QuillAudits

Audit Report
October, 2023

For

De.Fi

Table of Content

Table of Content

Executive Summary

Project Name	De.Fi
Project URL	*https://de.fi/*
Overview	The main purpose of the smart contracts is IDO/Vesting facilities. For IDO smart contracts it should support round system and allow payments in ETH and stablecoins(USDT/USDC). For vesting it should distribute tokens linearly and support multiple pools with own schedule.
Audit Scope	*https://github.com/de-dot-fi/eth-contracts*
Contracts in Scope	DeFiToken.sol IdoStorage.sol LaunchpadErc20Ido.sol LaunchpadNativeIdo.sol Vesting.sol
Commit Hash	0b1749460683c3e3bce195e51ae2ef824c624097
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated Tools, Functional Testing
Review 1	25th September 2023 - 5th October 2023
Updated Code Received	5th October 2023
Review 2	9th October 2023
Fixed In	a707340a03c923537420d14d369348a26621504a

Number of Security Issues per Severity

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	2	1	1

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage for
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw

Checked Vulnerabilities

- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Solhint, Mythril, Slither, Solidity Statistic Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - idoStorage.sol

High Severity Issues

A.1 Incorrect Condition Checks in **setMaxInvestment** and **setMinInvestment** Functions

Line

199

Functions - setMaxInvestment & setMinInvestment

```
function setMaxInvestment(uint256 investment_)
  external
  onlyRole(DEFAULT_ADMIN_ROLE)
{
  if (investment_ < _minInvestment) revert MinInvestmentErr(investment_, _minInvestment);

  _maxInvestment = investment_;

  emit MaxInvestmentUpdated(_maxInvestment);
}

function setMinInvestment(uint256 investment_)
  external
  onlyRole(DEFAULT_ADMIN_ROLE)
{
  if (investment_ > _maxInvestment) revert MaxInvestmentErr(investment_, _maxInvestment);

  _minInvestment = investment_;

  emit MinInvestmentUpdated(_minInvestment);
}
```

Description

The **setMaxInvestment** and **setMinInvestment** functions are incorrectly checking conditions. In **setMaxInvestment**, it compares the input **_investment** with **_minInvestment** which is not the intended comparison. Similarly, in **setMinInvestment**, it compares **_investment** with **_maxInvestment** which is contrary to the expected behavior. These incorrect condition checks can lead to unpredictable behavior and potential vulnerabilities in the investment limits of the IDO project.

A.1 Incorrect Condition Checks in **setMaxInvestment** and **setMinInvestment** Functions

Remediation

To resolve this issue and ensure correct condition checks, update the **setMaxInvestment** and **setMinInvestment** functions with the appropriate comparisons. Add additional checks as well to limit the max investment that can be allowed.

Status

Resolved

Contract - Vesting.sol

A.2 Lack of Duplicate Check in **setupBeneficiaryBatches** Function **_beneficiaries_** array

Line

90

Function - setupBeneficiaryBatches

```
function setupBeneficiaryBatches(uint8 idx_, address[] calldata _beneficiaries_, uint256[] calldata lockedAmounts_)
  external
  onlyRole(OPERATOR_ROLE)
{
  if (_pools[idx_].startTime == 0) revert PoolUndefinedErr(idx_);
  if (_beneficiaries_.length != lockedAmounts_.length) revert ArrayParamsInvalidLengthErr();

  uint256 totalLockedAmounts;
  for (uint8 i = 0; i < lockedAmounts_.length; i++) {
    totalLockedAmounts = totalLockedAmounts + lockedAmounts_[i];
  }
  _token.safeTransferFrom(_msgSender(), address(this), totalLockedAmounts);

  uint256 _beneficiariesLength = _beneficiaries_.length;
  for (uint8 i = 0; i < _beneficiariesLength; i++) {
    address beneficiary = _beneficiaries_[i];
    uint256 lockedAmount = lockedAmounts_[i];

    _beneficiaries[idx_][beneficiary].lockedAmount = _beneficiaries[idx_][beneficiary].lockedAmount + lockedAmount;
    _pools[idx_].totalLocked = _pools[idx_].totalLocked + lockedAmount;

    emit BeneficiaryAdded(idx_, beneficiary, lockedAmount);
  }
}
```

A.2 Lack of Duplicate Check in **setupBeneficiaryBatches** Function **_beneficiaries_** array

Description

The **setupBeneficiaryBatches** function does not perform a duplicate check on the **_beneficiaries_** array. Due to this oversight, it is possible to add the same beneficiary multiple times within the same batch, causing inconsistencies in the locking amounts and leading to potential discrepancies in the distribution of assets. This lack of a duplicate check undermines the integrity of the beneficiary setup process and can result in financial imbalances.

Remediation

To resolve this issue and ensure the uniqueness of beneficiaries within a batch, implement a duplicate check in the **setupBeneficiaryBatches** function.

