Infinitee Vault

Smart Contract Audit Report
Prepared for Infinitee Finance

Date Issued:	Jul 07, 2021
Project ID:	AUDIT2021007
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2021007
Version	v1.0
Client	Infinitee Finance
Project	Infinitee Vault
Auditor(s)	Weerawat Pawanawiwat Pongsakorn Sommalai Suvicha Buakhom
Author	Suvicha Buakhom
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jul 07, 2021	Full report	Suvicha Buakhom

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary **1**

1.1. Audit Result 1

1.2. Disclaimer 1

2. Project Overview **2**

2.1. Project Introduction 2

2.2. Scope 2

3. Methodology **4**

3.1. Test Categories 4

3.2. Audit Items 5

3.3. Risk Rating 6

4. Summary of Findings **7**

5. Detailed Findings Information **9**

5.1. Improper Withdrawal Logic on userEmergencyWithdraw() Function 9

5.2. Potential Centralized Control of State Variable 11

5.3. Transaction Ordering Dependence 13

5.4. Abuse of Reward Using Flashloan Attack 17

5.5. Dangerous Approval to External Contract 20

5.6. Design Flaw in emergencyWithdraw() Function of MasterChefWorker 23

5.7. Improper Kill-Switch Mechanism in MasterChefWorker 25

5.8. Improper Migration of Funds 27

5.9. Conflicting Permission 29

5.10. Improper Logic in claimReward() Function 31

5.11. Missing Input Validation 33

5.12. Use of Data From Multiple Sources 36

5.13. Improper Function Visibility 40

5.14. Inexplicit Solidity Compiler Version 43

5.15. Outdated Solidity Compiler Version 44

5.16. Unnecessary Function Declaration 45

6. Appendix **46**

6.1. About Inspex 46

6.2. References 47

1. Executive Summary

As requested by Infinitee Finance, Inspex team conducted an audit to verify the security posture of the Infinitee Vault smart contracts between Jun 23, 2021 and Jun 25, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Infinitee Vault smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found, and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 critical, 2 high, 5 medium, 3 low, 1 very low, and 4 info-severity issues. With the project team's prompt response, 1 critical, 2 high, 5 medium, 3 low, 1 very low, and 1 info-severity issues were resolved in the reassessment, while 3 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that Infinitee Vault smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

This smart contract passes Inspex's security verification standard, and is trustworthy.

Approved by Inspex on Jul 7, 2021

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inpex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Infinitee Vault is designed to integrate with other yield farming platforms. Users can stake a token to the vault, the vault would collectively stake the users' token to the integrated farm and periodically swap the yield farming reward harvested to another token specified in the contract.

Scope Information:

Project Name	Infinitee Vault
Website	https://infinitee.finance/vaults
Smart Contract Type	Ethereum Smart Contract
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jun 23, 2021 - Jun 25, 2021
Reassessment Date	Jul 6, 2021

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit (Commit: a73f9333dccbdbb043d06b28cf5c7713b67c8765):

Name	Location (URL)
InfiniteeFeeManager.sol	https://github.com/infiniteefinance/vault/blob/a73f9333dccbdbb043d06b28cf5c7713b67c8765/contracts/vault/InfiniteeFeeManager.sol
InfiniteeVault.sol	https://github.com/infiniteefinance/vault/blob/a73f9333dccbdbb043d06b28cf5c7713b67c8765/contracts/vault/InfiniteeVault.sol
MasterChefWithVaultWorker.sol	https://github.com/infiniteefinance/vault/blob/a73f9333dccbdbb043d06b28cf5c7713b67c8765/contracts/vault/MasterChefWithVaultWorker.sol
MasterChefWorker.sol	https://github.com/infiniteefinance/vault/blob/a73f9333dccbdbb043d06b28cf5c7713b67c8765/contracts/vault/MasterChefWorker.sol
Timelock.sol	https://github.com/infiniteefinance/vault/blob/a73f9333dccbdbb043d06b28cf5c7713b67c8765/contracts/timelock/Timelock.sol

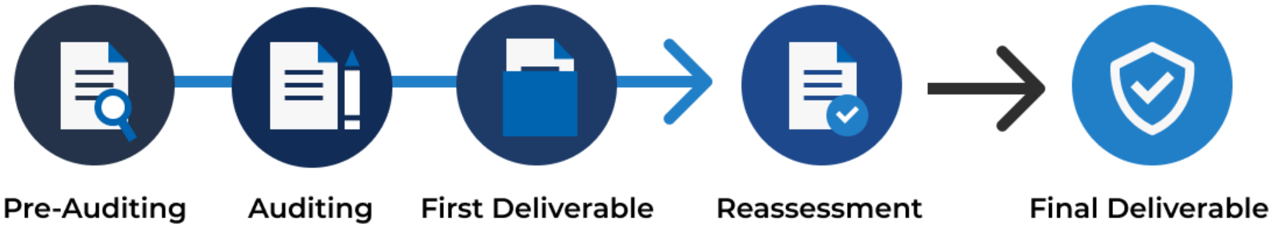
Reassessment (Commit: be42b0fa7a71f64cc8ef855af911c9bd95ff68ff):

Name	Location (URL)
InfiniteeFeeManager.sol	https://github.com/infiniteefinance/vault/blob/be42b0fa7a71f64cc8ef855af911c9bd95ff68ff/contracts/vault/InfiniteeFeeManager.sol
InfiniteeVault.sol	https://github.com/infiniteefinance/vault/blob/be42b0fa7a71f64cc8ef855af911c9bd95ff68ff/contracts/vault/InfiniteeVault.sol
MasterChefWithVaultWorker.sol	https://github.com/infiniteefinance/vault/blob/be42b0fa7a71f64cc8ef855af911c9bd95ff68ff/contracts/vault/MasterChefWithVaultWorker.sol
MasterChefWorker.sol	https://github.com/infiniteefinance/vault/blob/be42b0fa7a71f64cc8ef855af911c9bd95ff68ff/contracts/vault/MasterChefWorker.sol
Timelock.sol	https://github.com/infiniteefinance/vault/blob/be42b0fa7a71f64cc8ef855af911c9bd95ff68ff/contracts/timelock/Timelock.sol

