SMART CONTRACT AUDIT REPORT

for

deWeb

Prepared By: Yiqun Chen

PeckShield

July 4, 2021

Document Properties

Client	BBS Network
Title	Smart Contract Audit Report
Target	deWeb
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	July 4, 2021	Xiaotao Wu	Initial Draft

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 source code of the deWeb smart contract, 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 BBS Network

The BBS Network is a public network of interconnected message boards (nostalgically called BBSs, for the Bulletin Board Systems of the early Internet). Each and every post is stored as a unique NFT on-chain, and the BBS Network Token can be used to purchase those NFTs along with their associated ad real-estate (e.g. banners on posts), across the network. BBS tokens can also be staked to govern the network which is owned and operated by its community of token holders. BBS tokens are mined by generating verifiable engagement on user-created BBSs. In a sense, BBS can be likened to a "Public Reddit", distributed across multiple domains to prevent centralized censorship, while maintaining a network-effect and openness for anyone to build upon.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of deWeb

Item	Description
Target	deWeb
Website	https://www.bbsnetwork.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 4, 2021

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

- https://github.com/deweb-io/token.git (8b93dc9)

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

- https://github.com/deweb-io/token.git (ebcdcb0)

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 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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `deWeb` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
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 1 medium-severity vulnerability, 1 low-severity vulnerability and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Assumed Trust on Admin Keys	Security Features	Confirmed
PVE-002	Low	safeTransfer()/safeTransferFrom() Replacement	Coding Practices	Confirmed
PVE-003	Informational	Improved Validation of Function Arguments	Coding Practices	Fixed

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 Assumed Trust on Admin Keys

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

Description

In the BBSToken token contract, there is an owner account that plays a critical role such as minting a specified amount to the specified account.

```
    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
```

Listing 3.1: BBSToken::mint()

We note that the above mint() function allows for the owner to mint more tokens into circulation without being capped. We understand the need of the privileged functions 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 onlyOwner privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to owner explicit to BBSToken users.

Status This issue has been confirmed. Minting is the only extra privilege the owner has. deWeb will list this extra privilege granted to owner as part of the content on their web sites.

3.2 safeTransfer()/safeTransferFrom() Replacement

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: deWeb
- Category: Coding Practices [3]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`

In the following, we show the `distributeRewards()` routine in the `DailyRewards` contract. If the `ZRX` token is supported as the underlying token, the unsafe version of `bbsToken.transfer()` (line 70) may return false in the `ZRX` token contract's `transfer()` implementation (but the `IERC20` interface needs to consider it a failure)!

```
67   function distributeRewards() external {
68     require(block.timestamp - distributionTimestamp >= DISTRIBUTION_INTERVAL, "rewards distributed too recently");
69     for (uint16 rewardIndex = 0; rewardIndex < rewards.length; rewardIndex++) {
70         bbsToken.transfer(rewards[rewardIndex].beneficiary, rewards[rewardIndex].amountBBS);
71         emit RewardDistributed(rewards[rewardIndex].beneficiary, rewards[rewardIndex].amountBBS);
72     }
73     distributionTimestamp = block.timestamp;
74     emit RewardsDistributed();
75 }
```

Listing 3.3: `DailyRewards::distributeRewards()`

Note a number of routines can be similarly improved, including `Staking::declareReward()`, `Staking::lock()`, and `Staking::claim()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/transferFrom()`.

Status This issue has been confirmed. `deWeb` is dealing with the `BBS` token, which is fully inherited from OZ's ERC-20 contract, and never with any 3rd party tokens.

3.3 Improved Validation Of Function Arguments

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DailyRewards
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

The DailyRewards contract allows owner to declare rewards. Specifically, it provides an external declareRewards() function, which can only be called by the contract owner to declare a list of BBS token amounts to reward the corresponding beneficiaries on a daily basis. To elaborate, we show below this specific routine. It comes to our attention that the declareRewards() function has the inherent assumption on the same length of the given two arrays, i.e., beneficiariesToSet, and amountsToSet. However, this assumption is not always true.

```
function declareRewards(address[] calldata beneficiariesToSet, uint256[] calldata
    amountsToSet) external onlyOwner {
  delete declaredRewards;
  for (uint16 rewardIndex = 0; rewardIndex < beneficiariesToSet.length; rewardIndex++)
    {
      declaredRewards.push(Reward(beneficiariesToSet[rewardIndex], amountsToSet[
          rewardIndex]));
  }
  declarationTimestamp = block.timestamp;
  emit RewardsDeclared();
}
```

Listing 3.4: DailyRewards::declareRewards()

Recommendation Add the length check on all given arguments of declareRewards(). An example revision is shown below:

```
function declareRewards(address[] calldata beneficiariesToSet, uint256[] calldata
    amountsToSet) external onlyOwner {
  require(beneficiariesToSet.length == amountsToSet.length, "!length");
  delete declaredRewards;
  for (uint16 rewardIndex = 0; rewardIndex < beneficiariesToSet.length; rewardIndex++)
    {
      declaredRewards.push(Reward(beneficiariesToSet[rewardIndex], amountsToSet[
          rewardIndex]));
  }
  declarationTimestamp = block.timestamp;
  emit RewardsDeclared();
}
```

Listing 3.5: DailyRewards::declareRewards()

Status The issue has been fixed by this commit: `ebcdcb0`.

4 Conclusion

In this audit, we have analyzed the `deWeb` implementation. The system presents a unique, robust offering as a decentralized protocol to allow for a "Public Reddit", distributed across multiple domains to prevent centralized censorship, while maintaining a network-effect and openness for anyone to build upon. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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