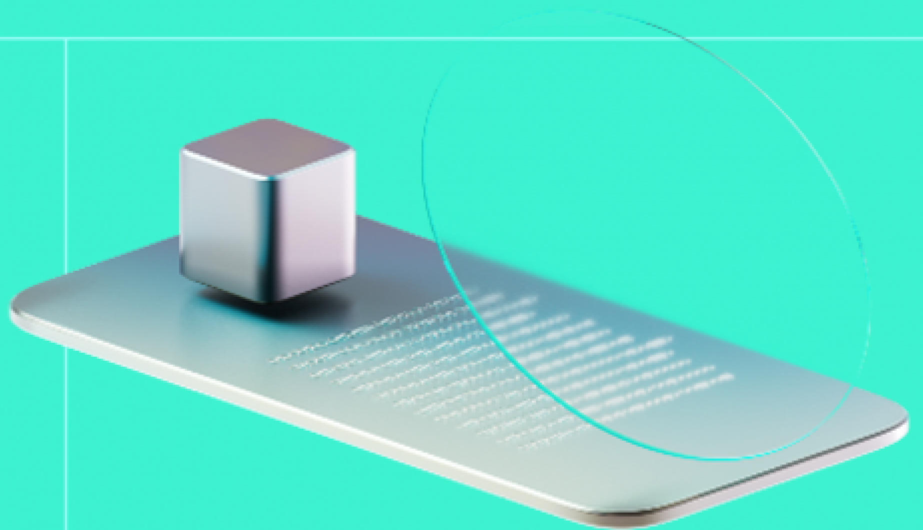
Smart Contract Code Review And Security Analysis Report



Customer: Qoodo

Date: 08/01/2024

We thank Qoodo for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

Qoodo integrates blockchain with AI and cloud technologies, offering a multifaceted platform for quality management and regulatory compliance, featuring transparent data tracking and product authentication.

Platform: Ethereum

Language: Solidity

Tags: ERC20 Staking

Timeline: - - -

Methodology: https://hackenio.cc/sc_methodology

Last Review Scope

Repository	https://github.com/NAPLOZZ/QDO_Staking
Commit	0cac0db

Audit Summary

10/10	8/10	100%	8/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.4/10

The system users should acknowledge all the risks summed up in the risks section of the report

16	15	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	1
High	2
Medium	1
Low	4

Vulnerability	Status
F-2023-0084 - Mismatch Between Documentation and Implementation	Accepted
F-2023-0065 - Missing Zero Address Validation	Fixed
F-2023-0067 - Redundant and Ineffective Implementation of Ownable	Fixed
F-2023-0068 - Floating Pragma	Fixed
F-2023-0069 - Inadequate Balance Checks in stake() and withdraw() functions	Fixed
F-2023-0070 - Redundancy of stakingTokenBalance Variable	Fixed
F-2023-0071 - Redundancy Of getStakingTokenBalance() Function	Fixed
F-2023-0072 - Potential underflow due to ambiguity in documentation	Fixed
F-2023-0076 - Lack of Phase Checks Leading to Potential Fund Lock	Fixed
F-2023-0077 - Absence of Events on Critical State Changes	Fixed
F-2023-0081 - Possible Underflow in stake() and withdraw() Functions Due to Reward Calculation	Fixed
F-2023-0082 - Absence of Emergency Withdrawal Function	Fixed
F-2023-0083 - Limitation in Reward Claiming Process Due to Combined withdraw() Function	Fixed
F-2023-0085 - Redundancy and Inefficiency in Admin Check in endPool() Function	Fixed
F-2024-0352 - Improper sanity check	Fixed
F-2024-0376 - Owner Can Renounce Ownership	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Qoodo
Audited By	Eren Gonen, Vladyslav Khomenko
Approved By	Grzegorz Trawinski
Website	https://qoodo.io/
Changelog	07/12/2023 - Preliminary Report
	08/01/2024 - Second Report

Table to Contents

System Overview	6
Executive Summary	7
Risks	8
Findings	9
Disclaimer	33
Appendix 1. Severity Definitions	34
Appendix 2. Scope	35

System Overview

Qoodo is a staking protocol with the following contracts:

Staking — Staking contract, represents a straightforward staking mechanism that enables users to stake ERC20 tokens and progressively accumulate rewards.

