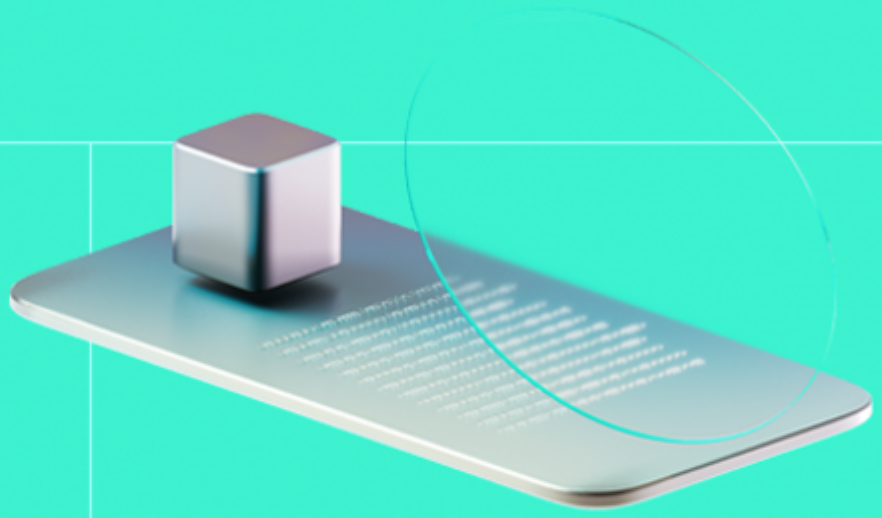
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Kyotoprotocol

Date: 16/04/2024

We express our gratitude to the Kyotoprotocol team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

KyotoProtocol is a carbon-negative blockchain designed to scale the Voluntary Carbon Market (VCM) and foster the growth of Regenerative Finance (ReFi) by leveraging Web3 technologies.

Platform: EVM

Language: Solidity

Tags: Staking, Vesting

Timeline: 09/04/2024 - 16/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/KyotoCarbonCo/chain-contracts
Commit	40b3c32

Audit Summary

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

Total 10/10

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

4	3	0	1
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	1
Medium	0
Low	2

Vulnerability	Status
F-2024-2102 - Missed pull over push pattern in RewardsCollector	Mitigated
F-2024-2099 - Faulty rewards accountancy logic in staking contract	Fixed
F-2024-2100 - Potential locking of staked NFTs due to changeable NFT address	Fixed
F-2024-2103 - Hardcoded APY in _rewardsPerTick function	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 Kyotoprotocol
Audited By	Niccolò Pozzolini
Approved By	Przemyslaw Swiatowiec
Website	https://kyotoprotocol.io
Changelog	11/04/2024 - Preliminary Report; 16/04/2024 - Remediation Report

Table of Contents

System Overview

The scope is composed by the chain contracts of the project KyotoProtocol:

- KyotoVault: Simple Vault contract that allows withdrawing funds by authorized addresses
- KyotoVesting: contract used to prevent migrated funds from flooding Kyoto Network with liquidity. After migration, tokens are put in the vesting contract and released accordingly to the vesting schedule - linearly each day over 18 months.
- MigrationStorage: contract used to handle the migration process for transferring funds to a vesting contract making sure that the same migration isn't processed twice.
- NodeNFTRegistry: an ERC721 NFT collection contract, used to represent ownership of Kyoto Network Nodes.
- NodeNFTStaking: staking contract for NFTs. Users can stake their NFTs in the contract and receive Kyoto in exchange at a fixed rate. It can be unstaked at any time with no penalties.
- RewardsCollector: contract that can receive Kyoto and distribute it to a configured set of targets with different weights.
- StakingPool: staking contract for Kyoto, with limits on how much can be staked at any time.
- ValidatorManager: contract that manages the validators of Kyoto Chain.