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

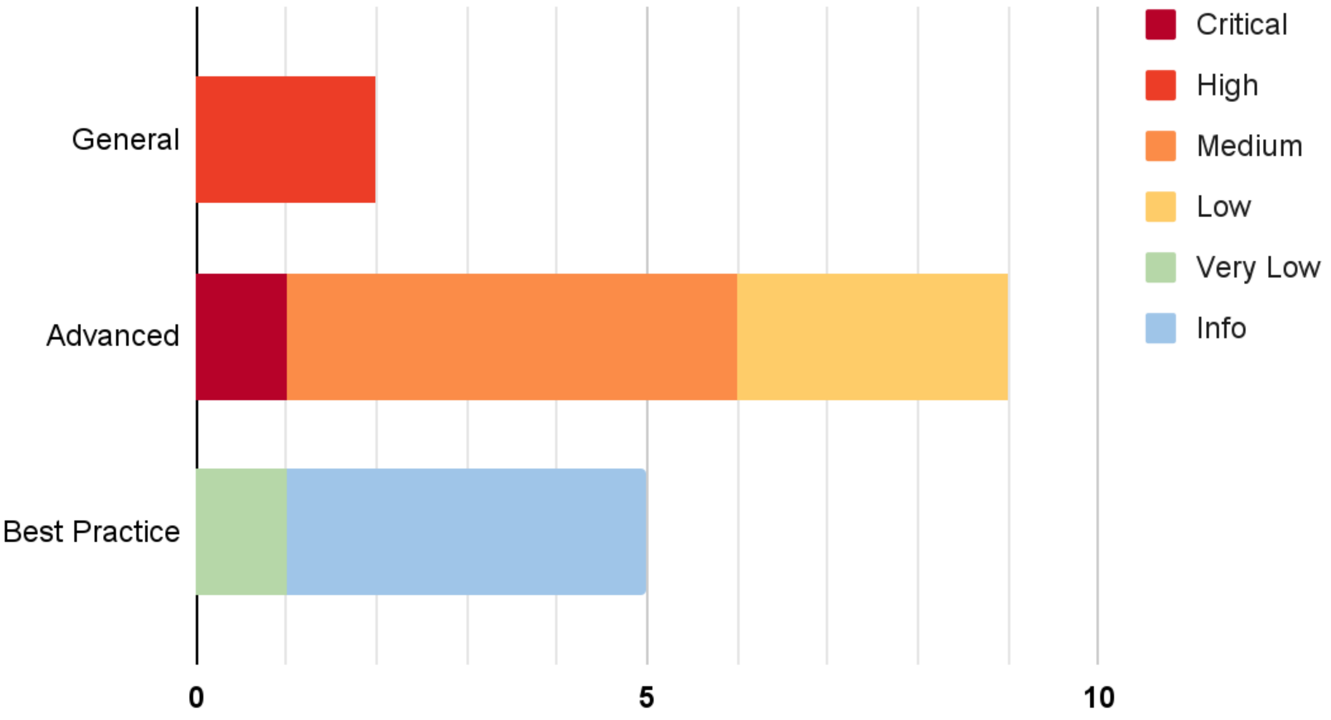
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

4. Summary of Findings

From the assessments, Inspex has found 16 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complication.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Improper Withdrawal Logic on userEmergencyWithdraw() Function	Advanced	Critical	Resolved
IDX-002	Potential Centralized Control of State Variable	General	High	Resolved
IDX-003	Transaction Ordering Dependence	General	High	Resolved
IDX-004	Abuse of Reward Using Flashloan Attack	Advanced	Medium	Resolved
IDX-005	Dangerous Approval to External Contract	Advanced	Medium	Resolved
IDX-006	Design Flaw in emergencyWithdraw() Function of MasterChefWorker	Advanced	Medium	Resolved
IDX-007	Improper Kill-Switch Mechanism in MasterChefWorker	Advanced	Medium	Resolved
IDX-008	Improper Migration of Funds	Advanced	Medium	Resolved
IDX-009	Conflicting Permission	Advanced	Low	Resolved
IDX-010	Improper Logic in claimReward() Function	Advanced	Low	Resolved
IDX-011	Missing Input Validation	Advanced	Low	Resolved
IDX-012	Use of Data From Multiple Sources	Best Practice	Very Low	Resolved
IDX-013	Improper Function Visibility	Best Practice	Info	No Security Impact
IDX-014	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-015	Outdated Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-016	Unnecessary Function Declaration	Best Practice	Info	Resolved

* The mitigations or clarifications by Infinitee Finance can be found in section 5.

