Infinity Yield Smart Contract Final Audit Report

Introduction 3
Scope of Audit 3
Check Vulnerabilities 3
Techniques and Methods 5
Structural Analysis 5
Static Analysis 5
Code Review / Manual Analysis 5
Gas Consumption 5
Tools and Platforms used for Audit 5
Issue Categories 6
High Severity Issues 6
Medium Severity Issues 6
Low Severity Issues 6
Informational 6
Number of issues per severity 6
Contract Details 7
Issues Found – Code Review / Manual Testing 8
High Severity Issues 8
Medium Severity Issues 8
Low Severity Issues 8
Informational 8
Closing Summary 9
Disclaimer 10

Scope of Audit

The scope of this audit was to analyze and document IFY smart contract codebase for quality, security, and correctness.

Check Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- 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

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.

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 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 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 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 to 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 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

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
OPEN	0	3	0	0
CLOSED	6	4	11	6

Introduction

During the period of **Jan 25th, 2021 to February 2nd, 2021** - Quillhash Team performed a security audit for **Infinity Yield** smart contracts.

The code for the audit was taken from following the official GitHub link**:**
https://github.com/InfinityYield/DeflationaryTokenWithStake

Bugs and Issues found After Code Updation

1. Ceil function is broken in all contracts

Severity - HIGH

Status: CLOSED

Line no:

- *Token Contract* - *Line 38*
- *Stake.sol* - *Line 159*
- *PreSale.sol* - **Line 160**

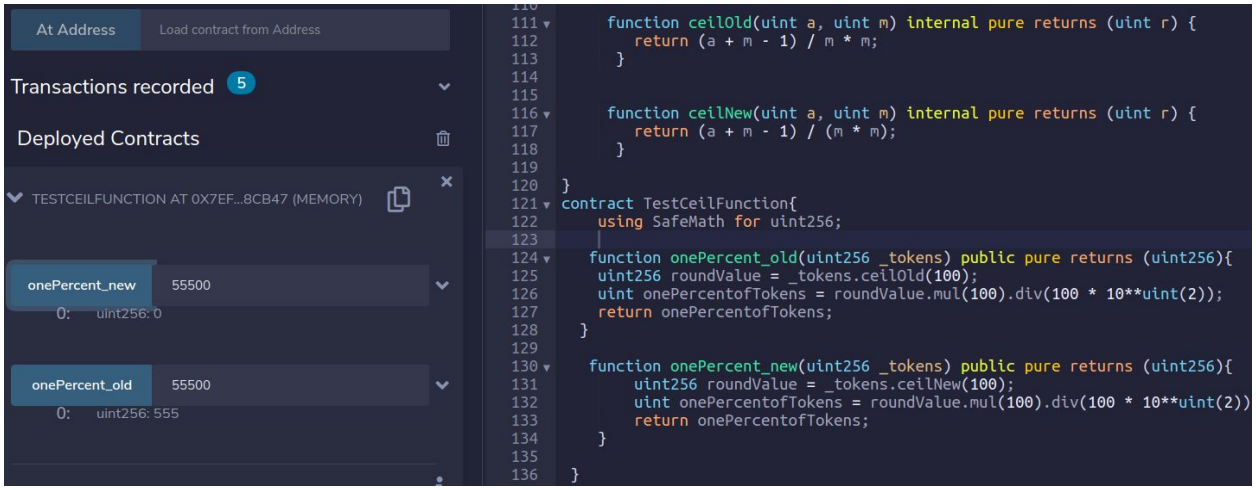
Description:

The **ceil** function implements an inaccurate arithmetic logic which lead to wrong round value result every time this function is called.

```
    function ceil(uint256 a, uint256 m) internal pure returns (uint256 r) {
        require(m != 0, "SafeMath: to ceil number shall not be zero");
        return (a + m - 1) / (m * m);
    }
```

Due to the current implementation of **ceil function,** the **onePercent function of Stake.sol** and **Token.sol** doesn't work as expected and returns a wrong result.