Privileged roles

- KyotoVault
  - DEFAULT_ADMIN_ROLE: can manage roles
  - WITHDRAWER_ROLE: can withdraw funds
- KyotoVesting
  - DEFAULT_ADMIN_ROLE: can manage roles
  - MIGRATOR_ROLE: can create new vesting schedules
- MigrationStorage
  - MIGRATOR_ROLE: can start the migration process
- NodeNFTRegistry
  - DEFAULT_ADMIN_ROLE: can manage roles and act as NODE_MANAGER_ROLE
  - NODE_MANAGER_ROLE: can mint new NFTs and change URI config
- NodeNFTStaking
  - DEFAULT_ADMIN_ROLE : can manage roles
  - MANAGER_ROLE: can change NFT collection that can be staked in the contract
- RewardsCollector
  - DEFAULT_ADMIN_ROLE: can update targets and percentages
- StakingPool
  - DEFAULT_ADMIN_ROLE: can manage roles
  - MANAGER_ROLE: can change financial parameters of the contract
- ValidatorManager
  - owner: can update the validators set

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 **10** out of **10**.

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

Code quality

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

- The development environment is configured.

Test coverage

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

- Deployment and basic user interactions are covered with tests.
- Negative cases are covered.
- Interactions by several users are tested thoroughly.

Security score

Upon auditing, the code was found to contain **0** critical, **1** high, **0** medium, and **2** low severity issues, leading to a security score of **10** out of **10**.

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 **10**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- **Dependency on Unaudited External Libraries**: The project utilizes libraries or contracts without security audits, potentially introducing vulnerabilities. This compromises the security of the audited system, making it susceptible to attacks exploiting these external weaknesses. Libray used: PRBMath.

Findings

Vulnerability Details

F-2024-2099 - Faulty rewards accountancy logic in staking contract - High

Description:

Both the staking contracts `NodeNFTStaking.sol` and `StakingPool.sol` implement a staking mechanism that allocates a reward rate to each user and only updates the `rewardsPool` variable upon unstaking. However, the current implementation has a flaw in the rewards accounting logic.

The issue lies in the fact that the `rewardsPool` variable only gets subtracted by the unwithdrawn rewards, and not by the total rewards accrued by the user. According to this implementation, if a user claims his rewards right before unstaking, the `rewardsPool` would stay untouched.

This faulty logic leads to an incorrect accountancy of rewards in the contract. Consequently, the output of the function `_rewardsMaxTick()` will be skewed upwards, violating the requirement that the contract holds sufficient balance to pay the rewards up to the tick returned by `_rewardsMaxTick()`.

Here is the problematic code snippet from `NodeNFTStaking`:

```
function unstake(uint256 tokenId) public {
uint256 amount = rewards(tokenId);

Stake storage stake_ = staking[tokenId];
stake_.stakeOwner = address(0);
totalRewardsCorrection -= stake_.appliedCorrection;
totalRewardsPerTick -= REWARDS_PER_TICK;
rewardsPool -= amount;

emit Unstaked(msg.sender, tokenId, amount);
_transfer(msg.sender, amount);
nft.transferFrom(address(this), msg.sender, tokenId);
}
```

Assets:

- NodeNFTStaking.sol [https://github.com/KyotoCarbonCo/chain-contracts]
- StakingPool.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Classification

Severity: High

Impact:

Likelihood [1-5]: 5
Impact [1-5]: 3
Exploitability [0-2]: 0
Complexity [0-2]: 0
Final Score: 4.0 (High)
Hacken Calculator Version: 0.6

Recommendations

Remediation:

The line `rewardsPool -= amount;` should be replaced with `rewardsPool -= (stake_.claimedRewards + amount);` to correctly account for all the rewards accrued by the user.

Remediation (commit: c23fe01): The client applied the suggested fix.

