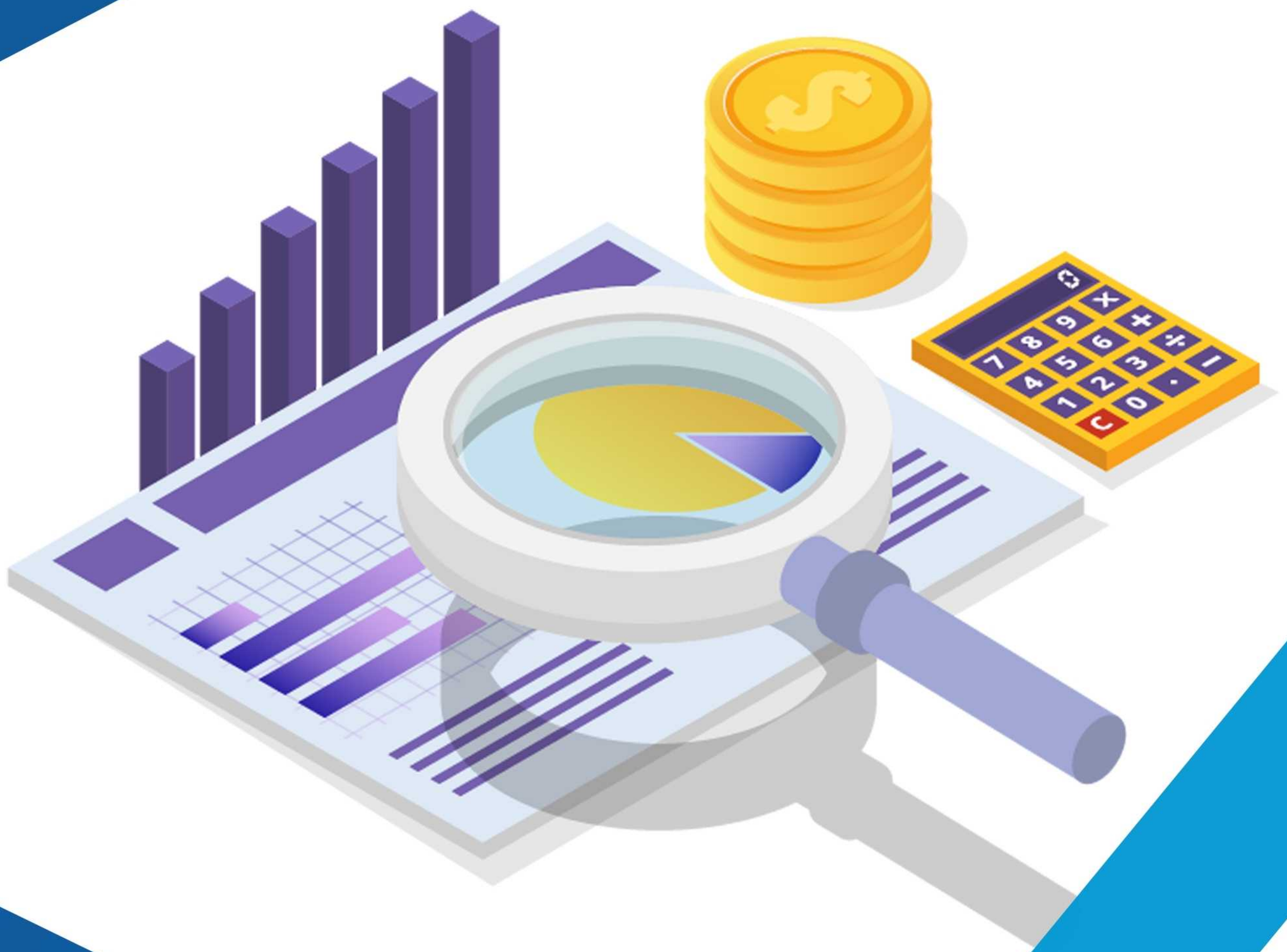
QuillAudits



Audit Report
April, 2021

OpenDeFi by OroPocket

Contents

Introduction

During the period of **March 23rd, 2021 to April 12th, 2021** – QuillHash Team performed security audit for **UniFarm(V4)** smart contracts. The code for audit was taken from the following link:

https://github.com/themohitmadan/unifarmv4 [commit: 66746bad593e825713f0156db9c80a09eb9ef50c]
https://drive.google.com/file/d/1lSeRnEbL9hYKrEdqlkragLEi065oZguU/view?usp=sharing

Updated Contracts:
https://github.com/themohitmadan/unifarmv4/tree/v4Audit

Commit Hash - **d3f99ce7804f479257896e15c5d0d2b036ccc880**

Overview of UniFarm

UniFarm is decentralized farming pool of DEFI's top projects. Common staking/farming programs allow users to farm only the token that they stake. Here they can farm multiple tokens. Users can unstake anytime There is no Lock-in. It provides one Place to Farm Instead of having to go to different exchanges etc to stake their different holdings, users can easily farm from one place while earning multiple. Tokens are held in self-custodian Wallet like Metamask—decentralised.

