SMART CONTRACT AUDIT REPORT

for

Beamswap

Prepared By: Xiaomi Huang

PeckShield

May 25, 2022

Document Properties

Client	Beamswap
Title	Smart Contract Audit Report
Target	Beamswap
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 25, 2022	Xiaotao Wu	Final Release
1.0-rc1	May 20, 2022	Xiaotao Wu	Release Candidate #1

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 Beamswap 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 Beamswap

The Beamswap is a decentralized exchange with an automated market maker with the support of liquidity provision and peer-to-peer transactions. Built on the Moonbeam network, Beamswap aims to provide new features in supporting the swap of crypto assets, both fungible and non-fungible, earning passive income from staking and yield farming, and even launching specific crypto projects on Moonbeam . The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Beamswap

Item	Description
Issuer	Beamswap
Website	https://beamswap.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 25, 2022

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

- https://github.com/BeamSwap/beamswap-stableamm.git (32697ca)

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 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 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) [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.

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 `Beamswap` protocol 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 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	0	
Informational	3	■ ■ ■
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 3 informational suggestions.

Table 2.1: Key Beamswap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-002	Informational	Redundant State/Code Removal	Coding Practices	Confirmed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-004	Informational	Inconsistency Between Document and Implementation	Coding Practices	Confirmed

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 Meaningful Events For Important State Changes

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SwapUtils
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

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 withdrawAdminFees() routine as an example. This routine is designed for the owner to withdraw all admin fees from the current pool to a given address. While examining the implementation of this routine, we notice when the withdrawAdminFees() is being called, there is no corresponding event being emitted to reflect the occurrence of withdrawAdminFees().

```
    /**
     * @notice withdraw all admin fees to a given address
     * @param self Swap struct to withdraw fees from
     * @param to Address to send the fees to
     */
    function withdrawAdminFees(Swap storage self, address to) external {
        IERC20[] memory pooledTokens = self.pooledTokens;
        for (uint256 i = 0; i < pooledTokens.length; i++) {
            IERC20 token = pooledTokens[i];
            uint256 balance =
                token.balanceOf(address(this)).sub(self.balances[i]);
            if (balance != 0) {
                token.safeTransfer(to, balance);
```

```
            }
        }
    }
```

Listing 3.1: `SwapUtils::withdrawAdminFees()`

Recommendation Properly emit the related event when the above-mentioned function is being invoked.

Status The issue has been fixed by this commit: `2b333f1`.

3.2 Redundant State/Code Removal

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Swap`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [3]

Description

The `Beamswap` contracts makes good use of a number of reference contracts, such as `SafeERC20`, `SafeMath`, `SwapUtils`, and `AmplificationUtils`, to facilitate its code implementation and organization. For example, the `Swap` smart contract has so far imported at least seven reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `Swap` contract. It defines the `TokenSwap`, `AddLiquidity`, `RemoveLiquidity`, `RemoveLiquidityOne`, `RemoveLiquidityImbalance`, `NewAdminFee`, `NewSwapFee`, `RampA`, and `StopRampA` events. These events are also defined either in `SwapUtils` or `AmplificationUtils`. With that, they could be removed from the `Swap` contract.

```
    event TokenSwap(
        address indexed buyer,
        uint256 tokensSold,
        uint256 tokensBought,
        uint128 soldId,
        uint128 boughtId
    );
    event AddLiquidity(
        address indexed provider,
        uint256[] tokenAmounts,
        uint256[] fees,
        uint256 invariant,
        uint256 lpTokenSupply
```

```
    );
    event RemoveLiquidity(
        address indexed provider,
        uint256[] tokenAmounts,
        uint256 lpTokenSupply
    );
    event RemoveLiquidityOne(
        address indexed provider,
        uint256 lpTokenAmount,
        uint256 lpTokenSupply,
        uint256 boughtId,
        uint256 tokensBought
    );
    event RemoveLiquidityImbalance(
        address indexed provider,
        uint256[] tokenAmounts,
        uint256[] fees,
        uint256 invariant,
        uint256 lpTokenSupply
    );
    event NewAdminFee(uint256 newAdminFee);
    event NewSwapFee(uint256 newSwapFee);
    event RampA(
        uint256 oldA,
        uint256 newA,
        uint256 initialTime,
        uint256 futureTime
    );
    event StopRampA(uint256 currentA, uint256 time);
```

Listing 3.2: Swap.sol

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been confirmed.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Swap
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Beamswap contracts implementation, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations. In the following, we show the representative functions potentially affected by the privileged owner.

To elaborate, we show below the code snippet of the privileged functions in the Swap contract, which give right to the owner to withdraw admin fees to the given account, set new admin/swap fee rates, and ramp A parameter, etc.

```
    /**
     * @notice Withdraw all admin fees to the contract owner
     */
    function withdrawAdminFees() external onlyOwner {
        swapStorage.withdrawAdminFees(owner());
    }

    /**
     * @notice Update the admin fee. Admin fee takes portion of the swap fee.
     * @param newAdminFee new admin fee to be applied on future transactions
     */
    function setAdminFee(uint256 newAdminFee) external onlyOwner {
        swapStorage.setAdminFee(newAdminFee);
    }

    /**
     * @notice Update the swap fee to be applied on swaps
     * @param newSwapFee new swap fee to be applied on future transactions
     */
    function setSwapFee(uint256 newSwapFee) external onlyOwner {
        swapStorage.setSwapFee(newSwapFee);
    }

    /**
     * @notice Start ramping up or down A parameter towards given futureA and futureTime
     * Checks if the change is too rapid, and commits the new A value only when it falls
        under
     * the limit range.
     * @param futureA the new A to ramp towards
     * @param futureTime timestamp when the new A should be reached
     */
```

```
    function rampA(uint256 futureA, uint256 futureTime) external onlyOwner {
        swapStorage.rampA(futureA, futureTime);
    }

    /**
     * @notice Stop ramping A immediately. Reverts if ramp A is already stopped.
     */
    function stopRampA() external onlyOwner {
        swapStorage.stopRampA();
    }
```

Listing 3.3: `Swap.sol`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms that multi-sig will be adopted for the privileged account.

3.4 Inconsistency Between Document and Implementation

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SwapUtils`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the protocol implementation.

An example comment can be found at line 135 of `SwapUtils::calculateWithdrawOneToken()`. The preceding function summary indicates that the input `tokenAmount` parameter is *"the amount to withdraw in the pool's precision"*. However, the implementation logic indicates it is *"the amount of the lp token to burn"*.

```
    /**
     * @notice Calculate the dy, the amount of selected token that user receives and
     * the fee of withdrawing in one token
     * @param tokenAmount the amount to withdraw in the pool's precision
     * @param tokenIndex which token will be withdrawn
     * @param self Swap struct to read from
     * @return the amount of token user will receive
     */
    function calculateWithdrawOneToken(
        Swap storage self,
        uint256 tokenAmount,
        uint8 tokenIndex
    ) external view returns (uint256) {
        (uint256 availableTokenAmount, ) =
            _calculateWithdrawOneToken(
                self,
                tokenAmount,
                tokenIndex,
                self.lpToken.totalSupply()
            );
        return availableTokenAmount;
    }
```

Listing 3.4: `SwapUtils::calculateWithdrawOneToken()`

Also, there is another inconsistency at line 188 of `SwapUtils::calculateWithdrawOneTokenDY()`.

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Beamswap` design and implementation. The `Beamswap` is a decentralized exchange with the support of liquidity provision and peer-to-peer transactions. The protocol is built on the `Moonbeam` network. The current code base is well structured and neatly organized. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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