A brief example of this can be seen below:



In the above image, the **onePercent function** is being called using using the current **ceil function(***ceilNew***)** as well as the older **ceil function(***ceilOld***).**

For an input of **55500,** the ***onePercent_old*** function(using *ceilOld*) returns its one percent as **555,** but the **onePercent_new** function returns **0** as the one percent of **55500**, which is incorrect.

2. Unnecessary Require Statement

Severity - LOW

Status: CLOSED

Line no:

- **Token Contract - Line 39**
- **Stake.sol - Line 159**
- **PreSale.sol - Line 161**

Description:

The **ceil** function includes a **require statement** to check that the value of the parameter "**m**" passed to the **ceil** function is **non-zero.**

```
function ceil(uint256 a, uint256 m) internal pure returns (uint256 r) {
    require(m != 0, "SafeMath: to ceil number shall not be zero");
    return (a + m - 1) / (m * m);
}
```

However, the parameter **m** is already set to be **100,** while calling the **ceil** function, in every contract. For instance, *Line 368 in Stake Contract.*
Therefore, it doesn't demand an additional check on parameter m to be non-zero.

```
function onePercent(uint256 _tokens) private pure returns (uint256){
    uint256 roundValue = _tokens.ceil(100);
    uint onePercentofTokens = roundValue.mul(100).div(100 * 10**uint(2));
    return onePercentofTokens;
}
```

A. Contract Details

Name - Token.sol

About the Contract:
Token.sol is an ERC20 token contract that implements the basic functionalities of an ERC20 token standard.

The contract includes an additional taxation mechanism on users while using the transfer functionalities of the contract.

Issues Found – Code Review / Manual Testing

High Severity Issues

A.1 Transfer function imposes Tax for token transfers to or from STAKING ADDRESS

Line no - 142

Status: CLOSED

Description:

The transfer function was designed in a way that it should not apply tax if the sender or receiver is **Staking Contract Address.**

However, the transfer function does impose a tax even if IFY tokens are being received or sent by the Staking Contract.

This leads to an unwanted scenario where tokens received by the Staking Contract is less than what was sent initially or tokens sent by the Staking contract is less than what is received by the end-user.

```
142     function transfer(address to, uint256 tokens) public override returns (bool success) {
143         require(address(to) != address(0));
144         require(balances[msg.sender] >= tokens);
145         require(balances[to] + tokens >= balances[to]);
146 
147         balances[msg.sender] = balances[msg.sender].sub(tokens);
148         uint256 deduction = 0;
149         // if the sender or receiver address is not staking address, apply tax
150         if (to != STAKING_ADDRESS|| msg.sender != STAKING_ADDRESS ){
151             deduction = onePercent(tokens).mul(tax); // Calculates the tax to be applied c
152             uint256 _OS = onePercent(deduction).mul(10); // 10% will go to owner
153             balances[owner] = balances[owner].add(_OS);
154             balances[STAKING_ADDRESS] = balances[STAKING_ADDRESS].add(deduction.sub(_OS));
155         }
156         balances[to] = balances[to].add(tokens.sub(deduction));
157         emit Transfer(msg.sender, to, tokens.sub(deduction));
158         return true;
159     }
```

The main reason behind this scenario is the incorrect implementation of the **if-else statement.**The current if-else statement in the transfer function includes an OR operation which means the **if** condition will execute even if any one of the given conditions are true.

Now, for instance, while sending IFY tokens to Staking Address, the ***receiver is a Staking Contract address but the sender is an externally owned address.***

Therefore, although **to != STAKING_ADDRESS** is false(*in the **if condition***), **msg.sender != STAKING ADDRESS** is true, thus allowing an entry to the taxation logic even when the receiver is a Staking contract Address.