The features & utility of the UniFarm ecosystem

- Simple UI. One place to farm all.
- No market exposure of staked tokens.
- Stake one, farm many tokens.
- Working jointly with several projects gives a wider audience to all participating projects in pool.
- Create multiple staking pools.
- Decentralized.
- Gamification done in a way to reduce any sell pressure.

UNIFARM TOKEN:

- Governance token for all protocol level changes.
- UniFarm tool will be added to all premium pools enabling token holders
- to stake UNIFARM and earn bunch of rewards.
- UniFarm is already cashflow positive.
- 25% of the revenue will be used to buy back UNIFARM token and lock
- them for a year.
- Total supply of 1 Billion tokens.
- 50% of the supply used for farming.

Details: https://unifarm.co/

Tokenomics

Tokenomics

Cap Table

Item	Number of tokens	Percent of tokens	Unlock Schedule
Fundraise	100000000	10	Variable for different rounds. Details below
Liquidity for exchanges	50000000	5	Unlocked for Balancer/Exchange liquidity. Liquidity will be added at $0.04
Farming	500000000	50	Unifarm pool farming
Team	100000000	10	Cliff for 1 year, monthly unlock from 13-24 months
Advisors	50000000	5	Cliff for 6 months, daily linear unlocking from days 181-360
Marketing	50000000	5	Cliff for 3 months; Monthly unlocking from month 4-24
Company Reserve	100000000	10	Cliff for 6 months; Quarterly unlock over 6 quarters
Community	50000000	5	Variable for community. Details below

Scope of Audit

The scope of this audit was to analyse UniFarm smart contracts codebase for quality, security, and correctness.

Check Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Address hardcoded
- Using delete for arrays
- Integer overflow/underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

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-20 token 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.
SmartCheck.

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

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerability 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 automated analysis were manually verified.

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

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned with 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 to be fixed before moving to a live environment.

Medium level 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 level severity issues

Low level severity issues can cause minor impact and or 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 severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	5	3
Closed	0	2	8	0

Issues Found – Code Review / Manual Testing

High severity issues

None.

Medium severity issues

1. **External calls inside a loop might lead to a denial-of-service attack**

   Each block has an upper bound on the amount of gas that can be spent, and thus the amount computation that can be done. This is the Block Gas Limit. The Ethereum blockchain transaction can only process a certain amount of gas due to the block gas limit. When a number of iteration costs go beyond the block gas limit, the transaction will fail and the contract can be stalled at a certain point. In this case, attackers may potentially attack the contract, and manipulate the gas.
   This leads to a couple possible Denial of Service vectors. Even in the absence of intentional attacks this can lead to problems. To minimize the damage caused by such failures, it is often better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically.
   If it’s absolutely necessary to loop over an array of unknown size, it is recommended to do transfer in batches or use favor pull over push payments technique.

   Favor pull over push payments pattern
   - https://fravoll.github.io/solidity-patterns/pull_over_push.html
   - https://blog.openzeppelin.com/onward-with-ethereum-smart-contract-security-97a827e47702/

   Code Lines:

   UnifarmV4.sol [#234, 270, 443-444, 451, 463]

   Auditors Remarks: Fixed.

2. Reentrancy

One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. It's recommended that the calls to external functions/events should happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit. When control is transferred to recipient, care must be taken to not create reentrancy vulnerabilities. OpenZeppelin has its own mutex implementation you can use called ReentrancyGuard. This library provides a modifier you can apply to any function called nonReentrant that guards the function with a mutex.
Consider using ReentrancyGuard or the checks-effects-interactions pattern.

