to regular users, significantly reduce the attack surface from unprivileged users. By restricting regular user actions to depositing and withdrawing assets, the

vault maintains a heightened level of security and mitigates potential risks associated with unauthorized access and malicious actors.

It is worth noting that two additional functions are accessible externally: `jediswap_v2_mint_callback` and `jediswap_v2_swap_callback,` both of which are used as callback functions from the pool on minting and swapping operations, and both have sufficient guard rails.

Attack Vectors That Were Covered

During the security review conducted by Ginger Security, our researchers meticulously evaluated the vault's resilience against various attack classes and vectors. The following attack classes and vectors were scrutinized to ensure the protocol's security:

1. **Frontrunning & Slippage Attacks:** The vault incorporates slippage protection mechanisms, crucial for safeguarding against manipulation during interactions with Automated Market Makers (AMMs).
2. **Share Manipulation & Inflation Attacks:** The architecture of the vault is designed to withstand potential share manipulation and inflation attacks. This is achieved by having the deposit and withdraw functions receive the amount of shares to deposit and withdraw from the user, rather than the amount of assets directly. Subsequently, the vault calculates the amount of assets to withdraw based on the shares requested by the user, ensuring it rounds up in favor of the vault's security measures.
3. **Rounding issues:** A common vulnerability shared among vault and vault-like contracts involves incorrect rounding. Specifically, rounding becomes an issue when depositing and withdrawing assets favor the user. Favorable rounding means that users receive more shares relative to the amount of assets that they deposited, and they receive more assets relative to the amount of shares that they redeem. Our researchers mapped all the instances in which the contract is rounding when calculating fractions and ensured that in all instances, the rounding does not favor the user.
4. **Functions Visibility and Access Control Issues:** Stringent access control measures are enforced within the vault's functions.
5. **Reentrancy Attacks:** To mitigate the risk of reentrancy attacks, proactive measures have been enacted by integrating the `ReentrancyGuardComponent` by OpenZeppelin. Our researchers meticulously verified this component's correct implementation and proper functioning to guarantee the integrity and atomicity of transactions within the vault. Through robust locking mechanisms, the vault remains fortified against potential exploits, ensuring the security of user assets. We recommend reducing the risk of reentrancy further by adding more guards and refactoring as many functions as possible to adhere to the CEI pattern.
6. **Felt252 Overflows and Underflows:** Cairo inherently protects against overflow and underflow attacks by default when dealing with integers and unsigned integers. However, when employing the `felt252` datatype, vulnerabilities to overflow and underflow

attacks may still exist. Our researchers meticulously examined all instances of felt252 usage and verified their safety from potential underflow and overflow risks.

7. **Accounting Issues:** The vault's accounting protocols adhere to robust standards, mitigating risks of discrepancies and ensuring accurate tracking.
8. **Logical Issues:** Thorough logic checks have been instituted to identify and rectify potential logical loopholes.
9. **External Libraries and Dependencies Issues:** The security researchers conducted thorough evaluations of external libraries and dependencies linked to third-party integrations.
10. **Deviations from Vault Solidity Implementation:** The vault's Solidity implementation undergoes an audit and manages significant funds, amounting to millions of dollars in Total Value Locked (TVL). Given its extensive usage and handling of substantial financial assets, it is considered a "black box" that has been extensively tested under various conditions. Our security researchers conducted a comprehensive comparison between the Solidity vault implementation and its counterpart in Cairo. The assessment affirmed the excellence of the implementation.
11. **Upgradability Issues:** In Cairo, upgradability is supported by default via `class_hash` replacement, rendering proxies unnecessary to achieve upgradability capabilities. The vault project leverages the OpenZeppelin Upgradable Cairo library to encapsulate this functionality. Our researchers meticulously verified that the implementation of upgradability aligns with the best practices outlined by OpenZeppelin.
12. **Coding Best Practices:** Our researchers ensured that the vault adheres to coding best practices in implementing Cairo smart contracts.

