SMART CONTRACT AUDIT REPORT

for

Buffer Protocol

Prepared By: Yiqun Chen

PeckShield

October 1, 2021

Document Properties

Client	Buffer Finance
Title	Smart Contract Audit Report
Target	Buffer
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 1, 2021	Xiaotao Wu	Final Release

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Buffer` 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 Buffer

`Buffer` is a non-custodial on-chain peer-to-pool options trading protocol built on `Binance Smart Chain`. It works like an `Automated Market Maker` (e.g. `PancakeSwap`) where traders can create, buy, and settle options against liquidity pool without the need of a counter-party (option writer). `Buffer` makes options trading accessible to everyone and much more efficient than its centralized counterpart.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Buffer

Item	Description
Name	Buffer Finance
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 2021

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

- https://github.com/Buffer-Finance/Buffer-Protocol (1c648bb)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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) [10], 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 design and implementation of the `Buffer` protocol smart contracts. 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	0	
Medium	1	■
Low	3	■ ■ ■
Informational	0	
Undetermined	1	■
Total	5	

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 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Handling Of Corner Cases In BufferBNBPool::provide()	Business Logic	Fixed
PVE-002	Undetermined	Potential Reentrancy Risks In BufferBN-BOptions::create()	Time and State	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Possible Costly LPs From Improper Pool Initialization	Time and State	Confirmed
PVE-005	Low	Lack Of OPTION_ISSUER_ROLE Initialization In BufferBNBPool	Coding Practices	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 Improved Handling Of Corner Cases In BufferBNBPool::provide()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BufferBNBPool`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [4]

Description

At the core of the `Buffer` protocol is the `BufferBNBPool` contract, which has an external `provide()` function for users to supply `BNB` to the pool and mint the corresponding shares of `rBFR-BNB` tokens to the users. While examining the routine, we notice its implementation can be improved.

To elaborate, we show below its full implementation. When this routine is called by a user, the user can specify a `referrer` address to receive possible `referralReward` (line 70). The current implementation requires `(referrer != address(0)&& referrer != msg.sender)` (line 77). However, if the specified `referrer` address is the contract itself, i.e., `address(this)`, the transfer of `referralReward` is also not needed.

```
70      function provide(uint256 minMint , address referrer) external payable returns (
            uint256 mint) {
71          lastProvideTimestamp[msg.sender] = block.timestamp;
72          uint256 supply = totalSupply();
73          uint256 balance = totalBalance();
74
75          uint256 amount = msg.value;
76
77          if(referrer != address(0) && referrer != msg.sender){
78              uint256 referralReward = ((msg.value * referralRewardPercentage)/ACCURACY)
                    /100;
79              amount = msg.value - referralReward;
80
```

```
            if (referralReward > 0){
                payable(referrer).transfer(referralReward);
            }
        }

        if (supply > 0 && balance > 0)
            mint = (amount * supply) / (balance - amount);
        else mint = amount * INITIAL_RATE;

        require(mint >= minMint, "Pool: Mint limit is too large");
        require(mint > 0, "Pool: Amount is too small");

        _mint(msg.sender, mint);

        emit Provide(msg.sender, amount, mint);
    }
```

Listing 3.1: `BufferBNBPool::provide()`

Note the same issue also exists in the `distributeSettlementFee()` routine of the `BufferBNBOptions` contract.

Recommendation Take into consideration the scenario that the the specified `referrer` address might be equal to the address of the `BufferBNBPool` contract.

Status This issue has been fixed in the following commit: `9e1b339`.

3.2 Potential Reentrancy Risks In BufferBNBOptions::create()

- ID: PVE-002
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: `BufferBNBOptions`
- Category: Time and State [9]
- CWE subcategory: CWE-841 [5]

Description

In the `BufferBNBOptions` contract, we notice the `create()` function is used to create a new option and mint the `ERC721` token, which represents the option created by user. Our analysis shows there is a potential reentrancy vulnerability in the function.