5. Detailed Findings Information

5.1. Improper Withdrawal Logic on userEmergencyWithdraw() Function

ID	IDX-001
Target	InfiniteeVault.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** Users can withdraw the whole balance of farmToken in the InfiniteeVault contract. **Likelihood: High** Any user that has staked tokens into the contract can call the userEmergencyWithdraw() function to drain the funds in the contract.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.1.1. Description

When users deposit farmToken to the contract through a deposit() function, user.amount value is increased by the amount deposited and ERC20 share token amount is increased by the _mint() function.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    worker.work();
    claimRewardAndPayFee();
    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }
    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);
    _mint(msg.sender, _amount);
    emit Deposit(msg.sender, _amount);
}
```

The `userEmergencyWithdraw()` function allows users to withdraw `farmToken` from the `InfiniteeVault` contract.

InfiniteeVault.sol

```
function userEmergencyWithdraw() external {
    uint256 amount = userInfos[msg.sender].amount;
    if (amount > 0) {
        IERC20(farmToken()).safeTransfer(msg.sender, amount);
    }
}
```

However, withdrawal through the `userEmergencyWithdraw()` function does not reduce `user.amount` and burn the `ERC20` share token.

As a result, users can withdraw the whole amount of `farmToken` in the `InfiniteeVault` contract.

5.1.2. Recommendation

Inspex recommends burning all `ERC20` share tokens and deducting the `user.amount` to 0 in `userEmergencyWithdraw()` function.

InfiniteeVault.sol

```
function userEmergencyWithdraw() external {
    UserInfo storage user = userInfos[msg.sender];
    uint256 amount = user.amount;
    if (amount > 0) {
        _burn(msg.sender, amount);
        user.amount = 0;
        IERC20(farmToken()).safeTransfer(msg.sender, amount);
    }
}
```

5.2. Potential Centralized Control of State Variable

ID	IDX-002
Target	InfiniteeFeeManager.sol InfiniteeVault.sol MasterChefWorker.sol MasterChefWithVaultWorker.sol
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity: High** **Impact: High** The controlling authorities can potentially change the critical state variables to drain all staked tokens. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved Infinitee Finance team has resolved this issue by implementing a timelock over the contracts deployed at following addresses: - `InfiniteeFeeManager`: 0x8a24b159d3eca84f2b991ed1d341cc3588884053 - `InfiniteeVault(1)`: 0x1B26b9a757B223b9f23997261cB4191122569452 - `MasterChefWithVaultWorker`: 0xfEA88aC042eFe36f25477447538ef861543B59C8 - `InfiniteeVault(2)`: 0x3e33A13aBada2950ce12C6161F7eB9B0cE31E4C1 - `MasterChefWorker`: 0x2cca191dC61DB6De52c4450E3EF59aDD7e560d5C The `Timelock` contract can be found at the following address: 0x8a3ac0b917fae02f2f11b394eec67734a09a4078

5.2.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, as the smart contracts are not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users if the owner is not set to `Timelock` contract.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
InfiniteeFeeManager.sol (L:43)	setFeeRateWithGovAmount()	OnlyOwner
InfiniteeFeeManager.sol (L:52)	setFeeRate()	OnlyOwner
InfiniteeVault.sol (L:180)	setWorker()	OnlyOwner
InfiniteeVault.sol (L:185)	setFeeManager()	OnlyOwner
InfiniteeVault.sol (L:190)	setDelayWithdrawalBlock()	OnlyOwner
MasterChefWorker.sol (L:125)	setVault()	OnlyOwner
MasterChefWithVaultWorker.sol (L:199)	setVault()	OnlyOwner

5.2.2. Recommendation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a `Timelock` contract to delay the changes for a reasonable amount of time

5.3. Transaction Ordering Dependence

ID	IDX-003
Target	MasterChefWorker.sol MasterChefWithVaultWorker.sol
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** High **Impact:** Medium The front running attack can be performed, resulting in a bad swapping rate and a lower reward. **Likelihood:** High This attack is not complex and can be done by anyone.
Status	Resolved Infinitee Finance team has resolved this issue by implementing `minOutFromFarm` and `minOutFromFairLaunch` parameters to pass the value from the client as recommended.

5.3.1. Description

The `work()` function of the worker contracts is called every time the `deposit()` or `withdraw()` functions are called by the user.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];

    worker.work();
    claimRewardAndPayFee();
```

InfiniteeVault.sol

```
function withdraw(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    require(user.amount >= _amount, "withdraw: not enough fund!");
    require(block.number >= user.withdrawableBlock, "withdraw: too fast after deposit!");

    worker.work();
    claimRewardAndPayFee();
```

It can also be called by the operator using the `work()` function in the `InfiniteeVault` contract.

InfiniteeVault.sol

```
function work() public override onlyOperator {
    worker.work();
    emit OperatorWork(rewardPerShare);
}
```

The `work()` function in the worker contracts is responsible for collecting the rewards and swapping them to the token defined as `userReward`.

MasterChefWorker.sol

```
function work() external override onlyVault {
    masterChef.deposit(poolId, 0);
    uint256 farmRewardBalance = farmReward.balanceOf(address(this));
    if (farmRewardBalance > 0) {
        uint256 beforeRewardBalance = userReward.balanceOf(address(this));
        router.swapExactTokensForTokens(farmRewardBalance, 0, rewardRoute, address(this), now);
        uint256 rewardBalance = userReward.balanceOf(address(this)).sub(beforeRewardBalance);
```

MasterChefWithVaultWorker.sol

```
function work() external override onlyVault whenNotPaused {
    masterChef.deposit(poolId, 0);
    fairLaunch.withdrawAll(address(this), fairLaunchPoolId);

    uint256 farmRewardBalance = farmReward.balanceOf(address(this));
    uint256 fairLaunchRewardBalance = fairLaunchReward.balanceOf(address(this));

    // Work on selling reward
    if (farmRewardBalance > 0) {
        router.swapExactTokensForTokens(farmRewardBalance, 0, rewardRoute, address(this), now);
    }

    // Work on selling extra reward from fair launch
    if (fairLaunchRewardBalance > 0) {
        router.swapExactTokensForTokens(fairLaunchRewardBalance, 0, fairLaunchRewardRoute, address(this), now);
    }

    uint256 rewardBalance = userReward.balanceOf(address(this));
```

However, as seen in the source code above, the `router.swapExactTokensForTokens()` function is called by setting the `amountOutMin` to 0. Therefore, the front running attack can be performed, resulting in a bad swapping rate and a lower bounty.

