SMART CONTRACT AUDIT REPORT

for

SquadSwap

Prepared By: Xiaomi Huang

PeckShield

February 15, 2024

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	February 15, 2024	Xuxian Jiang	Final Release
1.0-rc	January 29, 2024	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, powered by its native token $SQUAD, stands out as a decentralized exchange (DEX) backed by the PancakeSquad NFT community. It offers a user-friendly platform for seamless token swapping and ways to earn rewards, including liquidity provision, Farms, and Pools. A distinctive feature is the widget that allows communities to integrate a swap widget on their sites, leveraging SquadSwap's liquidity and smart contracts while earning fees from transactions processed through their widget. The use of $SQUAD as the driving force behind these operations fuels the ecosystem and enhances user engagement and platform growth. 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 15, 2024

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

- https://github.com/Bit5Tech/syrup.git (3f5d6a8)
- https://github.com/Bit5Tech/SquadSwap.git (e78743f)
- https://github.com/Bit5Tech/SquadSwap-v3.git (dad2277)
- https://github.com/Bit5Tech/SquadSwapClaimMerkle.git (e4df19f)
- https://github.com/Bit5Tech/SquadToken.git (a487042)
- https://github.com/Bit5Tech/mcv2.git (137ec38)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Bit5Tech/syrup.git (878af06)
- https://github.com/Bit5Tech/SquadSwap.git (e78743f)
- https://github.com/Bit5Tech/SquadSwap-v3.git (dad2277)
- https://github.com/Bit5Tech/SquadSwapClaimMerkle.git (e4df19f)
- https://github.com/Bit5Tech/SquadToken.git (a487042)
- https://github.com/Bit5Tech/mcv2.git (137ec38)

1.2 About PeckShield

PeckShield Inc. [12] 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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;

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

- 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.

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], 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 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	2	■■
Low	5	■■■■■
Informational	0	
Total	7	

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 medium-severity vulnerabilities and 5 low-severity vulnerabilities.

Table 2.1: Key SquadSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Liquidity Mining in SquadV3LmPool	Business Logic	Confirmed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Low	Incorrect SquadRate Initialization in MasterChefV2	Code Practices	Resolved
PVE-004	Low	Staking Incompatibility With Deflationary/Rebasing Tokens in SmartChefInitializable	Business Logic	Resolved
PVE-005	Low	Timely Pool Update Upon rewardPerBlock Change in SmartChefInitializable	Business Logic	Resolved
PVE-006	Low	Incorrect Pair Reserve Update Logic in SquadswapPair	Numeric Errors	Resolved
PVE-007	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Confirmed

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 Incorrect Liquidity Mining in SquadV3LmPool

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SquadV3LmPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `SquadSwap` protocol features a liquidity mining support with `UniswapV3 NFT`-like positions. In the process of examining the actual implementation, we notice a potential issue that may block a legitimate user from claiming the rewards.

To elaborate, we show below the implementation of the related `getRewardGrowthInside()` routine. As the name indicates, this routine is used to compute the reward growth data. However, it does not consider a possible underflow situation that may make the associated `MasterChef V3` smart contract unable to calculate reward for the positions whose initial `rewardGrowthInsideX128` values were negative underflow.

```
    function getRewardGrowthInside(
        mapping(int24 => LmTick.Info) storage self,
        int24 tickLower,
        int24 tickUpper,
        int24 tickCurrent,
        uint256 rewardGrowthGlobalX128
    ) internal view returns (uint256 rewardGrowthInsideX128) {
        Info storage lower = self[tickLower];
        Info storage upper = self[tickUpper];

        // calculate reward growth below
        uint256 rewardGrowthBelowX128;
        if (tickCurrent >= tickLower) {
            rewardGrowthBelowX128 = lower.rewardGrowthOutsideX128;
        } else {
```

```
            rewardGrowthBelowX128 = rewardGrowthGlobalX128 - lower.
                rewardGrowthOutsideX128;
        }

        // calculate reward growth above
        uint256 rewardGrowthAboveX128;
        if (tickCurrent < tickUpper) {
            rewardGrowthAboveX128 = upper.rewardGrowthOutsideX128;
        } else {
            rewardGrowthAboveX128 = rewardGrowthGlobalX128 - upper.
                rewardGrowthOutsideX128;
        }

        rewardGrowthInsideX128 = rewardGrowthGlobalX128 - rewardGrowthBelowX128 -
            rewardGrowthAboveX128;
    }
```

