SMART CONTRACT AUDIT REPORT

for

Revert UniswapV3Staker

Prepared By: Xiaomi Huang

PeckShield

September 7, 2022

Document Properties

Client	Revert Finance
Title	Smart Contract Audit Report
Target	UniswapV3Staker
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 7, 2022	Luck Hu	Final Release
1.0-rc	September 2, 2022	Luck Hu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Revert UniswapV3Staker, 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About UniswapV3Staker

The UniswapV3Staker of Revert Finance is a rewards program which is designed on the solid base of the canonical uniswap v3 staker contract. In addition, it introduces a new configuration value called vestingPeriod, which defines the minimal time a staked position needs to be in range to receive the full reward. The basic information of the audited UniswapV3Staker is as follows:

Table 1.1: Basic Information of UniswapV3Staker

Item	Description
Name	Revert Finance
Website	https://revert.finance//
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 7, 2022

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

- https://github.com/revert-finance/v3-staker.git (e42b172)

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

- https://github.com/revert-finance/v3-staker.git (af5bd83)

1.2 About PeckShield

PeckShield Inc. [8] 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 the OWASP Risk Rating Methodology [7]:

- 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.

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

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

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) [6], 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 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 implementation of the `UniswapV3Staker` smart 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 logic, 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	1	
Low	2	
Informational	0	
Total	3	

We have so far identified a list of potential issues. 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, the smart contract are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key UniswapV3Staker Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Initialization of secondsInsideInitial in _stakeToken()	Coding Practices	Fixed
PVE-002	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Mitigated
PVE-003	Low	Improved Validation of Function Arguments	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.

3 | Detailed Results

3.1 Proper Initialization of secondsInsideInitial in _stakeToken()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UniswapV3Staker
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [2]

Description

The UniswapV3Staker contract provides an incentive mechanism that rewards the staking of the supported Uniswap V3 LP tokens. The rewards are carried out by creating a number of incentive programs with the supported Uniswap V3 pools, the reward tokens and the vesting periods, etc. And staking users are rewarded as a function of in-range liquidity, i.e. the more trading volume the position helps generate, the more rewards user will earn.

To elaborate, we show below the code snippet of the _stakeToken(). As the name indicates, it is used for the LP token owner to stake the LP token into an incentive program. If there is no problem with the inputs, a new Stake record will be created. In particular, the new Stake records the initial secondsInside accumulated in the range (tickLower, tickUpper), which will be used to calculate the amount of time spent in the range for vesting. However, it comes to our attention that, it records the initial secondsInside in the new Stake only when the following requirement is met, i.e., (liquidity >= type(uint64).max). Otherwise, it misses to record the initial secondsInside in the Stake. As a result, it may calculate an unexpected amount of vested tokens to the user. So, it is suggested to properly record the initial secondsInside for the new Stake even when (liquidity < type(uint64).max).

```
    function _stakeToken(IncentiveKey memory key, uint256 tokenId) private {
        require(block.timestamp >= key.startTime, 'UniswapV3Staker::stakeToken:
            incentive not started');
        require(block.timestamp < key.endTime, 'UniswapV3Staker::stakeToken: incentive
            ended');
```

```
        bytes32 incentiveId = IncentiveId.compute(key);

        require(
            incentives[incentiveId].totalRewardUnclaimed > 0,
            'UniswapV3Staker::stakeToken: non-existent incentive'
        );
        require(
            _stakes[tokenId][incentiveId].liquidityNoOverflow == 0,
            'UniswapV3Staker::stakeToken: token already staked'
        );

        (IUniswapV3Pool pool, int24 tickLower, int24 tickUpper, uint128 liquidity) =
            NFTPositionInfo.getPositionInfo(factory, nonfungiblePositionManager, tokenId
                );

        require(pool == key.pool, 'UniswapV3Staker::stakeToken: token pool is not the
            incentive pool');
        require(liquidity > 0, 'UniswapV3Staker::stakeToken: cannot stake token with 0
            liquidity');

        deposits[tokenId].numberOfStakes++;
        incentives[incentiveId].numberOfStakes++;

        (, uint160 secondsPerLiquidityInsideX128, uint32 secondsInside) = pool.
            snapshotCumulativesInside(tickLower, tickUpper);

        if (liquidity >= type(uint64).max) {
            _stakes[tokenId][incentiveId] = Stake({
                secondsPerLiquidityInsideInitialX128: secondsPerLiquidityInsideX128,
                secondsInsideInitial: secondsInside,
                liquidityNoOverflow: type(uint64).max,
                liquidityIfOverflow: liquidity
            });
        } else {
            Stake storage stake = _stakes[tokenId][incentiveId];
            stake.secondsPerLiquidityInsideInitialX128 = secondsPerLiquidityInsideX128;
            stake.liquidityNoOverflow = uint64(liquidity);
        }

        emit TokenStaked(tokenId, incentiveId, liquidity);
    }
```

Listing 3.1: UniswapV3Staker::_stakeToken()

Recommendation Properly record the initial `secondsInside` for the new `Stake`.