Key Findings

The smart contracts are well-designed and engineered, but the implementation can be improved by addressing the following issues:

ID	Severity	Title	Status
M-01	Medium	Potential for asset drainage by Manager	Fixed
M-02	Medium	Missing Tests and inline comments	Fixed
M-03	Medium	Relies on unaudited external libraries	Acknowledged
M-04	Medium	Not using a two-step process for transferring ownership	Fixed
M-05	Medium	Error-prone tracking of `positions`	Fixed
M-06	Medium	Absence of Pausable Functionality	Fixed
M–07	Medium	Non-Adherence to CEI Pattern and Insufficient Reentrancy Guards	Fixed
L-01	Low	Prefer Enums or Constants over hardcoded values	Fixed
L-02	Low	Some internal functions don't start with an underscore	Fixed
L-03	Low	Open TODOs	Fixed
L-04	Low	There is no need to use the `bool find` variable	Fixed
L-05	Low	Inappropriate Naming of variable - vault vs. treasury	Fixed
L-06	Low	Misalignment with ERC-4626 Standard in Contract Function Naming	Acknowledged
L-07	Low	Presence of Unused Imports	Fixed
L-08	Low	Unoptimized storage	Acknowledged
L-09	Low	Constant variables kept in storage	Fixed
L-10	Low	Premature Read Operations	Fixed
L-11	Low	Repeated logic for calculating fractions	Fixed
L-12	Low	Repeated logic in swap fee collection	Fixed
L-13	Low	Inconsistent retrieval of total shares	Fixed
L-14	Low	Incorrect constant variable naming convention	Fixed

ID	Severity	Title	Status
L-15	Low	Code duplication of an internal function	Fixed

Finding Details

M-01: Potential for asset drainage by Manager

Impact: Potential for financial losses for users and diminished trust

Description

The current contract design allows the manager to manage the liquidity positions as well as manage swaps between `token0` and `token1`. Managing swaps is an essential functionality for a manager to create new liquidity positions effectively.

Currently, the amount of such operations a manager can execute is unlimited.

As such, a malicious manager can exploit the functionality as follows:

1. Remove all the vault's liquidities (to extract all of the liquidity into tokens).
2. Create a large personal liquidity position on the current tick (using external funds)
3. Generate a large amount of swaps for the entirety of the fund assets
    - Do this in a loop until most of the assets are drained due to swap fees
4. Redeem the personal liquidity position and profit from the swap fees

All of the above steps can be done in the same transaction and the same block to drain the vault's assets instantly. Also, note that this issue is also present in the Solidity version of the contract.

Recommended Mitigation Steps

1. Add a rate limit to the number of swaps that can be performed - for example, a legitimate manager will not need to perform more than one swap per block.
2. Set up external monitoring for the `vault_all_underlying_assets` function and alert the owner on any large changes to the value of the assets
3. Ensure the manager assigned is well-trusted and uses a multisig wallet.

M-02: Missing tests and inline comments

Impact: Potential of bugs and unintended behavior.

Description

The current codebase lacks sufficient unit test coverage, a crucial aspect to ensure the vault functionality operates as intended and to mitigate potential bugs. Comprehensive tests are prerequisites for an audit, and conducting tests during the audit period complicates the auditing process. Moreover, areas within the contract and certain states remain untested.

Furthermore, the contract itself lacks inline developer comments, both within functions and inside them, making it more challenging to review and audit effectively.

Note: It's worth noting an issue with `snforge` tests that caused unexpected behavior when utilizing the `prank` functionality alongside the `get_caller_address()` Starknet system call in the callback functions. Specifically, it returned the transaction initiator address instead of the last caller address, which should be the JediSwapV2 Pool Address. As a temporary workaround, we recommended hardcoding the pool address in the callback functions. However, **it's crucial to remove this temporary patch** before deploying to the mainnet to prevent unexpected behavior.