To elaborate, we show below the code snippet of the `create()` function. In the function, the `_safeMint()` function will be called (line 416) to mint an `ERC721` token for the option creator. A further examination of `_safeMint()` of `ERC721` shows the `_checkOnERC721Received()` function will be called to ensure the recipient confirms the receipt. If the recipient is a malicious one, she may launch a re-entrancy attack in the callback function. So far, we also do not know how an attacker can exploit this vulnerability to earn profit. After internal discussion, we consider it is necessary to bring this vulnerability up to the team. Though the implementation of the `create()` function is well designed, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `create()` function at the protocol level.

```
    function create(
        uint256 period,
        uint256 amount,
        uint256 strike,
        OptionType optionType,
        address referrer
    ) external payable returns (uint256 optionID) {
        (uint256 totalFee, uint256 settlementFee, uint256 strikeFee, ) = fees(
            period,
            amount,
            strike,
            optionType
        );

        require(
            optionType == OptionType.Call  optionType == OptionType.Put,
            "Wrong option type"
        );
        require(period >= 1 days, "Period is too short");
        require(period <= 90 days, "Period is too long");
        require(amount > strikeFee, "Price difference is too large");
        require(msg.value >= totalFee, "Wrong value");
        if (msg.value > totalFee) {
            payable(msg.sender).transfer(msg.value - totalFee);
        }

        uint256 strikeAmount = amount - strikeFee;
        uint256 lockedAmount = ((strikeAmount * optionCollateralizationRatio) / 100) +
            strikeFee;

        Option memory option = Option(
            State.Active,
            strike,
            amount,
            lockedAmount,
            totalFee - settlementFee,
            block.timestamp + period,
            optionType
        );

        optionID = createOptionFor(msg.sender);
        options[optionID] = option;

        uint256 stakingAmount = distributeSettlementFee(settlementFee, referrer);

        pool.lock{value: option.premium}(optionID, option.lockedAmount);

```

```
        emit Create(optionID, msg.sender, stakingAmount, totalFee);
    }
```

Listing 3.2: `BufferBNBOptions::create()`

```
    function createOptionFor(address holder) internal returns (uint256 id) {
        id = nextTokenId++;
        _safeMint(holder, id);
    }
```

Listing 3.3: `BufferBNBOptions::createOptionFor()`

Recommendation Add necessary reentrancy guards (e.g., `nonReentrant`) to prevent unwanted reentrancy risks.

Status This issue has been fixed in the following commit: `6f00433`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `Buffer` protocol, there is a privileged account that plays critical roles in governing and regulating the system-wide operations (e.g., configuring protocol parameters). In the following, we examine the privileged accounts and their related privileged accesses in current contracts.

To elaborate, we show below example privileged functions in the `BufferBNBOptions` contract. These routines allows the privileged account `_owner` to set a number of protocol parameters, including `impliedVolRate`, `settlementFeePercentage`, `settlementFeeRecipient`, `stakingFeePercentage`, `referralRewardPercentage`, and `optionCollateralizationRatio` for `BufferBNBOptions`. If these key parameters are set to unreasonable values by the `_owner`, the user assets may suffer unexpected losses.

```
    /**
    * @notice Used for adjusting the options prices while balancing asset's implied
        volatility rate
    * @param value New IVRate value
    */
   function setImpliedVolRate(uint256 value) external onlyOwner {
       require(value >= 1000, "ImpliedVolRate limit is too small");
       impliedVolRate = value;
```

```
}
```

Listing 3.4: `BufferBNBOptions::setImpliedVolRate()`

```
    /**
     * @notice Used for adjusting the settlement fee percentage
     * @param value New Settlement Fee Percentage
     */
    function setSettlementFeePercentage(uint256 value) external onlyOwner {
        require(value < 20, "SettlementFeePercentage is too high");
        settlementFeePercentage = value;
    }
```

Listing 3.5: `BufferBNBOptions::setSettlementFeePercentage()`

```
    /**
     * @notice Used for changing settlementFeeRecipient
     * @param recipient New settlementFee recipient address
     */
    function setSettlementFeeRecipient(IBufferStakingBNB recipient)
        external
        onlyOwner
    {
        require(address(recipient) != address(0));
        settlementFeeRecipient = recipient;
    }
```

Listing 3.6: `BufferBNBOptions::setSettlementFeeRecipient()`

```
    /**
     * @notice Used for adjusting the staking fee percentage
     * @param value New Staking Fee Percentage
     */
    function setStakingFeePercentage(uint256 value) external onlyOwner {
        require(value <= 100, "StakingFeePercentage is too high");
        stakingFeePercentage = value;
    }
```