Following link explains more about Reentrancy and ReentrancyGuard

https://docs.openzeppelin.com/contracts/2.x/api/utils#ReentrancyGuard
https://blog.openzeppelin.com/reentrancy-after-istanbul/

Code Lines:

- Admin.sol [#309-312, 315-319]
- UnifarmV4.sol [#69-123, 160-293, 315-319, 359-415, 419-475]

Auditors Remarks: Fixed.

Low level severity issues

1. Compiler version should be fixed

Solidity source files indicate the versions of the compiler they can be compiled with. It's recommended to lock the compiler version in code, as future compiler versions may handle certain language constructions in a way the developer did not foresee. Also, it's recommended to use latest compiler version.

Code Lines:

- Admin.sol [#1],
- UnifarmV4[#3]

Auditors Remarks: Not Fixed. Though the version used is same – remove the carat(^) sign to fix the compiler version in the contract.

2. Reentrancy

Coding style issues influence code readability and in some cases may lead to bugs in future. Smart Contracts have naming convention, indentation and code layout issues. It's recommended to use **Solidity Style Guide** to fix all the issues. Consider following the Solidity guidelines on formatting the code and commenting for all the files. It can improve the overall code quality and readability.

```
UnifarmV4.sol
  218:2  error  Line length must be no more than 120 but current length is 122  max-line-length
  223:2  error  Line length must be no more than 120 but current length is 122  max-line-length
  225:2  error  Line length must be no more than 120 but current length is 168  max-line-length
  234:2  error  Line length must be no more than 120 but current length is 138  max-line-length
  260:2  error  Line length must be no more than 120 but current length is 122  max-line-length
  262:2  error  Line length must be no more than 120 but current length is 168  max-line-length
  270:2  error  Line length must be no more than 120 but current length is 138  max-line-length
  315:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  378:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  419:2  error  Line length must be no more than 120 but current length is 144  max-line-length
  423:2  error  Line length must be no more than 120 but current length is 123  max-line-length
  438:2  error  Line length must be no more than 120 but current length is 155  max-line-length
  451:2  error  Line length must be no more than 120 but current length is 155  max-line-length

✖ 13 problems (13 errors, 0 warnings)
```

Auditors Remarks: Fixed.

3. Order of layout

The order of functions as well as the rest of the code layout does not follow solidity style guide.
Layout contract elements in the following order:

- Pragma statements
- Import statements
- Interfaces
- Libraries
- Contracts

Inside each contract, library or interface, use the following order:

- Type declarations
- State variables
- Events
- Functions

Please read following documentation links to understand the correct order:
- https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-layout
- https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-functions

Auditors Remarks: Not Fixed.

4. **Naming convention issues – Variables, Structs, Functions**

   It is recommended to follow all solidity naming conventions to maintain readability of code.
   Details:
   https://docs.soliditylang.org/en/v0.4.25/style-guide.html#naming-conventions

   Auditors Remarks: Fixed.

5. **Use local variable instead of state variable like length in a loop**

   For every iteration of for loop - state variables like length of non-memory array will consume extra gas. To reduce the gas consumption, it's advisable to use local variable.

   Code Lines:
   Admin.sol [#135, 165, 266]
   UnifarmV4.sol [#214, 442]

   Auditors Remarks: Not Fixed.

6. **Be explicit about which `uint` the code is using**

   `uint` is an alias for `uint256`, but using the full form is preferable. Be consistent and use one of the forms.

   Code Lines:
   UniFarmV4.sol [#435, 442, 445]

   Auditors Remarks: Fixed.

7. Implicit Visibility

It’s a good practice to explicitly define visibility of state variables, functions, interface functions and fallback functions. The default visibility of state variables – internal; function – public; interface function - external.

Code Lines:
Admin.sol [#92-96]
UniFarmV4.sol [#60-62]

Auditors Remarks: Not Fixed.

8. Consider using struct instead of multiple return values.

Replace multiple return values with a struct. It helps improve readability.

Code Lines:
UniFarmV4.sol [#485]

Auditors Remarks: Not Fixed.

9. Use external function modifier instead of public

The public functions that are never called by contract should be declared external to save gas. List of functions that can be declared external:

Admin.sol

- dmin.updateSequence(address,address[]) [#156-184]
- Admin.updateToken(address,uint256,uint256,uint256) [#186-204]
- Admin.lockableToken(address,uint8,uint256,bool) [#206-234]
- AdAdmin.addToken(address,uint256,uint256,uint256,uint8) [#98-122]
- Admin.setDailyDistribution(address[],address[],uint256[]) [#124-154]
- Amin.updateStakeDuration(uint256) [#236-243]
- Admin.updateOptionableBenefit(uint256) [#245-252]
- Admin.updateRefPercentage(uint256) [#254-261]
- Admin.updateIntervalDays(uint256[]) [#263-278]
- Admin.changeTokenBlockedStatus(address,address,bool) [#280-299]
- Admin.safeWithdraw(address,uint256) [#301-320]

UnifarmV4.sol

- Unifarmv4.viewStakingDetails(address) [#482-502]

Auditors Remarks: Fixed.