5.3.2. Recommendation

The tolerance value (`amountOutMin`) should not be set to 0. Inspex suggests calculating the expected amount out with the token price fetched from the price oracles or passed from the client directly, and setting it to the `amountOutMin` parameter while calling the `router.swapExactTokensForTokens()` function as shown in the following example:

InfiniteeVault.sol

```
function work(bytes calldata data) public override onlyOperator {
    worker.work(data);
    emit OperatorWork(rewardPerShare);
}
```

MasterChefWithVaultWorker.sol

```
function work(bytes calldata data) external override onlyVault whenNotPaused {
    (uint256 minOutFromFarm, uint256 minOutFromFairLaunch) = abi.decode(data, (uint256, uint256));
    masterChef.deposit(poolId, 0);
    fairLaunch.withdrawAll(address(this), fairLaunchPoolId);

    uint256 farmRewardBalance = farmReward.balanceOf(address(this));
    uint256 fairLaunchRewardBalance = fairLaunchReward.balanceOf(address(this));

    // Work on selling reward
    if (farmRewardBalance > 0) {
        router.swapExactTokensForTokens(farmRewardBalance, minOutFromFarm, rewardRoute, address(this), now);
    }

    // Work on selling extra reward from fair launch
    if (fairLaunchRewardBalance > 0) {
        router.swapExactTokensForTokens(fairLaunchRewardBalance, minOutFromFairLaunch, fairLaunchRewardRoute, address(this), now);
    }

    uint256 rewardBalance = userReward.balanceOf(address(this));

    if (rewardBalance > 0) {
        alpacaVault.deposit(rewardBalance);

        uint256 pendingAccrueReward = vaultPendingUpdateAccrueReward();
        pending = pendingAccrueReward;
        currentReward = currentReward.add(pendingAccrueReward);

        // Update vault reward value and reset pending for the next work
        vault.updateVault();
        pending = 0;

        // Work on deposit to vault
    }

    fairLaunch.deposit(address(this), fairLaunchPoolId,
alpacaVault.balanceOf(address(this)));
}
```

5.4. Abuse of Reward Using Flashloan Attack

ID	IDX-004
Target	MasterChefWorker.sol MasterChefWithVaultWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium A part of the pending reward can be claimed by the attacker. **Likelihood:** Medium This attack requires the use of a custom smart contract.
Status	Resolved Infinitee Finance team has resolved this issue by implementing a price oracle to prevent swapping when the price is under manipulation as recommended.

5.4.1. Description

The `work()` function of the worker contracts can be called by the users through the `deposit()` and `withdraw()` functions of the vault contract.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];

    worker.work();
    claimRewardAndPayFee();
```

InfiniteeVault.sol

```
function withdraw(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    require(user.amount >= _amount, "withdraw: not enough fund!");
    require(block.number >= user.withdrawableBlock, "withdraw: too fast after deposit!");

    worker.work();
    claimRewardAndPayFee();
```

The `work()` function harvests the pending rewards and performs swapping using the `router.swapExactTokensForTokens()` function.

MasterChefWorker.sol

```
function work() external override onlyVault {
    masterChef.deposit(poolId, 0);
    uint256 farmRewardBalance = farmReward.balanceOf(address(this));
    if (farmRewardBalance > 0) {
        uint256 beforeRewardBalance = userReward.balanceOf(address(this));
        router.swapExactTokensForTokens(farmRewardBalance, 0, rewardRoute,
address(this), now);
        uint256 rewardBalance =
userReward.balanceOf(address(this)).sub(beforeRewardBalance);
        pending = rewardBalance;
        vault.updateVault();
        pending = 0;
    }
}
```

MasterChefWithVaultWorker.sol

```
function work() external override onlyVault whenNotPaused {
    masterChef.deposit(poolId, 0);
    fairLaunch.withdrawAll(address(this), fairLaunchPoolId);

    uint256 farmRewardBalance = farmReward.balanceOf(address(this));
    uint256 fairLaunchRewardBalance =
fairLaunchReward.balanceOf(address(this));

    // Work on selling reward
    if (farmRewardBalance > 0) {
        router.swapExactTokensForTokens(farmRewardBalance, 0, rewardRoute,
address(this), now);
    }

    // Work on selling extra reward from fair launch
    if (fairLaunchRewardBalance > 0) {
        router.swapExactTokensForTokens(fairLaunchRewardBalance, 0,
fairLaunchRewardRoute, address(this), now);
    }

    uint256 rewardBalance = userReward.balanceOf(address(this));

    if (rewardBalance > 0) {
        alpacaVault.deposit(rewardBalance);

        uint256 pendingAccrueReward = vaultPendingUpdateAccrueReward();
        pending = pendingAccrueReward;
        currentReward = currentReward.add(pendingAccrueReward);

        // Update vault reward value and reset pending for the next work
        vault.updateVault();
        pending = 0;

        // Work on deposit to vault
    }

    fairLaunch.deposit(address(this), fairLaunchPoolId, alpacaVault.balanceOf(address(this)));
}
```

As the `work()` function can be executed by the users at any time, the attacker can use techniques such as flash loan to manipulate the price of the pool to gain profit from the swapping of `farmReward` and `fairLaunchReward`.

5.4.2. Recommendation

Inspex suggests implementing a mechanism to check the price of the token, such as a price oracle, to prevent the swapping from being done when the price is under manipulation.

5.5. Dangerous Approval to External Contract

ID	IDX-005
Target	MasterChefWithVaultWorker.sol MasterChefWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Medium **Impact:** Medium The external contract can steal all approved tokens from the worker contract. However, only reward tokens are usually stored in the worker contract. **Likelihood:** Medium It is unlikely that the external contract specifically defined by the owner will steal the tokens from the worker contract.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.5.1. Description

The `constructor()` function of `MasterChefWithVaultWorker` and `MasterChefWorker` contracts call the `_giveAllowances()` function.

MasterChefWithVaultWorker.sol

