SMART CONTRACT AUDIT REPORT

for

SquadSwap (Dynamo, Wow)

Prepared By: Xiaomi Huang

PeckShield

February 2, 2025

Document Properties

Client	SquadSwap
Title	Smart Contract Audit Report
Target	SquadSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 2, 2025	Xuxian Jiang	Final Release
1.0-rc	February 1, 2025	Xuxian Jiang	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 `SquadSwap` 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 SquadSwap

`SquadSwap` new version brings a new level of flexibility to its fee structure. Specifically, there was a fixed fee rate in the previous version. And the new version allows this fee rate to be adjusted. This changeable fee system makes it easier to adapt to market conditions and manage liquidity more effectively. By keeping the simplicity of the original system while adding this customization option, the revised protocol offers a smoother and more flexible trading experience. This audit covers the latest revision for customized fee support in `SquadSwap`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SquadSwap

Item	Description
Target	SquadSwap
Website	https://squadswap.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 2, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that this audit only covers the following contracts: `SquadswapFactory.sol`, `SquadswapPair.sol`, `V2_5Migrator.sol`, `SquadswapRouter02.sol`, `SquadswapLibrary.sol`, `V2SwapRouter.sol`, `MixedRouteQuoterV1.sol`, and `SmartRouterHelper.sol`.

- https://github.com/skyrocktech/SquadSwap.git (1cad0cd)
- https://github.com/skyrocktech/SquadSwapV3.git (23ca3dd)

1.2 About PeckShield

PeckShield Inc. [6] 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 [5]:

- 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 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) [4], 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 1.3: 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

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 new `SquadSwap` 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	0	
Medium	0	
Low	2	■ ■
Informational	1	■
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 2 low-severity vulnerabilities and and 1 informational recommendation.

Table 2.1: Key SquadSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Simplified swap() Logic in SquadswapPair	Code Practices	Confirmed
PVE-002	Low	Implicit Assumption Enforcement In AddLiquidity()	Business Logic	Resolved
PVE-003	Low	Improved Validation of Function Arguments in MixedRouteQuoterV1	Code Practices	Resolved

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 Simplified swap() Logic in SquadswapPair

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SquadV3Pool
- Category: Coding Practices [3]
- CWE subcategory: CWE-1109 [1]

Description

The main revision of SquadSwap is the customized fee support in SquadSwap. In the process of examining the pool-specific fee support, we notice the core swap() routine may be enhanced for improved gas efficiency.

To elaborate, we show below the implementation of the related swap() routine. This is a critical routine that implements the main token-swap logic. Note the customized fee is added by piggy-backing the return values of getReserves() with an extra field named _fee. However, it comes to our attention that current implementation makes a second call to explicitly request for the pool-specific swap fee, instead of reusing the prior getReserves() call. By simply reusing the prior getReserves() call, we can avoid three inter-contract calls: one for factory::poolManager(), another for factory::defaultFee(), and the remaining for poolManager::getFee().

```
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data)
        external lock {
        require(amount0Out > 0  amount1Out > 0, 'Squadswap: INSUFFICIENT_OUTPUT_AMOUNT')
            ;
        (uint112 _reserve0, uint112 _reserve1, ,) = getReserves(); // gas savings
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'Squadswap:
            INSUFFICIENT_LIQUIDITY');

        uint balance0;
        uint balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
```

```
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'Squadswap: INVALID_TO');
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
        if (data.length > 0) ISquadswapCallee(to).squadswapCall(msg.sender, amount0Out, amount1Out, data);
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
        require(amount0In > 0  amount1In > 0, 'Squadswap: INSUFFICIENT_INPUT_AMOUNT');
        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
        uint fee = getFee();
        uint balance0Adjusted = balance0 * (FEE_DENOMINATOR) - (amount0In * fee);
        uint balance1Adjusted = balance1 * (FEE_DENOMINATOR) - (amount1In * fee);
        require(balance0Adjusted * (balance1Adjusted) >= uint(_reserve0) * (_reserve1) * (FEE_DENOMINATOR**2), 'Squadswap: K');
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }
    ...
    function getFee() public view returns (uint fee) {
        address poolManagerAddress = ISquadswapFactory(factory).poolManager();
        ISquadV3PoolManagerStates poolManager = ISquadV3PoolManagerStates(
            poolManagerAddress
        );

        uint defaultFee = ISquadswapFactory(factory).defaultFee();
        fee = uint(poolManager.getFee(address(this), uint24(defaultFee)));
    }
    ...
    function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTimestampLast, uint _fee) {
        _reserve0 = reserve0;
        _reserve1 = reserve1;
        _blockTimestampLast = blockTimestampLast;
        _fee = getFee();
    }
```

Listing 3.1: `SquadV3Pool::swap()/getFee()/getReserves()`

Recommendation Revisit the above `swap()` routine for improved gas efficiency by avoiding redundant inter-contract calls.

