SMART CONTRACT AUDIT REPORT

for

StakingRewardsV3

Prepared By: Yiqun Chen

PeckShield

August 29, 2021

Document Properties

Client	StakingRewardsV3
Title	Smart Contract Audit Report
Target	StakingRewardsV3
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 29, 2021	Xuxian Jiang	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 the `StakingRewardsV3` 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 is well engineered and 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 StakingRewardsV3

The `StakingRewardsV3` protocol implements the classic `Synthetix StakingRewards` contract for `UniswapV3 NFT` positions. It has a rather standard simple to use interface since users can simply stake their `UniswapV3 NFT` positions and receive respective pro-rata rewards. In particular, the provided liquidity incentives are readily available on `UniswapV3` with respect to their `range positions` (proportional to liquidity provided in a given range).

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	StakingRewardsV3
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 29, 2021

In the following, we show the repository of reviewed files and the MD5 checksum hash value used in this audit.

- URL: https://gist.github.com/andrecronje/211f1a18857efd2fc8359e929f076ceb
- MD5: 3bed3c03983d335cf4265d8bab885a59

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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 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

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

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) [7], 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 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 implementation of the `StakingRewardsV3` protocol. 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	1	■
Low	2	■ ■
Informational	0	
Total	4	

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 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of StakingRewardsV3 Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper tokenId-Related Accounting	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Patterns	Time and State	Fixed
PVE-003	Low	Generation Of Meaningful Events	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 details.

3 Detailed Results

3.1 Proper tokenId-Related Accounting

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: StakingRewardsV3
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The current StakingRewardsV3 implements the classic Synthetix StakingRewards contract for UniswapV3 NFT positions. To properly keep track of each UniswapV3 NFT position, the StakingRewardsV3 contract maintains a number of tokenId-related states, including liquidityOf, elapsed, owners, tokenExists, and tokenIds. While analyzing the bookkeeping logic about these states, we notice the logic needs to be improved when a UniswapV3 NFT position is unstaked.

To elaborate, we show below the related deposit()/withdraw() functions. It comes to our attention that this deposit() function properly maintains all related states. However, the withdraw() function only updates liquidityOf and owners, but misses the logic to update the associated states on tokenExists and tokenIds.

```
    function deposit(uint tokenId) external update(tokenId) {
        (,,address token0, address token1,uint24 fee,int24 tickLower,int24 tickUpper,
            uint128 _liquidity,,,,) = nftManager.positions(tokenId);
        address _pool = PoolAddress.computeAddress(factory,PoolAddress.PoolKey({token0:
            token0, token1: token1, fee: fee}));
        (,uint160 _secondsPerLiquidityInside,) = UniV3(pool).snapshotCumulativesInside(
            tickLower, tickUpper);

        require(pool == _pool);
        require(_liquidity > 0);

        liquidityOf[tokenId] = _liquidity;
```

```
        elapsed[tokenId] = time(uint32(lastTimeRewardApplicable()),
            _secondsPerLiquidityInside);

        nftManager.transferFrom(msg.sender, address(this), tokenId);
        owners[tokenId] = msg.sender;

        if (!tokenExists[msg.sender][tokenId]) {
            tokenExists[msg.sender][tokenId] = true;
            tokenIds[msg.sender].push(tokenId);
        }
    }

    function withdraw(uint tokenId) public update(tokenId) {
        require(owners[tokenId] == msg.sender);
        liquidityOf[tokenId] = 0;
        owners[tokenId] = address(0);
        nftManager.safeTransferFrom(address(this), msg.sender, tokenId);
    }
```

Listing 3.1: `StakingRewardsV3::deposit()/withdraw()`

Recommendation Revise the above `withdraw()` to properly update all related states when a `UniswapV3 NFT` position is unstaked.

Status This issue has been fixed by properly updating the related state `tokenIds` and removing the need of `tokenExists`.

3.2 Suggested Adherence Of Checks-Effects-Interactions Patterns

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewardsV3`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `StakingRewardsV3` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 151) starts before effecting the update on internal states (lines 152−157), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function.

```
139     function deposit(uint tokenId) external update(tokenId) {
140         (,,address token0, address token1,uint24 fee,int24 tickLower,int24 tickUpper,
                uint128 _liquidity,,,,) = nftManager.positions(tokenId);
141         address _pool = PoolAddress.computeAddress(factory,PoolAddress.PoolKey({token0:
                token0, token1: token1, fee: fee}));
142         (,uint160 _secondsPerLiquidityInside,) = UniV3(pool).snapshotCumulativesInside(
                tickLower, tickUpper);
143 
144         require(pool == _pool);
145         require(_liquidity > 0);
146 
147         liquidityOf[tokenId] = _liquidity;
148 
149         elapsed[tokenId] = time(uint32(lastTimeRewardApplicable()),
                _secondsPerLiquidityInside);
150 
151         nftManager.transferFrom(msg.sender, address(this), tokenId);
152         owners[tokenId] = msg.sender;
153 
154         if (!tokenExists[msg.sender][tokenId]) {
155             tokenExists[msg.sender][tokenId] = true;
156             tokenIds[msg.sender].push(tokenId);
157         }
158     }
```

Listing 3.2: `StakingRewardsV3::deposit()`

Recommendation Apply necessary reentrancy prevention by following the common `checks-effects-interactions (CEI)` best practice or making use of the common `nonReentrant` modifier.

Status The issue has been fixed by adding the suggested `re-entrancy` protection.

3.3 Generation Of Meaningful Events

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StakingRewardsV3
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

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 StakingRewardsV3 contract as an example. This contract is designed to implement the classic Synthetix StakingRewards contract for UniswapV3 NFT positions. While examining the events that reflect the reward changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the reward is being updated, there is no respective event being emitted to reflect the changes.

```
    function getReward(uint tokenId) public update(tokenId) {
        uint _reward = rewards[tokenId];
        if (_reward > 0) {
            rewards[tokenId] = 0;
            _safeTransfer(reward, _getRecipient(tokenId), _reward);
        }
    }
```

Listing 3.3: StakingRewardsV3::getReward()

Moreover, a number of related events are suggested for necessary generation, including RewardPaid in getReward(), RewardAdded in notify(), Stake in deposit(), and Unstake in withdraw().

Recommendation Properly emit respective events to help off-chain monitoring and accounting tools.

Status This issue has been fixed by adding the suggested events.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StakingRewardsV3` protocol, which implements the classic `Synthetix StakingRewards` contract for `UniswapV3 NFT` positions. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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.

References

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

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.