SMART CONTRACT AUDIT REPORT

for

wBETH

Prepared By: Xiaomi Huang

PeckShield

March 28, 2023

Document Properties

Client	wBETH
Title	Smart Contract Audit Report
Target	wBETH
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 28, 2023	Xuxian Jiang	Final Release
1.0-rc	March 22, 2023	Luck Hu	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 4

1.1 About wBETH 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 Bypass of mintingAllowance in _mint() 11

3.2 Suggested immutable Usage for Gas Efficiency 12

3.3 Trust Issue of Admin Keys 14

4 Conclusion 16

References 17

1 Introduction

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

`Wrapped Binance Staked ETH` (`wBETH`) is an interest bearing liquid staking token for staked `ETH`. It will be live on `BSC` and `ETH` to enable users to participate in on-chain `Binance ETH` staking. The `Binance ETH Staking TVL` and the exchange rate of `wBETH:BETH` will be updated daily, and `wBETH` is expected to be used in various `DeFi` projects. The basic information of the audited contract is as follows:

Table 1.1: Basic Information of wBETH

Item	Description
Name	wBETH
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 28, 2023

In the following, we show the Git repository of reviewed file and the commit hash value used in this audit: Note the audit only covers the `contracts/withdraw/wBETHWithdraw.sol` file.

- https://github.com/earn-tech-git/wbeth/tree/develop (099b6c7b)

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

- https://github.com/earn-tech-git/wbeth/tree/develop (1a7fa2e7)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- 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

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) [7], 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 1.4: 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 wBETH 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 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	2	■ ■
Low	0	
Informational	1	■
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 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 and 1 informational recommendation.

Table 2.1: Key Audit Findings of wBETH Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Bypass of mintingAllowance in mint()	Business Logic	Fixed
PVE-002	Informational	Suggested immutable Usage for Gas Efficiency	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Bypass of mintingAllowance in mint()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: StakedTokenV1
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The StakedTokenV1 contract provides an interface, i.e., mint(), for the minters to mint new wBETH tokens to themselves. Each minter has a minting allowance (mintingAllowance) configured by Binance, which indicates how many tokens that minter is allowed to issue. While reviewing the implementation of the mint() routine, we notice the minter can issue more tokens than his/her minting allowance.

To elaborate, we show below the code snippet of the _mint() routine, which is called from the mint() routine to mint new wBETH tokens to the minter. As we can see from the comment of the _mint() routine (lines 280 – 281), the amount of tokens to mint must be less than or equal to the minting allowance of the caller. However, the implementation does not properly validate the input amount with the mintingAllowance. As a result, the minter can mint more tokens than he/she is allowed to.

Based on this, we suggest to validate the input amount with the minting allowance, and subtract the input amount from the minting allowance accordingly in the mint() routine.

```
277     /**
278      * @dev Function to mint tokens
279      * @param _to The address that will receive the minted tokens.
280      * @param _amount The amount of tokens to mint. Must be less than or equal
281      * to the minterAllowance of the caller.
282      * @return A boolean that indicates if the operation was successful.
283      */
284     function _mint(address _to, uint256 _amount)
285     internal
```

```
    whenNotPaused
    notBlacklisted(msg.sender)
    notBlacklisted(_to)
    returns (bool)
    {
        require(_to != address(0), "StakedTokenV1: mint to the zero address");
        require(_amount > 0, "StakedTokenV1: mint amount not greater than 0");

        totalSupply_ = totalSupply_.add(_amount);
        balances[_to] = balances[_to].add(_amount);

        emit Mint(msg.sender, _to, _amount);
        emit Transfer(address(0), _to, _amount);
        return true;
    }
```

Listing 3.1: StakedTokenV1::_mint

Recommendation Revisit the `mint()` routine to properly validate the input amount with the minting allowance, and subtract the input amount from the minting allowance accordingly.

Status The issue has been fixed by this commit: `508663f9`.

