SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Whitebit
Date: 18 July, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Whitebit
Approved By	Noah Jelich | Lead Solidity SC Auditor at Hacken OU
Tags	Vesting; Staking
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://whitebit.com
Changelog	29.05.2023 - Initial Review 30.06.2023 - Second Review 18.07.2023 - Third Review 24.07.2023 - Fourth Review

Table of contents

Introduction **5**
System Overview **5**
Executive Summary **6**
Risks **7**
Checked Items **8**
Findings **11**
Critical 11
High 11
H01. Integer Overflow 11
Medium 11
M01. Best Practice Violation: Unchecked Transfer 11
M02. Undocumented Functionality 12
M03. Division Before Multiplication 12
M04. Best Practice Violation: Checks-Effects-Interactions 13
M05. Uncontrolled Loop of Storage Interactions 13
M06. Insufficient Balance 13
M07. Access Control; Undocumented Functionality 14
M08. Integer Overflow 14
M09. Tx.origin As a Proxy for Msg.sender 15
Low 15
L01. Floating Pragma 15
L02. Missing Zero Address Validation 16
L03. Outdated Solidity Version 16
L04. Redundant Code 17
L05. Copy of Well-Known Contracts 17
L06. Missing Check 18
L07. Accumulation of Dust Values 18
L08. Redundant Code 19
L09. Inefficient Gas Model 19
L10. Wrong Check 20
L11. Copy of Well-Known Contracts 20
Informational 20
I01. Missing Events for Critical Value Updates 20
I02. Missing Event Indexed Parameters 21
I03. Boolean Equality 21
I04. Style Guide Violation: Order Of Layout 21
I05. Misleading Error Message 22
I06. Typo in Require Messages 22
I07. Variables That Can Be Set Immutable 22
I08. Functions That Can Be External 23
I09. Variables That Can Be Set Constant 23
Disclaimers **25**
Appendix 1. Severity Definitions **26**
Risk Levels 26

Impact Levels 27
Likelihood Levels 27
Informational 27
Appendix 2. Scope 28

Introduction

Hacken OÜ (Consultant) was contracted by Whitebit (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Whitebit is a suite of staking and vesting protocols with the following contracts:

- *FarmingRewards* – a contract that rewards users for staking their tokens. APY depends on the tokens provided by the owner and could not be calculated before reward tokens are deposited.
- *FarmingRewardsFactory* – a factory contract responsible for deploying new instances of FarmingRewards with some predefined values.
- *Treasure* – a contract for locking ERC20 tokens for a specific amount of time.
- *CustomOwnable* – custom implementation of OpenZeppelin’s Ownable without the ability to renounce ownership.

Privileged roles

- The owner of the FarmingRewardsFactory contract can change the lock amount for new deployments and the unlock commission percent.
- The owner of the Treasure contract can change the fee recipient address.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Technical description is provided.

Code quality

The total Code Quality score is **10** out of **10**.

- Development environment is configured
- Solidity Style Guide followed
- Efficient Gas model
- Updated Solidity version
- Best practices implemented

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- All possibilities are being tested.

Security score

As a result of the audit, the code contains **0** issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10**. The system users should acknowledge all the risks summed up in the risks section of the report.

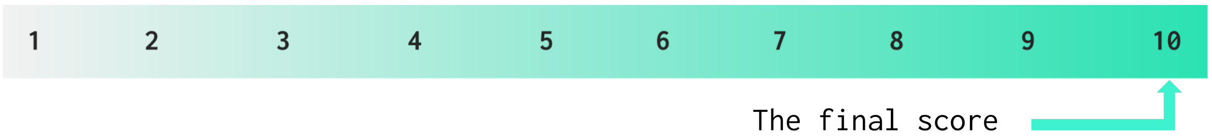

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
29 May 2023	7	8	1	0
30 June 2023	5	6	0	0

18 July 2023	2	0	0	0
24 July 2023	0	0	0	0

Risks

- Users can create their own pools, becoming the owners and introducing any token contracts as rewards and farming. **A malicious contract can be used for such tokens.**
- The system uses the contracts **timelock** and **WhiteSwapV2Factory**, which are **out of scope** and thus its logic **cannot be verified**.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	M08
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	L05
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	L01
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	M02
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	M07
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	M05
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	M03
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Passed	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Not Relevant	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	H01
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	H01
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	H01

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	M06
Style Guide Violation	Style guides and best practices should be followed.	Passed	I05
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. Data Consistency; Invalid Calculations

Impact	High
Likelihood	Medium

The contract *FarmingRewards* allows the *owner* to withdraw unused rewards balance after the staking period has reached the end.