Listing 3.1: `LmTick::getRewardGrowthInside()`

Recommendation Revisit the above `getRewardGrowthInside()` routine to handle the possible underflow situation. Here comes a possible extension to check whether an underflow situation occurs:

```
    function _getRewardGrowthInsideInternal(
        int24 tickLower,
        int24 tickUpper
    ) internal view returns (uint256 rewardGrowthInsideX128, bool isNegative) {
        (, int24 tick, , , , , ) = pool.slot0();
        LmTick.Info memory lower = lmTicks[tickLower];

        LmTick.Info memory upper = lmTicks[tickUpper];

        // calculate reward growth below
        uint256 rewardGrowthBelowX128;
        if (tick >= tickLower) {
            rewardGrowthBelowX128 = lower.rewardGrowthOutsideX128;
        } else {
            rewardGrowthBelowX128 = rewardGrowthGlobalX128 - lower.
                rewardGrowthOutsideX128;
        }

        // calculate reward growth above
        uint256 rewardGrowthAboveX128;
        if (tick < tickUpper) {
            rewardGrowthAboveX128 = upper.rewardGrowthOutsideX128;
        } else {
            rewardGrowthAboveX128 = rewardGrowthGlobalX128 - upper.
                rewardGrowthOutsideX128;
        }

        rewardGrowthInsideX128 = rewardGrowthGlobalX128 - rewardGrowthBelowX128 -
            rewardGrowthAboveX128;
        isNegative = (rewardGrowthBelowX128 + rewardGrowthAboveX128) >
```

```
        rewardGrowthGlobalX128;
}
```

Listing 3.2: Revised `_getRewardGrowthInsideInternal()`

Status The issue has been confirmed.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `SquadSwap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters and managing the reward pools). In the following, we show the representative functions potentially affected by the privilege of the account.

```
function setEmergency(bool _emergency) external onlyOwner {
    emergency = _emergency;
    emit SetEmergency(emergency);
}

function setReceiver(address _receiver) external onlyOwner {
    if (_receiver == address(0)) revert ZeroAddress();
    if (SQUAD.allowance(_receiver, address(this)) != type(uint256).max) revert();
    receiver = _receiver;
    emit NewReceiver(_receiver);
}

function setLMPoolDeployer(ILMPoolDeployer _LMPoolDeployer) external onlyOwner {
    if (address(_LMPoolDeployer) == address(0)) revert ZeroAddress();
    LMPoolDeployer = _LMPoolDeployer;
    emit NewLMPoolDeployerAddress(address(_LMPoolDeployer));
}
```

Listing 3.3: Example Privileged Operations in `MasterChefV3`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been mitigated with the use of `multisig` to manage the admin key.

3.3 Incorrect SquadRate Initialization in MasterChefV2

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChefV2`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

Description

The `SquadSwap` swap has its `MasterChefV2` contract to incentivize protocol users. This `MasterChefV2` contract is the only address with minting rights for `SQUAD` and is used to issue a constant number of `SQUAD` tokens per block. Also, there are two types of pools in `MasterChefV2`, i.e., `regular` and `special`. The issuance rates for these two pools are defined as `squadRateToRegularFarm` and `squadRateToSpecialFarm` respectively.

