SECURITY AUDIT OF

THE SINGLEFARM SMART CONTRACTS

Public Report

Jan 24, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jan 24, 2023. We would like to thank the deFarm for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the The SingleFarm Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About The SingleFarm Smart Contracts

deFarm is a new type of Social Finance platform that truly mixes the best of both worlds together. Their aim is to provide a social platform that encourages LPs to meet with Asset Managers, and cooperate with each other to achieve maximum capital efficiency.

Single Farm: A unique one-time type of DeFi vault on deFarm's platform. In Single Farm, a Farm Manager manages a single trade per vault. Investors can participate in a specific trading idea proposed by a Farm Manager and fund the corresponding farm. Once the trade is closed, realized profits or losses are instantly returned to the investors. This approach reduces risks for investors as their capital is dedicated to one trade, and it allows them to access unique trading opportunities proposed by expert Farm Managers.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of The SingleFarm Smart Contracts. It was conducted on commit `f1fe5256ce280e4401410854e21a963c844b2731` from git repository link: *https://github.com/de-farm/smartcontract/tree/features/beta*. The audit was performed on the `features/beta` branch.

The audit scope encompassed the following files:

- single/SingleFarm.sol
- single/SingleFarmFactory.sol
- single/Errors.sol
- utils/Administrable.sol
- utils/Constants.sol
- utils/Errors.sol
- utils/ETHFee.sol
- utils/OperatorManager.sol
- utils/ProtocolInfo.sol
- utils/SupportedDex.sol
- utils/VertexHandler.sol
- utils/WhitelistedTokens.sol

The most recent version of the audit report can be accessed on the `features/fix-audit` branch of this repository, specifically on commit `196d9dc2dad78876ee2db2fd74ae1663881e75b3`.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

deFarm acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. deFarm understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, deFarm agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the deFarm will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the deFarm, the final report will be considered fully accepted by the deFarm without the signature.

2. AUDIT RESULT

2.1. Overview

The SingleFarm Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.0`. The source code was written based on OpenZeppelin's library.

2.1.1. SingleFarmFactory

The Factory contract is used to create new SingleFarm contracts. For each investment pool, a SingleFarm contract will be created as a clone of implementation which set by the `admin` of Factory contract. By the default, the `manager` of the SingleFarm contract will be the caller which signed tx create farm. The factory contract has 2 main roles: `admin` and `owner`. The `admin` role manages configuration of the factory to create new SingleFarm contracts such as CapacityPerFarm, min/maxInvestmentAmount, ManagerFee, etc. The `owner` role manages the `admin` role and may withdraw the fund from the factory contract.

2.1.2. SingleFarm

Each Single Farm represents a distinct trading managed by a `Farm Manager`. The `manager` configures the setup during the creation process in SingleFarmFactory. Users have the option to invest in the farm by depositing ERC20 tokens during the fundraising period. Following this period, the `manager` utilizes the deposited funds to trade on Vertex Protocol (openPosition). The `manager` earns a performance fee based on the farm's profits. Throughout the open position, the `linkedSigner` account serves as the `manager`'s representation for trading on the Vertex. Upon closePosition, users can claim their share of the profits corresponding to the amount they initially deposited.

2.1.3. VertexHandler

The DEX handler contract facilitates the interaction between SingleFarm and the DEX. It assists SingleFarm in generating instructions to interact with the Vertex Protocol and calculates the account balance during the open position.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of The SingleFarm Smart Contracts.

2.2.1. Centralize issues CRITICAL

Affected files:

- SingleFarm.sol

- SingleFarmFactory.sol

Both `SingleFarm.sol` and `SingleFarmFactory.sol` have centralized roles: `owner` and `admin`.

With `owner`, it can set `admin` role and change the `dexHandler` address. If it's compromised, the attacker can update a dexHandler with stealing instruction to withdraw all user balance inside the SingleFarms. So we recommend using multisig wallet for `owner` role.

The compromise of the `admin` role also poses a risk. However, in line with the logical flow of the contracts, the `admin role` is required to execute tasks swiftly, making the use of multi-signature wallets impractical. Consequently, critical actions that demand careful consideration during the process may be delegated to the `owner`. Remaining tasks can be scrutinized more thoroughly through `admin` actions, as exemplified by the `liquidate` function in `SingleFarm.sol`.

```
/// @notice will change the status of the farm to `LIQUIDATED`
    /// @dev can be called once a farm is liquidated from the dex
    /// @dev can only be called by the `admin`
    function liquidate() external override onlyAdmin whenNotPaused {
        if (status != SfStatus.OPENED) revert NotOpened();
        status = SfStatus.LIQUIDATED;
        emit Liquidated();
    }
```

When the `admin` call liquidate, the contract status will be changed to `LIQUIDATED` even if when the position is not liquidated yet. When the status is `LIQUIDATED`, the contract will not work anymore, and logics for reward distribution or withdraw will not work anymore, and token will be stuck in dex contract forever.