The contract can use the same token for both deposits and rewards since it is not validated anywhere and the contract is not making correct accounting of that in *getRemainingRewardsForOwner()*.

Path: ./contracts/FarmingRewards.sol: getRemainingRewardsForOwner().

Recommendation: check the correct balances for rewards and deposits.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

■■ Medium

M01. Best Practice Violation: Unchecked Transfer

Impact	High
Likelihood	Low

The *ERC20* function *transfer()* is used repeatedly without the *SafeERC20* wrapper.

Tokens may not follow the *ERC20* standard and return *false* in case of transfer failure or not return any value at all. This can lead to a Denial of Service or unexpected behavior when dealing with some tokens. Hence, it is a best practice to use the *SafeERC20* wrapper when transferring tokens.

Paths:
./contracts/Treasure.sol: lock(), unlock().
./contracts/FarmingRewardsFactory.sol: deploy()

Recommendation: consider implementing the SafeERC20 library.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5).

M02. Undocumented Functionality

Impact	Medium
Likelihood	Medium

The check *totalReward >= epochDuration* compares tokens to time.

There is no apparent reason for this check and no documentation explaining why this is implemented in the code.

Path:
./contracts/FarmingRewardsFactory.sol

Recommendation: provide clear documentation about this check and/or update the code accordingly.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

M03. Floating Point Precision by Rounding Error

Impact	Low
Likelihood	High

The variable *wsFeeAmount* is calculated as a result of a division and later multiplication.

Solidity uses 256-bit precision for representing numbers, but some numbers cannot be accurately represented in 256 bits due to their fractional components. As a result, when performing arithmetic operations on such numbers, rounding errors may occur, leading to inaccurate results.

Since Solidity language does not have floating point numbers, performing divisions before multiplications result in a loss of precision.

Path:
./contracts/Treasure.sol: unlock().

Recommendation: it is recommended to perform divisions after multiplications to avoid loss of precision.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

M04. Best Practice Violation: Checks-Effects-Interactions

Impact	High
Likelihood	Low

State variables are updated after the external calls to the token contract.

As explained in Solidity Security Considerations, it is best practice to follow the checks-effects-interactions pattern when interacting with external contracts to avoid reentrancy-related issues.

Paths:
./contracts/Treasure.sol: lock().
./contracts/FarmingRewards: farm(), withdraw().
./FarmingRewardsFactory: deploy() → ITreasure.lock() contains a token contract call.

Recommendation: follow the checks-effects-interactions pattern when interacting with external contracts.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

M05. Uncontrolled Loop of Storage Interactions

Impact	Medium
Likelihood	Medium

The method *getRemainingRewardsForOwner()* calls *_earned()* for every account, whose number can be very high after some time.

The number of iterations of the loop in the function is uncontrolled as it depends on stored data, and it can reach the block Gas limit.

Path:
./contracts/FarmingRewards: getRemainingRewardsForOwner().

Recommendation: it is recommended to provide the possibility to separate the call in several transactions.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

M06. Insufficient Balance

Impact	High
Likelihood	Low

The contract *FarmingRewards* assumes that the expected amount of reward tokens was transferred to the contract without verifying its actual balance.

The deployment process will call the *transferFrom()* function from the rewards token, but the rewards token can be any token compliant with the *ERC20* interface and the token can transfer fewer tokens than expected (e.g. tokens with fees).

The *FarmingRewards* contract can not blindly trust the results of the deployment process from the factory contract. It should check if the data was correctly set and the expected amount of tokens was transferred in the initialization.

Transferring fewer than expected tokens can result in insufficient tokens balance for paying farm participants.

Path: ./contracts/FarmingRewards.sol: createEpoch().

