SMART CONTRACT AUDIT REPORT

for

VeChain VVET

Prepared By: Yiqun Chen

PeckShield

August 27, 2021

Document Properties

Client	VeChain
Title	Smart Contract Audit Report
Target	VVET
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 27, 2021	Xuxian Jiang	Final Release
1.0-rc1	August 25, 2021	Xuxian Jiang	Release Candidate #1

PeckShield

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

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the VVET token contract, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, 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 several issues related to either security or performance. This document outlines our audit results.

1.1 About VVET

VVET is a VIP-180 standard-compliant token that wraps and maintains a pegged 1:1 exchange rate with VET on VeChain. The VeChain is a cryptocurrency and smart contracts platform with a bi-token system that includes the VeChain Token (VET) and VeThor Token (VTHO). VET serves as the value-transfer medium to enable rapid value circulation. And VTHO represents the underlying cost and will be consumed (or, in other words, destroyed) after on-chain operations are performed. Note the holders of VVETs enjoy the same VTHO generation as if they were holding VETs.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of VVET

Item	Description
Target	VVET
Website	https://www.vechain.org/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 27, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/VeChainDEXCode/vvet.git (818fa11)

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

- https://github.com/VeChainDEXCode/vvet.git (6339f7f)

1.2 About PeckShield

PeckShield Inc. [12] 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

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

PeckShield

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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 accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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
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

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 or analysis 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.

In particular, we perform the audit according to the following procedure:

- 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 better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `vVET` 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	■ ■
Informational	2	■ ■
Total	4	

PeckShield

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-002	Informational	Redundant Check in Modifier checkItemBalance()	Coding Practices	Fixed
PVE-003	Low	approve()/transferFrom() Race Condition	Time and State	Confirmed
PVE-004	Informational	Suggested Event Generation For Allowance Changes	Coding Practices	Fixed

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 3 for details.

PeckShield

3 Detailed Results

3.1 Improved Precision By Multiplication-Before-Division

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: StakingModel
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [3]

Description

The lack of float support in Solidity may introduce a subtle and troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the calculateVTHO() (in the StakingModel contract) as an example. This routine is used to calculate the amount of VTHOs that will be rewarded to the user.

```
85     function calculateVTHO(
86         uint48 t1,
87         uint48 t2,
88         uint104 vetAmount
89     ) public pure returns (uint104 vtho) {
90         require(t1 <= t2, "t1 should be <= t2");
91         return ((vetAmount * 5) / (10**9)) * (t2 - t1);
92     }
```

Listing 3.1: StakingModel::calculateVTHO()

We notice the calculation of the amount of VTHOs (line 91) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., (vetAmount * 5)* (t2 - t1)/ (10**9).

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the commit: 695b76b.

3.2 Redundant Check in Modifier checkItemBalance()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: VVET9
- Category: Security Features [6]
- CWE subcategory: CWE-269 [4]

Description

In VVET9, there is a routine withdraw() that is used to withdraw VETs. To elaborate, we show below the code snippet of related functions.

```
28      function withdraw(uint256 wad) public {
29          require(vetBalance(msg.sender) >= wad);
30          removeVET(msg.sender, wad);
31          payable(msg.sender).transfer(wad);
32          emit Withdrawal(msg.sender, wad);
33      }
```

Listing 3.2: VVET9::withdraw()

```
36      function removeVET(address addr, uint256 amount) restrict(amount) internal {
37          _update(addr);
38          require(users[addr].balance >= uint104(amount), "insuffcient vet");
39          users[addr].balance -= uint104(amount);
40      }
```

Listing 3.3: StakingModel::removeVET()

We notice that the check on require(vetBalance(msg.sender)>= wad) in the withdraw() routine (line 29) is redundant because there is the same validity check in the removeVET() function, which checks users[addr].balance >= uint104(amount) (line 38). Note the same issue also exists on another routine, i.e., transferFrom().

Recommendation Remove the redundant checks on withdraw() and transferFrom().

Status The issue has been fixed by the following commits: 6339f7f and 27a1f4e.

3.3 approve()/transferFrom() Race Condition

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: VVET9
- Category: Time and State [7]
- CWE subcategory: CWE-362 [5]

Description

There is a known race condition issue regarding approve() / transferFrom() [1]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 VVETs to 1 VVET. The previously approved spender might race to transfer the amount the user initially approved (the 10 VVETs) and then additionally spend the new amount the user just approved (1 VVET). This breaks the user's intention of restricting the spender to the new amount, **not** the sum of old amount and new amount.

To alleviate such concern, it is recommended to apply a known workaround (e.g., increaseApproval()/decreaseApproval()) and further add necessary sanity checks when entering the approve() function.

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

Listing 3.4: VVET9::approve()

Recommendation Add additional sanity checks in approve() and workaround functions increaseApproval()/decreaseApproval().

```
    function approve(address guy, uint256 wad) public returns (bool) {
        require((wad ==0) || allowance[msg.sender][guy] ==0);
        allowance[msg.sender][guy] = wad;
        emit Approval(msg.sender, guy, wad);
        return true;
    }
    function increaseApproval(address guy, uint wad) external returns (bool) {
        allowance[msg.sender][guy] = add(allowance[msg.sender][guy], (wad));
        emit Approval(msg.sender, guy, allowance[msg.sender][guy]);
        return true;
    }
    function decreaseApproval(address guy, uint wad) external returns (bool) {
        allowance[msg.sender][guy] = sub(allowance[msg.sender][guy], (wad));
        emit Approval(msg.sender, guy, allowance[msg.sender][guy]);
        return true;
    }
```

Listing 3.5: Revised VVET9

Status The issue has been confirmed by the team.

3.4 Suggested Event Generation For Allowance Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: VVET9
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [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.

In the following, we use the transferFrom() routine as an example. This routine is used to transfer tokens from the sender to the recipient using the allowance mechanism, which deducts amount from the caller's allowance. While examining the events that reflect the protocol dynamics, we notice an Approval event could be emitted on calls to transferFrom(). This allows applications to reconstruct the allowance for all accounts just by listening to the events.

```
    function transferFrom(
        address src,
        address dst,
        uint256 wad
    ) public returns (bool) {
        require(vetBalance(src) >= wad);

        if (
            src != msg.sender && allowance[src][msg.sender] != type(uint256).max
        ) {
            require(allowance[src][msg.sender] >= wad);
            allowance[src][msg.sender] -= wad;
        }

        removeVET(src, wad);
        addVET(dst, wad);

        emit Transfer(src, dst, wad);

        return true;
    }
```

Listing 3.6: VVET9::transferFrom()

Recommendation Emits an `Approval` event to indicate the updated allowance.

Status This issue has been fixed in the commit: 473d87a.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the VVET token contract, which implements a VIP-180 standard token and maintains a pegged 1:1 exchange rate with VET. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

PeckShield

References

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

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

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[4] MITRE. CWE-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

PeckShield

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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

PeckShield