SMART CONTRACT AUDIT REPORT

for

DFORCE NETWORK

Prepared By: Shuxiao Wang

Hangzhou, China

Aug. 24, 2020

Document Properties

Client	dForce Network
Title	Smart Contract Audit Report
Target	Smart Contract
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author	Description
1.0	Aug. 24, 2020	Chiachih Wu	Final Release

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Smart Contract** design document and related smart contract source code, we in the report outline our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the white paper, 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 an issue related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About Smart Contract

The basic information of Smart Contract is as follows:

Table 1.1: Basic Information of Smart Contract

Item	Description
Issuer	dForce Network
Token Name	dForce
Token Symbol	DF
Decimals	18
Total Supply of Tokens	1,000,000,000
Token Type	ERC20
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	Aug. 24, 2020

In the following, we show the list of reviewed contracts used in this audit:

- Contract Address: 0x431ad2ff6a9C365805eBaD47Ee021148d6f7DBe0

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [3]:

- 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.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

We perform the audit according to the following procedures:

- 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 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 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.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit 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 an investment advice.

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
	Approve / TransferFrom Race Condition
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

2 Findings

2.1 Summary

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

2.2 Key Findings

Overall, the smart contract implementation has no significant issue, though there exists a low severity issue (Table 2.1). However, this low severity issue cannot be exploited in the wild and therefore should not be a main concern.

Table 2.1: Key Audit Findings

ID	Severity	Title	Type
PVE-001	Low	Better Handling of Ownership Transfers	Operational Issue

Please refer to Chapter 3 for details.

3 | Detailed Results

3.1 Better Handling of Ownership Transfers

- ID: PVE-001
- Severity: Low
- Description: The `setOwner()` function in Ownable contract allows the current admin of the contract to transfer her privilege to another address. However, in the `setOwner()` function, the `owner_` is directly stored into the storage, `owner`, after only validating that the `owner_` is a non-zero address (line 120).
- Details: DSToken.sol:116

```
116     function setOwner(address owner_)
117         public
118         onlyOwner
119     {
120         require(owner_ != address(0), "invalid owner address");
121         owner = owner_;
122         emit LogSetOwner(owner);
123     }
```

Listing 3.1: DSToken.sol

This is reasonable under the assumption that the `owner_` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `owner_` is provided, the contract owner may be forever lost, which might be devastating for DF token smart contract operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. In other words, this two-step procedure ensures that an owner public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

- Recommendation: As suggested, the ownership transition can be better managed with a two-step approach, such as, using these two functions: `setOwner()` and `updateOwner()`. Specifically, the `setOwner()` function keeps the new address in the storage, `newOwner`, instead of modifying the `owner` directly. The `updateOwner()` function checks whether `newOwner` is `msg.sender` to ensure that `newOwner` signs the transaction and verifies herself as the new owner. Only after the successful verification, `newOwner` would effectively become the `owner`.

```
    function setOwner(address owner_)
        public
        onlyOwner
    {
        require(owner_ != address(0), "invalid owner address");
        require(owner_ != owner, "the current and new owner cannot be the same");
        require(owner_ != newOwner, "cannot set the candidate owner to the same address");
        newOwner = owner_;
    }

    function updateOwner()
        public
    {
        require(newOwner != address(0), "candidate owner had not been set");
        require(msg.sender == newOwner, "msg.sender and newOwner must be the same");
        owner = newOwner;
        emit LogSetOwner(newOwner);
    }
```

Listing 3.2: DSToken.sol

3.2 Other Suggestions

As a common suggestion, due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always preferred to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.2;` instead of `pragma solidity ^0.5.2;`.

4 | Conclusion

The Smart Contract was analyzed in this audit. No critical or high level vulnerabilities had been discovered so far, though there exists an minor low severity issue that cannot be exploited in the wild. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [4, 5, 6, 7, 9].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [10] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.1.18 Approve / TransferFrom Race Condition

- Description: Whether the contract has the known race condition [2] regarding `approve()` / `transferFrom()`.
- Result: Not found
- Severity: Low

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

[4] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[5] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[6] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[7] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[9] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[10] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.