Recommended Mitigation Steps

It is important to achieve 100% unit test coverage before deployment to Mainnet, and we also recommend adding @notice, @dev, and @param comments to every function in the contract.

M-03: The vault relies on unaudited external libraries

Impact: Potential risk of using un-audited code

Description

YAS (yet another swap) states the following in repo:

"Currently, the project is in a development stage, it has not been audited yet and is not ready for production."

TeaVaultJediV2 contract imports the implementation of signed integers from YAS as well as other math utilities. TeaVaultJediV2 is required to do this due to the reliance on JediSwap, which relies on YAS.

This is a potential risk as the implementation is not audited and may contain vulnerabilities.

In addition, JediSwap has recently announced that the found an integer overflow bug - which means the reliance on JediSwap library should be considered unsafe until an audit is completed.

Recommended Mitigation Steps

Expand the scope of the audit to include the imported functions from YAS and or wait for JediSwap to get an audit first.

M-04: Not using a two-step process for transferring ownership

Impact: Ownership might be mistakenly transferred to a wrong address and be lost forever.

Description

The current implementation of ownership relies on the `OwnableCamelOnlyImpl` component from the `OpenZeppelin 0.8.0 Library. This version lacks the latest updates and does not include the two-step-transfer ownership component OwnableTwoStepCamelOnlyImpl, available in the latest version (0.9.0):

https://docs.openzeppelin.com/contracts-cairo/0.9.0/access#two_step_transfer

Recommended Mitigation Steps

Upgrade to the latest version of OpenZeppelin contracts (0.9.0) and integrate the `OwnableTwoStepCamelOnlyImpl` component instead of the `OwnableCamelOnlyImpl` component. In the updated version, the new owner must accept the transfer, ensuring that in the event of an erroneous transfer, the previous owner retains ownership and the issue can be fixed.

M-05: Error-prone tracking of `positions`

Impact: Reduced readability and maintainability. Increased code complexity.

Description

Currently, `positions` is a dictionary that acts like a List by keeping `position_length` as the index of the last position. Any time a new position is added or removed, `position_length` needs to be updated.

In addition, since there is no native `List` structure in Cairo - basic functionalities such as `push`, `pop` are implemented manually.

This is error-prone, untested, and can lead to bugs. It also makes the code more complex and less maintainable.

Recommended Mitigation Steps

It is recommended to either create a `struct List` to encompass this functionality and use this structure instead of keeping two storage variables in sync.

The custom `List` structure should be thoroughly tested if going with this route.

A better solution, though, is to use the community's `List` from the `alexandria` library, which is already efficiently written and well-tested:

https://github.com/keep-starknet-strange/alexandria/tree/main/src/storage#list

Additionally, it is advised to create two internal functions:

- `_pop_position(index) -> Position` which will combine `pop_front`
- `set_find_position_by_ticks(tick_lower, tick_upper) -> (index, Position)` which will encompass the logic contained in 4 different functions (`collect_position_swap_fee`, `remove_liquidity`, `add_liquidity`, `position_info_ticks`)

M-06: Absence of Pausable Functionality

Impact: Limited ability to mitigate critical situations promptly.

Description

The contract currently does not include any pausable functionality. Pausable mechanisms are vital in smart contracts for enabling a temporary halt to all critical operations in emergencies.

Having the capability to pause deposits becomes invaluable if issues arise within the JediSwapV2 Pools. This measure ensures that users cannot deposit new value into potentially problematic pools, thereby preventing further liquidity from being added to an already compromised situation. This becomes especially critical in light of recent incidents, such as the occurrence of a number overflow issue in the 0.05% ETH-USDC JediSwapV2 pool.

The absence of such a feature limits the ability of administrators or designated parties to react swiftly in mitigating unforeseen risks, such as security breaches or critical bugs.

Although the contract does have an “upgrade” functionality that can enable changing the contract’s functionality in case of an emergency - having a built-in method for pausing can shorten the reaction time during an emergency.