```
constructor(
    IERC20 _farmToken,
    IERC20 _farmRewardToken,
    IERC20 _userRewardToken,
    IERC20 _fairLaunchRewardToken,
    IAlpacaVault _alpacaVault,
    IFairLaunch _fairLaunch,
    IUniswapRouterETH _router,
    IMasterChef _masterChef,
    uint256 _poolId,
    uint256 _fairLaunchPoolId,
    address[] memory _rewardRoute,
    address[] memory _fairLaunchRewardRoute
) public {
    farm = _farmToken;
    farmReward = _farmRewardToken;
    userReward = _userRewardToken;
    fairLaunchReward = _fairLaunchRewardToken;
    alpacaVault = _alpacaVault;
    fairLaunch = _fairLaunch;
    router = _router;
    masterChef = _masterChef;
    poolId = _poolId;
    fairLaunchPoolId = _fairLaunchPoolId;
    rewardRoute = _rewardRoute;
    fairLaunchRewardRoute = _fairLaunchRewardRoute;

    _giveAllowances();
}
```

MasterChefWorker.sol

```
constructor(
    IERC20 _farmToken,
    IERC20 _farmRewardToken,
    IERC20 _userRewardToken,
    IUniswapRouterETH _router,
    IMasterChef _masterChef,
    uint256 _poolId,
    address[] memory _rewardRoute
) public {
    farm = _farmToken;
    farmReward = _farmRewardToken;
    userReward = _userRewardToken;
    router = _router;
    masterChef = _masterChef;
    poolId = _poolId;
    rewardRoute = _rewardRoute;

    _giveAllowances();
}
```

In the `_giveAllowances()` function, all tokens used in the contracts are approved to external contracts for the maximum number of uint256 as shown below.

MasterChefWithVaultWorker.sol

```
function _giveAllowances() internal {
    IERC20(farm).safeApprove(address(masterChef), uint256(-1));
    IERC20(farmReward).safeApprove(address(router), uint256(-1));
    IERC20(fairLaunchReward).safeApprove(address(router), uint256(-1));
    IERC20(userReward).safeApprove(address(alpacaVault), uint256(-1));
    IERC20(alpacaVault).safeApprove(address(alpacaVault), uint256(-1));
    IERC20(alpacaVault).safeApprove(address(fairLaunch), uint256(-1));
}
```

MasterChefWorker.sol

```
function _giveAllowances() internal {
    IERC20(farm).safeApprove(address(masterChef), uint256(-1));
    IERC20(farmReward).safeApprove(address(router), uint256(-1));
}
```

By approving an arbitrary number of allowance to external contracts, the external contracts can always steal all approved tokens from the worker contracts.

5.5.2. Recommendation

Inspex suggests removing `_giveAllowances()` function, approving only necessary number of allowance to the external contract, and revoking them after the process has finished, for example:

```
function work() external override onlyVault {
    masterChef.deposit(poolId, 0);

    uint256 farmRewardBalance = farmReward.balanceOf(address(this));

    if (farmRewardBalance > 0) {
        uint256 beforeRewardBalance = userReward.balanceOf(address(this));
        IERC20(farmReward).safeApprove(address(router), farmRewardBalance);
        router.swapExactTokensForTokens(farmRewardBalance, 0, rewardRoute,
address(this), now);
        IERC20(farmReward).safeApprove(address(router), 0);
        uint256 rewardBalance =
userReward.balanceOf(address(this)).sub(beforeRewardBalance);
        pending = rewardBalance;
        vault.updateVault();
        pending = 0;
    }
}
```

Please note that in the example, the remediations of other issues are not yet applied.

5.6. Design Flaw in emergencyWithdraw() Function of MasterChefWorker

ID	IDX-006
Target	MasterChefWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High All staked tokens will be stuck in the `MasterChefWorker` contract, so the stuck tokens cannot be withdrawn using the `userEmergencyWithdraw()` function. **Likelihood:** Low It is very rare that the `emergencyWithdraw()` function will be executed.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.6.1. Description

By design, the users will be able to perform `InfiniteeVault.userEmergencyWithdraw()` if and only if the owner has executed the `InfiniteeVault.emergencyWithdraw()` function.

By executing the `InfiniteeVault.userEmergencyWithdraw()` function, the user's staked tokens in the `InfiniteeVault` contract will be transferred back to the users as follows:

InfiniteeVault.sol

```
function userEmergencyWithdraw() external {
    uint256 amount = userInfos[msg.sender].amount;
    if (amount > 0) {
        IERC20(farmToken()).safeTransfer(msg.sender, amount);
    }
}
```

Therefore, when the `emergencyWithdraw()` function is executed, the worker must transfer all staked tokens to the `InfiniteeVault` contract. Thus, the users will be able to withdraw their staked tokens by executing the `userEmergencyWithdraw()` function.

However, after executing the `emergencyWithdraw()` function of `MasterChef` contract, the `emergencyWithdraw()` function of `MasterChefWorker` contract does not transfer all staked tokens to the `InfiniteeVault` contract as shown below.

MasterChefWorker.sol

```
function emergencyWithdraw() external override onlyOwner {
    masterChef.emergencyWithdraw(poolId);
}
```

As a result, all staked tokens will be stuck in the MasterChefWorker contract, so the `userEmergencyWithdraw()` function cannot be used to withdraw the stuck tokens.

5.6.2. Recommendation

Inspex suggests transferring all staked token back to the `InfiniteeVault` contract as shown in the following example:

MasterChefWorker.sol

