Ginger Security

Smart Contract Security Report

for

Teahouse Finance

(Post Update Security Review)

Prepared By: Jonathan S & Ron S

Ginger Security

11 April, 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	08.04.2024
Audit End Date	09.04.2024
Repository	TeaVaultJediV2
Commit Hash	99b7a199791c83a16c94f5687060146c36343515
Final Commit Hash (Post Mitigation)	6446f93cba248ebcb9466c22377ba18c2e8bc07f
Documentation	https://docs.teahouse.finance
Final Report Date (Post Mitigation)	11.04.2024
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.

In April 2024, the Tea House team made some minor modifications to the previously audited vault and requested Ginger Security to review the changes to ensure they don't introduce any security vulnerabilities to the code. Issues Identified during the security review:

Low Severity - 4

Scope

The scope of the project is the following smart contracts:

Filename	nCLOC (Cairo Lines of Code)
src/tea_vault_jedi_v2.cairo	1132
src/libraries/vault_utils.cairo	132
Total	1264

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

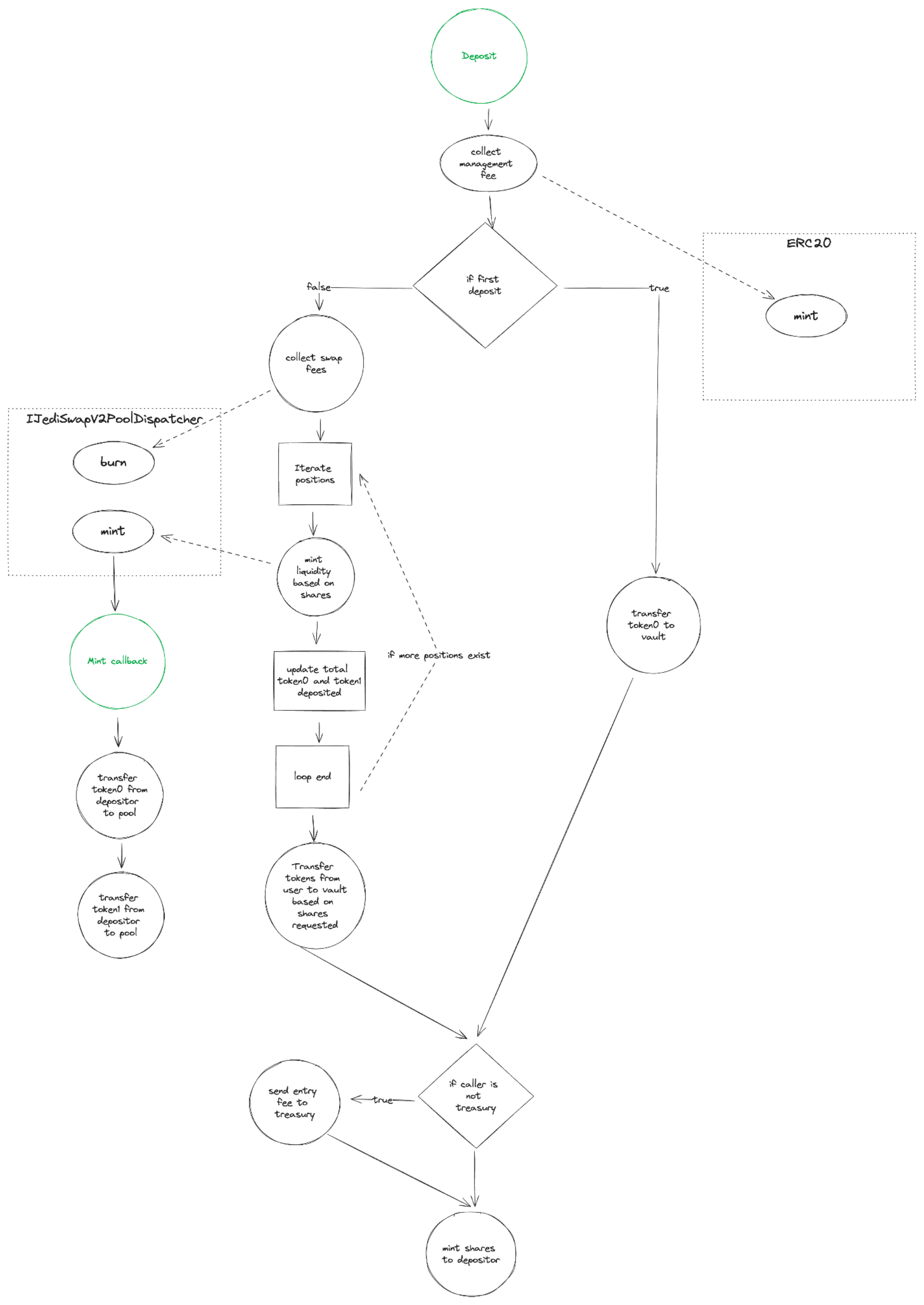
Users can participate in the vault by using the external deposit and withdraw functions.

