SMART CONTRACT AUDIT REPORT

for

PulsarSwap

Prepared By: Patrick Lou

PeckShield

April 21, 2022

Document Properties

Client	Pulsar
Title	Smart Contract Audit Report
Target	PulsarSwap
Version	1.0
Author	Luck Hu, Xiaotao Wu
Auditors	Luck Hu, Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 21, 2022	Luck Hu, Xiaotao Wu	Final Release
1.0-rc	April 1, 2022	Xiaotao Wu	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
1.1 About PulsarSwap . . . 4
1.2 About PeckShield . . . 5
1.3 Methodology . . . 5
1.4 Disclaimer . . . 7

2 Findings 9
2.1 Summary . . . 9
2.2 Key Findings . . . 10

3 Detailed Results 11
3.1 Missing Access Control in Pair . . . 11
3.2 Accommodation Of Non-ERC20-Compliant Tokens . . . 12
3.3 Improved WETH Token Handling in TWAMM . . . 15
3.4 Out-of-Gas Risk In executeVirtualOrdersUntilCurrentBlock() . . . 17
3.5 Lack Of Calling updatePrice() In Pair::provideInitialLiquidity() . . . 18
3.6 Incorrect orderIdStatusMap Update Logic In withdrawProceedsFromLongTermSwap() 20

4 Conclusion 22

References 23

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the PulsarSwap protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About PulsarSwap

PulsarSwap is the implementation of Time-Weighted Average Market Maker (TWAMM) that effectively combines embedded AMM, LongTerm Orders, Order Pool, and scalable reward distribution to enable not only Uniswap-like DEXs, but also other AMMs with algorithmic trading TWAP. Compared to AMM , TWAMM reduces the price slippage associated with large trades, thus reducing trader losses. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of PulsarSwap

Item	Description
Name	Pulsar
Website	https://pulsarswap.com/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 21, 2022

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

- https://github.com/PulsarSwap/TWAMM-Contracts.git (27b5b3b)

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

- https://github.com/PulsarSwap/TWAMM-Contracts.git (8c7d701)

1.2 About PeckShield

PeckShield Inc. [11] 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 the OWASP Risk Rating Methodology [10]:

- 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 checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 `PulsarSwap` smart contracts. 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	1	■
Medium	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	6	

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 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 high-severity vulnerability, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key PulsarSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Missing Access Control in Pair	Security Features	Resolved
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Medium	Improved WETH Token Handling in TWAMM	Business Logic	Resolved
PVE-004	Medium	Out-of-Gas Risk In executeVirtualOrdersUntilCurrentBlock()	Time and State	Confirmed
PVE-005	Medium	Lack Of Calling updatePrice() In Pair::provideInitialLiquidity()	Business Logic	Resolved
PVE-006	Low	Incorrect orderIdStatusMap Update Logic In withdrawProceedsFromLongTermSwap()	Business Logic	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 Missing Access Control in Pair

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Pair`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `PulsarSwap` protocol, there is a `Pair` contract which implements the actual pool for any two `ERC20` tokens. In the `Pair` contract, it provides a series of interfaces for `LPs` to add/remove liquidity, and for traders to exchange tokens. While examining these interfaces, we notice the existence of missed access control authorization that need to be corrected.

To elaborate, we show below the code snippet of the `removeLiquidity()` routine. As the name indicates, this routine is designed to remove liquidity from current pool for the given `LP` (identified by the `to` argument). It comes to our attention that there is no access control restriction enforced on this routine, which makes the `removeLiquidity()` routine opened to the public. As a result, anyone could invoke it to remove liquidity from the pool on behalf of any `LP`.

```
    function removeLiquidity(address to, uint256 lpTokenAmount)
        external
        override
        lock
        nonReentrant
    {
        require(
            lpTokenAmount <= totalSupply(),
            "Not Enough Lp Tokens Available"
        );
        updatePrice(reserveMap[tokenA], reserveMap[tokenB]);

        //execute virtual orders
```

```
        longTermOrders.executeVirtualOrdersUntilCurrentBlock(reserveMap);

        //the ratio between the number of underlying tokens and the number of lp tokens must remain invariant after burn
        uint256 amountAOut = (reserveMap[tokenA] * lpTokenAmount) /
            totalSupply();
        uint256 amountBOut = (reserveMap[tokenB] * lpTokenAmount) /
            totalSupply();

        reserveMap[tokenA] -= amountAOut;
        reserveMap[tokenB] -= amountBOut;

        _burn(to, lpTokenAmount);

        IERC20(tokenA).transfer(to, amountAOut);
        IERC20(tokenB).transfer(to, amountBOut);

        emit LiquidityRemoved(to, lpTokenAmount);
    }
```

Listing 3.1: Pair::removeLiquidity()

Our further study shows that the access control authorization could be granted to the `TWAMM` contract which is a dedicated router for the `Pair` contract. Note there are some other routines share the same issue in the `Pair` contract. Such as the `provideInitialLiquidity()/provideLiquidity()/instantSwapFromAToB()/longTermSwapFromAToB()` routines, etc.