```
function emergencyWithdraw() external override onlyOwner {
    masterChef.emergencyWithdraw(poolId);
    farm.safeTransfer(address(vault), farm.balanceOf(address(this)));
}
```

Please note that in the example, the remediations of other issues are not yet applied.

5.7. Improper Kill-Switch Mechanism in MasterChefWorker

ID	IDX-007
Target	MasterChefWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** High If an attack happens when the contract is unpausable, further damage cannot be prevented. **Likelihood:** Low It is unlikely for the `pause()` function to be required.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.7.1. Description

Immutability is one of the core principles of the blockchain. If the contract is designed to be non-upgradable, there is no mechanism to prevent contracts from potential failures.

For example, when the `MasterChefWorker` contract is deployed, there is no mechanism to stop the contract from being used when new issues are found.

MasterChefWorker.sol

```
function deposit() external override onlyVault {
    uint256 balance = farm.balanceOf(address(this));

    if (balance > 0) {
        masterChef.deposit(poolId, balance);
    }
}
```

5.7.2. Recommendation

Inspex recommends using the emergency stop pattern to protect the contract from potential failures.

In this case, it is recommended to inherit the `Pauseable` abstraction contract of OpenZeppelin to the `MasterChefWorker` contract as follows:

MasterChefWorker.sol

```
contract MasterChefWorker is YieldWorker, Ownable, Pauseable {
```

Then, implement the `pause()` and `unpause()` functions as shown below:

MasterChefWorker.sol

```
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

Finally, add the `whenNotPaused` modifier to critical external functions, for example:

MasterChefWorker.sol

```
function deposit() external override onlyVault whenNotPaused {
    uint256 balance = farm.balanceOf(address(this));

    if (balance > 0) {
        masterChef.deposit(poolId, balance);
    }
}
```

Please note that in the example, the remediations of other issues are not yet applied.

5.8. Improper Migration of Funds

ID	IDX-008
Target	InfiniteeVault.sol MasterChefWorker.sol MasterChefWithVaultWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The funds of the users can be stuck in the original smart contracts. **Likelihood:** Low It is unlikely for these functions to be called multiple times.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.8.1. Description

The `setWorker()` function can be used to set the address of the worker in the vault contract. The worker is responsible for managing the depositing and withdrawing of tokens.

InfiniteeVault.sol

```
function setWorker(YieldWorker _worker) public onlyOwner {
    worker = _worker;
    emit WorkerChanged(address(_worker));
}
```

Likewise, the `setVault()` function can be used to set the address of the vault in the worker contract. Vault is the contract recording the amount of tokens deposited by the users and responsible for distributing the reward.

MasterChefWorker.sol

```
function setVault(address _vault) external onlyOwner {
    vault = Vault(_vault);
}
```

MasterChefWithVaultWorker.sol

```
function setVault(address _vault) external onlyOwner {
    vault = Vault(_vault);
}
```

The `setWorker()` and `setVault()` function is allowed to be called multiple times without migrating the balance deposited to the new contracts, causing the balances to be stuck inside the original smart contracts.

5.8.2. Recommendation

Inspex suggests allowing only one execution of `setWorker()` and `setVault()` function, for example:

InfiniteeVault.sol

```
function setWorker(YieldWorker _worker) public onlyOwner {
    require(worker == address(0), "Worker is already set.");
    worker = _worker;
    emit WorkerChanged(address(_worker));
}
```

MasterChefWorker.sol

```
function setVault(address _vault) external onlyOwner {
    require(address(vault) == address(0), "Vault is already set.");
    vault = Vault(_vault);
}
```

MasterChefWithVaultWorker.sol

```
function setVault(address _vault) external onlyOwner {
    require(address(vault) == address(0), "Vault is already set.");
    vault = Vault(_vault);
}
```

5.9. Conflicting Permission

ID	IDX-009
Target	MasterChefWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The `emergencyWithdraw()` function cannot be called from the vault contract, but can still be called directly. **Likelihood:** Low The function is only used in an emergency situation.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.9.1. Description

The `emergencyWithdrawWorker()` function in the `InfiniteeVault` contract calls the `emergencyWithdraw()` function in the worker contract.

InfiniteeVault.sol

```
function emergencyWithdrawWorker() external onlyOwner {
    worker.emergencyWithdraw();
}
```

However, the `emergencyWithdraw()` function in the `MasterChefWorker` contract has the `onlyOwner` modifier.

MasterChefWorker.sol

```
function emergencyWithdraw() external override onlyOwner {
    masterChef.emergencyWithdraw(poolId);
}
```

Per the business design discussed with the Infinee team, the owner of the `MasterChefWorker` is not the `InfiniteeVault` contract; therefore, the `emergencyWithdrawWorker()` function is unusable.

5.9.2. Recommendation

Inspex suggests changing the function modifier from `onlyOwner` to `onlyVault`, for example:

MasterChefWorker.sol

```
function emergencyWithdraw() external override onlyVault {
    masterChef.emergencyWithdraw(poolId);
}
```

5.10. Improper Logic in claimReward() Function

ID	IDX-010
Target	MasterChefWithVaultWorker.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Low **Impact:** Medium The users cannot claim their reward. **Likelihood:** Low It is unlikely that the amount of reward token stored in the worker contract is greater than the claimed amount.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.10.1. Description

In the `claimReward()` function, if the remaining amount of reward token stored in the worker contract is fewer than the claiming amount, it will withdraw the reward token from the Alpaca vault as shown below.

MasterChefWithVaultWorker.sol

```
function claimReward(uint256 _amount) external override onlyVault whenNotPaused {
    if (_amount > 0) {
        uint256 workerBalance = userReward.balanceOf(address(this));
        uint256 share = vaultTokenAmountToShare(_amount.sub(workerBalance));
        fairLaunch.withdraw(address(this), fairLaunchPoolId, share);
        alpacaVault.withdraw(share);
        currentReward = currentReward.sub(_amount);
        userReward.safeTransfer(msg.sender, _amount);
    }
}
```

If the remaining amount of reward token stored in the worker contract is greater than the claiming amount, the transaction will be reverted because the subtraction overflow protection is triggered.

