SMART CONTRACT AUDIT REPORT

for

LuckyBid

Prepared By: Xiaomi Huang

PeckShield

June 27, 2023

Document Properties

Client	LuckyBid
Title	Smart Contract Audit Report
Target	LuckyBid
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 27, 2023	Luck Hu	Final Release
1.0-rc	June 9, 2023	Luck Hu	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 4
1.1 About LuckyBid 4
1.2 About PeckShield 5
1.3 Methodology 5
1.4 Disclaimer 7

2 Findings 9
2.1 Summary 9
2.2 Key Findings 10

3 Detailed Results 11
3.1 Potential Out-of-Service from VRF Service Module 11
3.2 Revised Distribution of Bid Profit in distribute() 13
3.3 Trust Issue on Admin Keys 14
3.4 Inconsistency of Ticket Purchase Prices in Bids&Tickets 16

4 Conclusion 18

References 19

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the LuckyBid 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 LuckyBid

LuckyBid is a decentralized omnichain NFT lottery marketplace that gives fair and verifiable opportunities for everyone to win bluechip NFTs with low investments and affordable gas fees. LuckyBid is built on LayerZero, Stargate, and Chainlink VRF/Automation, which ensures transparency and reliability in every draw. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of LuckyBid

Item	Description
Name	LuckyBid
Website	https://www.luckybid.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 27, 2023

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

- https://github.com/luckyBid/lucky-bid (f2629719)

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

- https://github.com/luckyBid/lucky-bid (47271a71)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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 accordingly classified into four categories, 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

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

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) [5], 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 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 1.4: 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 `LuckyBid` protocol implementations. 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	1	■
Informational	0	
Total	4	

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 1 high-severity vulnerability, 2 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Potential Out-of-Service from VRF Service Module	Business Logic	Fixed
PVE-002	Medium	Revised Distribution of Bid Profit in distribute()	Business Logic	Fixed
PVE-003	Medium	Trust Issue on Admin Keys	Security Features	Mitigated
PVE-004	Low	Inconsistency of Ticket Purchase Prices in Bids&Tickets	Business Logic	Confirmed

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 Potential Out-of-Service from VRF Service Module

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Bids`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

`LuckyBid` is built on `Chainlink VRF (Verifiable Random Function)` which is a provably fair and verifiable random number generator (`RNG`). `LuckyBid` relies on the unpredictable random numbers generated by `Chainlink VRF` to select the winner for the raffle. `Chainlink VRF` uses the `Request & Receive Data` cycle which calls back the `fulfillRandomWords()` function of the consuming contract with the produced random numbers.

One of the `VRF Security Considerations` requires that the `fulfillRandomWords()` callback function must not revert, or the `VRF` service will not attempt to call it a second time. While examining the `fulfillRandomWords()` implementation in the `ChainlinkVRFAdapter/Bids` contract, we notice the possibility of transaction revert.

In the following, we show the code snippet of the `Bids::drawCallback()` function which is called from the `ChainlinkVRFAdapter::fulfillRandomWords()` function. The `Bids::drawCallback()` function calculates the winner id of the raffle based on the random number generated by `VRF` and transfers the reward `NFT` to the winner by calling the `IERC721Upgradeable(rewardsCollection).safeTransferFrom()` (line 371). However, it comes to our attention that, if the winner is a contract but is not a valid `NFT` receiver or it reverts in its `onERC721Received()` function, the call to the `IERC721Upgradeable(rewardsCollection).safeTransferFrom()` will revert. As a result, the `VRF` service will not attempt to call the `fulfillRandomWords()` a second time , hence it fails to draw the raffle.

```
360     function drawCallback(
361         uint256 raffleId,
```

```
362         uint256 randomNumber
363     ) external onlyRandomizerAdapter {
364         RaffleInfo storage raffleInfo = raffles[raffleId];
365         require(raffleInfo.status == Status.Drawing, "NOT_RAFFLING");
366         raffleInfo.winnerNumber =
367             raffleInfo.start +
368             (randomNumber % (raffleInfo.end - raffleInfo.start + 1));

370         address to = ownerOf(raffleInfo.winnerNumber);
371         IERC721Upgradeable(rewardsCollection).safeTransferFrom(
372             address(this),
373             to,
374             raffleInfo.tokenId
375         );
376         raffleInfo.winner = to;

378         raffleInfo.status = Status.Drawn;

380         tickets.addProfit(
381             ticketId,
382             tickets.getMinPrice(ticketId) * (raffleInfo.end - raffleInfo.start + 1) -
                    raffleInfo.price
383         );

385         emit DrawCallback(raffleId, randomNumber, raffleInfo.winnerNumber, to);
386     }
```

Listing 3.1: Bids::drawCallback()

