SMART CONTRACT AUDIT REPORT

for

Boba

Prepared By: Xiaomi Huang

PeckShield

May 10, 2023

Document Properties

Client	Boba
Title	Smart Contract Audit Report
Target	Boba
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 10, 2023	Xuxian Jiang	Final Release
1.0-rc	March 31, 2023	Xuxian Jiang	Release Candidate

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 Boba network, 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 Boba

Boba is built on the Optimistic Rollup developed by Optimism. Aside from the focus on augmenting the compute capability, Boba differs from Optimism in a number of aspects: (1) It provides additional cross-chain messaging; (2) It supports different gas pricing logic; (3) It provides a swap-based system for rapid L2->L1 exits (without the 7 day delay); (4) It provides a community fraud-detector that allows transactions to be independently verified by anyone; (5) It interacts with L2 ETH using the normal ETH methods rather than as WETH; (6) It is organized as a DAO; (7) It supports native NFT bridging; and (8) It automatically relays classical 7-day exit messages to L1. The basic information of the audited network is as follows:

Table 1.1: Basic Information of Boba

Item	Description
Name	Boba
Website	https://boba.network
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 10, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit has a specific audit scope specified in https://docs.google.com/document/d/1uF1a_ce9y_x7sK_QXlNrwovfKxvV43Mj82RjpS29tyk (md5: eeb1ec7c7297984c27ebadc985eda9ab).

- https://github.com/bobanetwork/boba.git (6fb695a)

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

- https://github.com/bobanetwork/boba.git (668d83d)

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 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 list of check 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.

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 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.

Table 1.3: The Full List of Check Items

Category	Check Item
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 Logics	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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Boba implementation. 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	0	
Medium	2	■■
Low	4	■■■■
Informational	1	■
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 that 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 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Boba Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Target Address Validation in Lib_ResolvedDelegateProxy	Coding Practices	Resolved
PVE-002	Low	Timely Reward Update Upon Reward Retrieval	Business Logic	Resolved
PVE-003	Informational	Revisited ownerRevenue Accounting in BobaTuringCredit	Business Logic	Resolved
PVE-004	Low	Improved Parameter Updates in Boba_GasPriceOracle	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Incorrect Logic in DiscretionaryExit-Burn::payAndWithdraw()	Business Logic	Resolved
PVE-007	Low	Consistent Reentrancy/Pause Enforcement in L1NFTBridge	Time and State	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 Target Address Validation in Lib_ResolvedDelegateProxy

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Lib_ResolvedDelegateProxy
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The Boba network has a Lib_ResolvedDelegateProxy contract, which is designed to be a proxy. This proxy basically maintains a mapping to the intended target contract as the logic. While examining its logic, we notice the current implementation needs to be revised.

To elaborate, we show below the related _doProxyCall() routine. As the name indicates, this routine is used to perform a proxy call. It comes to our attention the target contract is not properly validated. In fact, it currently validates the addressManager["proxyOwner"] is not address(0) (line 96), which should be revised to validate addressManager["proxyTarget"] not address(0)!

```
88      /**
89       * Performs the proxy call via a delegatecall.
90       */
91      function _doProxyCall()
92          internal
93      {

95          require(
96              addressManager["proxyOwner"] != address(0),
97              "Target address must be initialized."
98          );

100         (bool success, bytes memory returndata) = addressManager["proxyTarget"].
                delegatecall(msg.data);
```

```
        if (success == true) {
            assembly {
                return(add(returndata, 0x20), mload(returndata))
            }
        } else {
            assembly {
                revert(add(returndata, 0x20), mload(returndata))
            }
        }
    }
```

Listing 3.1: `Lib_ResolvedDelegateProxy::_doProxyCall()`

Recommendation Revise the above routine to properly validate the proxy target.

Status The issue has been addressed by the following commit: `ba6e162`.

3.2 Timely Reward Update Upon Reward Retrieval

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To faciliate the cross-chain transfer, the `Boba` network has provided `liquidity pools` on both `L1` and `L2`. To encourage the participation from liquidity providers, the network provides rewards in proportion to their stake in the pool. While examining the current logic for reward retrieval, we notice the current implementation can be improved.

