# SMART CONTRACT AUDIT REPORT

for

# MAPLE TOKEN

Prepared By: Shuxiao Wang

PeckShield

April 4, 2021

## Document Properties

| | |
|---|---|
| Client | Maple Labs |
| Title | Smart Contract Audit Report |
| Target | Maple Token |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Yiqun Chen, Xuxian Jiang |
| Reviewed by | Shuxiao Wang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author | Description |
|---|---|---|---|
| 1.0 | April 4, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc | March 28, 2021 | Xuxian Jiang | Release Candidate |
| 0.1 | March 25, 2021 | Xuxian Jiang | First Draft |

## 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 **Maple Token** smart contract, we outline in the report 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 Maple Token

`Maple` is a decentralized corporate credit market that aims to provide capital to institutional borrowers through globally accessible fixed-income yield opportunities. In particular, `liquidity pools` are utilized to aggregate funding from `liquidity providers` and are loaned out to earn interest. The pools are professionally managed by `pool delegates` to provide as a sustainable yield source. And `Borrowers` request capital from the `Maple` protocol and pay associated interest fee. The `Maple` token is the protocol token of the `Maple` protocol. This audit covers the ERC20-compliance of the `Maple` token.

The basic information of Maple Token is as follows:

Table 1.1: Basic Information of Maple Token

| Item | Description |
|---|---|
| Issuer | Maple Labs |
| Website | https://maple.finance/ |
| Type | Ethereum ERC20 Token Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Audit Completion Date | April 4, 2021 |

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/maple-labs/maple-token.git (10d561f)

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

- https://github.com/maple-labs/maple-token.git (6903752)

## 1.2 About PeckShield

PeckShield Inc. [6] 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 [5]:

- 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.
- 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 1.3: 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 |

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.

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the Maple 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 | 2 | ■■ |
| Informational | 1 | ■ |
| Total | 3 | |

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 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Maple Token Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Accommodation of Non-ERC20-Compliant fundsToken | Business Logic | Fixed |
| PVE-002 | Informational | Inconsistency Between Document and Implementation | Coding Practices | Fixed |
| PVE-003 | Low | Consistency Between withdrawFunds() And withdrawFundsOnBehalf() | 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 no ERC20 inconsistency or incompatibility issue found in the audited Maple Token. 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-

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() | ✓ |

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/ERC2222), 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 Accommodation of Non-ERC20-Compliant fundsToken

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ERC2222
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### 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 the following, we examine the `transfer()` routine and related 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."*

```
64    function transfer(address _to, uint _value) returns (bool) {
65        //Default assumes totalSupply can't be over max (2^256 - 1).
66        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67            balances[msg.sender] -= _value;
68            balances[_to] += _value;
69            Transfer(msg.sender, _to, _value);
70            return true;
71        } else { return false; }
72    }
73
74    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 4.1: 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `withdrawFunds()` routine in the `ERC2222` contract. If the `USDT` token is supported as `tokenAddress`, the unsafe version of `fundsToken.transfer(msg.sender, withdrawableFunds)` (line 181) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
174     /**
175      * @dev Withdraws all available funds for a token holder
176      */
177     function withdrawFunds() public virtual override {
178         uint256 withdrawableFunds = _prepareWithdraw();
179
180         if (withdrawableFunds > uint256(0)) {
181             require(fundsToken.transfer(msg.sender, withdrawableFunds), "FDT:
                    TRANSFER_FAILED");
182
183             _updateFundsTokenBalance();
184         }
185     }
```

Listing 4.2: ERC2222::withdrawFunds()

**Recommendation** Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

**Status** The issue has been addressed by the following commit: `6903752`.

## 4.2 Inconsistency Between Document and Implementation

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: ERC2222
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

### Description

The Maple token is the native governance token of the Maple protocol and it is designed to be a FundsDistributionToken (FDT) token that inherits the ERC20 token standard and further implements the ERC2222 token standard.

