SMART CONTRACT AUDIT REPORT

for

StakingRewardsV3-1

Prepared By: Yiqun Chen

PeckShield

November 22, 2021

Document Properties

Client	StakingRewardsV3-1
Title	Smart Contract Audit Report
Target	StakingRewardsV3-1
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	November 22, 2021	Xuxian Jiang	Release Candidate

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 smart 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 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 tick range).

Table 1.1: Basic Information of the StakingRewardsV3 Contract

Item	Description
Name	StakingRewardsV3-1
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 22, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of smart contracts and this audit covers only the file named StakingRewardsV3-1.sol as well as the related dependencies.

- https://github.com/keep3r-network/StakingRewardsV3.git (bb2aeaa)

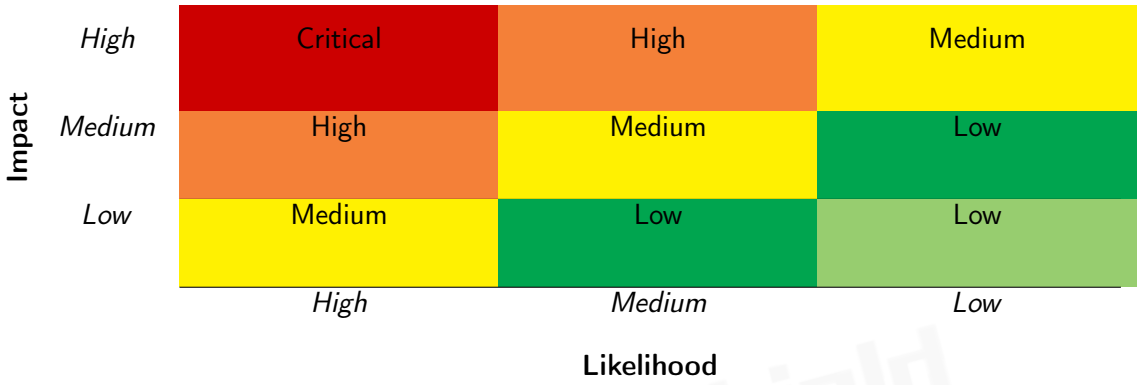
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/keep3r-network/StakingRewardsV3.git (TBD)

1.2 About PeckShield

PeckShield Inc. [7] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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

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

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) [5], 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` 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 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	3	■■■
Informational	0	
Total	3	

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 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of StakingRewardsV3-1 Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Generation of Meaningful Events For Governance Changes	Coding Practices	
PVE-002	Low	Proper Liquidity Range Validation in deposit()	Business Logic	
PVE-003	Low	Proper Storage Release in _withdraw()	Coding Practices	

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 Generation of Meaningful Events For Governance Changes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StakingRewardsV3
- Category: Coding Practices [3]
- 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 governance changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the governance is being updated, there is no respective event being emitted to reflect the changes.

```
151     function setGovernance(address _governance) external onlyGovernance {
152         nextGovernance = _governance;
153         delayGovernance = block.timestamp + DELAY;
154     }

156     function acceptGovernance() external {
157         require(msg.sender == nextGovernance && delayGovernance < block.timestamp);
158         governance = nextGovernance;
159     }
```

Listing 3.1: StakingRewardsV3::setGovernance()/acceptGovernance()

The same issue is also applicable when the `treasury` is changed.

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

Status

3.2 Proper Liquidity Range Validation in deposit()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakingRewardsV3`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `StakingRewardsV3` contract has a built-in incentive mechanism to encourage participating users to provide required liquidity. Specifically, users can provide liquidity around the current price for the given pool. Whether the provided liquidity is "in-range" is confirmed by comparing with the current price of the pool. In particular, the contract takes from users their tokenized positions through the `UniswapV3 NonfungiblePositionManager` and users are rewarded with a portion of the constant stream of reward tokens proportional to the share of total virtual liquidity they have provided.

To validate whether the provided liquidity is "in-range," the protocol has a handy validation `require(tickLower < _tick && _tick < tickUpper)` (line 237). This validation essentially checks the given position's `tick` falls within the range. However, the range validation needs to be performed as `require(tickLower <= _tick && _tick < tickUpper)`.

```
229     function deposit(uint tokenId) external update(tokenId) {
230         (,,address token0,address token1,uint24 fee,int24 tickLower,int24 tickUpper,
                uint128 _liquidity,,,,) = nftManager.positions(tokenId);
231         address _pool = PoolAddress.computeAddress(factory,PoolAddress.PoolKey({token0:
                token0, token1: token1, fee: fee}));
232
233         require(pool == _pool);
234         require(_liquidity > 0);
235
236         (,int24 _tick,,,,,) = UniV3(_pool).slot0();
237         require(tickLower < _tick && _tick < tickUpper);
238
239         nftManager.transferFrom(msg.sender, address(this), tokenId);
240
241         owners[tokenId] = msg.sender;
242         tokenIds[msg.sender].push(tokenId);
```

```
243
244         liquidityOf[tokenId] = _liquidity;
245         totalLiquidity += _liquidity;
246
247         emit Deposit(msg.sender, tokenId, _liquidity);
248     }
```

Listing 3.2: StakingRewardsV3::deposit()

Recommendation Revise the above deposit() routine to properly validate whether the given liquidity falls "in-range."

Status

3.3 Proper Storage Release in _withdraw()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StakingRewardsV3
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

To properly keep track of each position's contribution, te StakingRewardsV3 contract maintains a number of tokenId-related states, i.e., owners, liquidityOf, and elapsed. These states will be updated for each deposit and withdraw operations. While reviewing the accounting of these states, we notice the current withdrawal logic can be improved.

To elaborate, we show below the related _withdraw() function, what processes the withdraw request. It comes to our attention that it properly resets owners and liquidityOf of the removed tokenId, but leaves the elapsed state as is. Since freeing the Ethereum blockchain from storage variables and contracts leads to a gas refund, we suggest to reset the elapsed as well, i.e., adding the following statement delete elapsed[tokenId].

```
274     function _withdraw(uint tokenId) internal {
275         require(owners[tokenId] == msg.sender);
276         uint _liquidity = liquidityOf[tokenId];
277         liquidityOf[tokenId] = 0;
278         totalLiquidity -= _liquidity;
279         owners[tokenId] = address(0);
280         _remove(tokenIds[msg.sender], tokenId);
281         nftManager.transferFrom(address(this), msg.sender, tokenId);
282
283         emit Withdraw(msg.sender, tokenId, _liquidity);
```

```
284     }
```

Listing 3.3: `StakingRewardsV3::_withdraw()`

Recommendation Properly release all unused blockchain states to take the gas refund.

Status

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StakingRewardsV3` smart contract, 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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