Specifically, we show below the related code snippet from the `withdrawReward()` routine. As the name indicates, this routine allows the liquidity providers to withdraw their rewards. It comes to our attention that when the rewards are being requested, there is a need to timely update the pool's `accUserRewardPerShare` by invoking `updateUserRewardPerShare(_tokenAddress)`. The timely update of `accUserRewardPerShare` is necessary so that the liquidity provider can always return the latest reward. Note that the same issue is also applicable to other routines, including `ownerRecoverFee()` on both `L1LiquidityPool` and `L2LiquidityPool`.

```
    function withdrawReward(
        uint256 _amount,
        address _tokenAddress,
        address _to
```

```
    )
        external
        whenNotPaused
    {
        PoolInfo storage pool = poolInfo[_tokenAddress];
        UserInfo storage user = userInfo[_tokenAddress][msg.sender];

        require(pool.l2TokenAddress != address(0), "Token Address Not Registered");

        uint256 pendingReward = user.pendingReward.add(
            user.amount.mul(pool.accUserRewardPerShare).div(1e12).sub(user.rewardDebt)
        );

        require(pendingReward >= _amount, "Withdraw Reward Error");

        user.pendingReward = pendingReward.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accUserRewardPerShare).div(1e12);

        emit WithdrawReward(
            msg.sender,
            _to,
            _amount,
            _tokenAddress
        );

        if (_tokenAddress != address(0)) {
            IERC20(_tokenAddress).safeTransfer(_to, _amount);
        } else {
            (bool sent,) = _to.call{gas: SAFE_GAS_STIPEND, value: _amount}("");
            require(sent, "Failed to send Ether");
        }
    }
```

Listing 3.2: `L1LiquidityPool::withdrawReward()`

Recommendation Timely invoke `updateUserRewardPerShare()` when the rewards are being requested for retrieval.

Status The issue has been confirmed and the team considers it a tradeoff to make gas-efficient while also allowing users to self call update methods before withdrawing to withdraw all latest rewards altogether.

3.3 Revisited ownerRevenue Accounting in BobaTuringCredit

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BobaTuringCredit
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Boba network has a BobaTuringCredit contract to manage the credit subsystem for Boba Turing. Within this contract, there is a storage state ownerRevenue to presumably keep track of the revenue amount the owner is entitled to withdraw. However, our analysis shows this storage state is never updated to increase the ownerRevenue.

In fact, the only function that touches this state is the following withdrawRevenue() function. Though this function is properly designed to allow the owner to withdraw the revenue, the fact that the ownerRevenue state is never updated elsewhere indicates ownerRevenue is always 0. In other words, the owner may never be able to retrieve any reward at all.

```
    function withdrawRevenue(uint256 _withdrawAmount) public onlyOwner onlyInitialized {
        require(_withdrawAmount <= ownerRevenue, "Invalid Amount");

        ownerRevenue -= _withdrawAmount;

        emit WithdrawRevenue(msg.sender, _withdrawAmount);

        IERC20(turingToken).safeTransfer(owner, _withdrawAmount);
    }
```

Listing 3.3: BobaTuringCredit::withdrawRevenue()

Recommendation Revisit the logic to compute and accumulate the ownerRevenue in BobaTuringCredit .

Status The issue has been resolved as the BobaTuringCredit is a predeploy. The ownerRevenue state is updated through geth, when TuringCharge happens.

3.4 Improved Parameter Updates in Boba_GasPriceOracle

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Boba_GasPriceOracle
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Boba network is no exception. Specifically, if we examine the Boba_GasPriceOracle contract, it has defined a number of protocol-wide risk parameters, such as minPriceRatio and maxPriceRatio. In the following, we show the corresponding routines that allow for their changes.

```
    function updateMaxPriceRatio(uint256 _maxPriceRatio) public onlyOwner {
        require(_maxPriceRatio >= minPriceRatio && _maxPriceRatio > 0);
        maxPriceRatio = _maxPriceRatio;
        emit UpdateMaxPriceRatio(owner(), _maxPriceRatio);
    }

    /**
     * Update the minimum price ratio of ETH and BOBA
     * @param _minPriceRatio the minimum price ratio of ETH and BOBA
     */
    function updateMinPriceRatio(uint256 _minPriceRatio) public onlyOwner {
        require(_minPriceRatio <= maxPriceRatio && _minPriceRatio > 0);
        minPriceRatio = _minPriceRatio;
        emit UpdateMinPriceRatio(owner(), _minPriceRatio);
    }
```