With this case the liquidate function should verify the position health before changing the status to `LIQUIDATED`.

UPDATES

- *Jan 24, 2024*: This issue has been acknowledged and fixed by the deFarm team in commit `835367d65fa6d9f9cafb9056b892689b82ebd642`.

2.2.2. Conflict using between `admin` and `owner` LOW

Affected files:

- SingleFarm.sol
- SingleFarmFactory.sol

Currently, the `admin` and `owner` roles in both affected files are used not in a consistent way. In some function descriptions, the `owner` role is used to describe the role that can call the function, but it verifies `admin`.

```
/// SingleFarm.sol
/// @notice set the `fundDeadline` for a particular farm to cancel the farm early if needed
/// @dev can only be called by the `owner` or the `manager` of the farm
/// @param newFundDeadline new fundDeadline
    function setFundDeadline(uint256 newFundDeadline) external override {
        if (msg.sender != manager && msg.sender != IHasAdministrable(factory).admin())
revert NoAccess(manager, msg.sender); //<--@Audit: following the description, `owner`
should be used instead of `admin`
        if (newFundDeadline > 72 hours) revert AboveMax(72 hours, newFundDeadline);
        fundDeadline = newFundDeadline;
        emit FundDeadlineChanged(newFundDeadline);
    }

///SingleFarmFactory.sol
/// @notice Transfer `Eth` or token from this contract to the `receiver` in case of
emergency
/// @dev Can be called only by the `owner`
/// @param receiver address of the `receiver`
    function withdraw(
        address receiver,
        bool isEth,
        address token,
        uint256 amount
    ) external onlyAdmin returns (bool) {//<--@Audit: following the description, `owner`
should be used instead of `admin`
        if (isEth) {
            payable(receiver).transfer(amount);
        } else {
            IERC20Upgradeable(token).transfer(receiver, amount);
        }

        return true;
    }
```

UPDATES 1

- *Jan 22, 2024*: Team has updated the modifier and description in the several functions, but missing update the description of the `setFundDeadline` functions. We recommend updating the description of this function clearly.

UPDATES 2

- *Jan 24, 2024*: This issue has been acknowledged and fixed by the deFarm team in commit `196d9dc2dad78876ee2db2fd74ae1663881e75b3`.

2.2.3. Gas saving recommendations LOW

2.2.3.1. Use `calldata` instead of `memory` for variables in functions

In `external` function with array arguments, using `memory` will force solidity to copy that array to memory thus wasting more gas than using directly from calldata. Unless you want to write to the variable, always using `calldata` for external function.

UPDATES

- *Jan 24, 2024*: This issue has been acknowledged and fixed by the deFarm team in commit `a83c399d1206822f033e23e0cda26b52747b6e1e`.Recommend in `claimableAmount` function in `SingleFarm.sol`

```
function claimableAmount(address _investor) public view override returns (uint256 amount) {
        if (claimed[_investor] || status == SfStatus.OPENED) {
            amount = 0;
        } else if (status == SfStatus.CANCELLED || status == SfStatus.NOT_OPENED) {
            amount = (totalRaised * userAmount[_investor] * 1e18) / (actualTotalRaised * 
1e18); //<--@Audit: unnecessary multiplication by 1 in both numerator and denominator
        } else if (status == SfStatus.CLOSED) {
            amount = (remainingAmountAfterClose * userAmount[_investor] * 1e18) / 
(actualTotalRaised * 1e18); //<--@Audit: unnecessary multiplication by 1 in both numerator 
and denominator
        } else {
            amount = 0;
        }
        if (_investor == manager && managerFeeReceived > 0) {
            amount += managerFeeReceived;
        }
    }
```

In the second branch, the `totalRaised` and `userAmount` variables have already contained decimal in its value. So multiplying by 1e18 in both the numerator and denominator of the division operator are unnecessary.The third branch is the same as the second branch.

UPDATES

- *Jan 24, 2024*: This issue has been acknowledged and fixed by the deFarm team in commit `cf4fc14cdccb7227c7afc340996e53828566e271`.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jan 24, 2024*	Public Report	Verichains Lab

Table 2. Report versions history