SMART CONTRACT AUDIT REPORT

for

VITE TOKEN

Prepared By: Shuxiao Wang

PeckShield

April 12, 2021

Document Properties

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

Version Info

Version	Date	Author	Description
1.0	April 12, 2021	Yiqun Chen	Release Candidate
0.1	April 7, 2021	Yiqun Chen	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 **Vite** 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 some issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About Vite Token

`Vite` is a lightning-fast public blockchain where transactions incur zero fees. It is arguably one of DAG-based smart contract platforms with the flagship DApp `ViteX`, a trustless DEX deployed on the `Vite` chain. `ViteX` adopts the most cutting-edge decentralized exchange technology by implementing on-chain order matching, settlement, mining, and dividends distribution through smart-contracts on `Vite` chain. It is proposed and designed with the vision that many blockchains will grow to serve different needs and `Vite` aims to bridge current blockchains in a decentralized way.

The audited `Vite` token contract follows the `BEP20` standard and is deployed at the `Binance Smart Chain` (`BSC`). The basic information is as follows:

Table 1.1: Basic Information of Vite Token

Item	Description
Client	Vite Labs
Website	https://www.vite.org/
Type	BEP20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	April 12, 2021

In the following, we show the contract code deployed at the BSC chain with the following address:

- https://testnet.bscscan.com/address/0xb7f4f07c24b57dd931644a446c81300e9fcab378#code

And here is the revised contract code after all fixes have been checked in:

- https://testnet.bscscan.com/address/0x01b9600b266fa3dbb09f915e22844e4df38d556d#code

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 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 [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.

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.

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 `Vite` 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	1	■
Medium	1	■
Low	1	■
Informational	0	
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, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. However, there are two medium severity vulnerabilities and one high severity vulnerability, which requires an urgent fix-up. (shown in Table 2.1)

Table 2.1: Key Vite Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Blacklist Bypass Via transferFrom()	Business Logic	Fixed
PVE-002	Medium	Trust Issue Of Admin Roles	Business Logic	Confirmed
PVE-003	Low	Two-Step Transfer Of Privileged Account Ownership	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 Vite 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 Blacklist Bypass Via transferFrom()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `ERC20`
- Category: Coding Practices [4]
- 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 the following, we examine the additional functionality applied to the `transfer()` and `transferFrom()` and their code is shown in Listing 4.1.

Notice that there is an additional blacklist check before performing the intended token transfer. By design, the blacklist feature aims to prevent blacklisted accounts from transferring out their assets.

```
    function transfer(address _to, uint256 _value) public returns (bool) {
      require(tokenBlacklist[msg.sender] == false);
      require(_to != address(0));
      require(_value <= balances[msg.sender]);

      // SafeMath.sub will throw if there is not enough balance.
      balances[msg.sender] = balances[msg.sender].sub(_value);
      balances[_to] = balances[_to].add(_value);
      emit Transfer(msg.sender, _to, _value);
      return true;
    }
   function transferFrom(address _from, address _to, uint256 _value) public returns (bool) {
      require(tokenBlacklist[msg.sender] == false);
      require(_to != address(0));
      require(_value <= balances[_from]);
      require(_value <= allowed[_from][msg.sender]);

```

```
        balances[_from] = balances[_from].sub(_value);
        balances[_to] = balances[_to].add(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;
    }
```

Listing 4.1: StandardToken::transfer() & StandardToken::transferFrom()

However, it comes to our attenion that the designed blacklist feature can be bypassed. Specifically, blacklisted accounts can not only receive tokens from others, but also transfer their own tokens through non-blacklisted accounts by calling `approve()`, followed by `transferFrom()`.

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

Listing 4.2: StandardToken::approve()

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

Recommendation Validate all the addresses involved in functions `transfer()` and `transferFrom()`, i.e. `address(_from)& address (_to)& msg.sender`, and ensure they do not show up on the blacklist.

Status This issue has been fixed by validating all related addresses involved in token transfers.

4.2 Trust Issue Of Admin Roles

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ERC20`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the `Vite` token contract, the `owner` account plays a critical role in governing and regulating the entire operation and maintenance (e.g., account blacklisting, and mint/burn). It also has the privilege to pause the current token contract for withdrawals.

```
    function mint(address account, uint256 amount) onlyOwner public {

        totalSupply = totalSupply.add(amount);
        balances[account] = balances[account].add(amount);
        emit Mint(address(0), account, amount);
        emit Transfer(address(0), account, amount);
    }
```

Listing 4.3: CoinToken::mint()

To elaborate, we show above a privileged `mint()` function. This function allows for the `owner` account to mint more tokens into circulation without being capped. We understand the need of the privileged functions for contract maintenance, 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 Vite Token users.

Status This issue has been confirmed.

4.3 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ERC20`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1109 [1]

Description

The `Vite` token contract implements a rather basic access control mechanism that allows a privileged account, i.e., `owner`, to be granted exclusive access to typically sensitive functions (e.g., `mint()` in Section 4.2). Because of the privileged access and the implications of this sensitive function, the `owner` account is essential for the protocol-level safety and operation. In the following, we elaborate with the `owner` account.

```
    /**
     * @dev Allows the current owner to transfer control of the contract to a newOwner.
     * @param newOwner The address to transfer ownership to.
     */
    function transferOwnership(address newOwner) public onlyOwner {
      require(newOwner != address(0));
      emit OwnershipTransferred(owner, newOwner);
```

```
        owner = newOwner;
    }
```

Listing 4.4: Ownable::transferOwnership()

The current implementation provides a specific function, i.e., `transferOwnership()`, to allow for possible `owner` updates. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the `newOwner` parameter is always correctly provided. However, in the unlikely situation, when an incorrect `newOwner` is provided, the contract owner may be forever lost, which might be devastating for `Vite` 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 owner to an uncontrolled address. In other words, this two-step procedure ensures that a 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 Implement a two-step approach for owner update (or transfer): `transferOwnership()` and `acceptOwnership()`.

Status This issue has been fixed by following the suggested two-step procedure.

5 | Conclusion

In this security audit, we have examined the design and implementation of the `Vite` 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 vulnerabilities were discovered, we identified three issues of varying severities 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-1109: Use of Same Variable for Multiple Purposes. https://cwe.mitre.org/data/definitions/1109.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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