Recommended Mitigation Steps

The OpenZeppelin Pausable component facilitates the implementation of an emergency stop mechanism in contracts:
https://docs.openzeppelin.com/contracts-cairo/0.9.0/security#pausable

Ensure that the pausable functionality is governed by strict access controls, limiting the ability to pause and unpause the contract to the contract’s owner. Additionally, the `when_not_paused` check should be applied to the `deposit` and the `withdraw` functions.

Furthermore, developing an Incident Response plan and Runbooks detailing the step-by-step actions the manager and owner will take in response to various scenarios that may occur within the vault or the liquidity pools integrated with the vaults is essential. These documents will provide clear guidance and procedures to address unforeseen incidents and effectively maintain operational stability.

M-07: Non-Adherence to CEI Pattern and Insufficient Reentrancy Guards

Impact: Risk of reentrancy attacks.

Description

The contract currently does not fully adhere to the Checks-Effects-Interactions pattern, a recommended best practice in smart contract development to mitigate reentrancy risks. This pattern dictates that checks (such as validations and approvals) should be performed first, followed by effects (state changes), and finally, interactions with external contracts.

While the withdraw and deposit functions are protected with reentrancy guards, the manager functions lack such safeguards. Although the manager is currently trusted and there are no obvious reentrancy vulnerabilities, it is advised to protect all functions where the CEI pattern cannot be followed with reentrancy guards to reduce the risk of reentrancy attacks.

Recommended Mitigation Steps

Review and refactor the manager functions to adhere to the Checks-Effects-Interactions pattern where possible - this is always advised regardless of reentrency guards.

In addition, since not all functions can follow the CEI pattern, we advise adding reentrancy guards to all manager functions. For instance, the `add_liquidity` function can be refactored to follow the CEI pattern (which will significantly simplify it as well).

Like so:

```
fn add_liquidity(
    ref self: ContractState,
    tick_lower: i32,
    tick_upper: i32,
    liquidity: u128,
    amount0_min: u256,
    amount1_min: u256,
    deadline: u64
) -> (u256, u256) {
    // todo: add a reentrancy guard as well
    self.assert_only_manager();
    self.check_deadline(deadline);

    let position_length = self.position_length.read();
```

```
    let mut i: u8 = 0;

    loop {
        if i == position_length {
            assert(i < self.MAX_POSITION_LENGTH.read(),
Errors::POSITION_LENGTH_EXCEEDS_LIMIT);
            self.positions.write(i, Position { tick_lower: tick_lower,
tick_upper: tick_upper, liquidity: liquidity });
            self.position_length.write(position_length + 1);
            break;
        }

        let mut position = self.positions.read(i);
        if (position.tick_lower == tick_lower) && (position.tick_upper ==
tick_upper) {
            position.liquidity += liquidity;
            self.positions.write(i, position);
            break;
        }

        i += 1;
    };

    self._add_liquidity(tick_lower, tick_upper, liquidity, amount0_min,
amount1_min)
}
```

L-01: Prefer Enums or Constants over hardcoded values

Impact: Risk of making mistakes during development. Decreased maintainability and readability.

Description

Currently, the `callback_status` storage variable is either set to 1 or to 2 to represent the callback status. This is not a good practice as it makes the code less readable and increases the risk of making mistakes during development.

Another example is the `u128_max` constant which is initialized every time `_collect` is called. In addition to being inefficient, it is also error-prone.

Recommended Mitigation Steps

The `callback_status` storage variable should be either an enum or the values should be defined as constants.

To define an `enum` see:
https://book.cairo-lang.org/ch99-01-03-01-contract-storage.html#storing-custom-types

In addition, the `u128_max` variable should be a constant.