Recommendation: check if the correct amount of rewards tokens was transferred during initialization.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

M07. Access Control; Undocumented Functionality

Impact	Medium
Likelihood	Medium

An external contract, stored as *timeLock*, gets the ownership of *FarmingRewardsFactory*.

The contract that acts as *timeLock* is out of scope, there is no additional documentation or information about the contract, despite having privileged access to several functions of the contract.

Path: ./contracts/FarmingRewardsFactory.sol: timeLock.

Recommendation: additional documentation about this contract/role should be provided.

Found in: bc97735

Status: Mitigated (With Customer notice: The Timelock contract is set to implement some methods of the FarmingRewardsFactoryContract to provide the ability to change community variables through voting.)

M08. Integer Overflow

Impact	High

Likelihood	Low

endDate is set as the sum of *startDate* + *epochDuration*, which are introduced by users. An overflow is possible in this situation.

An overflow happens when an arithmetic operation reaches the maximum or minimum size of a type.

Integer overflow and underflow happen when a numerical operation exceeds the maximum or minimum limit of the data type used to store the value and return to *0*. This can lead to unexpected and potentially harmful behavior in a smart contract, such as reverts and exceptions, and can be exploited by attackers to manipulate the contract.

Path: ./contracts/FarmingRewardsFactory.sol: deploy().

Recommendation: it is recommended to use vetted safe math libraries for arithmetic operations (OpenZeppelin's SafeMath) or use a Solidity Compiler equal or above 0.8.0, which reverts on underflows/overflows. It can also be considered to add limits to those values.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

M09. Tx.origin As a Proxy for Msg.sender

Impact	High
Likelihood	Low

Using tx.origin to get the msg.sender of a function call is not recommended since it only works for EOA users.

If any user wants to interact with the protocol via smart contract, the farm() function will record the information of the EOA that initiated the transaction instead of such smart contract. As a consequence, there can be data consistency issues and confusions.

Path: ./contracts/FarmingRewards.sol: farm().

Recommendation: it is recommended to propagate msg.sender via msgSender() from OpenZeppelin's Context instead of using tx.origin.

Found in: 0be73ed

Status: Fixed (Revised commit: c0675f5)

Low

L01. Floating Pragma

Impact	Medium

Likelihood	Low

As stated in SWC-103, contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the *pragma* helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Path:
./contracts/*.sol

Recommendation: lock the pragma version in all contracts.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L02. Missing Zero Address Validation

Impact	Low
Likelihood	Low

Additional checks against the *0x0* address should be included in the reported functions to avoid unexpected results.

Path:
./contracts/Treasure.sol: changeFeeRecipient(),

Recommendation: it is recommended to add zero address checks.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L03. Outdated Solidity Version

Impact	Medium
Likelihood	Low

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Using the current version of Solidity is generally considered best practice because it includes the latest updates and bug fixes. Newer versions address security vulnerabilities that may have been discovered in previous versions, making them more secure to use. Additionally, newer versions include new features and improvements that make writing and deploying smart contracts easier and more efficient. Using an outdated version of Solidity may expose your contracts to potential security risks and make it more difficult to take advantage of newer features and capabilities.

Paths:
./contracts/*

Recommendation: consider using an up-to-date compiler version (SWC-102).

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

L04. Redundant Code

Impact	Medium
Likelihood	Low

In some places of the code there are redundant checks that are already checked somewhere else and do not need to be checked again.

The *lock()* function from *Treasure* contract checks the *account* balance and allowance for the contract, but *ERC20* *transferFrom* will already fail due to insufficient balance or insufficient allowance if applicable.

The *farm()* function from the *FarmingRewards* contract checks the *msg.sender* allowance for the contract, but *ERC20* *transferFrom* will already fail due to insufficient allowance if applicable.

The function *_updateRewardForEpoch()* in *FarmingRewards.sol* calls *_lastTimeRewardApplicable* three times and *_rewardPerToken* two times.

When calling the *unlock()* method, *endDate* is set again, despite it being already set in *lock()*.

Redundant code reduces readability and increases the Gas cost.

Paths:
./contracts/Treasure.sol: lock().
./contracts/FarmingRewards.sol: farm().
./contracts/Treasure.sol: unlock().