Status The issue has been confirmed and the team plans to revisit the issue after measuring possible gas gains.

3.2 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SquadswapRouter02
- Category: Coding Practices [3]
- CWE subcategory: CWE-628 [2]

Description

To facilitate the user interaction, SquadSwap has a built-in SquadswapRouter02 contract for users to add/remove pair liquidity pool and perform token swaps. While examining the pool-adding logic, we notice the addLiquidity() routine is provided to add amountADesired amount of tokenA and amountBDesired amount of tokenB. To elaborate, we show below the related code snippet.

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (ISquadswapFactory(factory).getPair(tokenA, tokenB) == address(0)) {
            ISquadswapFactory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB, ) = SquadswapLibrary.getReserves(factory, tokenA,
             tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = SquadswapLibrary.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'SquadswapRouter02:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = SquadswapLibrary.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'SquadswapRouter02:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
```

```
        }
    }
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual ensure(deadline) returns (uint amountA, uint amountB, uint
        liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = SquadswapLibrary.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = ISquadswapPair(pair).mint(to);
    }
```

Listing 3.2: `SquadswapRouter02::addLiquidity()`

It comes to our attention that the `SquadswapRouter02` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `SquadSwap V2 Router` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

Status This issue has been resolved as the team enforces these checks on the `UI` side. The desired value cannot be sent lower than the minimum values from the `UI`.

3.3 Improved Validation of Function Arguments in MixedRouteQuoterV1

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MixedRouteQuoterV1
- Category: Coding Practices [3]
- CWE subcategory: CWE-1109 [1]

Description

To facilitate the on-chain quotes for different SquadSwap variants, including SquadSwapV3, SquadSwapV2 , SquadSwapStable and MixedRoute, the protocol provides a helper MixedRouteQuoterV1 contract to return the token output amount when performing exact input swaps[1]. While reviewing the contract implementation, we notice the given user input can be better validated.

To elaborate, we show below the implementation of the related quoteExactInput() routine. As the name indicates, this routine is used to calculate the quote for an exactInput swap between an array of Stable, V2, and/or V3 pools. In particular, one function argument flag is used to indicate the pool type with 0 for V3, 1 for V2, 2 for 2pool, and 3 for 3pool. Meanwhile, it has another argument path to represent the swap path. With that, we need to validate the respect lengths should be the same, i.e., require(path.length == flag.length).

```
    function quoteExactInput(
        bytes memory path,
        uint256[] memory flag,
        uint256 amountIn
    )
        public
        override
        returns (
            uint256 amountOut,
            uint160[] memory v3SqrtPriceX96AfterList,
            uint32[] memory v3InitializedTicksCrossedList,
            uint256 v3SwapGasEstimate
        )
    {
        v3SqrtPriceX96AfterList = new uint160[](path.numPools());
        v3InitializedTicksCrossedList = new uint32[](path.numPools());
        ...
    }
```

Listing 3.3: MixedRouteQuoterV1::quoteExactInput()

[1]Note it allows getting the expected amount out for a given swap without executing the swap and it does not support exact output swaps since using the contract balance between exactOut swaps is not supported.

Recommendation Revisit the above routine to ensure the two parameters `path` and `flag` share the same length.

Status The issue has been resolved as the team enforces their consistency via the `UI`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the new version of `SquadSwap`, which brings a new level of flexibility to its fee structure. Specifically, there was a fixed fee rate. And the new version allows this fee rate to be adjusted. This changeable fee system makes it easier to adapt to market conditions and manage liquidity more effectively. By keeping the simplicity of the original system while adding this customization option, the revised protocol offers a smoother and more flexible trading experience. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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