SMART CONTRACT AUDIT REPORT

for

VANILLA

Prepared By: Shuxiao Wang

PeckShield

March 16, 2021

Document Properties

Client	Equilibrium
Title	Smart Contract Audit Report
Target	Vanilla
Version	1.0
Author	Ruiyi Zhang
Auditors	Ruiyi Zhang, Xuxian Jiang, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 16, 2021	Ruiyi Zhang	Final Release
1.0-rc1	March 08, 2021	Ruiyi Zhang	Release Candidate
0.2	March 01, 2021	Ruiyi Zhang	Additional Findings
0.1	February 28, 2021	Ruiyi Zhang	Initial 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 4
- 1.1 About Vanilla . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Possible Risk in Front-running _sellInUniswap() . . . 11
- 3.2 Improved Sanity Checks in estimateReward() . . . 13

4 Conclusion 15

References 16

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **Vanilla** protocol, 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 Vanilla

`Vanilla` is a trustless interface for all of DeFi that rewards users for making a profit trading and lending tokens. When users trade and lend through Vanilla's unified DeFi interface, users participate in `ProfitMining` whereby users mine VNL governance tokens for each fraction of an ether users make in profit.

The basic information of Vanilla is as follows:

Table 1.1: Basic Information of Vanilla

Item	Description
Client	Equilibrium
Website	https://equilibrium.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 16, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit. Note that Vanilla assumes a trusted entity to update timely and reliable market price feeds for supported assets.

- https://github.com/vanilladefi/contracts (9bf4f09)

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

- https://github.com/vanilladefi/contracts (f0a0587)

1.2 About PeckShield

PeckShield Inc. [5] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 the OWASP Risk Rating Methodology [4]:

- 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 1.3: The Full Audit Checklist

Category	Checklist Items
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 checklist of 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) [3], 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.

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 Logic	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 Vanilla implementation. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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 that 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 1 high-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Vanilla Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Possible Risk in Front-running _sellInUniswap()	Business Logic	Fixed
PVE-002	Informational	Improved Sanity Checks in estimateReward()	Business Logic	Fixed

Beside the identified issue, 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.

3 | Detailed Results

3.1 Possible Risk in Front-running _sellInUniswap()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: UniswapTrader.sol
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

In Vanilla, a user can do two things — buy and sell ERC-20 tokens.

- To buy a token, user calls VanillaRouter.buy()-function, along with the address of the traded token, the limit amount of tokens, and a deadline timestamp. VanillaRouter holds the direct ownership of the tokens and keeps record of the assets of each user.

- To sell a token, user calls VanillaRouter.sell()-function, along with the address of the traded token, number of sold tokens, the limit amount of Ether, and a deadline timestamp. Users can only sell the tokens, which they have bought themselves.

Also, trading transaction can revert for multiple reasons, e.g., Uniswap can not buy or sell the token for the given price (the constant-product invariant is violated). In this section, we examine one issue related to the above reason for revert. To elaborate, we show below the implementation of the _sellInUniswap() routine.

```
    address tokenCustody = address(this);
    uint256 balance = IERC20(_wethAddr).balanceOf(tokenCustody);

    // Use TransferHelper because we have no idea here how token.transfer() has been
        implemented
    TransferHelper.safeTransfer(token_, pairAddress, amount_);
    if (tokenFirst) {
        (uint112 tokenReserve, uint112 wethReserve, ) = pair.getReserves();
```

```
155         pair.swap(
156             uint256(0),
157             _amountToSwap(amount_, tokenReserve, wethReserve),
158             tokenReceiver_,
159             new bytes(0)
160         );
161         reserve = _updateTokenReserveOnSell(token_, wethReserve);
162     } else {
163         (uint112 wethReserve, uint112 tokenReserve, ) = pair.getReserves();
164         pair.swap(
165             _amountToSwap(amount_, tokenReserve, wethReserve),
166             uint256(0),
167             tokenReceiver_,
168             new bytes(0)
169         );
170         reserve = _updateTokenReserveOnSell(token_, wethReserve);
171     }
172     // finally check how the custody balance has changed after swap
173     numEth = IERC20(_wethAddr).balanceOf(tokenReceiver_) - balance;
174     // revert if the price diff between trade-time and execution-time was too large
175     require(numEth >= eth_, _ERROR_SLIPPAGE_LIMIT_EXCEEDED);
```

Listing 3.1: UniswapTrader::_sellInUniswap()

After swap, there is a check ensuring that the amount of the received WETH tokens is greater than `numEthLimit`. However, the `tokenReceiver_`'s balance gets the balance of `tokenCustody` after swap (line 149). Specifically, the WETH-balance of `tokenCustody` will practically always be zero, resulting in the subsequent check (line 175) not being triggered. It is worth mentioning that calling the `sellAndWithdraw()` function is not affected by the same issue. In that case, the `tokenCustody` and the `tokenReceiver` refer to the same address (the VanillaRouter contract) and the check (line L175) works as intended.

Recommendation Correct the logic of getting WETH-balance of the tokenReceiver before swap.

Status The issue has been fixed by this commit: `f0a0587`.

3.2 Improved Sanity Checks in estimateReward()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VanillaRouter.sol`
- Category: Business Logic [2]
- CWE subcategory: CWE-841 [1]

Description

Users are rewarded with a number of `VanillaGovernanceTokens`. The function `estimateReward()` estimates the reward for the given `owner` when selling `numTokensSold` tokens for `numEth` Ether. Also, it returns the individual components of the reward formula. However, the sanity check requires the `tokenPriceData.tokenSum` (line 316) of owner equals zero, while it should be greater than zero.

```
299     function estimateReward(
300         address owner,
301         address token,
302         uint256 numEth,
303         uint256 numTokensSold
304     )
305         external
306         view
307         returns (
308             uint256 profitablePrice,
309             uint256 avgBlock,
310             uint256 htrs,
311             uint256 vpc,
312             uint256 reward
313         )
314     {
315         PriceData storage prices = tokenPriceData[owner][token];
316         require(prices.tokenSum == 0, _ERROR_NO_TOKEN_OWNERSHIP);
317         profitablePrice = numTokensSold.mul(prices.ethSum).div(prices.tokenSum);
318         avgBlock = prices.weightedBlockSum.div(prices.tokenSum);
319         if (numEth > profitablePrice) {
320             uint256 profit = numEth.sub(profitablePrice);
321             uint128 wethReserve = wethReserves[token];
322             htrs = _estimateHTRS(avgBlock);
323             vpc = _estimateVPC(profit, wethReserve);
324             reward = _calculateReward(
325                 epoch,
326                 avgBlock,
327                 block.number,
328                 profit,
329                 wethReserve,
330                 reserveLimit
331             );
332         } else {
```

```
            htrs = 0;
            vpc = 0;
            reward = 0;
        }
    }
```

Listing 3.2: VanillaRouter::estimateReward()

Recommendation Correct the logic of the above sanity check.

```
    require(prices.tokenSum == 0, _ERROR_NO_TOKEN_OWNERSHIP);
```

Listing 3.3: VanillaRouter::estimateReward()

Status The issue has been fixed by this commit: `48d4bf0`.

4 Conclusion

In this audit, we have analyzed the Vanilla design and implementation. `Vanilla` is a trustless interface for all of DeFi that rewards users for making a profit trading and lending tokens. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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.

References

[1] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[2] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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