Listing 3.7: `BufferBNBOptions::setStakingFeePercentage()`

```
    /**
     * @notice Used for adjusting the referral reward percentage
     * @param value New Referral Reward Percentage
     */
    function setReferralRewardPercentage(uint256 value) external onlyOwner {
        require(value <= 100, "ReferralRewardPercentage is too high");
        referralRewardPercentage = value;
    }
```

Listing 3.8: `BufferBNBOptions::setReferralRewardPercentage()`

```
    /**
     * @notice Used for changing option collateralization ratio
     * @param value New optionCollateralizationRatio value
     */
    function setOptionCollaterizationRatio(uint256 value) external onlyOwner {
        require(50 <= value && value <= 100, "wrong value");
        optionCollateralizationRatio = value;
    }
```

Listing 3.9: `BufferBNBOptions::setOptionCollaterizationRatio()`

Another trust issue exists in the privileged accounts which are granted the `OPTION_ISSUER_ROLE` role. Only the `OPTION_ISSUER_ROLE` role has the right to call the functions `lock()/unlock()/send()` of the `BufferBNBPool` contract. If the `OPTION_ISSUER_ROLE` role is an EOA admin, this EOA admin can call the `send()` function to dismiss an active option.

```
    /*
     * @nonce calls by BufferCallOptions to lock the funds
     * @param amount Amount of funds that should be locked in an option
     */
    function lock(uint256 id, uint256 amount) external payable override {
        require(
            hasRole(OPTION_ISSUER_ROLE, msg.sender),
            "msg.sender is not allowed to excute the option contract"
        );
        require(id == lockedLiquidity[msg.sender].length, "Wrong id");
        require(totalBalance() >= msg.value, "Insufficient balance");   //Evan:
            totalBalance() = address(this).balance - lockedPremium;
        require(
            (lockedAmount + amount) <= ((totalBalance() - msg.value) * 8) / 10,
            "Pool Error: Amount is too large."
        );

        lockedLiquidity[msg.sender].push(LockedLiquidity(amount, msg.value, true));
        lockedPremium = lockedPremium + msg.value;
        lockedAmount = lockedAmount + amount;
    }

    /*
     * @nonce calls by BufferOptions to unlock the funds
     * @param id Id of LockedLiquidity that should be unlocked
     */
    function unlock(uint256 id) external override {
        require(
            hasRole(OPTION_ISSUER_ROLE, msg.sender),
            "msg.sender is not allowed to excute the option contract"
        );
        LockedLiquidity storage ll = lockedLiquidity[msg.sender][id];
        require(ll.locked, "LockedLiquidity with such id has already unlocked");
        ll.locked = false;

        lockedPremium = lockedPremium - ll.premium;
```

```
        lockedAmount = lockedAmount - ll.amount;

        emit Profit(id, ll.premium);
    }

    /*
     * @nonce calls by BufferCallOptions to send funds to liquidity providers after an
         option's expiration
     * @param to Provider
     * @param amount Funds that should be sent
     */
    function send(
        uint256 id,
        address payable to,
        uint256 amount
    ) external override {
        require(
            hasRole(OPTION_ISSUER_ROLE, msg.sender),
            "msg.sender is not allowed to excute the option contract"
        );
        LockedLiquidity storage ll = lockedLiquidity[msg.sender][id];
        require(ll.locked, "LockedLiquidity with such id has already unlocked");
        require(to != address(0));

        ll.locked = false;
        lockedPremium = lockedPremium - ll.premium;
        lockedAmount = lockedAmount - ll.amount;

        uint256 transferAmount = amount > ll.amount ? ll.amount : amount;
        to.transfer(transferAmount);

        if (transferAmount <= ll.premium)
            emit Profit(id, ll.premium - transferAmount);
        else emit Loss(id, transferAmount - ll.premium);
    }
```

Listing 3.10: `BufferBNBPool::lock()/unlock()/send()`

We understand the need of the privileged function for contract operation, but at the same time the extra power to the `_owner` may also be a counter-party risk to the contract 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 Make the list of extra privileges granted to `_owner` explicit to `Buffer` users.