During the analysis of the token contract, we notice a specific misleading comment located at line 213 of ERC2222::updateFundsReceived(). In particular, the preceding function summary indicates that "the contract computes the delta of the previous and the new funds token balance". However, the implemented logic (line 208) indicates it is the delta of the new and the previous funds token balance.

```
198     /**
199      * @dev Updates the current funds token balance
200      * and returns the difference of new and previous funds token balances
201      * @return A int256 representing the difference of the new and previous funds token
             balance
202      */
203     function _updateFundsTokenBalance() internal virtual returns (int256) {
204         uint256 _prevFundsTokenBalance = fundsTokenBalance;
205
206         fundsTokenBalance = fundsToken.balanceOf(address(this));
207
208         return int256(fundsTokenBalance).sub(int256(_prevFundsTokenBalance));
209     }
210
211     /**
212      * @dev Register a payment of funds in tokens. May be called directly after a
             deposit is made.
213      * @dev Calls _updateFundsTokenBalance(), whereby the contract computes the delta of
               the previous and the new
214      * funds token balance and increments the total received funds (cumulative) by delta
               by calling _registerFunds()
215      */
216     function updateFundsReceived() public virtual {
217         int256 newFunds = _updateFundsTokenBalance();
218
219         if (newFunds > 0) {
220             _distributeFunds(newFunds.toUint256Safe());
221         }
```

```
222     }
```

Listing 4.3: ERC2222::updateFundsReceived()

**Recommendation** Ensure the consistency between documents (including embedded comments) and implementation.

**Status** The issue has been addressed by the following commit: `6903752`.

## 4.3 Consistency Between withdrawFunds() And withdrawFundsOnBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ERC2222`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

### Description

As mentioned in Section 4.2, the `Maple` token follows the ERC2222 token standard that allows the token holders to be able to claim proportionate amounts of interest that have been paid over time. Specifically, there are two related routines, i.e., `withdrawFunds()` and `withdrawFundsOnBehalf()`, that can be used by token holders to withdraw all available funds, either directly by the token holders themselves or indirectly on behalf of token holders.

Our analysis with these two routines shows certain unnecessary inconsistency. In particular, For gas optimization purposes, the `withdrawFunds()` routine has implemented an enhancement that avoids making the asset-transferring call if the transferred amount is 0. However, the same enhancement has not been taken in the `withdrawFundsOnBehalf()` routine.

To elaborate, we show below the code snippet of `ERC2222::withdrawFundsOnBehalf()`. This routine allows for withdrawing all available funds on behalf of a token holder. However, it also makes the transfer request (line 193) even when the given `withdrawableFunds` is 0.

```
174     /**
175      * @dev Withdraws all available funds for a token holder
176      */
177     function withdrawFunds() public virtual override {
178         uint256 withdrawableFunds = _prepareWithdraw();
179 
180         if (withdrawableFunds > uint256(0)) {
181             require(fundsToken.transfer(msg.sender, withdrawableFunds), "FDT:
                    TRANSFER_FAILED");
```

```
182
183             _updateFundsTokenBalance();
184         }
185     }
186
187     /**
188      * @dev Withdraws all available funds for a token holder, on behalf of token holder
189      */
190     function withdrawFundsOnBehalf(address user) public virtual {
191         uint256 withdrawableFunds = _prepareWithdrawOnBehalf(user);
192
193         require(fundsToken.transfer(user, withdrawableFunds), "FDT:TRANSFER_FAILED");
194
195         _updateFundsTokenBalance();
196     }
```

Listing 4.4: ERC2222::withdrawFunds() and ERC2222::withdrawFundsOnBehalf()

**Recommendation** Avoid the token `transfer()` call when the transferred amount is 0 in both `withdrawFunds()` and `withdrawFundsOnBehalf()` routines.

**Status** The issue has been addressed by the following commit: 6903752.

# 5 | Conclusion

In this security audit, we have examined the Maple 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 two low severity issues 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

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