5.10.2. Recommendation

Inspex suggests withdrawing the reward tokens from the Alpaca vault only if the reward tokens stored in the worker contract are not enough as shown in the following example:

MasterChefWithVaultWorker.sol

```
function claimReward(uint256 _amount) external override onlyVault whenNotPaused
{
    if (_amount > 0) {
      uint256 workerBalance = userReward.balanceOf(address(this));
      if (_amount > workerBalance) {
          uint256 share = vaultTokenAmountToShare(_amount.sub(workerBalance));
          fairLaunch.withdraw(address(this), fairLaunchPoolId, share);
          alpacaVault.withdraw(share);
      }
      currentReward = currentReward.sub(_amount);
      userReward.safeTransfer(msg.sender, _amount);
    }
}
```

5.11. Missing Input Validation

ID	IDX-011
Target	InfiniteeVault.sol
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Low **Impact:** Medium The users cannot withdraw the staked tokens from the `InfiniteeVault` contract. **Likelihood:** Low It is very unlikely that the owner will set an improperly lengthy delay because there is no benefit in performing this action.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.11.1. Description

The `setDelayWithdrawalBlock()` function can be used to set the delay as the number of blocks before the user can withdraw after depositing.

InfiniteeVault.sol

```
function setDelayWithdrawalBlock(uint256 _delay) external onlyOwner {
    delayWithdrawalBlock = _delay;
}
```

It is used in the deposit function to determine the block number to be reached before the user can withdraw.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];

    worker.work();
    claimRewardAndPayFee();

    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }

    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);

    _mint(msg.sender, _amount);

    emit Deposit(msg.sender, _amount);
}
```

On the withdrawal, if the block number has not reached `user.withdrawableBlock`, the user cannot withdraw from the vault.

InfiniteeVault.sol

```
function withdraw(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    require(user.amount >= _amount, "withdraw: not enough fund!");
    require(block.number >= user.withdrawableBlock, "withdraw: too fast after deposit!");

    worker.work();
    claimRewardAndPayFee();

    if (_amount > 0) {
        uint256 balance = balanceOf(msg.sender);
        require(balance >= _amount, "withdraw: not enough token!");

        _burn(msg.sender, _amount);
        user.amount = user.amount.sub(_amount);
    }

    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);

    worker.withdraw(_amount);
    IERC20(farmToken()).safeTransfer(msg.sender, _amount);

    emit Withdraw(msg.sender, _amount);
}
```

However, there is no limit of delay in `setDelayWithdrawalBlock()` function, allowing the setting of improperly lengthy delay, making the users unable to withdraw from the smart contract.

5.11.2. Recommendation

Inspex suggests setting the upper limit of delay in `setDelayWithdrawalBlock()` function, for example:

InfiniteeVault.sol

```
function setDelayWithdrawalBlock(uint256 _delay) external onlyOwner {
    require(_delay <= MAX_DELAY, "Delay is longer than the limit")
    delayWithdrawalBlock = _delay;
}
```

Please note that the value of the MAX_DELAY variable should be defined in the smart contract as a reasonable amount of time.

5.12. Use of Data From Multiple Sources

ID	IDX-012
Target	InfiniteeVault.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Very Low **Impact:** Low Inconsistency can cause a small amount of reward miscalculation in the smart contract. **Likelihood:** Low It is very unlikely that the total supply will be inconsistent with the user staked amount.
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.12.1. Description

There are 2 sources of data stored to collect the amount of `farmToken` in the contract: the contract storage, and the amount of `ERC20` token minted by the contract.

The contract storage is mainly used to check the amount of the `farmToken` staked.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    worker.work();
    claimRewardAndPayFee();
    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }
    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);
    _mint(msg.sender, _amount);
    emit Deposit(msg.sender, _amount);
}
```

As discussed with the Infinitee team, the token minted will be used to stake in the future. However, the total supply of the contract's `ERC20` token is used to calculate the reward distribution.

InfiniteeVault.sol

```
function totalRewardPerShare() public view override returns (uint256) {
    uint256 _rewardPerShare = rewardPerShare;
    uint256 _pendingReward = pendingReward();
    uint256 _totalSupply = totalSupply();

    if (_pendingReward != 0 && _totalSupply != 0) {
        uint256 _pendingRewardPerShare =
            _pendingReward.mul(1e12).div(_totalSupply);
        _rewardPerShare = _rewardPerShare.add(_pendingRewardPerShare);
    }

    return _rewardPerShare;
}
```

The main usage of tokens minted is not the reward calculation. Therefore, using total supply to calculate the pending reward can cause inconsistency.

5.12.2. Recommendation

Inspex suggests storing the total amount of the contract's `ERC20` token minted in the contract, for example:

InfiniteeVault.sol

```
// The yield worker currently in use by the vault.
YieldWorker public worker;
// Fee Manager for calculate vault fee.
FeeManager public feeManager;
// Reward amount per share.
uint256 public rewardPerShare;
// Total share amount minted
uint256 public totalShare;
// Delay block for withdraw after deposit into vault.
uint256 public delayWithdrawalBlock;
// Info of each user that using vaults.
mapping(address => UserInfo) public userInfos;
// Operator address.
address public operator;
```

InfiniteeVault.sol

```
function totalRewardPerShare() public view override returns (uint256) {
    uint256 _rewardPerShare = rewardPerShare;
    uint256 _pendingReward = pendingReward();
    uint256 _totalSupply = totalShare;

    if (_pendingReward != 0 && _totalSupply != 0) {
        uint256 _pendingRewardPerShare =
            _pendingReward.mul(1e12).div(_totalSupply);
        _rewardPerShare = _rewardPerShare.add(_pendingRewardPerShare);
    }

    return _rewardPerShare;
}
```