Recommendation Add the necessary access control authorization to the above mentioned routines in the `Pair` contract.

Status This issue has been fixed in this commit: `8c7d701`.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pair`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transferFrom()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transferFrom()`, there is a check, i.e., `if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens from address _from to address _to, and MUST fire the Transfer event. The function SHOULD throw unless the _from account has deliberately authorized the sender of the message via some mechanism."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: `ZRX.sol`

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

In the following, we show the `Pair::provideLiquidity()` routine. If the `ZRX` token is supported as either of the tokenA/tokenB in the pool, the unsafe version of `IERC20(tokenA).transferFrom(to, address(this), amountAIn);` (line 158 and 159) may return false if the pool (spender) does not have enough allowance from the token owner (given by the `to` argument). But the `Pair::provideLiquidity()` routine expects the `transferFrom()` to revert on failure. Based on this, we may intend to replace the `transferFrom()` (line 158 and 159) with `safeTransferFrom()`.

```
132     function provideLiquidity(address to, uint256 lpTokenAmount)
133         external
134         override
```

```
        lock
        nonReentrant
    {
        require(
            totalSupply() != 0,
            "No Liquidity Has Been Provided Yet, Need To Call provideInitialLiquidity()"
        );
        updatePrice(reserveMap[tokenA], reserveMap[tokenB]);

        //execute virtual orders
        longTermOrders.executeVirtualOrdersUntilCurrentBlock(reserveMap);

        //the ratio between the number of underlying tokens and the number of lp tokens must remain invariant after mint
        uint256 amountAIn = (lpTokenAmount * reserveMap[tokenA]) /
            totalSupply();
        uint256 amountBIn = (lpTokenAmount * reserveMap[tokenB]) /
            totalSupply();

        reserveMap[tokenA] += amountAIn;
        reserveMap[tokenB] += amountBIn;

        _mint(to, lpTokenAmount);

        IERC20(tokenA).transferFrom(to, address(this), amountAIn);
        IERC20(tokenB).transferFrom(to, address(this), amountBIn);

        emit LiquidityProvided(to, lpTokenAmount);
    }
```

Listing 3.3: `Pair::provideLiquidity()`

Note the same issue also exists in other routines, such as the `provideInitialLiquidity()/removeLiquidity()/performInstantSwap()` routines in the `Pair` contract, and the `performLongTermSwap()/cancelLongTermSwap()/withdrawProceedsFromLongTermSwap()` routines in the `LongTermOrders` library.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()` and `transferFrom()`.

Status This issue has been fixed in this commit: `8c7d701`.

3.3 Improved WETH Token Handling in TWAMM

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: TWAMM
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the Pair contract supports users to create pair for any two ERC20 tokens, including the WETH token which is the ERC20 wrapper for the native ETH token. To make it convenient for users to exchange with the native ETH directly, the TWAMM contract is designed as a router to take the native ETH as the input/output token. While examining these ETH related routines in the TWAMM contract, we notice the existence of improper ETH/WETH handling that need to be corrected.

To elaborate, we show below the code snippet of the TWAMM::addInitialLiquidityETH() routine. As the name indicates, this routine is designed to add initial liquidity to ETH related pool. It comes to our attention that the routine receives ETH from the LP and deposits the ETH to the WETH contract (line 101). However, the new WETH tokens are minted to the TWAMM contract, while not the LP. Moreover, after the liquidity is added, the TWAMM contract will refund the remaining ETH back to the LP (line 105). But as mentioned earlier, all the received ETH has been deposited to the WETH contract (line 101).

```
86     function addInitialLiquidityETH(
87         address token,
88         uint256 amountToken,
89         uint256 amountETH,
90         uint256 deadline
91     ) external payable virtual override ensure(deadline) {
92         require(
93             IFactory(factory).getPair(token, WETH) != address(0),
94             "No Existing Pair Found, Create Pair First!"
95         );
96         address pair = Library.pairFor(factory, token, WETH);
97         (address tokenA, ) = Library.sortTokens(token, WETH);
98         (uint256 amountA, uint256 amountB) = tokenA == token
99             ? (amountToken, amountETH)
100            : (amountETH, amountToken);
101        IWETH10(WETH).deposit{value: msg.value}();
102        IPair(pair).provideInitialLiquidity(msg.sender, amountA, amountB);
103        // refund dust eth, if any
104        if (msg.value > amountETH) {
105            TransferHelper.safeTransferETH(msg.sender, msg.value - amountETH);
106        }
107    }
```