Privileged roles

- The admin of the Staking contract can
  - Set a new owner for the contract.
  - Set or modify the reward distribution speed.
  - Fund the contract for distribution of rewards with reward Tokens.
  - End the staking pool, transferring remaining tokens to the admin wallet.

Executive Summary

This report presents an in-depth analysis and scoring of the Customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **8** out of **10**.

- Technical description is sufficient.
- The description of how to run the project is missing.
- There is a mismatch between the documentation and the implementation.
- The code is covered with NatSpec comments.

Code quality

The total Code Quality score is **8** out of **10**.

- The code does not follow the Solidity best practices.

Test coverage

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

- Deployment and basic user interactions are covered with tests.
- Some test cases are failing.
- The project has not been adequately tested with multiple users.

Security score

Upon auditing, the code was found to contain **1** critical, **2** high, **1** medium, and **4** low severity issues, leading to a security score of **10** out of **10**. Subsequent to this evaluation, remedial actions have been successfully implemented, leading to the resolution of all identified issues across the critical, high, and medium severity categories, as well as 3 out of the 4 low severity issues, with one low severity issue remaining unresolved.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the Customer's smart contract yields an overall score of 1.9. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The admin has authority to change reward distribution speed after user staked their token

Findings

Vulnerability Details

F-2023-0076 - Lack of Phase Checks Leading to Potential Fund Lock - Critical

Description:

The current staking contract is missing checks if the contract was funded and if it was terminated using the `endPool()` function. This lack of validation leads to scenarios where user funds might be locked in the contract, either temporarily or permanently.

- **State of Unfunded Contract**
  The `stake()` function does not check if the admin has funded the contract. Consequently, users can stake their tokens irrespective of the contract's funding status. In this case, a problem arises when users try to withdraw their deposited tokens. The `withdraw()` function is designed to enable users to retrieve both their rewards and deposited tokens. However, if the admin has not funded the contract, the function will fail. When users attempt to withdraw, the `withdraw()` function tries to subtract the reward amount from `rewardsTokenBalance`. If the admin has not funded the contract, rewardsTokenBalance would be zero, leading to an underflow and contract revert. This results in a temporary lock of the deposited funds until the admin funds the contract.

```
 function withdraw(uint _amount) external updateReward(msg.sender) {
    ...
    rewardsTokenBalance -= reward;
    rewards[msg.sender] = 0;
    rewardsToken.safeTransfer(msg.sender, reward);
}
```

- **State of Contract Termination**
  In a scenario where the admin decides to terminate the pool, indicating that the contract will no longer be used, users can still stake their tokens due to a missing validation in the `stake()` function. Since there will not be sufficient rewards in this case, users' deposited funds become stuck in the contract. Whether this lock is temporary or permanent depends on the admin's subsequent actions.

Impact:

- **Risk of Fund Lock:** Users face the risk of their funds being temporarily or permanently locked in the contract due to the lack of phase checks.
- **Operational Risk:** The contract's inability to adapt to its funding and operational state can lead to user dissatisfaction and potential reputational damage.

This lack of checks creates potential scenarios that could lead to users' funds being locked within the contract.

Path: ./contracts/Staking.sol: endPool(), stake(), withdraw(), fund()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Critical

Impact: 5/5

Likelihood: 5/5

Recommendations

Recommendation:

Implement Funding Status Check:
To enhance the safety and reliability of the **Staking** contract, it is recommended to implement a mechanism that checks both the funding status of the contract and the sufficiency of funds to cover all potential user rewards before allowing users to execute the `stake()` function.

Pause Mechanism Post-Contract Termination:
Implement a pause mechanism: Once this function is called, indicating the termination of the contract, the `stake()` function should be disabled or paused to prevent any new stakes.

Enhanced Withdrawal Functionality:
Implement emergency withdrawal functionality to allow users to safely withdraw their staked tokens even in scenarios where the contract is paused or terminated. This ensures that users can always retrieve their staked assets irrespective of the contract's operational status.

Remediation(revised commit: c12e59c): The funded amount check was added to the `stake()` function. The funded amount would reset if the admin calls the `endPool()` function. It was no longer possible to stake any amount after `endPool()` was called, or if there were no funds inside the contract.

F-2023-0072 - Potential underflow due to ambiguity in documentation - High

Description:

The `endPool()` function in the Staking contract aims to close the staking pool and transfer any remaining reward tokens to the admin's wallet. However, the calculation involves subtraction of balances of potentially unrelated ERC20 tokens. It is not clear from the code or documentation whether tokens for staking and rewards are two different tokens or the same token.