Status The issue has been confirmed. The team further clarifies that after deploying the `Pool` and `Options` contracts, the team will assign the `OPTION_ISSUER_ROLE` to the `Options` contract and then renounce the `DEFAULT_ADMIN_ROLE` for the `Pool` contract.

3.4 Possible Costly LPs From Improper Pool Initialization

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BufferBNBPool`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [2]

Description

As mentioned in Section 3.1, the `BufferBNBPool` contract of `Buffer` protocol provides an external `provide()` function for users to supply `BNB` to the pool and mint the corresponding shares of `rBFR-BNB` tokens to the users. While examining the `LP` token calculation with the given `BNB` amount, we notice an issue that may unnecessarily make the pool token extremely expensive and the subsequent users may no longer be able to provide liquidity to the pool.

To elaborate, we show below the `provide()` routine. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function provide(uint256 minMint , address referrer) external payable returns (
        uint256 mint) {
        lastProvideTimestamp[msg.sender] = block.timestamp;
        uint256 supply = totalSupply();
        uint256 balance = totalBalance();

        uint256 amount = msg.value;

        if(referrer != address(0) && referrer != msg.sender){
            uint256 referralReward = ((msg.value * referralRewardPercentage)/ACCURACY)
                /100;
            amount = msg.value - referralReward;

            if (referralReward > 0){
                payable(referrer).transfer(referralReward);
            }
        }

        if (supply > 0 && balance > 0)
            mint = (amount * supply) / (balance - amount);
        else mint = amount * INITIAL_RATE;

        require(mint >= minMint, "Pool: Mint limit is too large");
        require(mint > 0, "Pool: Amount is too small");

        _mint(msg.sender, mint);

        emit Provide(msg.sender, amount, mint);
```

```
96      }
```

Listing 3.11: `BufferBNBPool::provide()`

Specifically, when the pool is being initialized, the `mint` value directly takes the value of `amount * INITIAL_RATE` (line 88), which is manipulatable by the malicious actor. As this is the first provide, the `totalBalance()` equals the `msg.value = 1 WEI`. With that, the actor can further transfer a huge amount of `BNB` to `BufferBNBPool` contract with the goal of making the `rBFR-BNB` extremely expensive.

An extremely expensive `rBFR-BNB` can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for provided `BNB` assets. The liquidity provider will suffer losses if this truncation happens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `provide()` to defensively calculate the mint amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first provide to avoid being manipulated.

Status This issue has been confirmed.

3.5 Lack Of OPTION_ISSUER_ROLE Initialization In BufferBNBPool

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BufferBNBPool`
- Category: Coding Practices [8]
- CWE subcategory: CWE-628 [3]

Description

The `BufferBNBPool` contract is designed for users to provide `BNB` as liquidity to the pool and receive write buffer tokens (`rBT`) accordingly. The provided liquidity is used to write both call and put options and the premium paid by the option buyer is distributed among the `rBT` holders as yield. While examining the implementation of this contract, we notice there is an important parameter that has not been properly initialized: i.e., `OPTION_ISSUER_ROLE`. Note only address that has been granted the `OPTION_ISSUER_ROLE` role can call the `lock()/unlock()/send()` functions of the `BufferBNBPool` contract.

By design, these functions should only be called by the `BufferBNBOptions` contract. If the address of `BufferBNBOptions` contract is not initialized to have the `OPTION_ISSUER_ROLE` role, the calling of these functions by the `BufferBNBPool` contract will not be successful.

```
    bytes32 public constant OPTION_ISSUER_ROLE =
        keccak256("OPTION_ISSUER_ROLE");

    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }
```

Listing 3.12: `BufferBNBPool::constructor()`

We understand the possibility of assigning the `OPTION_ISSUER_ROLE` to others by on-chain transactions. However, it is always helpful it is reflected in the smart contract implementation.

Recommendation Properly initialize the `OPTION_ISSUER_ROLE` parameter in the `BufferBNBPool::constructor()` function.

Status This issue has been resolved. The team clarifies that the `BNBOptions` contract address will not be known before deploying the `BNBPool` contract (as the `BNBOptions` contract needs the `BNBPool` contract as its constructor arguments.) With that, this issue is marked as resolved.

4 Conclusion

In this audit, we have analyzed the `Buffer` design and implementation. `Buffer` is a non-custodial on-chain peer-to-pool options trading protocol built on `Binance Smart Chain`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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