SMART CONTRACT AUDIT REPORT

for

Kyoko

Prepared By: Xiaomi Huang

PeckShield

September 4, 2023

Document Properties

Client	Kyoko
Title	Smart Contract Audit Report
Target	Kyoko
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 4, 2023	Xuxian Jiang	Final Release
1.0-rc	September 2, 2023	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Kyoko protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Kyoko

Kyoko NFT lending is a peer-to-pool platform with the goal of becoming the first decentralized, dual-rate NFT lending protocol that supports multiple NFT collections and provides a comprehensive solution for liquidity issues in the NFT market. Users can participate in the protocol either as depositors or borrowers. Depositors provide ETH liquidity to the market and earn passive income, while borrowers can obtain loans by using NFTs as collateral. The protocol uniquely offers lending pools for a wide range of NFT collections, rather than limiting itself to blue-chip NFT projects. This ensures greater inclusivity for NFT owners, making the platform accessible to a broader range of participants. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Kyoko

Item	Description
Name	Kyoko
Website	https://www.kyoko.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 4, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/kyoko-finance/kyoko-nft-lending.git (c684b82)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/kyoko-finance/kyoko-nft-lending.git (63449f4)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Kyoko protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	3	■■■
Low	2	■■
Informational	0	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Kyoko Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect START_BIT_POSITION in ReserveConfiguration	Coding Practice	Resolved
PVE-002	Medium	Refreshness of LiquidityIndex in KyokoPool::burnLiquidity()	Business Logic	Resolved
PVE-003	Low	Incorrect BorrowRate Emission in KyokoPool::_executeBorrow()	Business Logic	Resolved
PVE-004	Medium	Inconsistent Storage Layout Between KyokoPool And KyokoPoolLiquidator	Coding Practice	Resolved
PVE-005	High	Improved Input Validation in liquidationCall/bidCall/claimCall	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Improper totalSupply Adjustment in BasicERC20::_burn()	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect START_BIT_POSITION in ReserveConfiguration

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `ReserveConfiguration`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1099 [2]

Description

The `Kyoko` protocol has a flexible mechanism to keep track of the configuration of current reserve pool. This mechanism is mainly implemented in the `ReserveConfiguration` contract. In the process of reviewing this contract, we notice an internal bitmap logic to handle the reserve configuration should be improved.

In the following, we show the key fields in each reserve pool's `ReserveConfiguration`. To facilitate the access of different fields, the contract has defined the starting position for each field. Our analysis shows that four fields have the wrong starting positions, i.e., `BID_TIME_START_BIT_POSITION`, `IS_FROZEN_START_BIT_POSITION`, `LOCK_START_BIT_POSITION` and `TYPE_START_BIT_POSITION`. They are respectively defined as 155, 179, 180, and 212, which need to be corrected as 149, 173, 174, and 206.

```
library ReserveConfiguration {
    uint256 constant RESERVE_FACTOR_MASK =                        0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000; // prettier-ignore
    uint256 constant BORROW_RATIO_MASK =                          0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000FFFF; // prettier-ignore
    uint256 constant PERIOD_MASK =                                0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000000000FFFFFFFF; // prettier-ignore
```

```
    uint256 constant MIN_BORROW_TIME_MASK =                    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0000000000FFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant ACTIVE_MASK =                             0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant LIQUIDATION_THRESHOLD_MASK =              0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0001FFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant BORROWING_MASK =                          0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant STABLE_BORROWING_MASK =                   0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFBFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant LIQUIDATION_TIME_MASK =                   0xFFFFFFFFFFFFFFFFFFFFFFFFFFE00007FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant BID_TIME_MASK =                           0xFFFFFFFFFFFFFFFFFFFFE000001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant FROZEN_MASK =                             0xFFFFFFFFFFFFFFFFFFFFDFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant LOCK_MASK =                               0xFFFFFFFFFFFFC0000003FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore
    uint256 constant TYPE_MASK =                               0xFFFFFFFFFFC03FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF; // prettier-ignore

    /// @dev For the factor, the start bit is 0 (up to 15), hence no bitshifting is needed
    uint256 constant BORROW_RATIO_START_BIT_POSITION = 16;
    uint256 constant PERIOD_START_BIT_POSITION = 32;
    uint256 constant MIN_BORROW_TIME_START_BIT_POSITION = 72;
    uint256 constant IS_ACTIVE_START_BIT_POSITION = 112;
    uint256 constant LIQUIDATION_THRESHOLD_START_BIT_POSITION = 113;
    uint256 constant BORROWING_ENABLED_START_BIT_POSITION = 129;
    uint256 constant STABLE_BORROWING_ENABLED_START_BIT_POSITION = 130;
    uint256 constant LIQUIDATION_TIME_START_BIT_POSITION = 131;
    uint256 constant BID_TIME_START_BIT_POSITION = 155;
    uint256 constant IS_FROZEN_START_BIT_POSITION = 179;
    uint256 constant LOCK_START_BIT_POSITION = 180;
    uint256 constant TYPE_START_BIT_POSITION = 212;
  ...
}
```

