

Fist Farm

smart contracts
final audit report

May 2022



hashex.org



contact@hashex.org

Contents

1. Disclaimer	3
2. Overview	4
3. Found issues	6
4. Contracts	8
5. Conclusion	13
Appendix A. Issues' severity classification	14
Appendix B. List of examined issue types	15

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below - please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Fist Farm team to perform an audit of their smart contract. The audit was conducted between 16/05/2022 and 18/05/2022.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The updated code is available at the @alanwales223/FFCFarm GitHub repository after the commit [8779a87](#).

2.1 Summary

Project name	Fist Farm
URL	https://www.fistfarm.io/
Platform	Binance Smart Chain
Language	Solidity

2.2 Contracts

Name	Address
FFCFarmV2	
IStrategy & FFCToken	
OpenZeppelin imports	

3. Found issues



● High	1 (9%)
● Medium	3 (27%)
● Low	2 (18%)
● Info	5 (46%)

C1. FFCFarmV2

ID	Severity	Title	Status
C1-01	● High	Emission rate is not capped	✓ Resolved
C1-02	● Medium	Changing the reward token may block workflow	⌚ Acknowledged
C1-03	● Medium	EmergencyWithdraw may fail	✓ Resolved
C1-04	● Medium	Duplicated pools are allowed	✓ Resolved
C1-05	● Low	Gas optimizations	⚙️ Partially fixed
C1-06	● Low	massUpdate flag is optional	✓ Resolved
C1-07	● Info	Lack of documentation in code	⌚ Acknowledged
C1-08	● Info	General lack of events	✓ Resolved
C1-09	● Info	Inconsistent comments & typos	⚙️ Partially fixed
C1-10	● Info	Max supply of the FFC token may be slightly exceeded	⌚ Acknowledged

C1-11

● Info

Tokens with commissions aren't supported

☑ Acknowledged

4. Contracts

C1. FFCFarmV2

Overview

A typical minter farming contract derived from [MasterChef](#) by SushiSwap, but instead of holding the staked tokens, each pool of FFCFarmV2 contract works with an individual Strategy contract by calling `IStrategy(strat).deposit()` or `IStrategy(strat).withdraw()` methods.

Issues

C1-01 Emission rate is not capped

 High Resolved

Owner can set an arbitrary big value for the `FFCPerBlock` value. In such a case, the token will be devalued causing users to actually lose their rewards. The same goes for the `ownerFFCReward` part.

```
function setRewardPerBlock(uint256 _rewardPerBlock) public onlyOwner{
    massUpdatePools();
    FFCPerBlock = _rewardPerBlock;
}

function setOwnerFFCReward(uint256 _ownerFFCReward) public onlyOwner{
    ownerFFCReward = _ownerFFCReward;
}
```

Recommendation

Consider limiting the new value from above.

C1-02 Changing the reward token may block workflow

● Medium

✔ Acknowledged

Setting the **FFCv2** reward token address to a wrong value by a malicious or hacked owner would cause a blocked reward distribution. Though users can still call **emergencyWithdraw()**, changing the reward token address may lead to unexpected yield and frustration.

Recommendation

Consider setting the **FFCv2** token immutable.

C1-03 EmergencyWithdraw may fail

● Medium

✔ Resolved

The functions **emergencyWithdraw()** calls external contracts that are out of the scope of the current audit. If the strategy contract fails, users won't be able to withdraw their funds even with the **emergencyWithdraw()** function.

```
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    (...)
    IStrategy(poolInfo[_pid].strat).withdraw(msg.sender, amount);
    (...)
}
```

Also, the **emergencyWithdraw()** function does not check the actual amount of the tokens withdrawn from the strategy, and thus may fail if the strategy sends less tokens than has been requested.

Recommendation

Either this emergency function is useless or the external Strategy contract should have its own **emergencyWithdraw()** function implemented.