Listing 3.4: Boba_GasPriceOracle::updateMaxPriceRatio() and Boba_GasPriceOracle::updateMinPriceRatio()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of minPriceRatio or maxPriceRatio may make the current active priceRatio out-of-range, hence bringing undesirable consequence to the Boba network.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been resolved as the related contract is a predeploy and the method is controlled by the contract owner of the contract.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Boba` network, there is a privileged `owner` account that plays a critical role in governing and regulating the network-wide operations (e.g., set allowed tokens, configure system parameters, upgrade contracts, etc). In the following, we show the representative functions potentially affected by the privilege of the account.

```
function setAccessController(
  AccessControllerInterface _accessController
)
  public
  override
  onlyOwner()
{
  require(address(_accessController) != address(s_accessController), "Access
      controller is already set");
  s_accessController = _accessController;
  emit AccessControllerSet(address(_accessController), msg.sender);
}

function proposeFeed(
  address base,
  address quote,
  address aggregator
)
  external
  override
  onlyOwner()
{
  AggregatorV2V3Interface currentPhaseAggregator = _getFeed(base, quote);
  require(aggregator != address(currentPhaseAggregator), "Cannot propose current
      aggregator");
  address proposedAggregator = address(_getProposedFeed(base, quote));
  if (proposedAggregator != aggregator) {
    s_proposedAggregators[base][quote] = AggregatorV2V3Interface(aggregator);
    emit FeedProposed(base, quote, aggregator, address(currentPhaseAggregator), msg.
        sender);
  }
}
```

Listing 3.5: Example Privileged Operations in `FeedRegistry`

```
function withdrawFunds(address _recipient, uint256 _amount)
  external
  onlyOwner()
{
  updateAvailableFunds();
  uint256 available = uint256(recordedFunds.available);
  require(available.sub(requiredReserve(paymentAmount)) >= _amount, "insufficient
      reserve funds");
  require(bobaToken.transfer(_recipient, _amount), "token transfer failed");
  updateAvailableFunds();
}

function changeOracles(
  address[] calldata _removed,
  address[] calldata _added,
  address[] calldata _addedAdmins,
  uint32 _minSubmissions,
  uint32 _maxSubmissions,
  uint32 _restartDelay
)
  external
  onlyOwner()
{
  for (uint256 i = 0; i < _removed.length; i++) {
    removeOracle(_removed[i]);
  }

  require(_added.length == _addedAdmins.length, "need same oracle and admin count");
  require(uint256(oracleCount()).add(_added.length) <= MAX_ORACLE_COUNT, "max oracles
      allowed");

  for (uint256 i = 0; i < _added.length; i++) {
    addOracle(_added[i], _addedAdmins[i]);
  }

  updateFutureRounds(paymentAmount, _minSubmissions, _maxSubmissions, _restartDelay,
      timeout);
}
```

Listing 3.6: Example Privileged Operations in FluxAggregator

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 confirmed with the team. For the time being, it is mitigated with a multi-sig account.

3.6 Incorrect Logic in DiscretionaryExitBurn::payAndWithdraw()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DiscretionaryExitBurn`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Boba` network supports convenient cross-chain message passing and token swaps. While examining a related routine `payAndWithdraw()`, we notice its implementation can be improved.