Recommendation:
A possible solution to this issue is to implement the **if-else** condition in the following ways**:**

```
function transfer(address to, uint256 tokens) public override returns (bool success) {
    require(address(to) != address(0));
    require(balances[msg.sender] >= tokens);
    require(balances[to] + tokens >= balances[to]);

    balances[msg.sender] = balances[msg.sender].sub(tokens);
    uint256 deduction = 0;
    // if the sender or receiver address is not staking address, apply tax
  if (to == STAKING_ADDRESS || msg.sender == STAKING_ADDRESS ){
    balances[to] = balances[to].add(tokens);
    emit Transfer(msg.sender, to, tokens);
  }else{
    deduction = onePercent(tokens).mul(tax); // Calculates the tax to be applied on the
    uint256 _OS = onePercent(deduction).mul(10); // 10% will go to owner
    balances[owner] = balances[owner].add(_OS);
    balances[STAKING_ADDRESS] = balances[STAKING_ADDRESS].add(deduction.sub(_OS)); // ad
    balances[to] = balances[to].add(tokens.sub(deduction));
    emit Transfer(msg.sender, to, tokens.sub(deduction));
 }
    return true;
}
```

A. 2 TransferFrom function imposes Tax for token transfers to or from STAKING ADDRESS

Line no - 180

Status: CLOSED

Description:
The **transferFrom** function was designed in a way that it should not apply tax if the sender or receiver is **Staking Contract Address.**
However, the function imposes a tax even if IFY tokens are being received or sent by the Staking Contract.

This leads to an unwanted scenario, similar to the transfer function, where tokens received by the Staking Contract is less than what was sent initially or tokens sent by the Staking contract is less than what is received by the end-user.

```
180     function transferFrom(address from, address to, uint256 tokens) public override returns (bool success)
181         require(tokens <= allowed[from][msg.sender]); //check allowance
182         require(balances[from] >= tokens);
183         balances[from] = balances[from].sub(tokens);
184         allowed[from][msg.sender] = allowed[from][msg.sender].sub(tokens);
185
186         uint256 deduction = 0;
187         // if the sender or receiver address is not staking address, apply tax
188         if (to != STAKING_ADDRESS||from != STAKING_ADDRESS ){
189             deduction = onePercent(tokens).mul(tax); // Calculates the tax to be applied on the amount tra
190             uint256 _OS = onePercent(deduction).mul(10); // 10% will go to owner
191             balances[owner] = balances[owner].add(_OS);
192             balances[STAKING_ADDRESS] = balances[STAKING_ADDRESS].add(deduction.sub(_OS)); // add the tax
193         }
194
195         balances[to] = balances[to].add(tokens.sub(deduction)); // send rest of the amount to the receiver
196         emit Transfer(from, to, tokens.sub(deduction));
197         return true;
198     }
```

Recommendation:

A similar approach with the if-else condition, like the **transfer function,** will resolve this issue with **transferFrom** function as well.

Medium Severity Issues

A.3 transferOwnerShip function lacks a Zero Address Check.

[Line no - 60]

Status: CLOSED

Description:

The transferOwnership function doesn't validate the **_newOwner** address passed as a parameter.

```
60      function transferOwnership(address payable _newOwner) public onlyOwner {
61          owner = _newOwner;
62          emit OwnershipTransferred(msg.sender, _newOwner);
63      }
```

Recommendation:

The new owner's address must be checked with a **require** statement.

require(_newOwner != address(0),"Invalid address passed");

A.4 Division precedes Multiplication

Status : OPEN

Description:

The **ceil** function implements an inaccurate arithmetic logic which lead to wrong round value result every time this function is called.

```
function ceil(uint256 a, uint256 m) internal pure returns (uint256 r) {
    require(m != 0, "SafeMath: to ceil number shall not be zero");
    return (a + m - 1) / (m * m);
}
```

Low Severity Issues

A.5 ChangeTax function doesn't emit any event

Line no - 120

Status: CLOSED

Description:

The **tax** state variable is a crucial arithmetic parameter in the Token contract. Since there is no event emitted on updating this variable, it might be difficult to track it off chain.

Recommendation:

An event should be fired after changing the tax variable.

Informational

A.6 Absence of Error messages in Require Statements

Status: CLOSED

Description:

No require statement in the Token.sol contract includes an error message. While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:
Include error messages in every require statement.