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.

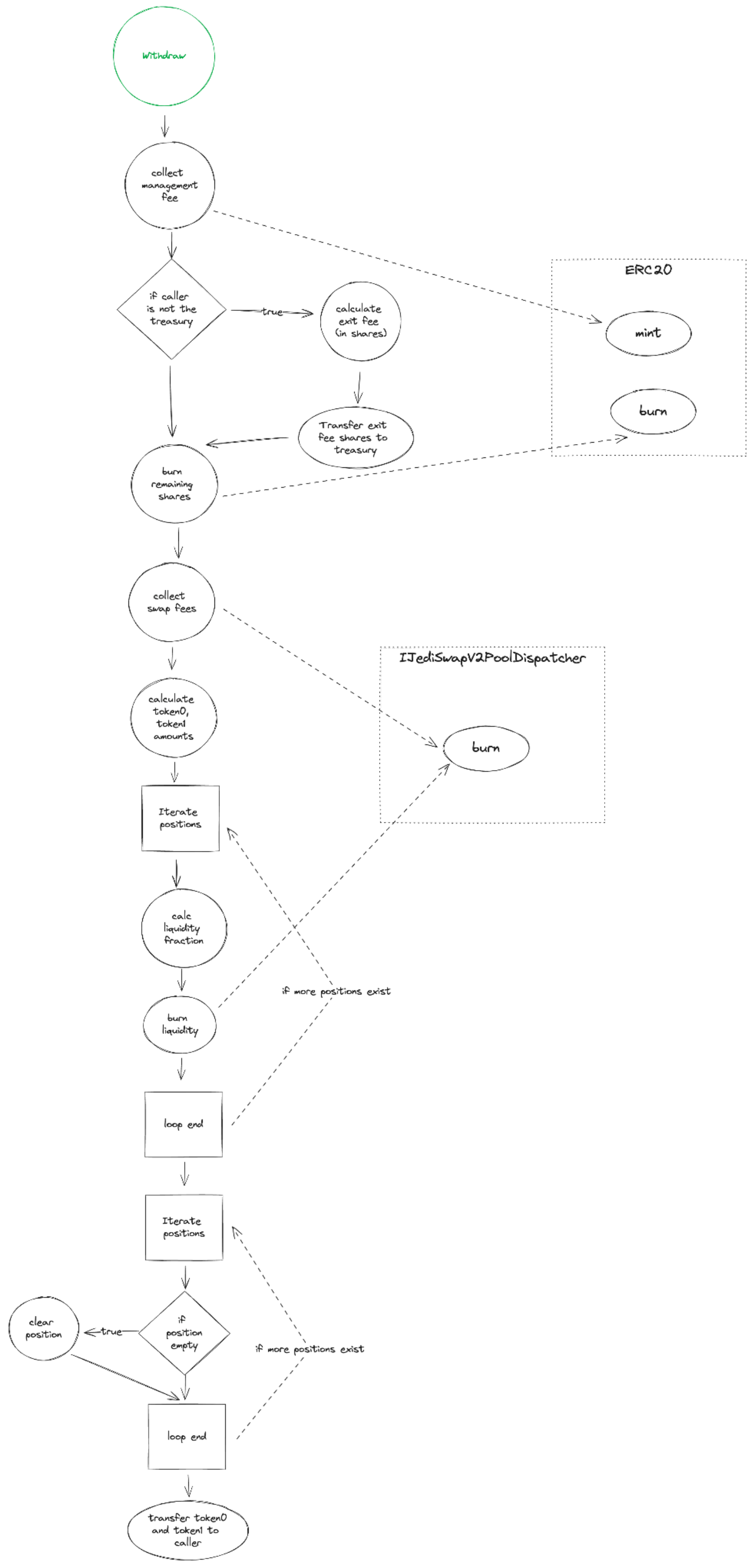
Below is a diagram illustrating the high-level logic of the `deposit` function:


Deposit
collect management fee
ERC20
mint
if first deposit
false
true
collect swap fees
IJediSwapV2PoolDispatcher
burn
mint
Iterate positions
mint liquidity based on shares
update total token0 and token1 deposited
loop end
if more positions exist
transfer token0 to vault
Mint callback
transfer token0 from depositor to pool
transfer token1 from depositor to pool
Transfer tokens from user to vault based on shares requested
if caller is not treasury
true
send entry fee to treasury
mint shares to depositor

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.