To elaborate, we show below its implementation. This routine has a rather straightforward logic in collecting the payment and then scheduling the withdrawal operation via the related `l2Bridge`. It comes to our attention that the payment validation is incomplete as it can be improved to ensure `require(msg.value != 0 || _l2Token != Lib_PredeployAddresses.OVM_ETH)`.

```
    function payAndWithdraw(
        address _l2Token,
        uint256 _amount,
        uint32 _l1Gas,
        bytes calldata _data
    ) external payable onlyWithBillingContract {
        // Collect the exit fee
        L2BillingContract billingContract = L2BillingContract(billingContractAddress);
        IERC20(billingContract.feeTokenAddress()).safeTransferFrom(msg.sender,
            billingContractAddress, billingContract.exitFee());

        require(!(msg.value != 0 && _l2Token != Lib_PredeployAddresses.OVM_ETH), "Amount
             Incorrect");

        if (msg.value != 0) {
            // override the _amount and token address
            _amount = msg.value;
            _l2Token = Lib_PredeployAddresses.OVM_ETH;
        }

        // transfer funds if users deposit ERC20
        if (_l2Token != Lib_PredeployAddresses.OVM_ETH) {
            IERC20(_l2Token).safeTransferFrom(msg.sender, address(this), _amount);
```

```
        }

        // call withdrawTo on the l2Bridge
        IL2ERC20Bridge(l2Bridge).withdrawTo(_l2Token, msg.sender, _amount, _l1Gas, _data
            );
    }
```

Listing 3.7: `DiscretionaryExitBurn::payAndWithdraw()`

Recommendation Improve the above routine to ensure the payment method is properly validated.

Status The issue has been resolved by following the above suggestion.

3.7 Consistent Reentrancy/Pause Enforcement in L1NFTBridge

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `re-entrancy` protection is not consistently enforced. Using the `L1NFTBridge` as an example, both `depositNFT()` `depositNFTTo()` functions (see the code snippet below) are properly enforced with associated modifiers `nonReentrant()` and `whenNotPaused()`. However, if we take a look at the `finalizeNFTWithdrawal()` routine, it is not consistently enforced with the same modifiers. To eliminate the possible risks from reentrancy, we strongly suggest to apply a consistent enforcement. Note that other contracts share the same issue, including `L1NFTBridge`, `L2NFTBridge`, `L1ERC1155Bridge` and `L2ERC1155Bridge`.

```
    function depositNFT(
        address _l1Contract,
        uint256 _tokenId,
```

```
        uint32 _l2Gas
    )
        external
        virtual
        override
        nonReentrant()
        whenNotPaused()
    {
        _initiateNFTDeposit(_l1Contract, msg.sender, msg.sender, _tokenId, _l2Gas, "");
    }

    //  /**
    //   * @inheritdoc iL1NFTBridge
    //   */
    function depositNFTTo(
        address _l1Contract,
        address _to,
        uint256 _tokenId,
        uint32 _l2Gas
    )
        external
        virtual
        override
        nonReentrant()
        whenNotPaused()
    {
        _initiateNFTDeposit(_l1Contract, msg.sender, _to, _tokenId, _l2Gas, "");
    }

    function finalizeNFTWithdrawal(
        address _l1Contract,
        address _l2Contract,
        address _from,
        address _to,
        uint256 _tokenId,
        bytes memory _data
    )
        external
        override
        onlyFromCrossDomainAccount(l2NFTBridge)
    {...}
```

Listing 3.8: `L1NFTBridge::depositNFT()/depositNFTTo()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been resolved. The team clarifies that the methods without `nonReentrant` are `xChain` methods that are supposed to be only called through `relayMessage()` on the `L1CrossDomainMessenger`, which have the `nonReentrant` and `whenNotPaused` modifiers, (or) by the `sequencer` for the `L2CrossDomainMessenger`

. The same holds for `L1LiquidityPool` as well. Some unused variable likes `extraGasRelay` have not been removed because these were slots on the proxy contract and cannot be removed.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Boba` network, which is built on the `Optimistic Rollup` developed by `Optimism`. Aside from the focus on augmenting the compute capability, `Boba` differs from `Optimism` in a number of aspects: (1) It provides additional cross-chain messaging; (2) It supports different gas pricing logic; (3) It provides a swap-based system for rapid `L2->L1` exits (without the 7 day delay); (4) It provides a community fraud-detector that allows transactions to be independently verified by anyone; (5) It interacts with `L2 ETH` using the normal `ETH` methods rather than as `WETH`; (6) It is organized as a `DAO`; (7) It supports native NFT bridging; and (8) It automatically relays classical 7-day exit messages to L1. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.