A.7 Explicit visibility declaration is missing
Status: CLOSED

Description:
The following elements in the contract have not been assigned any visibility explicitly:

- _totalSupply - Line 130
- address STAKING_ADDRESS Line 105
- mapping(address => uint256) balances; Line 106
- mapping(address => mapping(address => uint256)) allowed Line 107

Recommendation:
Visibility specifiers should be assigned explicitly in order to avoid ambiguity.

A.8 Constant declaration optimizes gas usage

Status: CLOSED

Description:
State variables that are not supposed to change throughout the contract should be declared as **constant.**

Recommendation*:*
The following state variables needs to be declared as constant:

```
100        string public symbol = "IFY";
101        string public  name = "Infinity Yeild";
102        uint256 public decimals = 18;
```

B. Contract Details

Name - PreSale.sol

About the Contract:

PreSale is a crowdsale contract that allows users to buy IFY tokens with ETH. Users will be allowed to buy tokens within the duration between the start of the sale(*3 Feb 2021*) and end of the sale(*10 Feb 2021*).

The user can claim their tokens once the **claim duration starts**, i.e., from ***10 Feb 2021, 8pm GMT*.**

Issues Found – Code Review / Manual Testing

High Severity Issues

B.1 Function logic of INVEST function is broken
Line 220

Status: CLOSED

Description:
The **Invest** function has been implemented in a way that it does not support more than one investment from the same address.

For instance, if **USER_A** deposits **5 ETH**, by calling the **Invest function,** and then calls the function once again with **2 ETH**, the contract only stores the investment details of **USER_A as 2 ETH**.
It doesn't keep a track of the 5 ETH investment made by the same user earlier.

This will result in permanent loss of user's **ETH**.

```
220     function Invest() public payable{
221         require( now > startSale && now < endSale , "Sale is closed");
222         uint256 tokens = getTokenAmount(msg.value);
223         investor[msg.sender] = tokens;
224         purchasedTokens = purchasedTokens + tokens;
225         owner.transfer(msg.value);
226     }
```

The **reason behind** this issue at **Line 223**.
The **investor mapping** simply replaces/updates a particular investor's initial token balance with the new amount of tokens. It never increments the balance of the investor in this mapping and, therefore, fails to keep the actual record of the total amount of tokens purchased by the investor.

Recommendation:
Increment the investor's tokens balance in the **investor mapping** so that ***it stores the total amount of tokens purchased*** instead of the ***latest token amount purchased.***

A possible approach to resolve this issue is mentioned below:

```
220     function Invest() public payable{
221         require( now > startSale && now < endSale , "Sale is closed");
222         uint256 tokens = getTokenAmount(msg.value);
223         investor[msg.sender] += tokens;
224         purchasedTokens = purchasedTokens + tokens;
225         owner.transfer(msg.value);
226     }
```

B.2 The function “getUnSoldTokens” might fail

Line no 248
Status: CLOSED

Description:
The **getUnSoldTokens** transfers the remaining tokens in the PreSale contract only if the following **require statement is true:**

```
252         require(tokensInContract > purchasedTokens, "no unsold tokens in contract");
```

However, if the tokensInContract variable is less than **purchasedTokens variable,** this require statement will revert back and IFY tokens locked in PreSale contract won’t be transferred back to owner’s address.
This might happen because:

- **tokensInContract** variable stores the current token balance of the PreSale contract which might change(*decrease*) as users call the **ClaimTokens** function which will transfer tokens from PreSale contract to user’s address.
- **purchasedTokens** variable is incremented with the amount of tokens purchased by any investor but never decremented when the user has claimed their invested amount. This doesn’t make it suitable for comparison with tokensInContract variable, which is more dynamic.