Status The issue has been fixed by this commit: `af5bd83`.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `UniswapV3Staker`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `UniswapV3Staker` contract provides an incentive mechanism that rewards the staking of the supported `Uniswap V3 LP` tokens with the reward tokens. The reward tokens are transferred into the contract by the incentive program creator when creating the incentive program via the `createIncentive()` routine. In the `createIncentive()` routine, the contract makes the use of `safeTransferFrom()` routine (line 125) to transfer assets into the contact. This routine works as expected with standard ERC20 tokens: namely the contract's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
100     function createIncentive(IncentiveKey memory key, uint256 reward) external override {
101         require(reward > 0, 'UniswapV3Staker::createIncentive: reward must be positive');
102         require(
103             block.timestamp <= key.startTime,
104             'UniswapV3Staker::createIncentive: start time must be now or in the future'
105         );
106         require(
107             key.startTime - block.timestamp <= maxIncentiveStartLeadTime,
108             'UniswapV3Staker::createIncentive: start time too far into future'
109         );
110         require(key.startTime < key.endTime, 'UniswapV3Staker::createIncentive: start time must be before end time');
111         require(
112             key.endTime - key.startTime <= maxIncentiveDuration,
113             'UniswapV3Staker::createIncentive: incentive duration is too long'
114         );

116         require(
117             key.vestingPeriod <= key.endTime - key.startTime,
118             'UniswapV3Staker::createIncentive: vesting time must be lte incentive duration'
119         );

121         bytes32 incentiveId = IncentiveId.compute(key);

123         incentives[incentiveId].totalRewardUnclaimed += reward;
```

```
        TransferHelperExtended.safeTransferFrom(address(key.rewardToken), msg.sender,
            address(this), reward);

        emit IncentiveCreated(key.rewardToken, key.pool, key.startTime, key.endTime, key
            .vestingPeriod, key.refundee, reward);
    }
```

Listing 3.2: UniswapV3Staker::createIncentive()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind the asset-transferring routines. In other words, the above operations, such as `createIncentive()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into `UniswapV3Staker` for support.

Recommendation Check the balance before and after the `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status This issue has been mitigated by the team that they will not be supporting rebasing/deflationary token and will make explicit that is the case in the UI.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV3Staker`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [1]

Description

The `UniswapV3Staker` contract provides a pair of routines, i.e., `createIncentive()/endIncentive()`, for the incentive program creator to create and end an incentive program. In particular, in the `endIncentive()` routine, if there are unclaimed rewards or locked rewards, they will be refunded to the `key.refundee` identified by the creator. However, in the `createIncentive()`, there is a lack of validation for the `key.refundee`. As a result, if the `key.refundee` is `address(0)`, the left rewards will be locked in the contract. So, it is suggested to add proper validation for the `key.refundee` to ensure it is a valid address.

```
    function createIncentive(IncentiveKey memory key, uint256 reward) external override
        {
        require(reward > 0, 'UniswapV3Staker::createIncentive: reward must be positive')
            ;
        require(
            block.timestamp <= key.startTime,
            'UniswapV3Staker::createIncentive: start time must be now or in the future'
        );
        require(
            key.startTime - block.timestamp <= maxIncentiveStartLeadTime,
            'UniswapV3Staker::createIncentive: start time too far into future'
        );
        require(key.startTime < key.endTime, 'UniswapV3Staker::createIncentive: start
            time must be before end time');
        require(
            key.endTime - key.startTime <= maxIncentiveDuration,
            'UniswapV3Staker::createIncentive: incentive duration is too long'
        );

        require(
            key.vestingPeriod <= key.endTime - key.startTime,
            'UniswapV3Staker::createIncentive: vesting time must be lte incentive
                duration'
        );

        bytes32 incentiveId = IncentiveId.compute(key);

        incentives[incentiveId].totalRewardUnclaimed += reward;

        TransferHelperExtended.safeTransferFrom(address(key.rewardToken), msg.sender,
            address(this), reward);

        emit IncentiveCreated(key.rewardToken, key.pool, key.startTime, key.endTime, key
            .vestingPeriod, key.refundee, reward);
    }
```

Listing 3.3: UniswapV3Staker::createIncentive()

What is more, the `UniswapV3Staker` contact provides the deposit owners the ability to transfer their deposits to new owners via the `transferDeposit()` routine. At the beginning of the `transferDeposit()` routine, it validates the new owner to be a valid address via `require(to != address(0))`. Our analysis shows that, there is a need to add a new validation `require(to != address(this))` to ensure the deposit can not be transferred to this contract. Otherwise, the LP token of the deposit will be locked in the contact.

```
    function transferDeposit(uint256 tokenId, address to) external override {
        require(to != address(0), 'UniswapV3Staker::transferDeposit: invalid transfer
            recipient');
        address owner = deposits[tokenId].owner;
        require(owner == msg.sender, 'UniswapV3Staker::transferDeposit: can only be
            called by deposit owner');
        deposits[tokenId].owner = to;
        emit DepositTransferred(tokenId, owner, to);
    }
```

Listing 3.4: UniswapV3Staker::transferDeposit()

Recommendation Improve the validations for the parameters of the above mentioned routines.

Status The issue has been fixed by this commit: `af5bd83`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `UniswapV3Staker` contact which is designed based on the canonical `uniswap` v3 staker contract. It introduces a new configuration value called `vestingPeriod`, which defines the minimal time a staked position needs to be in range to recieve the full reward. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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