Changes Overview

Here are the main changes that have been made to the protocol since our last security report:

Cairo Version Update

- Updated Cairo version from `2.4.0` to `2.6.3`.
- Using the two-step ownership transfer from OpenZeppelin instead of creating a cloned one.

ERC20 Token Updates

- Changed the name and symbol of ERC20 from `felt252` to `ByteArray`.

Integer Update for YAS

- Removed the YAS integer due to bugs reported by the JediSwap team.
- Using the integer that was implemented by the JediSwap team instead.

Error Handling in Alexandria Storage

- Implemented error handling due to warnings from the newer compiler version showing `must_use`.

Temporary Fixes for Jedi Core and Periphery

- Set the branch to `5-bug-fixes` temporarily.
- Will revert back to "main" after the merge.

Temporary Commit Fix for Alexandria Storage

- Set the commit to `bbc010b` temporarily
- A tag will be used instead after the next compatible release since the latest release doesn't support Cairo `2.6.3`.

Starknet DeFi Spring Campaign

- Added a `claim_reward()` function.
- Provided the reward claim contract.
- The process involves using proof to claim rewards, withdrawing and converting them to one of the underlying assets, and then donating back (compounding) to the vault.

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
L-01	Low	Can’t harvest rewards in some cases	Fixed
L-02	Low	Constant calculation can be pre-computed	Fixed
L-03	Low	Unindicative parameter name for internal function	Fixed
L-04	Low	Redundant additional read from storage	Fixed

Finding Details

L-01: Can’t harvest rewards in some cases

Impact

Possible loss of rewards in cases where the reward token is the same as one of the tokens in the vault strategy pair.

Description

In the new `claim_reward()` function, there's an assertion that restricts the address of the reward token. This limitation prevents the owner from sending either `token0` or `token1` which are managed by the vault, under the assumption that these tokens can never be a reward token.

According to the information in this post, the rewards for the StarkNet spring will be emitted as STRK tokens and some of the pairs that will be incentives are pairs that include the STRK token. The Tea House team could deploy a vault optimized for a liquidity provisioning strategy in JediSwap V2, where one of the tokens involved could be STRK (e.g., for pairs like STRK/USDC or STRK/ETH). If these pairs receive extra rewards, the vault owner won't be able to claim the extra STRK token rewards due to the mentioned assertion, which will prevent this action and revert the transaction.

Recommended Mitigation Steps

To claim the DeFi spring rewards in any scenario, we recommend removing the assertion that restricts the reward token from being either token0 or token1.

L-02: Constant calculation can be pre-computed

Impact

Gas Optimization

Description

The `_collect_management_fee` internal function contains a calculation wherein two constant values are multiplied.

```
let fee_multiplier_a_year: u256 = Constants::FEE_MULTIPLIER *
Constants::SECONDS_IN_A_YEAR;
```

This calculation can be defined as a constant since it relies entirely on other constants.

Recommended Mitigation Steps

Define the result of this calculation as a constant along with the other constants.

L-03: Unindicative parameter name for internal function

Impact

Readability and Maintainability

Description

The `_find_position_by_ticks` function has two modes of operation depending on the value of the `asset_not_found` parameter.

When `asset_not_found` is `false` the function will raise an error of `Errors::POSITION_DOES_NOT_EXIST` if the position is not found, and will always return `true` in the first member of the returned tuple.

When `asset_not_found` is `true`, the function does not throw an error but instead returns the result of the lookup in the first member of the returned tuple.

There are two issues here:

1. The parameter name of `asset_not_found` does not properly reflect the purpose of this flag
2. When the function is used, the return value is treated differently depending on the value of the passed parameter

Recommended Mitigation Steps

1. Change the name of the parameter to be more indicative of its purpose (e.g. `raise_error_if_not_found`)
2. Write two wrapper functions to reduce the burden on the caller of knowing the internals of this function
    a. `_find_position_by_ticks_if_exists`, which will simply call `_find_position_by_ticks` with `asset_not_found` set to false
        i. Alternative function name: `_ensure_position_and_find_by_ticks`
    b. `_get_position_by_ticks` which will only return a tuple of the position and the index without the boolean indicating whether the position is found. It should simply call `_find_position_by_ticks` with `asset_not_found` set to true and recreate the returned tuple to include only the last two members.

i. Alternative function names: `_try_find_position_by_ticks`, `_find_position_by_ticks_safely`

L-04: Redundant additional read from storage

Impact

Gas Optimization

Description

The `add_liquidity` function uses the `_find_position_by_ticks` function to search for a position. After a position is found, the returned position parameter is ignored.

The code later uses the index to retrieve the position object again. This is redundant as the object retrieved from `_find_position_by_ticks` can be used in instead.

Recommended Mitigation Steps

Don’t ignore the third member of the return value from `_find_position_by_ticks` and remove the redeclaration of the position.