What's more, after transferring the reward `NFT` to the winner, it calculates the bids profit via `tickets.getMinPrice(ticketId)* (raffleInfo.end - raffleInfo.start + 1)- raffleInfo.price` (line 382). However, we notice the `minPrices[id]]` can be updated by the governance in the `_setPrice()/ resetMinPrice()` routines (lines 115, 122). As a result, if the `minPrices[id]]` is updated after the raffle is created, the calculation at line 382 may revert because of mathmatic overflow/underflow, hence the call to the `fulfillRandomWords()` function reverts.

```
111 function _setPrice(uint256 id, uint256 amount, uint256 price) internal {
112     prices[id][amount] = price;
113     uint256 avgPrice = price / amount;
114     if (minPrices[id] == 0  avgPrice < minPrices[id]) {
115         minPrices[id] = avgPrice;
116     }

118     emit SetPrice(id, amount, price);
119 }

121 function resetMinPrice(uint256 id) external onlyGovernance {
122     minPrices[id] = 0;
123     emit ResetPrice(id);
124 }
```

Listing 3.2: Tickets::_setPrice()/resetMinPrice()

Recommendation Revisit the above `Bids::drawCallback()` function and ensure there is no chance it will revert.

Status The issue has been fixed by these commits: `107c84c7` and `47271a71`.

3.2 Revised Distribution of Bid Profit in distribute()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ProfitDistributor`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In the `LuckyBid` protocol, the bid profit can be withdrawn and distributed to the `treasury/multiFeeDistributor` by the `ProfitDistributor`. The profit is distributed to the `treasury/multiFeeDistributor` per a share ratio. While reviewing the distribution of the profit, we notice there is a lack of distributing the desired amount of profit to the `multiFeeDistributor`.

To elaborate, we show below the code snippet of the `ProfitDistributor::distribute()` routine. As the name indicates, it is used to distribute the bid profit. Firstly it calculates the profit amount that belongs to the `multiFeeDistributor` per the `shareRatio` (line 60). Then it transfers the remaining amount of profit, i.e., `amount - shareAmount`, to the `treasury` (line 61). However, it comes to our attention that the routine does not transfer the `shareAmount` of profit to the `multiFeeDistributor`, though the `shareAmount` may be 0 if the `shareRatio` is 0. As a result, the profit that belongs to the `multiFeeDistributor` is locked in the contract.

```
54    function distribute(address token, uint256 amount) external {
55        require(treasury != address(0), "UNSET_TREASURY");
56        require(multiFeeDistributor != address(0), "UNSET_MULTI_FEE_DISTRIBUTOR");

58        IERC20(token).transferFrom(msg.sender, address(this), amount);

60        uint256 shareAmount = amount * shareRatio / 100;
61        IERC20(token).transferFrom(address(this), treasury, amount - shareAmount);

63        // TODO, distribute to MultiFeeDistribution
64    }
```

Listing 3.3: ProfitDistributor :: distribute ()

Recommendation Properly transfer the `shareAmount` of profit to the `multiFeeDistributor`.

Status The issue has been fixed by this commit: `107c84c7`.

3.3 Trust Issue on Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `LuckyBid` protocol, there is a privileged `owner` account and certain privileged roles, i.e., `GOVERNANCE_ROLE` `/OPERATOR/EMERGENCY_ADMIN_ROLE`, that play critical roles in governing and regulating the system-wide operations (e.g., grant other roles, set ticket price, withdraw funds). Our analysis shows that these privileged accounts need to be scrutinized. In the following, we use the `Tickets` contract as an example and show the representative functions potentially affected by the privileges of the privileged accounts.

Specifically, the privileged functions in `Tickets` allow for the `EMERGENCY_ADMIN_ROLE` to pause/unpause the ticket purchase, allow for the `GOVERNANCE_ROLE` to set ticket price, reset the `minPrices[id]` which can impact the bid balance/profit, set the profit distributor, withdraw bid profit, and allow for the `owner` to set the ticket `URI`, etc.

```
    function pause() external onlyEmergencyAdmin {
        _pause();
    }

    function unpause() external onlyEmergencyAdmin {
        _unpause();
    }

    function setPrice(
        uint256 id,
        uint256 amount,
        uint256 price
    ) external onlyGovernance {
        _setPrice(id, amount, price);
    }

    function resetMinPrice(uint256 id) external onlyGovernance {
        minPrices[id] = 0;
        emit ResetPrice(id);
    }

    function setBaseURI(string memory newBaseUri) external onlyOwner {
        _setBaseURI(newBaseUri);
    }
```

```
    function setURI(uint256 tokenId, string memory tokenURI) external onlyOwner {
        _setURI(tokenId, tokenURI);
    }

    function setProfitDistributor(address _profitDistributor) external onlyGovernance {
        _setProfitDistributor(_profitDistributor);
    }

    function withdrawProfit(address bids, uint256 id, uint256 amount) external
        onlyGovernance {
        require(profitDistributor != address(0), "UNSET_PROFIT_DISTRIBUTOR");
        require(bidsProfit[bids][id] >= amount, "INSUFFICIENT_PROFIT");

        bidsProfit[bids][id] -= amount;
        bidsBalance[bids][id] -= amount;

        WETH.approve(profitDistributor, amount);
        IProfitDistributor(profitDistributor).distribute(address(WETH), amount);

        emit WithdrawProfit(bids, profitDistributor, id, amount);
    }

    function claimWETH(uint256 id, uint256 amount) external onlyGovernance {
        require(profitDistributor != address(0), "UNSET_PROFIT_DISTRIBUTOR");
        require(getAvailableBalance(id) >= amount, "NOT_ALLOWED_AMOUNT");

        totalBalance[id] -= amount;

        WETH.approve(profitDistributor, amount);
        IProfitDistributor(profitDistributor).distribute(address(WETH), amount);

        emit ClaimWETH(profitDistributor, id, amount);
    }
```

