SMART CONTRACT AUDIT REPORT

for

3WM Token

Prepared By: Yiqun Chen

PeckShield

January 26, 2022

Document Properties

Client	3WM
Title	Smart Contract Audit Report
Target	3WM Token
Version	1.0
Author	Patrick Liu
Auditors	Patrick Liu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	January 26, 2022	Patrick Liu	Final Release

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 design document and related source code of the 3WM token 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 can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About 3WM Token

3WM is a global industrial project dedicated to protect the environment and improving living conditions. Its ambition is to revolutionize the way we manage waste and plastics, the way we treat water, the way we move, by creating the world's largest network of bioenergy plants and the world's first green intelligence network. With that, 3WM is a BEP20 and ERC20 compliant cross chain token that acts as the reward token for the 3WM project.

The basic information of the audited token contract is as follows:

Table 1.1: Basic Information of 3WM Token

Item	Description
Issuer	3WM
Website	https://3wm.io/
Type	BSC BEP20/Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	January 26, 2022

In the following, we show the bscscan link for the 3WM token contract used in this audit.

- https://bscscan.com/address/0xd065ca6460ae6d379c93ec0422084c2bc7048d77

And the etherscan link for the `3WM` token contract is shown below.

- https://etherscan.io/address/0xf2157ea9c60547c9bdd50123e924bd55d388de1a

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.

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 3WM token contract. 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	1	🟦
Low	0	
Informational	2	🟦 🟦
Total	3	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20/BEP20 specification and other known best practices, and validate its compatibility with other similar ERC20/BEP20 tokens and current DeFi protocols. The detailed ERC20/BEP20 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/BEP20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, and 2 informational issues.

Table 2.1: Key 3WM Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Non ERC20-Compliance Of Transfer Event	Coding Practices	Confirmed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-003	Informational	isVestedlisted Logic Bypass Via transferFrom()	Coding Practices	Confirmed

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 4 for details.

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).

Note as BEP20 extends ERC20 specification and all the ERC20 validations here apply to BEP20 token contract too.

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

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 audited 3WM Token. And the details are elaborated in Section 4.1. 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/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 Non ERC20-Compliance Of Transfer Event

- ID: PVE-001
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: BEP20TokenContract
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [2]

Description

In the 3WM token contract, there is a mint() function that is responsible for minting additional 3WM tokens into circulation. As specified in the ERC20 specification, when new tokens are minted, a Transfer event will be emitted. Our analysis shows this event can be improved for ERC20 compliance.

To elaborate, we show below the related code snippet . The ERC20 standard specifies that "a token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created." However, in current mint() function, it emits the Transfer event by specifying the contract itself as the _from. For better ERC20 compliance, it is suggested to strictly follow the ERC20 standard.

```
  function mint(address _to, uint256 _amount) public onlyAuthorized canMint returns (
    bool) {
    totalSupply = totalSupply.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Mint(_to, _amount);
    emit Transfer(address(this), _to, _amount);
    return true;
   }
```

Listing 4.1: BEP20TokenContract::mint()

Recommendation Revise the 3WM token implementation to ensure its ERC20-compliance.

Status This issue has been confirmed.

4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High

- Target: BEP20TokenContract
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In 3WM, there is a privileged owner account that plays a critical role in governing and regulating the token-related operations. To elaborate, we show below the privileged mint()/lockToken()/setICO() functions that allow the owner to add new tokens into circulation, disable token transfer and set the privileged ICO address respectively.

```
function mint(address _to, uint256 _amount) public onlyAuthorized canMint returns (bool)
     {
    totalSupply = totalSupply.add(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Mint(_to, _amount);
    emit Transfer(address(this), _to, _amount);
    return true;
  }
```

Listing 4.2: BEP20TokenContract::unlockToken()()/lockToken()/setICO()

```
      function lockToken() public onlyAuthorized returns (bool) {
        locked = true;
        emit LockToken();
        return true;
      }

      function setICO(address _icocontract) public onlyOwner returns (bool) {
        require(_icocontract != address(0));
        ico = _icocontract;
        emit SetICO(_icocontract);
        return true;
      }
```

Listing 4.3: BEP20TokenContract::lockToken()/setICO()

Our analysis shows that the owner address is currently configured as 0x8667d6122408028d913ce2143e0910143207bd1a which is an EOA account. The administrative key concern may be alleviated with a multi-sig account, though it is far from perfect. Specifically, a better approach is to eliminate the administrative key concern by transferring the privileged roles to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Promptly transfer the `owner` privilege of `3WM` token to the intended governance contract. And 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.

4.3 isVestedlisted Logic Bypass Via transferFrom()

- ID: PVE-003
- Severity: Informational
- Likelihood: NA
- Impact: NA

- Target: `BEP20TokenContract`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized BEP20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the additional functionalities applied to the `transfer()` and `transferFrom()` functions.

To elaborate, we show below their implementation. Notice that there is an additional `!isVestedlisted` check (line 268) before performing the intended token transfer. By design, the `vestedlist` feature aims to prevent those accounts which are in `vestedlist` from transferring out their assets.

```
266     function transfer(address _to, uint256 _value) public canTransfer returns (bool) {
267         require(_to != address(0));
268         require (!isVestedlisted(msg.sender));
269         require(_value <= balances[msg.sender]);
270         require (msg.sender != address(this));
271 
272         // SafeMath.sub will throw if there is not enough balance.
273         balances[msg.sender] = balances[msg.sender].sub(_value);
274         balances[_to] = balances[_to].add(_value);
275         emit Transfer(msg.sender, _to, _value);
276         return true;
277       }
```

Listing 4.4: `BEP20TokenContract::transfer()`

However, it comes to our attention that the designed `vestedlist` feature can be bypassed. Specifically, those accounts in `vestedlist` can transfer their own tokens through non-`vestedlist` accounts by calling `approve()`, followed by `transferFrom()`.

```
332     function approve(address _spender, uint256 _value) public returns (bool) {
333         allowed[msg.sender][_spender] = _value;
334         emit Approval(msg.sender, _spender, _value);
335         return true;
```

```
    }
```

Listing 4.5: `BEP20TokenContract::approve()`

In particular, the `approve()` function enables every account including an account in `vestedlist` to authorize others to transfer a certain amount of tokens out on behalf of itself. As a result, with the help of a non-vestedlist account, a user in `vestedlist` can completely bypass the `isVestedlisted()` check, which apparently goes against the design.

Recommendation Override `transferFrom()` to apply the same logic as `transfer()`.

Status This issue has been confirmed.

5 Conclusion

In this security audit, we have examined the design and implementation of the 3WM token contract. 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 three issues of varying severities. 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-1109: Use of Same Variable for Multiple Purposes. https://cwe.mitre.org/data/definitions/1109.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

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