Listing 3.1: The `ReserveConfiguration` Contract

Recommendation Correct the above starting positions of affected fields in `ReserveConfiguration`

.

Status The issue has been fixed by the following commit: 83a5036.

3.2 Refreshness of LiquidityIndex in KyokoPool::burnLiquidity()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: KyokoPool
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

The Kyoko protocol allows depositors provide ETH liquidity to the market and earn passive income, while borrowers can obtain loans by using NFTs as collateral. The deposited liquidity is managed with a KyokoPool contract. While examining one public function burnLiquidity(), we notice it may not use the latest liquidity state.

To elaborate, we show below the burnLiquidity() function. It implements a rather straightforward logic in burning the initial liquidity in KToken of a reserve. However, it simply reuses the LiquidityIndex (from the current reserve state), which may not be timely updated yet. To fix, we need to update the reserve state and refresh the interest rate.

```
    function burnLiquidity(
        uint256 reserveId,
        uint256 amount
    ) external override {
        DataTypes.ReserveData storage reserve = _reserves[reserveId];
        address kToken = reserve.kTokenAddress;
        IKToken(kToken).burn(msg.sender, amount, reserve.liquidityIndex);
    }
```

Listing 3.2: KyokoPool::burnLiquidity()

Recommendation Timely update the reserve state before burning the requested liquidity in the above burnLiquidity() function.

Status The issue has been fixed by the following commit: 83a5036.

3.3 Incorrect BorrowRate Emission in KyokoPool::_executeBorrow()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: KyokoPool
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

The Kyoko protocol follows the key concept from the popular Aave protocol with the unique extensions to support peer-to-pool NFT lending. In the process of analyzing the current borrow logic, we notice the emitted Borrow event contains incorrect information.

To elaborate, we show below the code snippet from the affected _executeBorrow() function. If we focus on the final emitted event Borrow, the last field emitBorrowRate denotes the current borrow rate. We notice this rate depends on the current borrow mode: if it is a stable borrow mode, then it should be assigned as currentStableRate; otherwise, it should have the reserve.currentVariableBorrowRate. In other words, emitBorrowRate = stable ? currentStableRate : reserve.currentVariableBorrowRate, not the current emitBorrowRate = flag ? currentStableRate : reserve.currentVariableBorrowRate.

```
    function _executeBorrow(
        uint256 reserveId,
        address asset,
        uint256 nftId,
        uint256 interestRateMode,
        uint256 floorPrice,
        address onBehalfOf
    ) internal returns (uint256) {
        ExecuteBorrowParams memory vars;
        vars.reserveId = reserveId;
        vars.nft = asset;
        vars.nftId = nftId;
        vars.interestRateMode = interestRateMode;
        vars.floorPrice = floorPrice;
        vars.user = onBehalfOf;
        DataTypes.ReserveData storage reserve = _reserves[vars.reserveId];
        bool flag = _reservesNFTList[vars.reserveId].contains(vars.nft);
        DataTypes.InterestRateMode rateMode = DataTypes.InterestRateMode(
            vars.interestRateMode
        );
        ...
        uint256 emitBorrowRate = flag
            ? currentStableRate
            : reserve.currentVariableBorrowRate;

```

```
        emit Borrow(
            vars.reserveId,
            borrowId,
            vars.nft,
            vars.nftId,
            vars.interestRateMode,
            amountToBorrow,
            emitBorrowRate
        );
        return amountToBorrow;
    }
```

Listing 3.3: `KyokoPool::_executeBorrow()`

Recommendation Emit the `Borrow` event with the correct borrow rate.