For example: `const U128_MAX: u128 = 0xffffffffffffffffffffffffffffffff;`

L-02: Some internal functions don't start with an underscore

Impact: Risk of exposing mistakenly a sensitive internal function

Description

The majority of the internal functions in the contract start with an underscore which is a good convention. However, some of the internal functions such as `check_deadline`, `check_liquidity`, and `assert_only_manager`, don't start with an underscore.

Maintaining consistency in naming conventions enhances code readability and comprehension while mitigating the risk of exposing internal functions. We recommend aligning all internal function names with the established convention of commencing with an underscore.

Recommended Mitigation Steps

All internal functions should start with an underscore. Add an underline (_) prefix to the following function names: `check_deadline`, `check_liquidity`, and `assert_only_manager`.

L-03: Open TODOs

Impact: Potential issues and bugs in the vault

Description

While reviewing the codebase we noticed some open TODO inline comments in both `tea_vault_jedi_v2.cairo` and `vault_utils.cairo` files:

https://github.com/TeahouseFinance/TeaVaultJediV2/blob/b40d307929f82805311e2ccf2204085db8f1ef4c/src/tea_vault_jedi_v2.cairo#L337

https://github.com/TeahouseFinance/TeaVaultJediV2/blob/b40d307929f82805311e2ccf2204085db8f1ef4c/src/tea_vault_jedi_v2.cairo#L338

https://github.com/TeahouseFinance/TeaVaultJediV2/blob/b40d307929f82805311e2ccf2204085db8f1ef4c/src/tea_vault_jedi_v2.cairo#L614

https://github.com/TeahouseFinance/TeaVaultJediV2/blob/b40d307929f82805311e2ccf2204085db8f1ef4c/src/libraries/vault_utils.cairo#L69

Recommended Mitigation Steps

As an integral step toward achieving production readiness, it is important to address developer comments, especially TODOs. Often, these comments signify missing functionalities that require implementation or essential validation checks that need to be incorporated. We recommend resolving all the TODOs and removing the inline comments prior to mainnet deployment.

L-04: There is no need to use the `bool find` variable

Impact: Unnecessary gas costs

Description

In numerous functions where iteration occurs through all positions within a `loop`, a boolean variable named `find` indicates whether the position was found. If the position is not found, the `find` variable remains false, leading to the transaction reverting due to the `assert` statement. However, in such cases, if a position is not found, both `amount0` and `amount1` returned from the loop will be `0`. Therefore, instead of tracking an additional memory mutable variable like `find` to determine if the position was found, a simple check of `amount0 > 0 || amount1 > 0` can be implemented. This code pattern is being implemented in the following functions:

1. `position_info_ticks`
2. `add_liquidity`
3. `remove_liquidity`
4. `collect_position_swap_fee`

Recommended Mitigation Steps

In all the aforementioned functions, remove the `find` state variable and update the assert statements to:

```
assert(amount0 > 0 || amount1 > 0, Errors::POSITION_DOES_NOT_EXIST);
```

L-05: Inappropriate Naming of variable - vault vs. treasury

Impact: Confusion about the function and purpose of the contract, potentially leading to misinterpretation of its operations.

Description

We observed that fees (performance, exit, entry) are all sent to a contract defined under the variable `vault` in the `FeeConfig` struct. Since the contract is called a `Vault` this variable name is misleading. The functionality of this variable aligns more with `treasury` rather than `vault`.

A discrepancy was noted in the naming convention of a specific contract variable. Specifically, the `vault` variable within the `FeeConfig` struct is used to direct various fees (performance, exit, entry) to a designated address. However, the term "`Vault`" implies the contract in question which is used to store and manage assets - which can be misleading.

In practice, this address behaves more like a treasury, managing and allocating funds rather than merely storing them. This misalignment between the variable name and its functional role could lead to confusion or misunderstanding of the contract's operation.

Also, naming two different addresses using the same term is confusing.

Recommended Mitigation Steps

Change the variable name from `vault` to `treasury` within the `FeeConfig struct`. This change should be reflected in all instances where the variable is used throughout the contract code.