Status

Acknowledged

Contract - launchPadNativeIdo.sol

A.3 Lack of Oracle Response Validation for Stale Prices

Line

310

Function - _getTokenAmount and _updatePurchaseState

```
function _getTokenAmount(uint256 investment_, IIdoStorage.Vesting vesting_)
    internal
    view
    returns (uint256)
{
    uint8 priceDecimals = _priceFeed.decimals();
    (, int256 price, , ,) = _priceFeed.latestRoundData();

    return (investment_ * uint256(price) * PRECISION) / _idoStorage.getPrice(vesting_) / (10 ** priceDecimals);
  }
}
```

A.3 Lack of Oracle Response Validation for Stale Prices

Line

217

Function - _getTokenAmount and _updatePurchaseState

```
function _updatePurchasingState(address beneficiary_, uint256 investment_, uint256 tokensSold_, address referral_, uint256 ma
  internal
{
  uint8 priceDecimals = _priceFeed.decimals();
  (, int256 price, , ,) = _priceFeed.latestRoundData();

  _raised = _raised + investment_;
  uint256 normalizedInvestment = (investment_ * uint256(price) * PRECISION) / (10 ** (TOKEN_DECIMALS + priceDecimals));
  _idoStorage.setPurchaseState(beneficiary_, ETH, normalizedInvestment, tokensSold_, referral_, mainReward_, tokenReward_);
}
```

Description

The **_getTokenAmount** and **_updatePurchasingState** functions are not checking the freshness of the price received from the oracle before using it in calculations. This vulnerability can lead to inaccurate conversion rates and investment calculations. If stale or outdated prices are used, it can result in incorrect token amounts being allocated to investors, causing financial losses and potential disputes.

Remediation

To address this issue and ensure the accuracy of price conversions, implement a check to validate the freshness of the oracle response before using it in calculations.

Status

Resolved

Medium Severity Issues

Contract - Vesting.sol

A.4 Lack of Proper Controls in **setupBeneficiary**, **setupBeneficiaryBatches** and **disableBeneficiary** Functions

Line

77, 90, 115

Functions - setupBeneficiary, setupBeneficiaryBatches and disableBeneficiary

```
function setupBeneficiary(uint8 idx_, address beneficiary_, uint256 lockedAmount_)
  external
  onlyRole(OPERATOR_ROLE)
{
  if (_pools[idx_].startTime == 0) revert PoolUndefinedErr(idx_);

  _token.safeTransferFrom(_msgSender(), address(this), lockedAmount_);
  _beneficiaries[idx_][beneficiary_].lockedAmount = _beneficiaries[idx_][beneficiary_].lockedAmount + lockedAmount_;
  _pools[idx_].totalLocked = _pools[idx_].totalLocked + lockedAmount_;

  emit BeneficiaryAdded(idx_, beneficiary_, lockedAmount_);
}
```

```
function setupBeneficiaryBatches(uint8 idx_, address[] calldata _beneficiaries_, uint256[] calldata lockedAmounts_)
  external
  onlyRole(OPERATOR_ROLE)
{
  if (_pools[idx_].startTime == 0) revert PoolUndefinedErr(idx_);
  if (_beneficiaries_.length != lockedAmounts_.length) revert ArrayParamsInvalidLengthErr();

  uint256 totalLockedAmounts;
  for (uint8 i = 0; i < lockedAmounts_.length; i++) {
    totalLockedAmounts = totalLockedAmounts + lockedAmounts_[i];
  }
  _token.safeTransferFrom(_msgSender(), address(this), totalLockedAmounts);

  uint256 _beneficiariesLength = _beneficiaries_.length;
  for (uint8 i = 0; i < _beneficiariesLength; i++) {
    address beneficiary = _beneficiaries_[i];
    uint256 lockedAmount = lockedAmounts_[i];

    _beneficiaries[idx_][beneficiary].lockedAmount = _beneficiaries[idx_][beneficiary].lockedAmount + lockedAmount;
    _pools[idx_].totalLocked = _pools[idx_].totalLocked + lockedAmount;

    emit BeneficiaryAdded(idx_, beneficiary, lockedAmount);
  }
}
```

A.4 Lack of Proper Controls in **setupBeneficiary**, **setupBeneficiaryBatches** and **disableBeneficiary** Functions

```
function disableBeneficiary(uint8 idx_, address beneficiary_)
  external
  onlyRole(OPERATOR_ROLE)
{
  if (_pools[idx_].startTime == 0) revert PoolUndefinedErr(idx_);
  if (_beneficiaries[idx_][beneficiary_].disabled) revert BeneficiaryDisabledErr(beneficiary_);

  uint256 securedAmount = _beneficiaries[idx_][beneficiary_].lockedAmount - _beneficiaries[idx_][beneficiary_].withdrawn;
  _beneficiaries[idx_][beneficiary_].withdrawn = _beneficiaries[idx_][beneficiary_].lockedAmount;
  _beneficiaries[idx_][beneficiary_].disabled = true;

  _token.safeTransfer(_msgSender(), securedAmount);

  emit BeneficiaryDisabled(idx_, beneficiary_, securedAmount);
}
```