Recommendation: remove redundant code.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L05. Copy of Well-Known Contracts

Impact	Low

Likelihood	Medium

Well-known contracts from projects like OpenZeppelin should be imported directly from source as the projects are in development and may update the contracts in future.

Path: ./contracts/CustomOwnable.sol

Recommendation: import the contract directly from source instead of modifying it.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L06. Missing Check

Impact	Medium
Likelihood	Low

Users are unable to create locks with duration less than one year. Trying to do so (e.g. creating a lock with 3 month duration) will result in a new lock with one year of duration. This behavior is not expected by the final user and can cause unexpected lock of funds for more time than wanted.

Path:
./contracts/Treasure: lock().

Recommendation: consider reverting when users try to create locks with durations less than one year or allow users to create locks with less than one year.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L07. Accumulation of Dust Values

Impact	Low
Likelihood	Medium

The function *farm()* introduces a require check *block.timestamp >= startDate*, which won't allow any user to engage in farming until the start time is reached.

Since users can not start deposits before the start time, it is unlikely the *FarmingRewards* contract will distribute all rewards balances to farmers.

Path:
./contracts/FarmingRewards: farm().

Recommendation: consider redesigning the require check so that users can farm from the start time of the farming.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

L08. Redundant Code

Impact	Low
Likelihood	Medium

The function *lock()* introduces a require check *_lockDuration == lockDurationModified*, which is added due to redundant logic in the function. The *lockDurationModified* variable is useless as *_lockDuration* could be used directly.

Paths:
./contracts/Treasure: lock().

Recommendation: consider refactoring the function code in order to be more assertive and straightforward, e.g by removing the *lockDurationModified* variable and, instead, only checking if *_lockDuration >= ONE_YEAR*.

Found in: 0be73ed

Status: Mitigated (With Customer notice: According to our business logic WSD lock is a form of payment for deploying your farming pool. Minimum WSD lock duration is 1 year, and if it's over 1 year, then WSD lock duration = farming epoch duration. We will make sure it will readily be apparent throughout our documentation and farming interface that you'll lock your WSD for a minimum of 1 year as a form of payment for farming pool deployment.)

L09. Inefficient Gas Model

Impact	Low
Likelihood	Medium

In the FarmingRewards constructor, the variable rewardRate is calculated as a function of the state variables rewardAmount and epochDuration instead of _rewardAmount and _epochDuration, expending more Gas than necessary.

In the same function, the emission of FarmingPoolInfo() follows the same pattern of using state variables instead of _stardDate, _endDate, _rewardAmount, _epochDuration and _minimumStakingExitTime.

Paths:
./contracts/Treasure: lock().

Recommendation: consider refactoring the function code in order to be more assertive and straightforward, e.g by removing the *lockDurationModified* variable and, instead, only checking if *_lockDuration >= ONE_YEAR*.

Found in: 0be73ed

Status: Fixed (Revised commit: c0675f5)

L10. Wrong Check

Impact	Medium
Likelihood	Low

In the function lock(), the check farmingStart > 0 should be checking epochDuration instead, since it seems to be a typo from the review from the first version.

Paths:
./contracts/Treasure: lock().

Recommendation: review the code and check to make sure it is the correct one.

Found in: 0be73ed

Status: Fixed (Revised commit: c0675f5)

L11. Copy of Well-Known Contracts

Impact	Low
Likelihood	Medium

Well-known contracts from projects like OpenZeppelin should be imported directly from source as the projects are in development and may update the contracts in future.

Path: ./contracts/FarmingRewardsFactory.sol: deployFarmingRewards()

Recommendation: it is recommended to use the [OpenZeppelin library create2] to deploy and get the address of a create2-generated contract.

Found in: 0be73ed

Status: Fixed (Revised commit: dce4b97)

Informational

I01. Missing Events for Critical Value Updates

Events should be *emitted* after sensitive changes take place, to facilitate tracking and notify off-chain clients following the contract’s activity.

Paths:
./contracts/Treasure.sol: constructor() → FeeRecipientChanged.
./contracts/FarmingRewards.sol: _updateRewardsForEpoch().
./contracts/FarmingRewardsFactory.sol: constructor() → LockAmountChanged, WSUnlockComission.