Ensure that all related documentation is updated to reflect this change.

L-06: Misalignment with ERC-4626 Standard in Contract Function Naming

Impact: Confusion among users and developers familiar with the ERC-4626 standard

Description

The ERC-4626 standard is used to unify yield-bearing vaults. Although Teahouse Vaults are not in line with the ERC-4626 standard, it will be beneficial for users and auditors if the terms used are similar.

Teahouse's vault uses the terms `deposit` and `withdraw` while the ERC-4626 standard typically uses the terms `mint` and `redeem` for these operations. The current names `withdraw` and `deposit` do not accurately reflect the operations being performed in line with the ERC-4626 standard. This can lead to confusion, especially for users familiar with the standard, and can potentially cause misinterpretation when interacting with the contract.

By using similar terminology like "`mint`," "`redeem`," and "`assets/shares`" for relevant functions and state variables, allows auditors to quickly understand and make comparisons between your custom vault-like implementation and the ERC-4626 standard.

Recommended Mitigation Steps

1. Rename the withdraw function to redeem and the deposit function to mint. This change should align the function names with the terminology used in the ERC-4626 standard, providing clarity and consistency.
2. Update all related user-facing documentation on the website to align with this change.

L-07: Presence of Unused Imports

Impact: Unnecessary clutter in the codebase

Description

Several imports in the smart contract code are not being utilized. These include `ContractAddressZero`, and `position_swap_fee`. Such unused imports suggest either remnants of previous versions of the code or placeholders for future functionalities that have not been implemented.

Keeping these unused imports in the code is not a best practice as it can confuse developers and auditors, leading them to spend time understanding the relevance of these imports which, in reality, have none.

Recommended Mitigation Steps

Remove all unused imports from the code. This will clean up the codebase, making it more efficient and easier to maintain.

L-08: Unoptimized storage

Impact: Unnecessary gas costs

Description

Most of the gas fees associated with a transaction to a smart contract are due to storage operations. Since, in Cairo, every type is derived from `felt252` which uses 252 bits to store the value, it is important to optimize storage usage. The methodology recommended by the Cairo team is to pack multiple values into a single storage slot using `StorePacking`.

Recommended Mitigation Steps

Combine multiple small storage variables into a single struct.

Pack and unpack the values by implementing the `StorePacking` trait.

See: https://book.cairo-lang.org/ch99-01-03-05-optimizing-storage.html

L-09: Constant variables kept in storage

Impact: Unnecessary gas costs

Description

Some of the storage variables are initialized by the constructor to constant values.

Since storage access is expensive, it is best to define constant variables as `const` - allowing the compiler to optimize their use. Specifically, the following variables are static and can be defined as `const`:
`SECONDS_IN_A_YEAR`, `FEE_MULTIPLIER`, `MAX_POSITION_LENGTH`

Recommended Mitigation Steps

Define constant variables as `const` like this:

```
const SECONDS_IN_A_YEAR: u256 = consteval_int!(365 * 24 * 60 * 60);
```

See: https://book.cairo-lang.org/ch02-01-variables-and-mutability.html#constants

L-10: Premature Read Operations

Impact: Unnecessary gas costs

Description

In the current implementation of the `deposit` function, the `FEE_MULTIPLIER` and `fee_config` storage variables are read outside the if statement (`caller != fee_config.vault`) but are used only inside the if statement.

In cases where this condition is not satisfied, the initial reading of `fee_config` and `FEE_MULTIPLIER` becomes redundant, leading to unnecessary gas usage.

Recommended Mitigation Steps

Modify the code to move the reading of `fee_config` and `FEE_MULTIPLIER` inside the if statement. This ensures that these values are only read when necessary, i.e., when the caller is different from `fee_config.vault`.