Status The issue has been fixed by the following commit: `83a5036`.

3.4 Inconsistent Storage Layout Between KyokoPool And KyokoPoolLiquidator

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `KyokoPoolLiquidator`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

To facilitate the protocol management, `Kyoko` keeps the liquidation-related logic in a standalone `KyokoPoolLiquidator` contract and then delegates all liquidation calls to it. Naturally, there is a requirement on the storage consistency between `KyokoPool` and `KyokoPoolLiquidator`. Our analysis shows that there is a different storage layout which needs to be resolved before the deployment.

To elaborate, we show below the storage layouts of `KyokoPool` and `KyokoPoolLiquidator`. We notice it inherits from a few parent contracts, including `Initializable`, `IKyokoPool`, `KyokoPoolStorage`, `ContextUpgradeable`, `IERC721ReceiverUpgradeable`, and `KyokoPoolStorageExt`. For the delegate call, we expect the `KyokoPoolLiquidator` contract shares the same storage layout. However, it only inherits `Initializable`, `IKyokoPool`, `KyokoPoolStorage`, `ContextUpgradeable`, and `IERC721ReceiverUpgradeable`, but not `KyokoPoolStorageExt`. With that, we suggest the addition of `KyokoPoolStorageExt` at the end of inheriting contracts in `KyokoPoolLiquidator`.

```
contract KyokoPool is
    Initializable,
```

```
    IKyokoPool,
    KyokoPoolStorage,
    ContextUpgradeable,
    IERC721ReceiverUpgradeable,
    KyokoPoolStorageExt
{
    ...
}
```

Listing 3.4: Key Storage States Defined in KyokoPool

```
contract KyokoPoolLiquidator is
    Initializable,
    IKyokoPoolLiquidator,
    KyokoPoolStorage,
    ContextUpgradeable
{
    ...
}
```

Listing 3.5: Key Storage States Defined in KyokoPoolLiquidator

Recommendation Ensure the storage consistency between KyokoPool and KyokoPoolLiquidator.

Status The issue has been fixed by the following commit: 83a5036.

3.5 Improved Input Validation in liquidationCall/bidCall/claimCall

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: KyokoPoolLiquidator
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [3]

Description

As mentioned in Section 3.4, the Kyoko protocol makes use of a standalone KyokoPoolLiquidator contract to handle loan liquidation. In the process of analyzing liquidation-related functions, we notice they can be improved to validate the user input.

To elaborate, we show below an example function – liquidationCall(). As the name indicates, it allows users to liquidate the loans that have expired. This function takes three arguments: the first one reserveId represents the related reserve, the second one borrowId denotes the specific borrow for liquidation, the last one amount shows the liquidation amount. The first two parameters need to be cross-checked to ensure that the specific borrow occurs at the given `reserveId`. However, this cross-check is not performed, which may be abused to corrupt the liquidation process. The same issue is also applicable to other two routines, including `bidCall()` and `claimCall()`. The lack of cross-check may be exploited to steal the auctioned `NFT` at a skewed price.

```
    function liquidationCall(
        uint256 reserveId,
        uint256 borrowId,
        uint256 amount
    ) external payable override returns (uint256, string memory) {
        uint256 innerBorrowId = borrowId;
        uint256 _reserveId = reserveId;
        DataTypes.BorrowInfo storage info = borrowMap[innerBorrowId];
        DataTypes.ReserveData storage reserve = _reserves[_reserveId];
        uint256 amountToLiquidation = amount;

        (uint256 stableDebt, uint256 variableDebt) = Helpers
            .getUserDebtOfAmount(info.user, reserve, info.principal);

        ...
    }
```

Listing 3.6: `KyokoPoolLiquidator::liquidationCall()`

Recommendation Validate the given input to the above functions is consistent and expected. Note that the functions `ValidationLogic::validateRepay()` and `KToken::initialize()` can be similarly improved.