The issue arises when subtracting `_totalSupply`, the total amount of staked tokens, from the balance of reward tokens `(rewardsToken.balanceOf(address(this)))`.

In the Ethereum blockchain, ERC20 tokens can have varying decimals, such as USDT and others. For instance, consider a scenario where the `rewardsToken` has a decimal of `6`, and the `stakingToken` has a decimal of `18`. If the admin deposits 10,000 reward tokens into the contract using the `fund()` function, the contract's reward token balance would increase to `1E10 (10,000,000,000)`, given the reward token's decimal is 6.

Should a user stake 1 staking token, `_totalSupply` would rise to `1E18 (1,000,000,000,000,000,000)`. The subtraction in the `endPool()` function:

```
function endPool() public {
        ...
        rewardsToken.safeTransfer(
            msg.sender,
            rewardsToken.balanceOf(address(this)) - _totalSupply
        );
    }
```

would then cause an underflow error, as `_totalSupply` exceeds `rewardsToken.balanceOf(address(this))`. This discrepancy can result in the reward funds being locked in the contract, especially if the admin needs to cancel the pool in emergency situations.

Path: ./contracts/Staking.sol: endPool()
Found In: 0cac0db

Status: Fixed

Classification

Severity: High

Impact:	5/5
Likelihood:	1/5

Recommendations

Recommendation:

Modify the `endPool()` function to correctly handle the transfer of remaining reward tokens. Instead of subtracting `_totalSupply` from `rewardsToken.balanceOf(address(this))`, consider implementing a more reliable method to calculate the amount of reward tokens that can be safely transferred to the admin.

Remediation(revised commit: c12e59c): The reward token and stake token will be set to the same address. The documentation was updated accordingly.

F-2023-0081 - Possible Underflow in stake() and withdraw() Functions Due to Reward Calculation - High

Description:

The `stake()` and `withdraw()` functions in the contract are modified by `updateReward()`, which updates the contract's state variables and the variables associated with `msg.sender` before executing the function. The `updateReward()` modifier internally calls the `earned()` function, which calculates the rewards for a given account. This calculation involves a call to `rewardPerToken()` function The `rewardPerToken()` function is designed to return zero under specific conditions, such as when `_totalSupply` equals zero, indicating no tokens are deposited in the contract.

This design can lead to issues in certain scenarios. For instance, consider a situation where the staking contract is active and correctly funded. Suppose a user executes the `stake()` function, which increases `_totalSupply`. If this user is the only one in the pool and decides to withdraw their entire deposited amount an hour later, the withdraw function will transfer the user's current rewards and update the `userRewardPerTokenPaid[account]` variable. In this scenario, the user successfully withdraws their entire amount. However, since they were the only person in the pool and withdrew everything, `_totalSupply` drops to zero. If the user attempts to execute the `stake()` function again, the `updateReward` modifier, which eventually calls the `earned()` function, will be triggered. This function will revert due to the subtraction:

```
rewardPerToken() - userRewardPerTokenPaid[account]
```

Actual code:

```
function earned(address account) public view returns (uint) {
    return
        ((_balances[account] *
            (rewardPerToken() - userRewardPerTokenPaid[account])) / 1e18) +
        rewards[account];
}
```

Since the `userRewardPerTokenPaid[account]` variable has increased due to the successful reward claim, and `_totalSupply` has decreased to zero (because the user withdrew all their tokens), `rewardPerToken()` will return zero. This is due to its conditional statement:

```
function rewardPerToken() public view returns (uint) {
        if (_totalSupply == 0) {
            return 0;
        }
        ...
}
```

Furthermore, there's another scenario where this calculation could result in an underflow for some users. If the admin reduces the `rewardRate` to a value lower than the current rate via `setNew()` function, and some users have already claimed their rewards, the `rewardPerToken()` function will return a value lower than `userRewardPerTokenPaid[account]`. Consequently, if these users attempt to execute `withdraw()` function, it will revert with an underflow error.

Therefore, the subtraction mentioned above can lead to an underflow in specific scenarios, temporarily blocking the stake function and potentially hindering withdrawals as well.