3.2 Suggested immutable Usage for Gas Efficiency

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Deployer`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variable `tokenOwner` in the `Deployer` contract (line 10). If there is no need to dynamically update this key variable, it can be declared as either constant or `immutable` for gas efficiency. In particular, the above state variable can be defined as `immutable` as it will not be changed after its initialization in `constructor()` (line 24).

Note the same issue is also applicable to the following state variables: `proxyAdmin/minter/operator/ethReceiver/oracle`.

```
8  contract Deployer {
9    ...
10   address public tokenOwner;
11   address public proxyAdmin;
12   address public minter;
13   address public operator;
14   address public ethReceiver;
15   address public oracle;

17   constructor(address _tokenOwner, address _proxyAdmin, address _minter, address
         _operatorWallet, address _ethReceiver) {
18       require(_tokenOwner != address(0), "zero _tokenOwner");
19       require(_proxyAdmin != address(0), "zero _proxyAdmin");
20       require(_minter != address(0), "zero _minter");
21       require(_operatorWallet != address(0), "zero _operatorWallet");
22       require(_ethReceiver != address(0), "zero _ethReceiver");

24       tokenOwner = _tokenOwner;
25       proxyAdmin = _proxyAdmin;
26       minter = _minter;
27       operator = _operatorWallet;
28       ethReceiver = _ethReceiver;
29       oracle = address(new ExchangeRateUpdater());
30   }
31   ...
32 }
```

Listing 3.2: `Deployer.sol`

Recommendation Revisit the state variables definition and make extensive use of `immutable` states for gas efficiency.

Status The issue has been fixed by this commit: `057cbb95`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the wBETH protocol, there is a privileged account, i.e., owner, that plays a critical role in governing and regulating the system-wide operations (e.g., update the oracle, update the ethReceiver). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the StakedTokenV1 contract as an example and show the representative functions potentially affected by the privileges of the owner account.

Specifically, the privileged functions in StakedTokenV1 allow for the owner to update the oracle who can set the exchange rate of wBETH:BETH, update the ethReceiver who will receive the staked ETH, update the operator who can move the staked ETH to the ethReceiver, etc.

```
    function updateOracle(address newOracle) external onlyOwner {
        require(
            newOracle != address(0),
            "StakedTokenV1: oracle is the zero address"
        );
        require(
            newOracle != oracle(),
            "StakedTokenV1: new oracle is already the oracle"
        );
        bytes32 position = _EXCHANGE_RATE_ORACLE_POSITION;
        assembly {
            sstore(position, newOracle)
        }
        emit OracleUpdated(newOracle);
    }

    function updateEthReceiver(address newEthReceiver) external onlyOwner {
        require(
            newEthReceiver != address(0),
            "StakedTokenV1: newEthReceiver is the zero address"
        );

        address currentReceiver = ethReceiver();
        require(newEthReceiver != currentReceiver, "StakedTokenV1: newEthReceiver is
            already the ethReceiver");

        bytes32 position = _ETH_RECEIVER_POSITION;
        assembly {
```

```
            sstore(position, newEthReceiver)
        }
        emit EthReceiverUpdated(currentReceiver, newEthReceiver);
    }

    function updateOperator(address newOperator) external onlyOwner {
        require(
            newOperator != address(0),
            "StakedTokenV1: newOperator is the zero address"
        );

        address currentOperator = operator();
        require(newOperator != currentOperator, "StakedTokenV1: newOperator is already the operator");

        bytes32 position = _OPERATOR_POSITION;
        assembly {
            sstore(position, newOperator)
        }
        emit OperatorUpdated(currentOperator, newOperator);
    }
```

Listing 3.3: Example Privileged Operations in the `StakedTokenV1` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 and the team will use a secure cold wallet scheme to manage the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of `wBETH`, which is short for `Wrapped Binance Staked ETH`. It is an interest bearing liquid staking token for staked `ETH`, which will be live on `BSC` and `ETH` to enable users to participate in on-chain `Binance ETH` staking. The `Binance ETH Staking TVL` and the exchange rate of `wBETH:BETH` will be updated daily, and `wBETH` is expected to be used in various `DeFi` projects. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

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

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

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

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

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

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