SMART CONTRACT AUDIT REPORT

for

STN TOKEN

Prepared By: Shuxiao Wang

PeckShield

February 26, 2021

Document Properties

Client	StoneDeFi
Title	Smart Contract Audit Report
Target	STN Token
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuan Li, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	February 26, 2021	Xuxian Jiang	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 design document and related source code of the **STN Token** smart contract, 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 documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC20 compliance issues or security concerns. This document outlines our audit results.

1.1 About STN Token

The `Stone` token (`STN`) is the native utility token of the `Stone` protocol, which acts as the central part of the incentive structure of `Stone` DeFi ecosystem and interests of all ecosystem participants. Specifically, the `STN` token is the governance token that plays a crucial role in the ecosystem. Moreover, it is empowered to reward liquidity provision during the bootstrapping, incentivize portfolio rebalancing, pay transfer fees for the cross-chain execution, and act as the security deposit to safeguard funds in the liquid staked assets.

The basic information of STN Token is as follows:

Table 1.1: Basic Information of STN Token

Item	Description
Issuer	StoneDeFi
Website	https://www.stondefi.io/
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	February 26, 2021

In the following, we show the contract file and the MD5/SHA checksum value of the contract file:

- File: stone-token
- MD5: a144d69fb15b5abed31bd22c2ab0d177
- SHA: c92d89f26f7d678ea395374421c826854fe6ab19

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 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 [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.

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.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.2: Vulnerability Severity Classification

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

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 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 1.3: 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
ERC20 Compliance Checks	Compliance Checks (Section 3)
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

Here is a summary of our findings after analyzing the STN Token. 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 ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	1	🟦
Informational	1	🟦
Total	2	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, a minor ERC20 compliance issue was found and our detailed checklist can be found in Section 3. Also, there is no critical or high severity issue, although the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key STN Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Race Condition Between approve() And transferFrom()	Time and State	Fixed
PVE-002	Informational	Improved Event Generation With Indexed Assets	Coding Practices	Fixed

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 our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue found in the audited STN Token. Specifically, the `decimals` is currently defined as `uint256` while the specification requires the use of `uint8`. In particular, the related `getter` has the following definition:

`function decimals()public view returns (uint8).`

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	—
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

4 Detailed Results

4.1 Race Condition Between approve() And transferFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `Stone`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [3]

Description

The current `STN` ERC20 implementation has a known race condition issue between `approve()` and `transferFrom()` [1]. Specifically, when a user intends to reduce the spending allowance amount previously approved from, say, 10 `STN` to 1 `STN`. The user may race to spend all of the previously approved allowance (the 10 `STN`) and then additionally spend the new allowance amount just approved (1 `STN`). This breaks the user's intention of restricting the spending allowance to the new amount, **not** the sum of old amount and new amount.

```
    /**
        @notice Approve an address to spend the specified amount of tokens on behalf of
            msg.sender
        @dev Beware that changing an allowance with this method brings the risk that
            someone may use both the old
             and the new allowance by unfortunate transaction ordering. One possible
                 solution to mitigate this
             race condition is to first reduce the spender's allowance to 0 and set the
                 desired value afterwards:
             https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        @param _spender The address which will spend the funds.
        @param _value The amount of tokens to be spent.
        @return Success boolean
     */
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }
```

Listing 4.1: STN::approve()

```
    /**
        @notice Transfer tokens from one address to another
        @param _from The address which you want to send tokens from
        @param _to The address which you want to transfer to
        @param _value The amount of tokens to be transferred
        @return Success boolean
     */
    function transferFrom(
        address _from,
        address _to,
        uint256 _value
    )
        public
        returns (bool)
    {
        require(allowed[_from][msg.sender] >= _value, "Insufficient allowance");
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        _transfer(_from, _to, _value);
        return true;
    }
```

Listing 4.2: STN::transferFrom()

In order to properly approve the spending allowance, there also exists a known workaround: users can utilize the non-standard `increaseAllowance()` and `decreaseAllowance()` functions.

Recommendation Add the suggested workaround functions `increaseAllowance()` and `decreaseAllowance()` to mitigate the well-known issues around setting allowances. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

Status This issue has been fixed.

4.2 Improved Event Generation With Indexed Assets

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Stone
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [2]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

We have examined the generation of events per ERC20 specification. In the following, we list a few representative events that have been defined in STN Token.

```
contract Token {

    using SafeMath for uint256;

    string public symbol;
    string public name;
    uint256 public decimals;
    uint256 public totalSupply;

    mapping(address => uint256) balances;
    mapping(address => mapping(address => uint256)) allowed;

    event Transfer(address from, address to, uint256 value);
    event Approval(address owner, address spender, uint256 value);
```

Listing 4.3: Defined events in STN

It comes to our attention that the both events Transfer and Approval do not have the address information indexed. Note that each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, which means it will be attached as data (instead of a separate topic). Considering that the asset is typically queried, it is typically treated as a topic, hence the need of being indexed.

Recommendation Revise the above events by properly indexing the emitted asset information.

Status This issue has been fixed by indexing the address information in the emitted events.

4.3 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 0.6.0;` instead of `pragma solidity ^0.6.0;`.

5 Conclusion

In this security audit, we have examined the STN Token design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified a low severity issue and an informational suggestion that were promptly confirmed and fixed by the team. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

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

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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