Path: ./contracts/Staking.sol: stake(), withdraw(), updateReward(), earned(), rewardPerToken(), setNew()
Found In: 0cac0db

Status: Fixed

Classification

Severity: High

Impact: 4/5

Likelihood: 4/5

Recommendations

Recommendation: Modify the `earned()` and `rewardPerToken()` functions to handle scenarios where `_totalSupply` is zero or `rewardRate` is reduced, preventing arithmetic underflow.

Remediation(revised commit: c12e59c): If the `rewardPerToken()` function returns zero, the `earned()` function will immediately return zero, making underflow no longer possible.

Evidences

Forge Test

Reproduce:

POC Steps:

Initial Staking and Withdrawal:

- A user stakes tokens, increasing **_totalSupply**.
- Later, the user withdraws their entire stake, leading to **_totalSupply** being reduced to zero.

Reward Calculation Post-Withdrawal:

- Upon withdrawal, the user's **userRewardPerTokenPaid[account]** is updated with their claimed reward amount.
- If the user attempts to stake again, the **updateReward()** modifier is invoked, which calls **earned()**.
- Inside **earned()**, the operation **rewardPerToken() - userRewardPerTokenPaid[account]** is performed.

Underflow in Reward Calculation:

- Since **rewardPerToken()** returns zero (due to **_totalSupply** being zero), and **userRewardPerTokenPaid[account]** is a positive value (from the previous reward claim), this subtraction results in an underflow.

Temporary Blockage of stake():

- The underflow in the **earned()** function causes a revert, temporarily preventing the user from staking again.

POC Code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../contracts/Staking.sol";
import "../contracts/Qoodo.sol";
import "../contracts/MockERC20.sol";
import "ds-test/test.sol";
import "forge-std/console.sol";

contract EarnedDOSPOC is Test {
    Staking public staking;
    MockERC20 public stakingToken;
    MockERC20 public rewardToken;
    address public alice;
    uint256 constant FUND_AMOUNT = 1E18;
    uint256 constant USER_DEPOSIT_AMOUNT = 1E18;
    uint256 constant ONE_HOUR = 3600;

    function setUp() public {
        stakingToken = new MockERC20(18);
        rewardToken = new MockERC20
```

See more

Results:

```
[FAIL. Reason: panic: arithmetic underflow or overflow
(0x11)] testEarnedDOS() (gas: 386907)

Logs:

Alice Reward:  3600
```

```
Alice Staked:  1000000000000000000

Reward Per Token:  3600

Stored Reward Per Token:  0

Last Update Time:  0

Total Deposit:  0

Reward Per Token:  0

User Reward Per Token Paid:  3600
```

Files: EarnedDOSPOC.t.sol

MockERC20.sol

F-2024-0352 - Improper sanity check - Medium

Description: When user unstakes, the contract needs to make sure that user:

- only gets only as much as they staked
- gets rewards according to their stake but only as much as there is undistributed rewards

Current implementation of the contract also checks the balance of the user and only allows them to withdraw as much as they already have.

```
require(stakingToken.balanceOf(address(msg.sender)) >=
_amount, 'Cannot withdraw more than the balance');
```

This means if user stakes all their tokens, they wouldn't be able to withdraw anything unless they acquire some tokens elsewhere.
Path: ./contracts/Staking.sol
Found in: a0597db

Assets:

- Staking.sol [https://github.com/hknio/QDO_Staking-7318c880426a78dd17d14ce273e]

Status: Fixed

Classification

Severity: Medium

Impact: 2/5

Likelihood: 3/5

Recommendations

Recommendation: Remove the following line:

```
require(stakingToken.balanceOf(address(msg.sender)) >=
_amount, 'Cannot withdraw more than the balance');
```

Remediation(revised commit: 3ae5704): The mentioned line was removed. The `withdraw()` function will not revert under the previously described specific conditions anymore.

F-2023-0069 - Inadequate Balance Checks in stake() and withdraw() functions - Low

Description:

The **Staking** contract, lacks adequate balance checks in its `stake()` and `withdraw()` functions. This lack of proper checks leads to inappropriate revert messages and potential confusion for users.

Inadequate Balance Validation in `stake()`:
The `stake()` function in the Staking contract lacks a preliminary check to verify if the user has a sufficient token balance before executing the `safeTransferFrom()` call. While `safeTransferFrom()` inherently reverts if the balance is insufficient, the absence of an explicit check upfront can lead to less descriptive revert messages. This could potentially confuse users about the reason for the transaction failure. Introducing an initial balance validation could provide a clearer, custom error message and potentially save gas by avoiding execution of subsequent operations if the transaction is destined to fail.

Insufficient Balance Checks in `withdraw()`:
The `withdraw()` function lacks a validation to ensure that the user's balance is sufficient to cover the withdrawal amount. If a user who has never staked tokens (balance = 0) attempts to withdraw, the contract will revert due to an underflow error instead of insufficient balance during the subtraction `_balances[msg.sender] -= _amount` This could be misleading and confusing, as it does not clearly indicate an insufficient balance.

Additionally, the `withdraw()` function does not validate if the provided withdrawal amount is less than or equal to `_totalSupply`, `stakingTokenBalance`, and `rewardsTokenBalance`. If the amount exceeding these values leads to a revert due to underflow, resulting in unclear error messages for users.

Such issues may result in users receiving error messages that do not accurately describe the problem, like insufficient balance. These unclear revert messages and underflow errors can be particularly baffling for users not well-versed in the contract's inner workings.

Path: ./contracts/Staking.sol: stake(), withdraw()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: **Implement Pre-Transfer Checks in `stake()`:**

- Add a balance check to ensure the user has enough tokens to stake before executing the transfer. This prevents state changes in case of insufficient funds.

Balance Validation in `withdraw()`:

- Introduce checks to confirm that the user's staking balance covers the withdrawal amount and that the withdrawal amount does not exceed the `stakingTokenBalance`, `_balances[msg.sender]`, `_totalSupply`, `rewardsTokenBalance` . Provide clear and informative revert messages for insufficient balance scenarios.

Remediation(revised commit: 3ae5704): The necessary checks was added to the `stake()` and `withdraw()` functions. The structure of the `withdraw()` function was changed; if there is not a necessary reward amount, users will be able to withdraw their initial amount without any revert.

[F-2023-0083] - Limitation in Reward Claiming Process Due to Combined withdraw() Function - Low

Description: In the **Staking** contract, the `withdraw()` function is designed to handle both the withdrawal of staked tokens and the claiming of reward tokens. However, this combined functionality presents a limitation for users who wish to claim their reward tokens without withdrawing their staked tokens. The function includes a condition

`require(_amount > 0, "Cannot withdraw 0");`

Actual code:

```
function withdraw(uint _amount) external updateReward(msg.sender) {
    require(_amount > 0, "Cannot withdraw 0");
    _totalSupply -= _amount;
    stakingTokenBalance -= _amount;
    _balances[msg.sender] -= _amount;
    stakingToken.safeTransfer(msg.sender, _amount);
    uint reward = rewards[msg.sender];
    rewardsTokenBalance -= reward;
    rewards[msg.sender] = 0;
    rewardsToken.safeTransfer(msg.sender, reward);
}
```

which implies that users are required to withdraw a minimum of one staked token to claim their rewards. This design choice not only restricts user flexibility but also potentially increases gas costs due to the necessity of two token transfers (staking token and reward token) within the same transaction.

Path: ./contracts/Staking.sol: withdraw()
Found In: 0cac0db

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: Consider either splitting the `withdraw()` function into two distinct functions – one for withdrawing staked tokens and another for claiming reward tokens –

or alternatively, modifying the existing `withdraw()` function to allow users to claim rewards without having to withdraw their staked tokens.

Remediation(revised commit: 3ae5704): The structure of the `withdraw()` function was changed; if there is not enough reward amount in the contracts, users will be able to withdraw their initial amount.

F-2023-0084 - Mismatch Between Documentation and Implementation - Low

Description: The documentation for the **Staking** contract specifies that the `endPool()` function is designed to terminate the staking pool, transfer the remaining tokens to the admin, and self-destruct the contract. However, upon reviewing the actual implementation of the `endPool()` function, it is evident that there is no self-destruct functionality present. This discrepancy between the documentation and the contract's code leads to confusion and potential misunderstandings about the contract's behavior and capabilities.

Path: ./contracts/Staking.sol: endPool()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Accepted

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: Update the documentation to accurately reflect the current implementation of the `endPool()` function.

Remediation: Since the initial review, no changes related to self-destruct in the documentation was observed.

F-2024-0376 - Owner Can Renounce Ownership - Low

Description: The project's contract utilizes **OpenZeppelin's Ownable**, which includes the `renounceOwnership` function. This function allows the current owner to permanently transfer ownership of the contract to the zero address, effectively leaving the contract without an owner. While this feature can be useful in certain scenarios where decentralization and immutability are desired, it also poses significant risks. If ownership is renounced unintentionally or without due consideration, it can lead to the loss of control over certain critical functionalities like `fund` that are restricted to the owner.

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Low

Impact: 1/5

Likelihood: 1/5

Recommendations

Recommendation: Override `renounceOwnership` and disable its functionality.

Remediation(revised commit: 3ae5704): The function was overridden; a revert statement was implemented.

F-2023-0065 - Missing Zero Address Validation - Info

Description: In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "**zero address**". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.

The "**Missing zero address control**" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.

For instance, consider a contract that includes a function to change its owner. This function is crucial, as it determines who has administrative access. However, if this function lacks proper validation checks, it might inadvertently permit the setting of the owner to the zero address. Consequently, the administrative functions will become unusable.

There `constructor()` and `setNewOwner()` functions is not protected against use of **zero address**.

Path: ./contracts/Staking.sol: constructor(), setNewOwner()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: Implement zero address validation for the given parameters. This can be achieved by adding require statements that ensure address parameters are not the zero address.

Remediation(revised commit: c12e59c): Zero address validation was implemented.

F-2023-0067 - Redundant and Ineffective Implementation of Ownable - Info

Description:

The **Staking** contract, designed for ERC20 token staking and reward distribution, imports the **Ownable.sol** from the OpenZeppelin library but does not inherit or implement its functionality. Instead, the contract defines an **admin** variable and utilizes a custom `onlyOwner()` modifier for administrative functions like `setNewOwner()` and `setNew()`. This approach leads to a redundancy in importing **Ownable.sol**, as it is features are not utilized.

Path: ./contracts/Staking.sol
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation:

Custom ownership management implementations may pose risks if not meticulously crafted, particularly when compared to the proven **Ownable** or **Ownable2Step** pattern. It's advisable to refactor the contract to inherit from **Ownable** or **Ownable2Step**, making use of its established ownership management functions. This approach guarantees a more robust and secure ownership feature.

Remediation(revised commit: 3ae5704): The **Ownable** pattern was implemented, and the mentioned redundant functions and variables was removed.

F-2023-0068 - Floating Pragma - Info

Description: A **floating pragma** in Solidity refers to the practice of using a pragma statement that does not specify a fixed compiler version but instead allows the contract to be compiled with any compatible compiler version. This issue arises when pragma statements like pragma solidity `^0.8.0` are used without a specific version number, allowing the contract to be compiled with the latest available compiler version. This can lead to various compatibility and stability issues.

Version Compatibility: Using a floating pragma makes the contract susceptible to potential breaking changes or unexpected behavior introduced in newer compiler versions. Contracts that rely on specific compiler features or behaviors may break when compiled with a different version.

Interoperability Issues: Contracts compiled with different compiler versions may have compatibility issues when interacting with each other or with external services. This can hinder the interoperability of the contract within the Ethereum ecosystem.

The project uses floating pragma `^0.8.20`.

Path: ./contracts/Qoodo.sol,
./contracts/Staking.sol
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: It is recommended to use a fixed pragma statement that specifies a known, well-tested compiler version. This helps ensure the stability, security, and predictability of the smart contract throughout its lifecycle.

Remediation(revised commit: c12e59c): The **Solidity** version is set to version **0.8.20**

F-2023-0070 - Redundancy of stakingTokenBalance Variable - Info

Description: In the Staking contract, there are two variables, `stakingTokenBalance` and `_totalSupply`, that are updated simultaneously in the `stake()` and `withdraw()` functions. Both variables essentially track the same quantity – the total amount of staking tokens in the contract. This redundancy leads to unnecessary complexity, gas consumption and potential confusion, as both variables are accessible externally and provide the same information.

Path: ./contracts/Staking.sol
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: Consider removing the `stakingTokenBalance` variable and rely solely on `_totalSupply` for tracking the total staked tokens.

Remediation(revised commit: c12e59c): The variable was removed.

F-2023-0071 - Redundancy Of getStakingTokenBalance() Function - Info

Description: The `stakingTokenBalance` variable defined as **public** and also there is a `getStakingTokenBalance()` getter function created in order to access to this public variable. In Solidity, when a state variable is declared as public, the compiler automatically creates a getter function for it. This feature provides external access to the variable's value without the need for manually defined getter functions. Despite this, the contract includes explicitly defined `getStakingTokenBalance()` function to access this `stakingTokenBalance` public variable. This redundancy results in duplicate functions, which unnecessarily increase the contract's size and deployment cost.

Path: ./contracts/Staking.sol: getStakingTokenBalance()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: Consider removing the redundant function to streamline the code and improve efficiency.

Remediation(revised commit: c12e59c): The redundant function was removed.

F-2023-0077 - Absence of Events on Critical State Changes - Info

Description: The functions does not emit events on change of important values. The events not emitted for the following functions:

- `setNewOwner()`
- `setNew()`

In Solidity, events are crucial for logging significant state changes, especially for functions that alter critical contract parameters or ownership. The absence of such events in these functions leads to a lack of traceability, which is essential for effective contract monitoring.

Path: contracts/Staking.sol: setNewOwner(), setNew()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: Introduce events and emit in the `setNewOwner()` and `setNew()` functions to log changes in ownership and reward distribution speed, respectively.

Remediation(revised commit: 3ae5704): The `setNewOwner()` function was removed. An event was implemented in the `setNew()` function.

F-2023-0082 - Absence of Emergency Withdrawal Function - Info

Description: The **Staking** contract currently lacks an emergency withdrawal function, which is a significant oversight in terms of user fund security and contract flexibility. An emergency withdrawal function is essential to allow users to retrieve their staked funds in situations where normal contract operations are disrupted. This could include scenarios where the contract is prematurely ended by the admin, becomes underfunded, or encounters other operational issues that prevent users from withdrawing their funds along with their rewards.

Path: contracts/Staking.sol
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: **Implement an Emergency Withdrawal Function:**

- Introduce a function that allows users to withdraw their staked funds without claiming rewards in emergency situations. This function should be accessible when the contract is in a non-operational state or when normal withdrawal mechanisms fail.

Define Emergency Conditions:

- Clearly specify the conditions under which the emergency withdrawal function can be activated, such as contract termination by the admin or insufficient reward funds.

Remediation(revised commit: 3ae5704): The structure of the `withdraw()` function was changed. Users will be able to withdraw their initial amount in the event that `endPool()` is called or if the contract does not have enough funds to cover user funds. Under these circumstances, an emergency function is not required.

F-2023-0085 - Redundancy and Inefficiency in Admin Check in endPool() Function - Info

Description: In the **Staking** contract, the `endPool()` function manually checks if the caller is the admin using a require statement;