Recommendation:
The **getUnsoldTokens** could implement a simpler logic as mentioned below:

```
265     function getUnSoldTokens() onlyOwner external{
266         require(block.timestamp > endSale, "sale is not closed");
267         // check unsold tokens
268         uint256 tokensInContract = IERC20(tokenAddress).balanceOf(address(this));
269         require(tokensInContract > 0, "no unsold tokens in contract");
270         require(IERC20(tokenAddress).transfer(owner, tokensInContract), "transfer of token failed");
271     }
```

B.3 Tax is applicable on PreSale Contract Address

Status: CLOSED

Description:

Since the presale contract address is not mentioned in the transfer function of the Token contract, the taxation procedure is applicable for the presale contract.

It means whenever the owner transfers IFY tokens to the PreSale contract address, the actual amount of tokens received by the PreSale contract is less than the amount actually sent. Similarly if the PreSale contract sends IFY tokens to any other user, the token amount received by the user is lesser than than tokens sent by the contract.

This is because of the taxation mechanism which sends deducts a particular percentage of the transferred tokens and transfers 10% of the deducted amount to the owner's address and the remaining to the staking contract address.

Recommendation

Is this behaviour intended?

If this is not an intended behaviour the **transfer** and **transferFrom** functions in the **Token** contract must be modified in a way that it doesn't impose any tax charges when token is sent from or received by the PreSale contract Address.

Medium Severity Issues

B.4 transferOwnerShip function lacks a Zero Address Check.

Line no - 183

Status: CLOSED

Description:

The transferOwnership function doesn't validate the **_newOwner** address passed as a parameter.

```
60      function transferOwnership(address payable _newOwner) public onlyOwner {
61          owner = _newOwner;
62          emit OwnershipTransferred(msg.sender, _newOwner);
63      }
```

Recommendation:

The new owner's address must be checked with a **require** statement.

require(_newOwner != address(0),"Invalid address passed");

B.5 Division precedes Multiplication

Status : OPEN

Description:

The **ceil** function implements an inaccurate arithmetic logic which lead to wrong round value result every time this function is called.

```
    function ceil(uint256 a, uint256 m) internal pure returns (uint256 r) {
        require(m != 0, "SafeMath: to ceil number shall not be zero");
        return (a + m - 1) / (m * m);
    }
```

Low Severity Issues

B.6 External visibility should be preferred

Line no - 183

Status: CLOSED

Description:
Those functions that are never called throughout the contract should be marked as ***external*** visibility instead of ***public*** visibility.

This will effectively result in Gas Optimization as well.

Therefore, the following function must be marked as **external** within the contract:

```
183     function transferOwnership(address payable _newOwner) public onlyOwner {
184         owner = _newOwner;
185         emit OwnershipTransferred(msg.sender, _newOwner);
186     }
```

Informational

B.7 Explicit visibility declaration is missing

Line no - 211

Status: CLOSED

Description:
The following elements in the contract have not been assigned any visibility explicitly:

```
211         mapping(address => uint256) investor;
```

Recommendation:
Visibility specifiers should be assigned explicitly in order to avoid ambiguity.

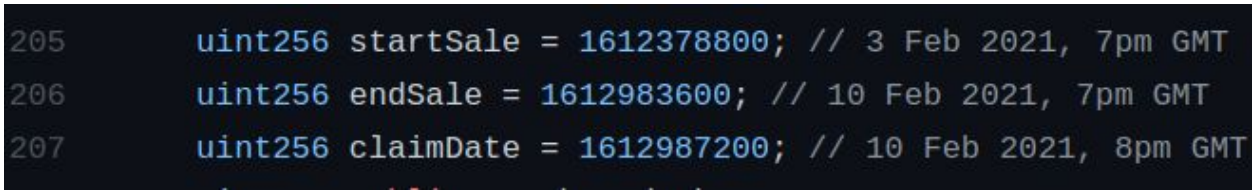
B.8 Constant declaration optimizes gas usage

Line no- 205 to 207

Status: CLOSED

Description:
State variables that are not supposed to change throughout the contract should be declared as **constant.**

Recommendation*:*
The following state variables needs to be declared as constant:



C. Contract Details

Name - Stake.sol

About the Contract:
Stake.sol is a staking contract which allows users to stake their IFY tokens in 4 different categories for high ROIs as follows:

- *1 week - 5% ROI*
- *1 month - 25% ROI*
- *3 months - 100% ROI*

- *6 months - 245% ROI*

The user can withdraw(*claim*) their staked tokens only when their selected stake period has come to an end.
Moreover, users can claim their reward after the staking period is over.