Listing 3.4: TWAMM::addInitialLiquidityETH()

What is more, when we further look into the code of the `Pair::provideInitialLiquidity()` routine (as shown below), we notice that the input tokens for liquidity providing are directly transferred from the `LP` (given by the `to` argument), while not the `msg.sender` (`TWAMM` in our example). That is to say, the `LP` shall keep both the input tokens in its balance before adding the liquidity. So in the `TWAMM::addInitialLiquidityETH()` routine, the `WETH` shall be minted to the `LP`, not the `TWAMM`.

```
    function provideInitialLiquidity(
        address to,
        uint256 amountA,
        uint256 amountB
    ) external override lock nonReentrant {
        require(
            totalSupply() == 0,
            "Liquidity Has Already Been Provided, Need To Call provideLiquidity()");

        reserveMap[tokenA] = amountA;
        reserveMap[tokenB] = amountB;

        //initial LP amount is the geometric mean of supplied tokens
        uint256 lpAmount = amountA
            .fromUint()
            .sqrt()
            .mul(amountB.fromUint().sqrt())
            .toUint(); // - MINIMUM_LIQUIDITY;
        // _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens // TODO: uncomment
        _mint(to, lpAmount);
        IERC20(tokenA).transferFrom(to, address(this), amountA);
        IERC20(tokenB).transferFrom(to, address(this), amountB);

        emit InitialLiquidityProvided(to, amountA, amountB);
    }
```

Listing 3.5: Pair::provideInitialLiquidity()

Note similar issues exist in all other ETH related routines in the `TWAMM` contract, such as the `addLiquidityETH()/withdrawLiquidityETH()/instantSwapTokenToETH()/instantSwapETHToToken()` routines, etc.

Recommendation Revise all the `ETH` related routines to properly route the `ETH` and the `WETH` between the `LP` and the pool.

Status This issue has been fixed in this commit: `8c7d701`.

3.4 Out-of-Gas Risk In executeVirtualOrdersUntilCurrentBlock()

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: LongTermOrdersLib
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

In the PulsarSwap protocol, the executeVirtualOrdersUntilCurrentBlock() function will be triggered when a user provides liquidity, removes liquidity, performs instant swap, performs long term swap, cancels long term swap, or withdraws proceeds from long term swap. This executeVirtualOrdersUntilCurrentBlock() function will execute all pending virtual orders until current block is reached. However, if the operations that can trigger the execution of the executeVirtualOrdersUntilCurrentBlock() function does not occur for a long time, then the number of pending virtual orders could be large enough such that the subsequent execution of executeVirtualOrdersUntilCurrentBlock() could lead to out-of-gas (lines 284-291).

```
275     ///@notice executes all virtual orders until current block is reached.
276     function executeVirtualOrdersUntilCurrentBlock(
277         LongTermOrders storage self,
278         mapping(address => uint256) storage reserveMap
279     ) internal {
280         uint256 nextExpiryBlock = self.lastVirtualOrderBlock -
281             (self.lastVirtualOrderBlock % self.orderBlockInterval) +
282             self.orderBlockInterval;
283         //iterate through blocks eligible for order expires, moving state forward
284         while (nextExpiryBlock < block.number) {
285             executeVirtualTradesAndOrderExpiries(
286                 self,
287                 reserveMap,
288                 nextExpiryBlock
289             );
290             nextExpiryBlock += self.orderBlockInterval;
291         }
292         //finally, move state to current block if necessary
293         if (self.lastVirtualOrderBlock != block.number) {
294             executeVirtualTradesAndOrderExpiries(
295                 self,
296                 reserveMap,
297                 block.number
298             );
299         }
```

```
    }
```

Listing 3.6: `LongTermOrdersLib::executeVirtualOrdersUntilCurrentBlock()`

Recommendation Take into consideration the scenario where there may exist a large number of virtual orders waiting to be executed.

Status This issue has been confirmed.

3.5 Lack Of Calling updatePrice() In Pair::provideInitialLiquidity()

- ID: PVE-005
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Pair`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `PulsarSwap` protocol, the `Pair` contract provides an external `provideInitialLiquidity()` for users to provide initial liquidity. This function can be called by a user only when the `totalSupply` of the `Pair` contract is equal to 0. When analyzing this initial liquidity-providing routine `provideInitialLiquidity()`, we notice there is a lack of invoking `updatePrice()` to update the price accumulators before transferring assets to the contracts.

```
    ///@notice provide initial liquidity to the amm. This sets the relative price
        between tokens
    function provideInitialLiquidity(
        address to,
        uint256 amountA,
        uint256 amountB
    ) external override lock nonReentrant {
        require(
            totalSupply() == 0,
            "Liquidity Has Already Been Provided, Need To Call provideLiquidity()"
        );

        reserveMap[tokenA] = amountA;
        reserveMap[tokenB] = amountB;

        //initial LP amount is the geometric mean of supplied tokens
        uint256 lpAmount = amountA
            .fromUint()
            .sqrt()
```

```
            .mul(amountB.fromUint().sqrt())
            .toUint(); // - MINIMUM_LIQUIDITY;
        // _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens // TODO: uncomment
        _mint(to, lpAmount);
        IERC20(tokenA).transferFrom(to, address(this), amountA);
        IERC20(tokenB).transferFrom(to, address(this), amountB);

        emit InitialLiquidityProvided(to, amountA, amountB);
    }
```