Evidences

rewardsPool is not decreased if rewards are collected before unstaking

Reproduce:

```
function testAudit_DirectUnstake() public {
// initial state
console.log("initial rewardsPool: ");
console.logUint(stakingPool.rewardsPool());
// stake some
stakingPool.stake{value: 5 ether}(address(this));
// time passes
_warp(YEAR);
// direct unstake
uint balancePre = address(this).balance;
stakingPool.unstake(0);
uint balancePost= address(this).balance;
// final state
console.log("balance difference: ");
console.logUint(balancePost - balancePre);
console.log("final rewardsPool: ");
console.logUint(stakingPool.rewardsPool());
}
function testAudit_ClaimThenUnstake() public {
// initial state
console.log("initial rewardsPool: ");
console.logUint(stakingPool.rewardsPool());
// stake some
stakingPool.stake{value: 5 ether}(address(this));
// time passes
_warp(YEAR);
// claim rewards first, then unstake
uint balancePre = address(this).balance;
uint amt = stakingPool.rewards(address(this),0);
stakingPool.claimRewards(0, amt);
stakingPool.unstake(0);
uint balancePost= address(this).balance;
// final state
console.log("balance difference: ");
console.logUint(balancePost - balancePre);
console.log("final rewardsPool: ");
console.logUint(stakingPool.rewardsPool());
}
```

Results:

[PASS] testAudit_DirectUnstake() (gas: 128098)
Logs:
initial rewardsPool:
1000000000000000000
balance difference:
5999999999999995520
final rewardsPool:
4480

[PASS] testAudit_ClaimThenUnstake() (gas: 161047)
Logs:
initial rewardsPool:
1000000000000000000
balance difference:
5999999999999995520
final rewardsPool:
1000000000000000000

F-2024-2100 - Potential locking of staked NFTs due to changeable NFT address - Low

Description:

In the `NodeNFTStaking.sol` contract, the `nft` variable, which represents the address of the NFT collection that can be staked in this contract, can be changed through the `setNft()` function. This function can be called by anyone with the `MANAGER_ROLE`.

The issue arises when this function is triggered during active stakings. If the `nft` address is changed while NFTs are staked, those staked NFTs will get locked in this contract until the previous `nft` address gets restored. This is because the `unstake()` function uses the current `nft` address to perform the `transferFrom` operation.

Here is the problematic code snippet:

```
function setNft(address newNft) external onlyRole(MANAGER_ROLE) {
nft = ERC721Enumerable(newNft);
}
```

Assets:

- NodeNFTStaking.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 3
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 2.1 (Low)
Hacken Calculator Version: 0.6

Recommendations

Remediation:

To solve this issue, the `Stake` struct should include an additional field `address nft` which gets populated in the `stake()` function. Then, the `unstake()` function should use this stored `nft` address to perform the `transferFrom` operation. This ensures that the correct NFT address is always used, regardless of any changes to the `nft` variable in the contract.

Remediation (commit: c23fe01): The `nft` variable has been made immutable.

F-2024-2102 - Missed pull over push pattern in RewardsCollector - Low

Description:

In the `RewardsCollector.sol` contract, the `distribute()` function uses a push pattern to distribute funds to the configured targets. This pattern can potentially allow a malicious receiver to block the distribution process, leading to a Denial of Service (DoS) attack.

Here is the problematic code snippet:

```
function distribute() public {
//...
for (uint256 i; i < targetsLength; i++) {
uint256 distributionTargetAmt = (totalDistAmt * percentages[i]) / DE
NOMINATOR;
_distributeToAddress(targets[i], distributionTargetAmt); // @audit p
ush pattern used here
}
}
```

The line `_distributeToAddress(targets[i], distributionTargetAmt);` pushes the funds to the target addresses. If one of these addresses is a contract that reverts or consumes all the gas in the transaction, it could block the distribution process.

Assets:

- RewardsCollector.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Mitigated

Classification

Severity: Low

Impact:

Likelihood [1-5]: 2
Impact [1-5]: 3
Exploitability [0-2]: 0
Complexity [0-2]: 0
Final Score: 2.5 (Low)
Hacken Calculator Version: 0.6

Recommendations

Remediation:

A safer approach would be to use a pull pattern. This could be implemented by setting the withdrawable amounts for each target in the `distribute()` function and adding a `withdraw()` function that allows

the receivers to withdraw their funds. This way, even if one receiver fails to withdraw their funds, it won't affect the others.

Mitigation notes (commit: c23fe01): Only trusted addresses and EOAs should be provided as distribution targets. The documentation has been changed accordingly.