To elaborate, we show below the key parameters defined in `MasterChefV2`. We notice that the sum of `squadRateToRegularFarm` and `squadRateToSpecialFarm` is not equal to `SQUAD_RATE_TOTAL_PRECISION` . With that, there is a need to adjust these three parameters to ensure the following invariant: `SQUAD_RATE_TOTAL_PRECISION = squadRateToRegularFarm + squadRateToSpecialFarm`.

```
    /// @notice Basic boost factor, none boosted user's boost factor
    uint256 public constant BOOST_PRECISION = 100 * 1e10;
    /// @notice Hard limit for maximum boost factor, it must greater than
        BOOST_PRECISION
    uint256 public constant MAX_BOOST_PRECISION = 200 * 1e10;
    /// @notice total squad rate = toRegular + toSpecial
    uint256 public constant SQUAD_RATE_TOTAL_PRECISION = 1e12;
    /// @notice SQUAD distribute percentage for regular farm pool
    uint256 public squadRateToRegularFarm = 10 * 1e10;
    /// @notice SQUAD distribute percentage for special pools
    uint256 public squadRateToSpecialFarm = 15 * 1e10;
```

Listing 3.4: The Key Parameters in `MasterChefV2`

Recommendation Revisit the above rates so that the token issuance invariant is maintained.

Status The issue has been resolved as these two parameters have been correctly updated via `updateSquadRate()`.

3.4 Staking Incompatibility With Deflationary/Rebasing Tokens in SmartChefInitializable

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmartChefInitializable`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `SquadSwap` protocol, there is a `SmartChefInitializable` contract that allows users to stake tokens to receive or farm rewards. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and records the staked amount in the farming pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint256 _amount) external nonReentrant {
        UserInfo storage user = userInfo[msg.sender];

        if (squadProfile != address(0)) {
            // Checks whether the user has an active profile
            require(
                (!squadProfileIsRequested && squadProfileThresholdPoints == 0)
                ISquadProfile(squadProfile).getUserStatus(msg.sender),
                "Deposit: Must have an active profile"
            );

            uint256 numberUserPoints = 0;

            if (squadProfileThresholdPoints > 0) {
                require(squadProfile != address(0), "Deposit: SquadProfile is not exist"
                    );
                (, numberUserPoints, , , , ) = ISquadProfile(squadProfile).
                    getUserProfile(msg.sender);
            }

            require(
                squadProfileThresholdPoints == 0  numberUserPoints >=
                    squadProfileThresholdPoints,
                "Deposit: User is not get enough user points"
```

```
            );
        }

        userLimit = hasUserLimit();

        require(!userLimit  ((_amount + user.amount) <= poolLimitPerUser), "Deposit:
            Amount above limit");

        _updatePool();

        if (user.amount > 0) {
            uint256 pending = (user.amount * accTokenPerShare) / PRECISION_FACTOR - user
                .rewardDebt;
            if (pending > 0) {
                rewardToken.safeTransfer(address(msg.sender), pending);
            }
        }

        if (_amount > 0) {
            user.amount = user.amount + _amount;
            stakedToken.safeTransferFrom(address(msg.sender), address(this), _amount);
        }

        user.rewardDebt = (user.amount * accTokenPerShare) / PRECISION_FACTOR;

        emit Deposit(msg.sender, _amount);
    }
```

Listing 3.5: `SmartChefInitializable::deposit()`

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 these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, 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 expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` 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 `SquadSwap`. However, as a `DEX` protocol, it may not be possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been resolved as there is no need to support deflationary/rebasing tokens.

3.5 Timely Pool Update Upon rewardPerBlock Change in SmartChefInitializable

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SmartChefInitializable
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the SmartChefInitializable contract provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of staked tokens in the reward pool.

When analyzing the reward rate update routine updateRewardPerBlock(), we notice the need of timely invoking _updatePool() to update the reward distribution before the new reward rate becomes effective.