Issues Found – Code Review / Manual Testing

High Severity Issues

None

Medium Severity Issues

C.1 The “Stake” function doesn’t follow the Check Effects Interaction Pattern

Line no - 282
Status: CLOSED

Description:
The **STAKE** function implements an external call at Line 290 before updating the state variables in the contract as shown below:

```
288
289         // transfer the tokens from caller to staking contract
290         IFY.transferFrom(msg.sender, address(this), _amount);
291
292         // record it in contract's storage
293         stakers[msg.sender].stakedAmount = stakers[msg.sender].stakedAmount.add(_amount); // add to the stake or fresh stake
294         stakers[msg.sender].stakingOpt = optionNumber;
295         stakers[msg.sender].stakingEndDate = block.timestamp.add(stakingOptions[optionNumber.sub(1)].stakingPeriod);
296         stakers[msg.sender].rewardPercentage = stakingOptions[optionNumber.sub(1)].stakingPercentage;
```

Although the external call is made to the IFY token contract itself, it is not considered to update state variables after making an external call.

Recommendation:
Follow the **Check Effect Interaction Pattern.**

C.2 transferOwnerShip function lacks a Zero Address Check.

Line no - 212

Status: CLOSED

Description:

The transferOwnership function doesn't validate the **_newOwner** address passed as a parameter.

```
60      function transferOwnership(address payable _newOwner) public onlyOwner {
61          owner = _newOwner;
62          emit OwnershipTransferred(msg.sender, _newOwner);
63      }
```

Recommendation:

The new owner's address must be checked with a **require** statement.

require(_newOwner != address(0),"Invalid address passed");

C.3 Division precedes Multiplication

Status : OPEN

Description:

The **ceil** function implements an inaccurate arithmetic logic which lead to wrong round value result every time this function is called.

```
function ceil(uint256 a, uint256 m) internal pure returns (uint256 r) {
    require(m != 0, "SafeMath: to ceil number shall not be zero");
    return (a + m - 1) / (m * m);
}
```

Low Severity Issues

C.4 Event emitted after External Call

Line no [298](), [328](), [351]()

Status: CLOSED

Description:
As per the Check Effects Interaction Pattern, events should be emitted before the external call.

The following functions emits events after the external call:

- Stake
- ClaimReward
- Unstake

C.5 Return Value of External call is never used
Line no - 290, 326

Status: CLOSED

Description:
The return value of the externals made in the following functions are never used:

- Stake function at line **290**
- Unstake function at line **326**

Recommendation:
Effective use of all return values must be ensured within the contract.

Informational

C.6 Explicit visibility declaration is missing

Status: CLOSED

Description:
The following elements in the contract have not been assigned any visibility explicitly:

- Line 221
- Line 242

Recommendation:
Visibility specifiers should be assigned explicitly in order to avoid ambiguity.

C.7 External visibility should be preferred

Line no - 212

Status: CLOSED

Description:
Those functions that are never called throughout the contract should be marked as ***external*** visibility instead of ***public*** visibility.

This will effectively result in Gas Optimization as well.

Recommendation:

Therefore, the following function must be marked as **external** within the contract:

```
212     function transferOwnership(address payable _newOwner) public onlyOwner {
213         owner = _newOwner;
214         emit OwnershipTransferred(msg.sender, _newOwner);
215     }
```

Closing Summary

Overall, the smart contracts are very well written and adhere to guidelines.

While the majority of issues and bugs have been fixed, some new issues were found in the modified that might lead to unexpected behavior.

In order to achieve an effective performance from the contract, it is highly recommended to fix the new issues explained in the report above.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Infinity Yield 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 IFY Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.