Recommendation: consider emitting events in said functions.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

I02. Missing Event Indexed Parameters

Some *events* are not using *indexed* parameters. If such *events* provide relevant enough information to fetch, they should use the keyword *indexed* in order to be properly searched and tracked off-chain.

Path:
./contracts/FarmingRewards.sol: FarmingPoolInfo() -> stakingToken, rewardToken.

Recommendation: consider adding indexed parameters for events that should be searchable.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

I03. Boolean Equality

Boolean values can be used directly and do not need to be compared to *true* or *false*.

Paths:
./contracts/FarmingRewards.sol: initialize().

Recommendation: remove boolean equality.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

I04. Style Guide Violation: Order Of Layout

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

1. Type declarations
2. State variables

3. Events
4. Errors
5. Modifiers
6. Functions
    a. constructor
    b. initializer (if exists)
    c. receive function (if exists)
    d. fallback function (if exists)
    e. external
    f. public
    g. internal
    h. private

Paths:
./contracts/FarmingRewards.sol
./contracts/FarmingRewardsFactory.sol

Recommendation: change order of layout to fit Official Style Guide.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

I05. Misleading Error Message

In the contract *FarmingRewards* in the *createEpoch* function, when a user inputs a start date that is less than current date, the contract will throw an error which says *“Provided start date too late”* when actually the start date is too early thus it is invalid.

Path:
./contracts/FarmingRewards.sol

Recommendation: fix the error message with the correct information.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

I06. Typo in Require Messages

In the contract *FarmingRewardsFactory* there is a typo present in four of the require statements in the deploy function:

then -> than.

Path:
./contracts/FarmingRewardsFactory.sol

Recommendation: fix typos.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

I07. Variables That Can Be Set Immutable

Impact	Low
Likelihood	Low

Use *immutable* keywords on state variables to limit changes to their state and save Gas.

Note that, by implementing changes in H01, *rewardToken* and *stakingToken* can be declared *immutable*.

Path:

./contracts/Treasure.sol: wsd, farmingRewardFactory.

Recommendation: consider using the keyword immutable for said variables.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

I08. Functions That Can Be External

Impact	Medium
Likelihood	Low

Public functions that are never called by the contract should be declared *external* to save Gas.

Path:
./contracts/FarmingRewards.sol: lastTimeRewardApplicable(), rewardPerToken().

Recommendation: remove redundant code.

Found in: bc97735

Status: Fixed (Revised commit: c0675f5)

I09. Variables That Can Be Set Constant

Impact	Low
Likelihood	Low

Use *constant* keywords on state variables that are set by default and never change to save Gas.

Use CAP_WORDS to declare said variables.

Paths:

./contracts/FarmingRewardsFactory.sol: MAX_COMISSION, minStakingExitTime, maxStakingExitTime, minEpochDuration.
./contracts/Treasure.sol: ONE_YEAR, MAX_COMISSION.

Recommendation: consider using the keyword constant for said variables.

Found in: bc97735

Status: Fixed (Revised commit: 0be73ed)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	Medium
Medium Likelihood	**High**	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/john-whitebit/farming
Commit	bc97735
Whitepaper	-
Requirements	Link
Technical Requirements	-
Contracts	File: eth/contracts/CustomOwnable.sol SHA3: c0da1caa2f57aa3288d6feba598a19c56e4d63ac011d34abf5bf42eeb0935754 File: eth/contracts/FarmingRewards.sol SHA3: ec84cf0adf1f2045b7f9eca0b41f5b5bca336bea47b2d07a14351bdcdf8f6e6a File: eth/contracts/FarmingRewardsFactory.sol SHA3: 3c0ba245c90688194dd4fe5a6a279948ba433b0ca317e1a94bb07acfc12ea9a1 File: eth/contracts/Treasure.sol SHA3: ba90cf6cf0ec8c9c7d5219510e5db2bcaf26915c87bca22f922b62a5794b7c66 File: eth/contracts/interfaces/IFarmingRewards.sol SHA3: 305be4c62d32dfdbbfd497788a9e2b38204de54798a44780a3bfdf14568a9283 File: eth/contracts/interfaces/ITreasure.sol SHA3: 9e4145318a062d5d367e825515ab9429c1ae0b67205a7cf6a18b7b1ca6436528 File: eth/contracts/interfaces/IWhiteSwapV2Factory.sol SHA3: 68dcd5a4e1925da756c414a0e50d19809f23fe26d415bb432519deb5871f0b1d