```
    /*
     * @notice Update reward per block
     * @dev Only callable by owner.
     * @param _rewardPerBlock: the reward per block
     */
    function updateRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {
        require(block.number < startBlock, "Pool has started");
        rewardPerBlock = _rewardPerBlock;
        emit NewRewardPerBlock(_rewardPerBlock);
    }
```

Listing 3.6: SmartChefInitializable ::updateRewardPerBlock()

If the call to _updatePool() is not immediately invoked before updating the reward rate, it is possible that the new reward rate is applied earlier than expected. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Recommendation Timely invoke `_updatePool()` when the reward rate is being updated. An example revision is shown as below:

```
    /*
     * @notice Update reward per block
     * @dev Only callable by owner.
     * @param _rewardPerBlock: the reward per block
     */
    function updateRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {
        require(block.number < startBlock, "Pool has started");
        _updatePool();
        rewardPerBlock = _rewardPerBlock;
        emit NewRewardPerBlock(_rewardPerBlock);
    }
```

Listing 3.7: Revised SmartChefInitializable ::updateRewardPerBlock()

Status This issue has been resolved as the team confirms to refresh the reward rate before applying the new reward per block.

3.6 Incorrect Pair Reserve Update Logic in SquadswapPair

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `SquadswapPair`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The `SquadSwap` protocol is in essence a decentralized exchange (`DEX`). While reviewing the `UniswapV2`-based `Squadswap` engine, we notice a possible underflow issue that may break the basic `DEX` functionality.

To elaborate, we show below the related `_update()` routine. As the name indicates, this routine is designed to update the pool reserves with actual token balances. Within this routine, it maintains the so-called `TWAP` price oracle for external queries. However, we notice this contract has the following pragma, i.e., `pragma solidity ^0.8.0`, which indicates the built-in support of arithmetic overflow/underflow validation in regular arithmetic operation. In other words, the local variable `timeElapsed = blockTimestamp - blockTimestampLast` (line 75) may be reverted if there is an arithmetic underflow. If such underflow occurs, any call to `_update()` will be simply reverted. And this `_update()` routine may be called in a number of places, including `mint()`, `burn()`, `swap()`, and `sync()`. As a result, it may potentially affected all these functions.

```
71      // update reserves and, on the first call per block, price accumulators
```

```
    function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private {
        require(balance0 <= type(uint112).max && balance1 <= type(uint112).max, 'Squadswap: OVERFLOW');
        uint32 blockTimestamp = uint32(block.timestamp % 2**32);
        uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
            price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
        }
        reserve0 = uint112(balance0);
        reserve1 = uint112(balance1);
        blockTimestampLast = blockTimestamp;
        emit Sync(reserve0, reserve1);
    }
```

Listing 3.8: `SquadswapPair::_update()`

Recommendation Properly revise the above routine to ensure the arithmetic underflow will not break the swap functionality.

Status The issue has been resolved. Note the state of `blockTimestamp` has 32 bits and `2**32 = 4,294,967,292 seconds`, which means about `136.12 years`. With that, current `block.timestamp = Date.now()/ 1000 = 1,706,685,140`. So the valid time period is `2,588,282,152 seconds (4,294,967,292 - 1,706,685,140)`, which is converted to `82 years`. After all these pair contracts will be valid for `82 years`, the overflow error won't happen during this period. Moreover, every time the `_update()` function is called, this counter will be reset.

3.7 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SquadswapRouter02
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [4]

Description

In the SquadswapRouter02 contract, the addLiquidity() routine (see the code snippet below) is provided to add amountADesired amount of tokenA and amountBDesired amount of tokenB into the pool as liquidity via the SquadswapRouter02::addLiquidity() routine. 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 3.9: `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 The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SquadSwap` protocol, which is a decentralized exchange (`DEX`) backed by the `PancakeSquad NFT` community. It offers a user-friendly platform for seamless token swapping and ways to earn rewards, including liquidity provision, `Farms`, and `Pools`. A distinctive feature is the widget that allows communities to integrate a swap widget on their sites, leveraging `SquadSwap`'s liquidity and smart contracts while earning fees from transactions processed through their widget. The use of `$SQUAD` as the driving force behind these operations fuels the ecosystem and enhances user engagement and platform growth. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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