Status The issue has been fixed by the following commit: `83a5036`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [4]

Description

In the `Kyoko` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and contract upgrade). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contracts.

```
    function freezeReserve(uint256 reserveId) external onlyPoolAdmin {
        DataTypes.ReserveConfigurationMap memory currentConfig = _pool
            .getConfiguration(reserveId);

        currentConfig.setFrozen(true);

        _pool.setConfiguration(reserveId, currentConfig.data);

        emit ReserveFrozen(reserveId);
    }

    /**
     * @dev Unfreezes a reserve
     * @param reserveId The id of the reserve
     **/
    function unfreezeReserve(uint256 reserveId) external onlyPoolAdmin {
        DataTypes.ReserveConfigurationMap memory currentConfig = _pool
            .getConfiguration(reserveId);

        currentConfig.setFrozen(false);

        _pool.setConfiguration(reserveId, currentConfig.data);

        emit ReserveUnfrozen(reserveId);
    }

    /**
     * @dev Updates the reserve factor of a reserve
     * @param reserveId The id of the reserve
     * @param reserveFactor The new reserve factor of the reserve
     **/
    function setReserveFactor(uint256 reserveId, uint256 reserveFactor)
        external
        onlyPoolAdmin
    {
        DataTypes.ReserveConfigurationMap memory currentConfig = _pool
            .getConfiguration(reserveId);

        currentConfig.setReserveFactor(reserveFactor);

        _pool.setConfiguration(reserveId, currentConfig.data);

        emit ReserveFactorChanged(reserveId, reserveFactor);
    }

    /**
     * @dev Updates the borrow ratio of a reserve
     * @param reserveId The id of the reserve
     * @param ratio The new borrow ratio of the reserve
     **/
    function setBorrowRatio(uint256 reserveId, uint256 ratio)
        external
```

```
        onlyPoolAdmin
    {
        DataTypes.ReserveConfigurationMap memory currentConfig = _pool
            .getConfiguration(reserveId);

        currentConfig.setBorrowRatio(ratio);

        _pool.setConfiguration(reserveId, currentConfig.data);

        emit ReserveBorrowRatioChanged(reserveId, ratio);
    }
```

Listing 3.7: Example Privileged Operations in `KyokoPoolConfigurator`

```
    function authorizeLendingPool(address lendingPool) external onlyAdmin {
        WETH.approve(lendingPool, type(uint256).max);
    }
```

Listing 3.8: Example Privileged Operations in `KyokoPool`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain `EOA` account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed `DAO`.

Moreover, it should be noted that current contracts are to be deployed behind a proxy. And naturally, there is a need to properly manage the admin privileges as they are capable of upgrading the entire protocol implementation.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigatd as the team clarifies the use of a multisig account.

3.7 Improper totalSupply Adjustment in BasicERC20::_burn()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasicERC20`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [5]

Description

The `Kyoko` protocol tokenizes the user liquidity and debt positions. The main tokenization support is based on a common `BasicERC20` contract. While examining the basic token-related logic, we notice the current burn-related implementation needs to properly keep track of the `totalSupply` state by maintaining the invariant: i.e., `totalSupply` is the same as the sum of all holders' balances.

To elaborate, we show below the `_burn()` function. It implements a rather straightforward logic in burning the requested token amount. Naturally, we need to decrease the `totalSupply` by the burnt amount. However, it comes to our attention that the `totalSupply` is instead increased by the burnt amount (line 218).

```
212     function _burn(address account, uint256 amount) internal virtual {
213         require(account != address(0), "ERC20: burn from the zero address");
214 
215         _beforeTokenTransfer(account, address(0), amount);
216 
217         uint256 oldTotalSupply = _totalSupply;
218         _totalSupply = oldTotalSupply + amount;
219 
220         uint256 oldAccountBalance = _balances[account];
221         require(oldAccountBalance >= amount, "ERC20: burn amount exceeds balance");
222         _balances[account] = oldAccountBalance - amount;
223     }
```

Listing 3.9: `BasicERC20::_burn()`

Recommendation Properly update the `totalSupply` state within the token-burning logic.

Status The issue has been fixed by the following commit: `ef12062`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Kyoko` protocol, which aims to become the first decentralized, dual-rate, peer-to-pool `NFT` lending protocol and provide a comprehensive solution for liquidity issues in the `NFT` market. Users can participate in the protocol either as depositors or borrowers. Depositors provide `ETH` liquidity to the market and earn passive income, while borrowers can obtain loans by using `NFTs` as collateral. The protocol uniquely offers lending pools for a wide range of `NFT` collections, rather than limiting itself to blue-chip `NFT` projects. This ensures greater inclusivity for `NFT` owners, making the platform accessible to a broader range of participants. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.