L-11: Repeated logic for calculating fractions

Impact: Error-prone and less readable code

Description

The `withdraw` and `deposit` functions have to calculate assets and liquidity as a fraction of the requested shares to mint or redeem. The operation looks like this: `mul_div(total_assets, shares_requested, total_shares)`

The `mul_div_rounding_up` function is used when rounding up is necessary. In total the operation is repeated 6 times in the code. A similar operation the calculates fees is repeat 5 more time. For such a critical operation, it is best to export it to a separate function and test it.

Recommended Mitigation Steps

Create an new internal function, e.g.

```
_fraction_of_shares(total_of_asset: u256, rounding: Rounding) -> u256
```

Rounding can be defined as a `struct` with `Ceil` and `Floor` to handle rounding.

A similar function can be defined for the fee calculation, e.g.

```
_fraction_of_fees(total_of_asset: u256, fee: u32, rounding: Rounding)
-> u256
```

L-12: Repeated logic in swap fee collection

Impact: Reduced maintainability

Description

There are two functions used to collect swap fees: `_collect_position_swap_fee` which collects the fees of a specific position and `_collect_all_swap_fee` which is used to collect the fees from all positions.

The code in `_collect_all_swap_fee` is nearly identical except for two things: it uses a loop to go through all of the positions and collects the performance fees using `_collect_performance_fee` at the end. Following the DRY principle, the code should be exported to a separate function which can then be reused.

Recommended Mitigation Steps

Rename `_collect_position_swap_fee` to `_collect_performance_and_position_swap_fee` and create a new `_collect_position_swap_fee` function which can be reused in `_collect_all_swap_fee`

Like so:

```
fn _collect_position_swap_fee(ref self: ContractState, position: Position)
-> (u128, u128) {
    let pool_dispatcher = IJediSwapV2PoolDispatcher { contract_address:
self.pool.read() };
    pool_dispatcher.burn(position.tick_lower, position.tick_upper, 0);
    self._collect(position.tick_lower, position.tick_upper)
}

fn _collect_performance_and_position_swap_fee(ref self: ContractState,
position: Position) -> (u128, u128) {
    let (amount0, amount1) = self._collect_position_swap_fee(position);
    self._collect_performance_fee(amount0, amount1);
    (amount0, amount1)
}

fn _collect_all_swap_fee(ref self: ContractState) -> (u128, u128) {
    // ...
    loop {
```

```
        if i == position_length {
            break;
        }
        let position = self.positions.read(i);
        let (amount0, amount1) = self._collect_position_swap_fee(position);
        // ...
    };
    self._collect_performance_fee(total0, total1);
    (total0, total1)
}
```

L-13: Inconsistent retrieval of total shares

Impact: Reduced maintainability

Description

The contract extracts the total amount of shares using `self.erc20.total_supply()` in most of the functions. In the function `_collect_management_fee` the total amount of shares is instead retrieved by calling `self.total_supply()`.

While it does not hurt code functionality, it does make the code less readable and maintainable. It is good practice to maintain consistency and call component functions by referencing the component storage binding.

Recommended Mitigation Steps

Convert the call to `self.erc20.total_supply()` in the `_collect_management_fee` function to align it with the rest of the contract.

L-14: Incorrect constant variable naming convention

Impact: Reduced readability

Description

The `VaultUtils` module defines to constant values: `q96` and `q128`. The convention is to define a constant variable with an all-caps notation.

Recommended Mitigation Steps

Rename the constant variables to `Q96` and `Q128`.

L-15: Code duplication of an internal function

Impact: Reduced maintainability

Description

The `VaultUtils` module defines the function `get_amounts_for_liquidity` which wraps the JediSwap function of the same name.

Then the `position_info` function rewrites the same wrapper by calling `LiquidityAmounts::get_amounts_for_liquidity` directly instead of reusing the wrapper function which adds boilerplate to the `position_info` function and reduces maintainability.

Recommended Mitigation Steps

Reuse the internal `VaultUtils::get_amounts_for_liquidity` function and remove the initialization of the `sqrt_price_x96` in the `position_info` function.