F-2024-2103 - Hardcoded APY in _rewardsPerTick function - Info

Description: In the `StakingPool.sol` contract, the `_rewardsPerTick()` function contains a hardcoded value representing the Annual Percentage Yield (APY) of 20%.

Assets:

- StakingPool.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Classification

Severity: Info

Recommendations

Remediation: To improve code readability and maintainability, this value should be declared as a constant at the top of the contract. This way, it's clear what this value represents and it can be easily updated if the APY changes in the future.

Remediation (commit: c23fe01): The APY has been defined as a constant variable.

Observation Details

F-2024-2098 - Redundant assignment in MigrationStorage's constructor - Info

Description:

In the `MigrationStorage.sol` contract, the constructor contains a redundant assignment where the `admin` variable is assigned to itself. This operation does not have any practical effect on the functionality of the contract but it does incur a slight gas cost. More importantly, it can lead to confusion and reduce the readability of the code. It is recommended to remove this redundant assignment to improve the clarity of the code.

```
/**
* @notice Constructor to initialize the contract with the admin, mig
rator, and KyotoVesting contract address.
* @param admin The address of the admin role.
* @param migrator The address of the migrator role.
* @param vesting The address of the KyotoVesting contract.
*/
constructor(address admin, address migrator, IKyotoVesting vesting)
{
admin = admin;
vestingContract = vesting;
_grantRole(DEFAULT_ADMIN_ROLE, admin);
_grantRole(MIGRATOR_ROLE, migrator);
}
```

Assets:

- MigrationStorage.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Recommendations

Remediation:

It is suggested to remove the redundant line `admin = admin;`

Remediation (commit: c23fe01): The redundant assignment has been removed.

F-2024-2101 - Redundant math in stake function - Info

Description:

In the `NodeNFTStaking.sol` contract, the `stake()` function contains redundant math during the computation of the `correction` variable. The term `(int256(totalRewardsPerTick) - oldRewardsPerTick)` is essentially equivalent to `REWARDS_PER_TICK` as `totalRewardsPerTick` is incremented by `REWARDS_PER_TICK` just before this calculation.

The same applies to the `StakingPool.sol` contract.

Assets:

- NodeNFTStaking.sol [https://github.com/KyotoCarbonCo/chain-contracts]
- StakingPool.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Recommendations

Remediation:

The line `int256 correction = (int256(totalRewardsPerTick) - oldRewardsPerTick) * int256(_currentTick() - 1);` should be replaced with `int256 correction = REWARDS_PER_TICK * int256(_currentTick() - 1);` to improve gas usage and code readability. This change simplifies the calculation and makes the code more straightforward to understand.

Remediation (commit: c23fe01): The suggested fix has been applied.

F-2024-2104 - Lack of Parameter Validation in setValidators Function - Info

Description: In the `ValidatorManager.sol` contract, the `setValidators()` function does not validate the length of the `validators` array parameter. This could potentially allow for an empty set of validators, which should not be permitted.

Assets:

- ValidatorManager.sol [https://github.com/KyotoCarbonCo/chain-contracts]

Status: Fixed

Recommendations

Remediation: To resolve this issue, a check should be added to ensure that the `validators` array has a minimum length before it is assigned to `_validators`.

Remediation (commit: c23fe01): The input array is now validated to have length > 0.

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 **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

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

Repository	https://github.com/KyotoCarbonCo/chain-contracts
Commit	40b3c3292e7bccf8cc04c0608dbe224e75a83789
Requirements	kyoto-flows.pdf
Technical Requirements	kyoto-flows.pdf, readme.md

Contracts in Scope

./src/KyotoVault.sol

./src/KyotoVesting.sol

./src/MigrationStorage.sol

./src/NodeNFTRegistry.sol

./src/NodeNFTStaking.sol

./src/RewardsCollector.sol

./src/StakingPool.sol

./src/ValidatorManager.sol

./src/interfaces/IKyotoVault.sol

./src/interfaces/IKyotoVesting.sol

./src/interfaces/IRewardsCollector.sol

./src/interfaces/ValidatorSmartContractInterface.sol

./src/libs/Errors.sol