10. **Boolean equality check**

    Boolean constants can be used directly. No need to be compare to true or false.

    Code Lines:
    Admin.sol [#216]
    UnifarmV4.sol [#160-293, 377-380, 422-425]

    Auditors Remarks: Fixed.

11. **Use of different compiler versions of Solidity**

    Different compiler versions of solidity are in use - Version used: ['>=0.6.0<=0.8.0', '^0.7.0'].

    - ^0.7.0 [Admin.sol#1]
    - ^0.7.0 [UnifarmV4.sol#3]
    - >=0.6.0<=0.8.0 [abstract\Ownable.sol#6]
    - ^0.7.0 [interface\IERC20.sol#2]
    - ^0.7.0 [library\SafeMath.sol#1]

    It is recommended to a single fixed compiler version in all files including the local Openzeppelin contracts.

    Auditors Remarks: Fixed.

12. **Uninitialized local variable**

    It is always a good practice to initialize all the variables. If a variable is meant to be initialized to zero, explicitly set it to zero.

    Code Lines:
    UnifarmV4.sol [#442]

    Auditors Remarks: Fixed.

13. Use require for error handling while using transfer

The ERC-20 transfer function returns a Boolean value if the value was transferred successfully or not. It’s recommended to use require for error handling when transfer function is called.

Code Lines:
UnifarmV4.sol [#399, 440, 463]

Auditors Remarks: Fixed.

Informational

1. Use of block.timestamp should be avoided

Malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. It is risky to use block.timestamp, as block.timestamp can be manipulated by miners. Therefore block.timestamp is recommended to be supplemented with some other strategy in the case of high-value/risk applications.

Code Lines:

- Admin.sol [#119, 149, 180, 202, 219, 232, 241, 250, 259, 274, 297, 318]
- UnifarmV4.sol [#61, 95, 105, 122, 141, 142, 201, 241, 277, 350,367, 373, 412, 458, 473]

2. Files not in use can be removed from contracts folder

Context.sol and Pausable.sol are not in use. It's recommended to remove these files.

3. Gas optimization tips

The contract has costly loops and consumes a lot more gas than normal. It is highly recommended to implement gas optimization techniques to reduce gas consumption.
https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6

Automated Testing

Slither

Slither is an open-source Solidity static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses.

```
INFO:Printers:
Compiled with solc
Number of lines: 783 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 6 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 13
Number of informational issues: 13
Number of low issues: 2
Number of medium issues: 0
Number of high issues: 2
ERCs: ERC20

+-----------+-------------+-------+----------------------+---------------+----------------------+
|   Name    | # functions |  ERCS |     ERC20 info       | Complex code |      Features        |
+-----------+-------------+-------+----------------------+---------------+----------------------+
|  Admin    |      25     |       |                      |      No       | Tokens interaction   |
|  IERC20   |      6      | ERC20 |     No Minting       |      No       |                      |
|           |             |       | Approve Race Cond.   |               |                      |
|           |             |       |                      |               |                      |
| SafeMath  |      8      |       |                      |      No       |                      |
+-----------+-------------+-------+----------------------+---------------+----------------------+
INFO:Slither:Admin.sol analyzed (6 contracts)
```

```
INFO:Printers:
Compiled with solc
Number of lines: 1287 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 7 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 14
Number of informational issues: 22
Number of low issues: 24
Number of medium issues: 5
Number of high issues: 5
ERCs: ERC20

+-----------+-------------+-------+----------------------+---------------+----------------------+
|    Name   | # functions |  ERCS |     ERC20 info       | Complex code |      Features        |
+-----------+-------------+-------+----------------------+---------------+----------------------+
| Unifarmv4 |      34     |       |                      |      Yes      | Tokens interaction   |
|  IERC20   |      6      | ERC20 |     No Minting       |      No       |                      |
|           |             |       | Approve Race Cond.   |               |                      |
|           |             |       |                      |               |                      |
| SafeMath  |      8      |       |                      |      No       |                      |
+-----------+-------------+-------+----------------------+---------------+----------------------+
INFO:Slither:UnifarmV4.sol analyzed (7 contracts)
```

Slither didn't raise any critical issue with smart contracts. The smart contracts were well tested and all the minor issues that were raised have been documented in the report. Also, all other vulnerabilities of importance have already been covered in the manual audit section of the report.