Listing 3.4: Example Privileged Operations in `Tickets`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the `LuckyBid` users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged accounts to the intended `DAO`-like governance contract. All changes to the 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 The issue has been mitigated as the team confirmed they will transfer the ownership to a multi-sig account in short term and consider implementing a `DAO` and timelock mechanism in the future.

3.4 Inconsistency of Ticket Purchase Prices in Bids&Tickets

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Bids`, `Tickets`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `LuckyBid` protocol, a user can buy tickets to bid for a raffle. The ticket price can be updated by governance. Generally, there are two ways for the user to buy ticket. One way is calling the `Bids::bidWithETH()` routine which will mint the desired amount of tickets on behalf of the user. The other way is calling the `Tickets::mintWithETH()` routine which directly mints the desired amount of tickets for the user. While examining the two ways to buy tickets for users, we notice they are using different prices.

In the following, we show below the code snippets of the `Bids::bidWithETH()/Tickets::mintWithETH()` routines. In the `Bids::bidWithETH()` routine, it uses the full price, i.e., `prices[id][1]`, as the price to calculate the payment (line 266). In the `Tickets::mintWithETH()` routine, it directly uses the discount price, i.e., `prices[id][amount]`, as the payment (line 186). However, it comes to our attention that the calculated payment amount may be different, i.e., `prices[id][1] * amount != prices[id][amount]`, because of the possible discount which is designed to encourage users to purchase as many tickets as they desire.

As a result, a user may spend different amounts of payment to buy the same amount of tickets via different purchase ways.

```
261   function bidWithETH(
262     uint256 amount,
263     bytes memory data,
264     bytes32 _referralCode
265 ) external payable whenNotPaused {
266     uint256 paymentAmount = tickets.getFullPrice(ticketId) * amount;
267     tickets.mintWhenBidding{value: paymentAmount}(ticketId, amount, data);
268 
269     tickets.burn(address(this), ticketId, amount);
270     _bid(msg.sender, amount);
271     _setTraderReferralCodeTry(_referralCode);
272 
273     uint256 refund = msg.value - paymentAmount;
274     if (refund > 0) {...}
275 }
```

Listing 3.5: `Bids::bidWithETH()`

```
function mintWithETH(
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) external payable whenNotPaused {
    uint256 price = prices[id][amount];
    require(price > 0, "AMOUNT_NOT_ALLOWED");
    require(price <= msg.value, "ETH_AMOUNT_INSUFFICIENT");
    _mint(to, id, amount, data);

    totalBalance[id] += price;
    WETH.deposit{value: price}();

    if (price < msg.value) {...}
}

function getFullPrice(uint256 id) public view returns (uint256) {
  require(prices[id][1] != 0, "PRICE_ERROR");
  return prices[id][1];
}

function getPrice(
    uint256 id,
    uint256 amount
) public view returns (uint256) {
    return prices[id][amount];
}
```

Listing 3.6: `Tickets::mintWithETH()`

Recommendation Revisit the above `Bids::bidWithETH()/Tickets::mintWithETH()` routines and use the same purchase price for users to buy tickets.

Status The issue has been confirmed and the team clarified that: this is business-related. Users who use different buy functions have an equal chance to win. The reason we designed different buy functions is to guide users in purchasing tickets. From the platform's perspective, we encourage users to buy more tickets. By selling tickets, the platform can raise funds more rapidly. Once the funds are raised, the platform can acquire `NFTs` earlier.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `LuckyBid` protocol. `LuckyBid` is a decentralized `omnichain NFT` lottery marketplace that gives fair and verifiable opportunities for everyone to win `bluechip NFTs` with low investments and affordable gas fees. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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