SMART CONTRACT AUDIT REPORT

for

ANKR Protocol

Prepared By: Patrick Lou

PeckShield

July 9, 2022

Document Properties

Client	ANKR
Title	Smart Contract Audit Report
Target	ANKR Protocol
Version	1.0
Author	Patrick Lou
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Xiaotao Wu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc1	March 19, 2022	Patrick Lou	Release Candidate #1
1.0	July 9, 2022	Patrick Lou	Final Release

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction

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

The ANKR liquid staking is an enhanced method of staking on the BNB Smart Chain (previously BSC). The enhancement allows the users to stake their funds through the corresponding smart contracts on ANKR, accumulate rewards, and receive their stakes as well as rewards when unstaking. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ANKR Protocol

Item	Description
Name	ANKR
Type	BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 9, 2022

In the following, we show the MD5 hash value of the compressed file used in this audit:

- MD5 (BNB.zip) = f44d26579438b7b2f43b6cd6d5ed659f

And here is the final MD5 hash value of the compressed file after all fixes for the issues found in the audit have been checked in:

- MD5 (BNB.zip) = b8616b15df08e972030e2a757fd9bde3

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

	Likelihood		
Impact	*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 [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 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

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

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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the ANKR 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	1	🟦
Medium	1	🟦
Low	1	🟦
Informational	0	
Total	3	

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, 1 medium-severity vulnerability and 1 informational recommendation.

Table 2.1: Key ANKR Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Properly Update State Variables in burnAndSetPending()/updatePendingBurning()	Coding Practices	Fixed
PVE-002	Low	Improved Validation Checks In distributeManual()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides 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 Properly Update State Variables in burnAndSetPending()/updatePendingBurning()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `BinancePool_R2`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The `ANKR` liquid staking allows the users to stake their funds through the corresponding smart contracts on the `BNB smart chain`, accumulate rewards, and receive their stakes/rewards when unstaking. While reviewing the `unstake()` logic, we notice that the function logic of `aBNBb_R2::burnAndSetPending()` needs to be corrected.

To elaborate, we show below this `burnAndSetPending()` function. The function is called at the end of the `BinancePool_R2::unstake()` to burn the corresponding staked bonds as well as reset some contract state variables to reflect the amount changes of the unstaked bonds (lines 214-217). It comes to our attention that it does not update the `_totalStaked` state variable which stores the value of total staked bonds. In other words, the `_totalStaked` state is incorrectly updated, hence leading to an incorrect calculation of the exchange ratio between bonds amount and underlying shares (in the `updateRatio()` function). More specifically, there is a need to reflect the deduction of total staked bonds.

```
209  function burnAndSetPending(address account, uint256 amount)
210  public
211  override
212  onlyBondMinter
213 {
214  _pendingBurn[account] = _pendingBurn[account] + amount;
215  _pendingBurnsTotal = _pendingBurnsTotal + amount;
```

```
 uint256 sharesToBurn = _bondsToShares(amount);
 _totalUnbondedBonds += amount;
 _burn(account, sharesToBurn);
 emit Transfer(account, address(0), amount);
}
```

Listing 3.1: `aBNBb_R2::burnAndSetPending()`

Also we notice there is another similar case which does not update the `_totalUnbondedBonds` state variable. During the unstaking process, upon the cross-transaction completion, the `BNB` backend service executes `BinancePool_R2::distributeRewards()` to distribute stakes and rewards to the users. It will invoke the following `updatePendingBurning()` function in which the related state variables are updated (lines 229-230). It comes to our attention that the `_totalUnbondedBonds` is not updated as it should be the same as in `burnAndSetPending()` function. Namely, we need to reflect the total unbounded bonds amount change by the following statement: `"_totalUnbondedBonds -= amount"`.

```
function updatePendingBurning(address account, uint256 amount)
public
override
onlyBondMinter
{
uint256 pendingBurnableAmount = _pendingBurn[account];
require(pendingBurnableAmount >= amount, "amount is wrong");
_pendingBurn[account] = pendingBurnableAmount - amount;
_pendingBurnsTotal = _pendingBurnsTotal - amount;
}
```

Listing 3.2: `aBNBb_R2::updatePendingBurning()`

Recommendation Properly update the `_totalStaked` and `_totalUnbondedBonds` state variables as suggested above.

Status This issue has been fixed in the following latest code package.