Smartcheck

SmartCheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. SmartCheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR).

```
ruleId: SOLIDITY_OVERPOWERED_ROLE
patternId: j83hf7
severity: 2
line: 124
column: 4
content: functionsetDailyDistribution(address[]memorystakedToken,address[]memoryrewardToken,uint256[]memorydailyDist
=rewardToken.length&&rewardToken.length==dailyDistribution.length,"Invalid Input");for(uint8i=0;i<stakedToken.length
tokenDetails[rewardToken[i]].isExist,"Token not exist");tokenDailyDistribution[stakedToken[i]][rewardToken[i]=daily
oken[i],rewardToken[i],dailyDistribution[i],block.timestamp);}}

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 1
column: 16
content: ^

ruleId: SOLIDITY_SAFEMATH
patternId: 837cac
severity: 1
line: 17
column: 4
content: usingSafeMathforuint256;

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 92
column: 4
content: constructor(address_owner)Ownable(_owner){stakeDuration=90days;refPercentage=5ether;optionableBenefit=2;}

SOLIDITY_VISIBILITY :1
SOLIDITY_SAFEMATH :1
SOLIDITY_OVERPOWERED_ROLE :1
SOLIDITY_PRAGMAS_VERSION :1
SOLIDITY_EXTRA_GAS_IN_LOOPS :3
SOLIDITY_GAS_LIMIT_IN_LOOPS :3
```

```
ruleId: SOLIDITY_SAFEMATH
patternId: 837cac
severity: 1
line: 12
column: 4
content: usingSafeMathforuint256;

ruleId: SOLIDITY_SHOULD_RETURN_STRUCT
patternId: 83hf3l
severity: 1
line: 485
column: 16
content: (address[]memory,address[]memory,bool[]memory,uint256[]memory,uint256[]memory,uint256[]memory)

ruleId: SOLIDITY_UNCHECKED_CALL
patternId: f39eed
severity: 3
line: 206
column: 8
content: sendToken(userAddress,stakingDetails[userAddress].tokenAddress[stakeId],stakingDetails[userAddress].to

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 60
column: 4
content: constructor(address_owner)Admin(_owner){poolStartTime=block.timestamp;}

SOLIDITY_VISIBILITY :1
SOLIDITY_SAFEMATH :1
SOLIDITY_PRAGMAS_VERSION :1
SOLIDITY_EXTRA_GAS_IN_LOOPS :2
SOLIDITY_GAS_LIMIT_IN_LOOPS :2
SOLIDITY_UNCHECKED_CALL :1
SOLIDITY_SHOULD_RETURN_STRUCT :1
```

SmartCheck did not detect any high severity issue. All the considerable issues raised by SmartCheck are already covered in the code review section of this report.

Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.
Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the code review section of this report.

Remix IDE

Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally.

The UniFarm smart contracts were tested for various compiler versions. The smart contract compiled without any error on explicitly setting compiler version **0.7.0>= and <0.8.0.**

It is recommended to use any fixed compiler versions from 0.7.0 to 0.7.6. The contract failed to compile from version 0.8.0 onwards as few functionalities in solidity have been deprecated.

The contract was successfully deployed on Rinkbey Network (0x8388DB68211E8322794f7665a605A28Ee2B0A563). Consumption of gas while deploying the contract was found to be extremely high. It is recommended to make the contract gas efficient.

Auditors Remarks: The updated contract was deployed on the Rinkbey testnet [https://rinkeby.etherscan.io/tx/0x7d688fac5527edf1061c666b2e4f05ff2a6c41f745bdd1c81eeced9702aeebda]. Consumption of gas was found to be normal.

Closing Summary

Overall, the smart contracts are adhered to ERC-20 guidelines. Several issues of medium and low severity were found during the audit. There were no critical or major issues found that can break the intended behaviour. All the medium issues and 8 low severity issues have been fixed.

Disclaimer

QuillHash audit is not a security warranty, investment advice, or an endorsement of the UniFarm platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the UniFarm Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

OpenDeFi

by OroPocket

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com