C1-04 Duplicated pools are allowed

● Medium

✓ Resolved

`add()` function allows the owner to add pools with identical strat addresses. In such a case, these pool's rewards would be calculated incorrectly:

```
function updatePool(uint256 _pid) public {
    (...)
    uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
    pool.accFFCPerShare = pool.accFFCPerShare.add(
        FFCReward.mul(1e12).div(sharesTotal)
    );
    (...)
}
```

Recoomendation

We recommend denying the duplicated pools by requiring `stratAddress[_strat]` to be `false` in `add()` function.

C1-05 Gas optimizations

● Low

🔧 Partially fixed

Allowances for strategies could be increased to max once, upon the pool creation.

`FFCMaxSupply` and `startBlock` variables have no updaters and should be declared constants.

Public functions that are not called within the contract can be declared as `external`.

C1-06 massUpdate flag is optional

● Low

✓ Resolved

`add()` and `set()` functions have an optional `_withUpdate` flag which calls `massUpdatePools()` if set `false` and may cause unfair rewards in case of rarely updated pools, see more info [here](#).

C1-07 Lack of documentation in code

● Info

☑ Acknowledged

Public and external methods in the smart contract lack documentation in the [NatSpec](#) format.

C1-08 General lack of events

● Info

☑ Resolved

Most of the governance **onlyOwner** functions don't emit any specific event, complicating the public control over the contract.

C1-09 Inconsistent comments & typos

● Info

🔄 Partially fixed

L1431 mentions that the default value of **ownerFFCReward** should be 12% instead of 0.

L1435 start block is set to January 2021.

L1564 typo in 'rewawrd'

C1-10 Max supply of the FFC token may be slightly exceeded

● Info

☑ Acknowledged

The contract has a public variable **FFCMaxSupply** for the max token supply. After the total supply of the FFC token reaches or exceeds **FFCMaxSupply**, minting from the FFCFarm stops by setting multiplier for zero:

```
function getMultiplier(uint256 _from, uint256 _to)
    public
    view
    returns (uint256)
{
    if (IERC20(FFCv2).totalSupply() >= FFCMaxSupply) {
        return 0;
    }
    return _to.sub(_from);
}
```

Tokens are minted in the `updatePool()` function that does not check that the sum of the current amount and the amount to mint exceeds `FFCMaxSupply`.

C1-11 Tokens with commissions aren't supported

● Info

☑ Acknowledged

Deposit function doesn't have a check on the actual transferred amount of deposited tokens. The owner must not add pools with tokens with fees on transfers.

```
function deposit(uint256 _pid, uint256 _wantAmt) public nonReentrant {
    (...)
    pool.want.safeTransferFrom(
        address(msg.sender),
        address(this),
        _wantAmt
    );
    (...)
}
```

C2. IStrategy & FFCToken

Overview

Interfaces for external calls. No issues were found.

C3. OpenZeppelin imports

Overview

Ownable, ReentrancyGuard, SafeERC20, EnumerableSet, Address, ERC20, IERC20, SafeMath, and Context contract are imported from the OpenZeppelin's repository. No issues were found, but EnumerableSet is not in use.

5. Conclusion

1 high and 3 medium severity issues were found. 1 high and 2 medium of them were fixed with the update.

The reviewed contracts are highly dependent on the owner's account. Users using the project have to trust the owner and that the owner's account is properly secured. We also recommend adding unit tests with coverage of at least 90% to any introduced functionality. It must be also noted that the audited FFCFarm contract is also highly dependant on the implementations of strategy contracts which are out of scope of the current audit. FFC token reward calculations are made via strategy contracts, user funds are also held on strategy contracts.

This audit includes recommendations on improving the code and preventing potential attacks.

Appendix A. Issues' severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Informational.** Issues that do not impact the contract operation. Usually, informational severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 [@hashex_manager](https://t.me/hashex_manager)

 blog.hashex.org

 [linkedin](https://www.linkedin.com/company/hashex)

 [github](https://github.com/hashex)

 [twitter](https://twitter.com/hashex)

#HashEx
BLOCKCHAIN SECURITY