Description

The **setupBeneficiary** and **disableBeneficiary** functions lack sufficient controls, leading to potential issues in the beneficiary management process. There are three critical shortcomings in the current implementation:

1. **Cross-Pool Deactivation:** Disabled beneficiaries from one pool can still be added to other pools. This allows a beneficiary who has been disabled in one pool to potentially access benefits in another pool, leading to inconsistencies and potential unfairness.
2. **Beneficiary Existence Check:** During the **disableBeneficiary** function, there is no check to verify if the beneficiary even exists for the specified pool. This lack of validation can lead to attempts to disable non-existing beneficiaries, causing unnecessary transactions and potential gas wastage.

Remediation

To address these issues, implement the following controls and checks in the **setupBeneficiary** and **disableBeneficiary** functions:

1. **Cross-Pool Deactivation Check:** Implement a check in the **disableBeneficiary** function to prevent disabled beneficiaries from being added to other pools. If a beneficiary is disabled in one pool, prevent them from being added to any other pool.
2. **Beneficiary Existence Check:** Implement a check during **disableBeneficiary** to ensure that the beneficiary being disabled actually exists in the specified pool. If the beneficiary does not exist, revert the transaction to prevent unnecessary operations.

Status

Resolved

A.5 Lack of Token Rescue Mechanism in LaunchPadErc20Ido Contract

Line

N/A

Function - N/A

N/A

Description

The **LaunchPadErc20Ido** contract does not implement a token rescue mechanism, which means there is no way to recover tokens or native tokens (such as ETH) sent to the contract mistakenly or under exceptional circumstances. In the absence of a token rescue feature, any tokens sent to the contract address, either intentionally or accidentally, become irretrievable, leading to potential financial losses for users or the project itself.

Remediation

To mitigate this issue and allow for the recovery of tokens sent to the contract address, implement a token rescue function in the **LaunchPadErc20Ido** contract. The function should be accessible only to authorized administrators and should allow the withdrawal of mistakenly sent tokens to a designated wallet.

Status

Resolved

Low Severity Issues

A.6 State-changing methods are missing event emissions

Line

99

Function - setupReferrals

```
function setupReferrals(
  address[] calldata referrals_,
  uint256[] calldata mainRewards_,
  uint256[] calldata secondaryRewards_
)
  external
  onlyRole(OPERATOR_ROLE)
{
  if (isClosed()) revert IdoClosedErr();
  if (referrals_.length != mainRewards_.length || referrals_.length != secondaryRewards_.length) revert ArrayParamsInvalidLengthErr();

  uint256 length = referrals_.length;
  for (uint256 i = 0; i < length; i++) {
    _referrals[referrals_[i]] = Referral({
      defined: true,
      enabled: true,
      mainReward: mainRewards_[i],
      secondaryReward: secondaryRewards_[i]
    });
  }
}
```

Description

Critical functions within the contract lack the emission of events. Events play a vital role in providing transparency, enabling external systems to react to changes, and offering a way to track important contract activities. The absence of events can hinder monitoring and auditing efforts, making it difficult to detect and respond to critical contract actions.

Remediation

To address this issue and improve the transparency and auditability of the contract, you should add appropriate event emissions in critical functions. These events should capture essential information about the function's execution, including input parameters and outcomes. Additionally, consider emitting events both before and after critical state changes, where applicable.

Status

Resolved

Informational Issues

A.7 General Recommendation

In the case of stable coins, the hierarchy of authority plays a crucial role and our team recommends that the roles with privileges must be given to the accounts with proven authority and functions like “Minting” and “Burning” must be used cautiously because once these functions are called on the mainnet then could be no minting or burning. Hence, it will be an irreversible change.

Status

Fixed

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Should be able to grant Minter, Burner and Asset Protection Roles to accounts
- ✓ Should be able to Mint and Burn tokens (from the owner's account as well as from any other account)
- ✓ Should be able to transfer tokens
- ✓ Should be able to transfer ownership and revert for a zero address
- ✓ Should revert if transfer amount exceeds balance
- ✓ Should revert if Minter and Burners don't have desired roles
- ✓ Should be able to set a fee receiver account
- ✓ Should be able to deduct the fee from the transfer amount and transfer it to the fee recipient
- ✓ Should not deduct fee if sender is owner, receiver is fee recipient, excluded from fee, or the fee percentage amount is not greater than zero
- ✓ Should be able to freeze and unfreeze accounts and revert if the caller does not have the asset protection role
- ✓ Owner should be able to stop minting, burning and pause/unpause the contract

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the De.Fi. We performed our audit according to the procedure described above.

Some issues of High,Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in De.Fi smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of De.Fi smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the De.Fi to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

850+
Audits Completed

$30B
Secured

$30B
Lines of Code Audited

Follow Our Journey

Audit Report
October, 2023

For

De.Fi

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com