Second review scope

Repository	https://github.com/john-whitebit/farming
Commit	0be73ed
Whitepaper	-
Requirements	Link
Technical Requirements	-
Contracts	File: eth/contracts/FarmingRewards.sol SHA3: e1bce7cf13a12bc810b5d065d9ed5fce0984f6a12c4b3e2bd8a12946a3e60880 File: eth/contracts/FarmingRewardsFactory.sol SHA3: d7c5fa5b774aea20eece5ed4848d321c80bb3a5d20d221e8af848df455f0fcca

	File: eth/contracts/Treasure.sol SHA3: b0acbc32027d6142d566f1694d2f649efb30a50b88b823d99a12a70140315d17 File: eth/contracts/interfaces/IFarmingRewards.sol SHA3: 8b775148f78cf087d402678827d1011f3ecfc6a4b18c999c9eb1a39a8d1df2d1 File: eth/contracts/interfaces/ITreasure.sol SHA3: aa64c2cb1f58fd27bd1c3a8aeaff83fae2034e98a7df8df2de7f41ce7fa49d04 File: eth/contracts/interfaces/IWhiteSwapV2Factory.sol SHA3: 23897ee7a1daa19c4b183bf05163bf1b3ca19c0efcce90645fafc51312b2f34d

Third review scope

Repository	https://github.com/john-whitebit/farming
Commit	c0675f5
Whitepaper	-
Requirements	Link
Technical Requirements	-
Contracts	File: eth/contracts/FarmingRewards.sol SHA3: 506e990a6af03ac2006c2ee1ca3176dd9e0b42c279f5f9fedbe14a5764818cbd File: eth/contracts/FarmingRewardsFactory.sol SHA3: 56a59b1b6b5b12b1a8b33413edc5b2f82b2547e1065b7c041dc6f53428bd40b5 File: eth/contracts/Treasure.sol SHA3: 61b2a54e63939a7b6c79a626b338e8a58b1978939808fe39e4abef5349ea478a File: eth/contracts/interfaces/IFarmingRewards.sol SHA3: 698c114ef45fb23259737a2ac3a164e45e7f6db815c969de94df507a57806da0 File: eth/contracts/interfaces/ITreasure.sol SHA3: 9e1c3d1cc286de28895d9230053b3683b10ee5644e50dd83dad6f5b58770cce4 File: eth/contracts/interfaces/IWhiteSwapV2Factory.sol SHA3: d608c7a2af02ac0603010eb560dfc9a2664367fe587c8a430cd4f4cd3a968632

Fourth review scope

Repository	https://github.com/Whiteswap-exchange/farming-eth
Commit	dce4b97
Whitepaper	-
Requirements	Link
Technical Requirements	-

Contracts	File: eth/contracts/FarmingRewards.sol SHA3: 68d2e9950d9a57341bb3f69eb2b683cd3d0188a0fe5331cc398350d855832c13 File: eth/contracts/FarmingRewardsFactory.sol SHA3: 866c115abef80b0c9245773a9ea595d5ddb15d8becd4d0d4a398d3bdea5c203f File: eth/contracts/Treasure.sol SHA3: 103002c0f50e43b20b2ad901ab343d2c85c54973bed5899cb8c5a7c156be8fcb File: eth/contracts/interfaces/IFarmingRewards.sol SHA3: 698c114ef45fb23259737a2ac3a164e45e7f6db815c969de94df507a57806da0 File: eth/contracts/interfaces/ITreasure.sol SHA3: 9e1c3d1cc286de28895d9230053b3683b10ee5644e50dd83dad6f5b58770cce4 File: eth/contracts/interfaces/IWhiteSwapV2Factory.sol SHA3: d608c7a2af02ac0603010eb560dfc9a2664367fe587c8a430cd4f4cd3a968632