```
function endPool() public {
        // Check if the caller is the admin
        require(msg.sender == admin, "You are not the admin");
…
        );
    }
```

However, the contract already includes an `onlyOwner()` modifier designed for this purpose. The manual implementation of the admin check in `endPool()` is redundant and less efficient compared to using the existing `onlyOwner()` modifier. This redundancy not only increases the contract size but also affects code readability and maintainability.

Path: contracts/Staking.sol: endPool(), onlyOwner()
Found In: 0cac0db

Assets:

- Staking.sol [https://github.com/NAPLOZZ/QDO_Staking]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation: Replace the manual admin check in the `endPool()` function with the `onlyOwner()` modifier to leverage its existing functionality and reduce redundancy.

Remediation(revised commit: 3ae5704): The `onlyOwner()` modifier was implemented.

Observation Details

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.

Severity	Description
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 will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details - Initial

Repository	https://github.com/NAPLOZZ/QDO_Staking
Commit	0cac0dbf7270674ad0ec193b028351ef05eb5123
Whitepaper	https://docs.qoodo.io/
Requirements	Confidential
Technical Requirements	Confidential

Scope Details - Second

Repository	https://github.com/NAPLOZZ/QDO_Staking
Commit	c12e59c03dd5b31ea8ecddcf6ffecedf4b2c6118
Whitepaper	https://docs.qoodo.io/
Requirements	Confidential
Technical Requirements	Confidential

Scope Details - Third

Repository	https://github.com/NAPLOZZ/QDO_Staking
Commit	3ae5704cde07ce295cec474528d9339c0e8bdbf4
Whitepaper	https://docs.qoodo.io/
Requirements	Confidential
Technical Requirements	Confidential

Contracts in Scope

./contracts/Staking.sol