- MD5 (bnb.zip) = b8616b15df08e972030e2a757fd9bde3

3.2 Improved Validation Checks In distributeManual()

- ID: PVE-002
- Severity: Informational
- Likelihood: NA
- Impact: NA
- Target: `BinancePool_R2`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

In the `BinancePool_R2` contract, the `distributeManual()` function is used to distribute stakes and rewards to the specific user manually if the `distributeRewards()` function failed to do so for the user. It is used as a way to make sure the specific user can receive the stakes and rewards in case of the user fails to get them via `distributeRewards()`.

While reviewing the function, we notice that there is an issue with the condition check on whether to proceed with the function execution. To elaborate, we show below this `distributeManual()` function. Specifically, it validates the balance of this contract to ensure there is enough `BNB` left to transfer to the user (lines 207-210). In this case, the `amount` variable stores the amount of `BNB` that will be sent to the user and it only needs to check `address(this).balance >= amount` instead of the current logic `address(this).balance >= amount + stashedForManualDistributes` (line 208).

```
192  function distributeManual(uint256 id) external nonReentrant {
193      require(
194          markedForManualDistribute[id],
195          "not marked for manual distributing"
196      );
197      address[] memory claimers = new address[](1);
198      uint256[] memory amounts = new uint256[](1);
199
200      address claimer = _pendingClaimers[id];
201      address payable wallet = payable(address(claimer));
202      uint256 amount = pendingClaimerUnstakes[claimer];
203
204      markedForManualDistribute[id] = false;
205      stashedForManualDistributes -= amount;
206
207      require(
208          address(this).balance >= amount + stashedForManualDistributes,
209          "insufficient pool balance"
210      );
211      claimers[0] = claimer;
212      amounts[0] = amount;
213      IInternetBond(_bondContract).updatePendingBurning(claimer, amount);
214      pendingClaimerUnstakes[claimer] = 0;
215
216      (bool result, ) = wallet.call{value: amount}("");
217      require(result, "failed to send rewards to claimer");
218      delete _pendingClaimers[id];
219
220      emit RewardsDistributed(claimers, amounts, 0);
221 }
```

Listing 3.3: `BinancePool_R2::distributeManual()`

Recommendation Revise the above `distributeManual()` function to properly validate the contract balance check as below.

```
        require(
            address(this).balance >= amount,
            "insufficient pool balance"
        );
```

Listing 3.4: `BinancePool_R2::BinancePool_R2()`

Status This issue has been fixed in the following latest code package.

- MD5 (bnb.zip) = b8616b15df08e972030e2a757fd9bde3

3.3 Trust Issue of Admin Keys

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

Description

In the `ANKR` protocol, there are privileged `owner/minter/operator` accounts that play a critical role in governing and regulating the system-wide operations (e.g., mint and burn aBNBb tokens). These privileged accounts also have the capability of controlling or governing the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine these privileged accounts and the related privileged accesses in current contracts.

To elaborate, we show below example privileged routines in multiple contracts. These routines allow the `owner/minter/operator` accounts to mint new `aBNBb`, set system parameters, distribute rewards, etc.

```
function mint(address account, uint256 shares) public onlyMinter {
    _lockedShares = _lockedShares - int256(shares);
    _mint(account, shares);
    emit Transfer(address(0), account, shares);
}

function burn(address account, uint256 amount) public override onlyMinter {
    uint256 shares = _bondsToShares(amount);
    _lockedShares = _lockedShares + int256(shares);
    _burn(account, shares);
    emit Transfer(account, address(0), amount);
}
```

Listing 3.5: `aBNBb_R2::mint()/burn()`

```
    function setMinimumStake(uint256 minStake) external onlyOperator {
        _minimumStake = minStake;
    }

    function getRelayerFee() external view returns (uint256) {
        return _tokenHub.getMiniRelayFee();
    }

    function changeIntermediary(address intermediary) external onlyOwner {
        _intermediary = intermediary;
    }

    function changeBondContract(address bondContract) external onlyOwner {
        _bondContract = bondContract;
    }

    function changeTokenHub(address tokenHub) external onlyOwner {
        _tokenHub = ITokenHub(tokenHub);
    }
```

Listing 3.6: `BinancePool_R2::Multiple Functions`

It would be worrisome if each privileged `owner/minter/operator` 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the ANKR protocol, which allows the users stake their funds through the corresponding smart contracts, accumulate rewards, and receive their stakes and rewards when unstaking. During the audit, we notice that the current code base is well organized and those identified issues are 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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