Listing 3.7: `Pair::provideInitialLiquidity()`

If the call to `updatePrice()` is not invoked in the `provideInitialLiquidity()` routine, the value of the state variable `blockTimestampLast` will remain `0`. The calculation of `timeElapsed` in the subsequent call of `updatePrice()` will be not correct (line 85), thus the calculations for state variables `priceACumulativeLast/priceBCumulativeLast` will also be not correct.

```
82      // update price accumulators, on the first call per block
83      function updatePrice(uint256 reserveA, uint256 reserveB) private {
84          uint32 blockTimestamp = uint32(block.timestamp % 2**32);
85          uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
86          if (timeElapsed > 0 && reserveA != 0 && reserveB != 0) {
87              // * never overflows, and + overflow is desired
88              priceACumulativeLast +=
89                  uint256(
90                      UQ112x112.encode(uint112(reserveB)).uqdiv(uint112(reserveA))
91                  ) *
92                  timeElapsed;
93              priceBCumulativeLast +=
94                  uint256(
95                      UQ112x112.encode(uint112(reserveA)).uqdiv(uint112(reserveB))
96                  ) *
97                  timeElapsed;
98          }
99          blockTimestampLast = blockTimestamp;
100         emit UpdatePrice(reserveA, reserveB);
101     }
```

Listing 3.8: `Pair::updatePrice()`

Note similar issue also exists in the `executeVirtualOrders()` routine of the same contract.

Recommendation Timely invoke `updatePrice()` for the above-mentioned functions.

Status This issue has been fixed in this commit: `8c7d701`.

3.6 Incorrect orderIdStatusMap Update Logic In withdrawProceedsFromLongTermSwap()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LongTermOrdersLib
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the PulsarSwap protocol, a user can withdraw proceeds from his/her long term swap order and this can be done before or after the order has expired. If the order has expired, the total proceeds of this order will be sent to the user. If the order has expired, the total proceeds collected for this order will be sent to the user. If the order has not yet expired, the proceeds accumulated so far for this order will be sent to the user. When analyzing this withdrawProceedsFromLongTermSwap() routine, we notice the current update logic for mapping type orderIdStatusMap is not correct.

To elaborate, we show below its code snippet. Specifically, an order status should be updated to false only when this order has expired, instead of updating the order status to false regardless of whether the order has expired or not (line 216).

```
189     ///@notice withdraw proceeds from a long term swap (can be expired or ongoing)
190     function withdrawProceedsFromLongTermSwap(
191         LongTermOrders storage self,
192         address sender,
193         uint256 orderId,
194         mapping(address => uint256) storage reserveMap
195     ) internal {
196         //update virtual order state
197         executeVirtualOrdersUntilCurrentBlock(self, reserveMap);
198 
199         Order storage order = self.orderMap[orderId];
200         require(order.owner == sender, "Sender Must Be Order Owner");
201 
202         OrderPoolLib.OrderPool storage OrderPool = self.OrderPoolMap[
203             order.sellTokenId
204         ];
205         uint256 proceeds = OrderPool.withdrawProceeds(orderId);
206 
207         //charge LP fee
208         uint256 proceedsMinusFee = (proceeds * (10000 - LP_FEE)) / 10000;
209 
210         require(proceedsMinusFee > 0, "No Proceeds To Withdraw");
211         //transfer to owner
212         IERC20(order.buyTokenId).transfer(sender, proceedsMinusFee);
213 
```

```
        // delete orderId from account list
        // removeOrderId(self, orderId, msg.sender);
        self.orderIdStatusMap[orderId] = false;
    }
```

Listing 3.9: `LongTermOrdersLib::withdrawProceedsFromLongTermSwap()`

Recommendation Update the order status to false only when this order has expired.

Status This issue has been fixed in this commit: `8c7d701`.

4 Conclusion

In this audit, we have analyzed the PulsarSwap design and implementation. PulsarSwap is the implementation of Time-Weighted Average Market Maker (TWAMM) that effectively combines embedded AMM, LongTerm Orders, Order Pool, and scalable reward distribution to enable not only Uniswap-like DEXs, but also other AMMs with algorithmic trading TWAP. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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