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];

    worker.work();
    claimRewardAndPayFee();

    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }

    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);

    _mint(msg.sender, _amount);
    totalShare = totalShare.add(_amount);

    emit Deposit(msg.sender, _amount);
}
```

InfiniteeVault.sol

```
function withdraw(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    require(user.amount >= _amount, "withdraw: not enough fund!");
    require(block.number >= user.withdrawableBlock, "withdraw: too fast after
deposit!");

    worker.work();
    claimRewardAndPayFee();

    if (_amount > 0) {
        uint256 balance = balanceOf(msg.sender);
        require(balance >= _amount, "withdraw: not enough token!");

        _burn(msg.sender, _amount);
        totalShare = totalShare.sub(_amount);
        user.amount = user.amount.sub(_amount);
    }

    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);

    worker.withdraw(_amount);
    IERC20(farmToken()).safeTransfer(msg.sender, _amount);

    emit Withdraw(msg.sender, _amount);
}
```

Please note that in the example, the remediations of other issues are not yet applied.

5.13. Improper Function Visibility

ID	IDX-013
Target	InfiniteeVault.sol MasterChefWithVaultWorker.sol Timelock.sol
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	**No Security Impact** The Infinitee Finance team has acknowledged this issue and resolved this issue only in `InfiniteeVault.sol` and `MasterChefWithVaultWorker.sol`, but not in `Timelock.sol`.

5.13.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `deposit()` function of the `InfiniteeVault` is set to public and it is never called from any internal function.

InfiniteeVault.sol

```
function deposit(uint256 _amount) public override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    worker.work();
    claimRewardAndPayFee();
    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }
    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);
    _mint(msg.sender, _amount);
    emit Deposit(msg.sender, _amount);
}
```

The following table contains all functions that have public visibility and are never called from any internal function.

Target	Function
InfiniteeVault.sol (L:106)	deposit()
InfiniteeVault.sol (L:130)	withdrawAll()
InfiniteeVault.sol (L:159)	work()
InfiniteeVault.sol (L:164)	updateVault()
InfiniteeVault.sol (L:180)	setWorker()
InfiniteeVault.sol (L:185)	setFeeManager()
MasterChefWithVaultWorker.sol (L:203)	pause()
Timelock.sol (L:54)	setDelay()
Timelock.sol (L:63)	acceptAdmin()
Timelock.sol (L:71)	setPendingAdmin()
Timelock.sol (L:84)	queueTransaction()
Timelock.sol (L:95)	cancelTransaction()
Timelock.sol (L:115)	executeTransaction()

5.13.2. Recommendation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

InfiniteeVault.sol

```
function deposit(uint256 _amount) external override nonReentrant {
    UserInfo storage user = userInfos[msg.sender];
    worker.work();
    claimRewardAndPayFee();
    if (_amount > 0) {
        IERC20(farmToken()).safeTransferFrom(
            msg.sender,
            address(worker),
            _amount
        );
        worker.deposit();
        user.amount = user.amount.add(_amount);
        user.withdrawableBlock = block.number.add(delayWithdrawalBlock);
    }
    user.rewardDebt = user.amount.mul(totalRewardPerShare()).div(1e12);
    _mint(msg.sender, _amount);
    emit Deposit(msg.sender, _amount);
}
```

5.14. Inexplicit Solidity Compiler Version

ID	IDX-014
Target	InfiniteeFeeManager.sol InfiniteeVault.sol MasterChefWithVaultWorker.sol MasterChefWorker.sol
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	No Security Impact Infinitee Finance team has acknowledged this issue.

5.14.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

InfiniteeVault.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0 <0.8.0;
```

5.14.2. Recommendation

Inspex suggests fixing the solidity compiler to the latest stable version.

During the audit activity, the latest stable versions of Solidity compiler in each major are as follows:

- Major 0.6: v0.6.12
- Major 0.7: v0.7.6
- Major 0.8: v0.8.6

5.15. Outdated Solidity Compiler Version

ID	IDX-015
Target	Timelock.sol
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	No Security Impact Infinitee Finance team has acknowledged this issue.

5.15.1. Description

The Solidity compiler version specified in the smart contract was outdated. This version has publicly known inherent bugs that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

Timelock.sol

```
pragma solidity 0.6.6;
```

5.15.2. Recommendation

Inspex suggests upgrading the Solidity compiler to the latest stable version.

During the audit activity, the latest stable versions of Solidity compiler in each major are as follows:

- Major 0.6: v0.6.12
- Major 0.7: v0.7.6
- Major 0.8: v0.8.6

5.16. Unnecessary Function Declaration

ID	IDX-016
Target	MasterChefWorker.sol
Category	Smart Contract Best Practice
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact:** None **Likelihood:** None
Status	Resolved Infinitee Finance team has resolved this issue as recommended.

5.16.1. Description

The `_removeAllowances()` function in `MasterChefWorker` is never used in the contract and should be removed for reducing gas used during deployment.

For proof of concept, the following source code shows that the `_removeAllowances()` function of the `InfiniteeVault` is declared as internal visibility.

MasterChefWorker.sol

```
function _removeAllowances() internal {
    IERC20(farm).safeApprove(address(masterChef), 0);
    IERC20(farmReward).safeApprove(address(router), 0);
}
```

5.16.2. Recommendation

Inspex suggests removing unused internal functions if they are not called from any function in the same contract.

In this case, it is recommended to remove `_removeAllowances()` function from the `MasterChefWorker` contract.

6. Appendix

6.1. About Inspex

inspex

CYBERSECURITY
PROFESSIONAL
SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE