SMART CONTRACT AUDIT REPORT

for

Easyswap

Prepared By: Xiaomi Huang

PeckShield

October 15, 2023

Document Properties

Client	Easyswap
Title	Smart Contract Audit Report
Target	Easyswap
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Jinzhuo Shen, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 15, 2023	Xuxian Jiang	Final Release
1.0-rc	September 26, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the Easyswap 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 Easyswap

Easyswap is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from Solidly/Velodrome with a unique AMM. The DEX is compatible with all the standard features as popularized by UniswapV2 with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps, as well as a built-in NFT-based voting mechanism and associated token emissions. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Easyswap

Item	Description
Name	Easyswap
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 15, 2023

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

- https://github.com/ScrollSwapfi/ScrollSwap.git (6f7813f)

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

- https://github.com/ScrollSwapfi/ScrollSwap.git (8232821)

1.2 About PeckShield

PeckShield Inc. [13] 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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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 accordingly classified into four categories, 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

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

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) [11], 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 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 design and implementation of the `Easyswap` protocol 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	3	■■■
Low	8	■■■■■■■■
Informational	0	
Total	13	

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 2 high-severity vulnerabilities, 3 medium-severity vulnerabilities, and 8 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Pair Initialization With Minimal Liquidity Enforcement	Time And State	Resolved
PVE-002	Low	Possible Sandwich/MEV For Reduced Returns	Time And State	Resolved
PVE-003	Low	Revisited Proposal State For Cancellation in Governor	Business Logic	Resolved
PVE-004	High	Incorrect Delegate/Voting Balance Accounting in VotingEscrow	Business Logic	Resolved
PVE-005	Medium	Voting Delegate Denial-of-Service With Dust Delegates	Business Logic	Resolved
PVE-006	Low	Possible Rebase Reward Lockup For Expired NFTs	Business Logic	Resolved
PVE-007	Low	Inconsistent K Invariants Between Pair And Router	Business Logic	Resolved
PVE-008	Low	Improved Validation on Protocol Parameters	Coding Practices	Resolved
PVE-009	Low	Revisited withdraw() Logic in IDOSale	Business Logic	Resolved
PVE-010	High	Potentially Out-of-sync rewardRate in Gauge	Business Logic	Confirmed
PVE-011	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-012	Low	Improper VotingEscrow Query in RewardsDistributor	Coding Practices	Resolved
PVE-013	Low	Timely Reward Resume Upon Reviving Gauges	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 Improved Pair Initialization With Minimal Liquidity Enforcement

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Pair`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [3]

Description

The `Easyswap` protocol has the built-in curve ($x^3y + xy^3 = k$) for efficient stable swaps. While examining the k calculation, we notice a possible denial-of-service issue that may allow the first stable swap LP to brick the (new) pair.

In the following, we show the implementation of the related `_k()` routine, which basically computes the k value from the above curve. However, it comes to our attention that the internal variable `_a` (line 445) may yield 0, which effectively computes the return value to be 0 and cascadingly meets the k invariant maintained in the `swap()` routine. As a result, the first LP of a stable swap pair may abuse it to initialize the pair with $k = 0$ and then empty the pool by resetting `reserve0` and/or `reserve1` to be 0. With that, any later LP may find infeasible to add new liquidity to the pair, hence effectively bricking the pair.

```
441     function _k(uint x, uint y) internal view returns (uint) {
442         if (stable) {
443             uint _x = x * 1e18 / decimals0;
444             uint _y = y * 1e18 / decimals1;
445             uint _a = (_x * _y) / 1e18;
446             uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
447             return _a * _b / 1e18;  // x3y+y3x >= k
448         } else {
449             return x * y; // xy >= k
450         }
```

```
    }
```

Listing 3.1: `Pair::_k()`

Recommendation To fix the above issue, there is a need to ensure the initial k upon the stable swap pair initialization will be larger than `MINIMUM_LIQUIDITY*MINIMUM_LIQUIDITY`. An example revision is shown as below:

```
    function mint(address to) external lock returns (uint liquidity) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        uint _balance0 = IERC20(token0).balanceOf(address(this));
        uint _balance1 = IERC20(token1).balanceOf(address(this));
        uint _amount0 = _balance0 - _reserve0;
        uint _amount1 = _balance1 - _reserve1;

        uint _totalSupply = totalSupply; // gas savings, must be defined here since
            totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(_k(_amount0, _amount1)) - MINIMUM_LIQUIDITY;
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
                MINIMUM_LIQUIDITY tokens
        } else {
            liquidity = Math.min(_amount0 * _totalSupply / _reserve0, _amount1 *
                _totalSupply / _reserve1);
        }
        require(liquidity > 0, 'ILM'); // Pair: INSUFFICIENT_LIQUIDITY_MINTED
        _mint(to, liquidity);

        _update(_balance0, _balance1, _reserve0, _reserve1);
        emit Mint(msg.sender, _amount0, _amount1);
    }
```

Listing 3.2: Revised `Pair::mint()`

Status This issue has been fixed in the following commit: `69598e2`.

3.2 Possible Sandwich/MEV For Reduced Return

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SingleLiquidityProvider`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [4]

Description

To facilitate the liquidity provision, the `Easyswap` protocol has a `SingleLiquidityProvider` contract to allow users to add single-sided liquidity. With that, there is a need to swap from one token to another. And we notice the token swap needs to be aware of possible MEV risks.

```
    function _slpIn(
        address _tokenToSlp,
        uint256 _tokenAmountIn,
        address _lpToken,
        uint256 _tokenAmountOutMin
    ) internal returns (uint256 lpTokenReceived) {
        if(_tokenToSlp != WETHAddress){
        require(_tokenAmountIn >= MINIMUM_AMOUNT, "Slp: Amount too low");
        }

        address token0 = IPair(_lpToken).token0();
        address token1 = IPair(_lpToken).token1();

        require(_tokenToSlp == token0   _tokenToSlp == token1, "Slp: Wrong tokens");

        // Retrieve the path
        IRouter.route[] memory routerRoutes = new IRouter.route[](1);


        routerRoutes[0].from = _tokenToSlp;

        // Initiates an estimation to swap
        uint256 swapAmountIn;

        {
            // Convert to uint256 (from uint112)
            (uint256 reserveA, uint256 reserveB, ) = IPair(_lpToken).getReserves();

            require((reserveA >= MINIMUM_AMOUNT) && (reserveB >= MINIMUM_AMOUNT), "Slp:
                Reserves too low");

            if (token0 == _tokenToSlp) {
                swapAmountIn = _calculateAmountToSwap(_lpToken, _tokenAmountIn);
                routerRoutes[0].to = token1;
                if(_tokenToSlp != WETHAddress){
                require(reserveA / swapAmountIn >= maxSlpReverseRatio, "Slp: Quantity
                    higher than limit");
                }
            } else {
                swapAmountIn = _calculateAmountToSwap(_lpToken, _tokenAmountIn);
                routerRoutes[0].to = token0;
                if(_tokenToSlp != WETHAddress){
                require(reserveB / swapAmountIn >= maxSlpReverseRatio, "Slp: Quantity
                    higher than limit");
                }
            }
        }
        ...
    }
```

Listing 3.3: SingleLiquidityProvider :: _slpIn()

To elaborate, we show above the related `_slpIn()` routine. We notice the conversion is routed to `scrollerRouter` for token swaps. Moreover, the applied slippage control is based on the instantaneous `reserve0` (line 371) and `reserve1` (line 377), which is therefore vulnerable to possible front-running attacks. In other words, these two reserves may be inflated right before the token swap, resulting in possible loss in this round of liquidity addition.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of LP users.

Status This issue has been fixed in the following commit: `69598e2`.

3.3 Revisited Proposal State For Cancellation in Governor

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Governor`, `L2Governor`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Easyswap` protocol has a built-in governance to facilitate the protocol operation and management. In particular, each protocol has its own lifecycle and its associated protocol state. While reviewing the possible protocol states, we notice the current protocol cancellation operation makes use of an incorrect protocol state.

To elaborate, we show below the related code snippet `_cancel()`, which validates the current state not in `Canceled`, `Expired`, and `Executed`. Our analysis shows that the state of `Expired` here should be replaced with `Defeated` – as the current `state()` routine never returns the `Expired` state.

```
    function _cancel(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) internal virtual returns (uint256) {
        uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);
        ProposalState status = state(proposalId);

        require(
            status != ProposalState.Canceled && status != ProposalState.Expired &&
                status != ProposalState.Executed,
            "Governor: proposal not active"
        );
        _proposals[proposalId].canceled = true;
```

```
        emit ProposalCanceled(proposalId);

        return proposalId;
    }
```

Listing 3.4: `L2Governor::_cancel()`

Moreover, another related routine `execute()` is invoked to execute the protocol actions. We notice one specific validation – `require(status == ProposalState.Succeeded || status == ProposalState.Queued)` (line 301), which potentially checks the `Queued` state. However, the `state()` routine never returns the `Queued` state.

```
291     function execute(
292         address[] memory targets,
293         uint256[] memory values,
294         bytes[] memory calldatas,
295         bytes32 descriptionHash
296     ) public payable virtual override returns (uint256) {
297         uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);

299         ProposalState status = state(proposalId);
300         require(
301             status == ProposalState.Succeeded  status == ProposalState.Queued,
302             "Governor: proposal not successful"
303         );
304         _proposals[proposalId].executed = true;

306         emit ProposalExecuted(proposalId);

308         _beforeExecute(proposalId, targets, values, calldatas, descriptionHash);
309         _execute(proposalId, targets, values, calldatas, descriptionHash);
310         _afterExecute(proposalId, targets, values, calldatas, descriptionHash);

312         return proposalId;
313     }
```

Listing 3.5: `L2Governor::execute()`

Recommendation Revise the above two routines to properly examine possible protocol states.

Status This issue has been fixed in the following commit: `69598e2`.

3.4 Incorrect Delegate/Voting Balance Accounting in VotingEscrow

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: VotingEscrow
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Easyswap protocol has a core VotingEscrow contract that escrows the governance tokens in the form of an ERC-721 NFT. It also has a built-in delegation feature that allows a user to delegate the voting power to another user. In the process of reviewing the delegation feature, we notice the current implementation is flawed.

In particular, we show below the logic of a core routine that implements the delegation feature. As the name indicates, this _moveAllDelegates() routine records the changes of the owner's NFTs as part of the delegate operation. However, it comes to our attention that the previously delegated NFTs are being duplicated in srcRepNew when nextSrcRepNum = srcRepNum-1, which could seriously affect the voting balance calculation. Note another routine _moveTokenDelegates() shares the same issue.

```
    function _moveAllDelegates(
        address owner,
        address srcRep,
        address dstRep
    ) internal {
        // You can only redelegate what you own
        if (srcRep != dstRep) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint[] storage srcRepOld = srcRepNum > 0
                    ? checkpoints[srcRep][srcRepNum - 1].tokenIds
                    : checkpoints[srcRep][0].tokenIds;
                uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
                uint[] storage srcRepNew = checkpoints[srcRep][
                    nextSrcRepNum
                ].tokenIds;
                // All the same except what owner owns
                for (uint i = 0; i < srcRepOld.length; i++) {
                    uint tId = srcRepOld[i];
                    if (idToOwner[tId] != owner) {
                        srcRepNew.push(tId);
                    }
                }
```

```

            numCheckpoints[srcRep] = srcRepNum + 1;
        }

        if (dstRep != address(0)) {
            uint32 dstRepNum = numCheckpoints[dstRep];
            uint[] storage dstRepOld = dstRepNum > 0
                ? checkpoints[dstRep][dstRepNum - 1].tokenIds
                : checkpoints[dstRep][0].tokenIds;
            uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
            uint[] storage dstRepNew = checkpoints[dstRep][
                nextDstRepNum
            ].tokenIds;
            uint ownerTokenCount = ownerToNFTokenCount[owner];
            require(
                dstRepOld.length + ownerTokenCount <= MAX_DELEGATES,
                "dstRep would have too many tokenIds"
            );
            // All the same
            for (uint i = 0; i < dstRepOld.length; i++) {
                uint tId = dstRepOld[i];
                dstRepNew.push(tId);
            }
            // Plus all that's owned
            for (uint i = 0; i < ownerTokenCount; i++) {
                uint tId = ownerToNFTokenIdList[owner][i];
                dstRepNew.push(tId);
            }

            numCheckpoints[dstRep] = dstRepNum + 1;
        }
    }
}
```

Listing 3.6: `VotingEscrow::_moveAllDelegates()`

Recommendation Revise the above delegate logic to properly record the set of NFTs being delegated.

Status This issue has been fixed in the following commit: `35fa34c`.

3.5 Voting Delegate Denial-of-Service With Dust Delegates

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

Description

As mentioned in Section 3.4, the VotingEscrow contract in Easyswap escrows the governance tokens in the form of an ERC-721 NFT, which can be delegated from one user to another. While analyzing the delegation logic, we notice a possible denial-of-service issue.

In the following, we show the implementation of the related _moveTokenDelegates() routine. This routine is designed to move the delegated NFT from one user to another. We notice the recipient-side requirement upon the NFT delegation, i.e., dstRepOld.length + 1 <= MAX_DELEGATES (line 1230). In other words, there is a limit on the maximum number of received NFTs. Further, our analysis shows it is possible to create dust NFT and delegate them to one victim user. As a result, the victim user may not be able to recieve legitimate delegation once the number of total delegates reaches the threshold, i.e., MAX_DELEGATES.

```
1193    function _moveTokenDelegates(
1194        address srcRep,
1195        address dstRep,
1196        uint _tokenId
1197    ) internal {
1198        if (srcRep != dstRep && _tokenId > 0) {
1199            if (srcRep != address(0)) {
1200                uint32 srcRepNum = numCheckpoints[srcRep];
1201                uint[] storage srcRepOld = srcRepNum > 0
1202                    ? checkpoints[srcRep][srcRepNum - 1].tokenIds
1203                    : checkpoints[srcRep][0].tokenIds;
1204                uint32 nextSrcRepNum = _findWhatCheckpointToWrite(srcRep);
1205                uint[] storage srcRepNew = checkpoints[srcRep][
1206                    nextSrcRepNum
1207                ].tokenIds;
1208                // All the same except _tokenId
1209                for (uint i = 0; i < srcRepOld.length; i++) {
1210                    uint tId = srcRepOld[i];
1211                    if (tId != _tokenId) {
1212                        srcRepNew.push(tId);
1213                    }
1214                }

1216                numCheckpoints[srcRep] = srcRepNum + 1;
1217            }
```

```
            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint[] storage dstRepOld = dstRepNum > 0
                    ? checkpoints[dstRep][dstRepNum - 1].tokenIds
                    : checkpoints[dstRep][0].tokenIds;
                uint32 nextDstRepNum = _findWhatCheckpointToWrite(dstRep);
                uint[] storage dstRepNew = checkpoints[dstRep][
                    nextDstRepNum
                ].tokenIds;
                // All the same plus _tokenId
                require(
                    dstRepOld.length + 1 <= MAX_DELEGATES,
                    "dstRep would have too many tokenIds"
                );
                for (uint i = 0; i < dstRepOld.length; i++) {
                    uint tId = dstRepOld[i];
                    dstRepNew.push(tId);
                }
                dstRepNew.push(_tokenId);

                numCheckpoints[dstRep] = dstRepNum + 1;
            }
        }
    }
```

Listing 3.7: `VotingEscrow::_moveTokenDelegates()`

Recommendation Improve the above logic by restricting possible dust delegation. Note another routine `_moveAllDelegates()` shares the same issue.

Status This issue has been fixed in the following commit: `69598e2`.

3.6 Possible Rebase Reward Lockup For Expired NFTs

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardsDistributor`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

To compensate the locked protocol tokens for voting, the `Easyswap` protocol issues rebase rewards to users based on their pro-rata voting weight. These rebase rewards may be claimed via the `RewardsDistributor` contract. While reviewing the reward-claiming logic, we notice the claim may fail if the respective lockup expiry is passed.

To elaborate, we show below the implementation of this `claim()` routine. While it properly computes the reward amount to claim, the rewards will be eventually deposited into the `VotingEscrow` contract with the `deposit_for()` routine, which requires the given `_tokenId` must be valid, including not expired.

```
function claim(uint _tokenId) external returns (uint) {
    if (block.timestamp >= time_cursor) _checkpoint_total_supply();
    uint _last_token_time = last_token_time;
    _last_token_time = _last_token_time / WEEK * WEEK;
    uint amount = _claim(_tokenId, voting_escrow, _last_token_time);
    if (amount != 0) {
        IVotingEscrow(voting_escrow).deposit_for(_tokenId, amount);
        token_last_balance -= amount;
    }
    return amount;
}
```

Listing 3.8: `RewardsDistributor::claim()`

Recommendation Revise the above routine to send the rewards to the owner if the given `_tokenId` is expired.

Status This issue has been fixed in the following commit: `69598e2`.

3.7 Inconsistent K Invariants Between Pair And Router

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `Easyswap` protocol has a built-in curve, which is different from $xy = k$. While analyzing the k usages between `Pair` and `Router` routines, we notice certain inconsistency that needs to be resolved before deployment.

To elaborate, we show below the related `quoteLiquidity()` routine from the `Router` contract. While it always follows the traditional curve $xy = k$ to compute the output amount, it does not take into account the new curve when the given pair is a stable swap one.

```
// given some amount of an asset and pair reserves, returns an equivalent amount of
    the other asset
function quoteLiquidity(uint amountA, uint reserveA, uint reserveB) internal pure
    returns (uint amountB) {
```

```
        require(amountA > 0, 'Router: INSUFFICIENT_AMOUNT');
        require(reserveA > 0 && reserveB > 0, 'Router: INSUFFICIENT_LIQUIDITY');
        amountB = amountA * reserveB / reserveA;
    }
```

Listing 3.9: `Router::quoteLiquidity()`

Recommendation Revise the above logic to properly compute the quote amount for both stable and volatile pairs.

Status This issue has been fixed in the following commit: `69598e2`.

3.8 Improved Validation on Protocol Parameters

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Easyswap` protocol is no exception. Specifically, if we examine the `IDOSale` contract, it has defined a number of protocol-wide risk parameters, such as `_hardCap` and `_startTime`. In the following, we show the corresponding constructor routine that initializes their values.

```
    constructor(
        uint256 hardCap_,
        uint256 startTime_,
        uint256 endTime_,
        IERC20 tokenTokenAddress_,
        uint256 tokenprice_
    ) {
        _hardCap = hardCap_;
        _startTime = startTime_;
        _endTime = endTime_;
        _tokenTokenAddress = tokenTokenAddress_;
        _tokenprice = tokenprice_;

        emit PoolIsUpcoming();
    }
```

Listing 3.10: IDOSale::constructor()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `_startTime` may make the `IDO` process infeasible, hence incurring cost or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Note the same issue is also applicable to other contracts, including `publicPool` and `whitelistPool`.

Status This issue has been fixed in the following commit: `69598e2`.

3.9 Revisited withdraw() Logic in IDOSale

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IDOSale`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Easyswap` protocol has a built-in `IDOSale` contract to raise funds for the protocol development and community engagement. While reviewing the withdrawal logic in `IDOSale`, we notice it makes an unnecessary assumption and its implementation can be greatly improved.

To elaborate, we show below the related `withdraw()` routine. It computes the `tokentokensToReceive` based on the specified `_tokenprice` (line 149) with an implicit assumption, i.e., the protocol token has 18 as its decimals. Moreover, the storage state of `tokenfinalTokens` is updated and then reset, introducing redundant storage updates. Moreover, the routine also introduces a local variable `tokenfinaltokenreceive`, which can be simply replaced with an earlier one `tokentokensToReceive`.

```
145     function withdraw() external poolIsFinished nonReentrant returns (bool) {
146         uint256 ethersToSpend = _balanceOf[msg.sender];
147         require(ethersToSpend > 0, "No amount present to withdraw");

149         uint256 tokentokensToReceive = (ethersToSpend /_tokenprice * 10 ** 18);


152         require(
153             (IERC20(_tokenTokenAddress).allowance(owner(), address(this))) >=
154                 tokentokensToReceive,
155             "Not enough allowance for project tokens"
156         );
```

```
        _balanceOf[msg.sender] = 0;
        tokenfinalTokens[msg.sender] = tokentokensToReceive;

        uint256 tokenfinaltokenreceive = tokenfinalTokens[msg.sender];

        tokenfinalTokens[msg.sender] = 0;


        IERC20(_tokenTokenAddress).transferFrom(
            owner(),
            msg.sender,
            tokenfinaltokenreceive
        );

        emit Claim(msg.sender, tokenfinaltokenreceive);


        return true;
    }
```

Listing 3.11: `IDOSale::withdraw()`

Recommendation Simplify the above routine and remove the implicit decimals assumption. Note the same issue is also applicable to other contracts, including `publicPool` and `whitelistPool`.

Status This issue has been fixed in the following commit: `69598e2`.

3.10 Potentially Out-of-sync rewardRate in Gauge

- ID: PVE-010
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Gauge`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `Easyswap` protocol creates a gauge for the supported pool and each gauge will receive the emission of protocol tokens from the `voter`. While reviewing the protocol token redistribution within each gauge to its stakers, we notice the current redistribution logic may make use of an out-of-sync reward state.

To elaborate, we show below the related code snippet `_calcRewardPerToken()`, which calculates the `rewardPerToken` for the given reward token as well as the related total supply. However, it comes to our attention that it always uses the latest `rewardRate`, which may not be in the same `epoch`. As a result, the LP stakers may be rewarded with incorrectly-computed reward amount.

```
    function _calcRewardPerToken(address token, uint timestamp1, uint timestamp0, uint supply, uint startTimestamp) internal view returns (uint, uint) {
        uint endTime = Math.max(timestamp1, startTimestamp);
        return (((Math.min(endTime, periodFinish[token]) - Math.min(Math.max(timestamp0, startTimestamp), periodFinish[token])) * rewardRate[token] * PRECISION / supply), endTime);
    }
```

Listing 3.12: `Gauge::_calcRewardPerToken()`

Recommendation Revise the above routine to properly emit the reward redistribution among LP stakers in each gauge.

Status This issue has been confirmed. The team has considered storing the user's epoch separately to keep track of all LP stakers, but this is difficult to implement due to the unpredictability of staking. Also, since rewards decrease weekly, there is no issue with running out of rewards. The team will notify LP stakers and rewards will be applied at the latest `rewardRate`.

3.11 Trust Issue of Admin Keys

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Easyswap` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and adding new allowed tokens). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setPause(bool _state) external {
        require(msg.sender == pauser);
        isPaused = _state;
    }

    function setFeeManager(address _feeManager) external {
        require(msg.sender == feeManager, 'not fee manager');
        pendingFeeManager = _feeManager;
    }

    function acceptFeeManager() external {
```

```
        require(msg.sender == pendingFeeManager, 'not pending fee manager');
        feeManager = pendingFeeManager;
    }

    function setFee(bool _stable, uint256 _fee) external {
        require(msg.sender == feeManager, 'not fee manager');
        require(_fee <= MAX_FEE, 'fee too high');
        require(_fee != 0, 'fee must be nonzero');
        if (_stable) {
            stableFee = _fee;
        } else {
            volatileFee = _fee;
        }
    }
```

Listing 3.13: Example Privileged Functions in `PairFactory`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

3.12 Improper VotingEscrow Query in RewardsDistributor

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardsDistributor`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

To incentivize the long-time stakers with inflation, the `Easyswap` protocol has a built-in `RewardsDistributor` contract to calculate inflation and adjust emission balances accordingly. While reviewing the current logic, we notice three different routines can be improved.

To elaborate, we show below one example `ve_for_at()` routine. This routine is proposed to calculate the voting power of the given `NFT` at the specific timestamp. It comes to our attention that the resulting `int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.ts))))` (line 140) may be negative. However, when it is negative, the type cast to `uint` makes it positive, which leads to an incorrect calculation of voting power (in this case, the resulting voting power should be 0.). The same issue is also applicable to two other routines `_checkpoint_total_supply()` and `_claim()`.

```
135     function ve_for_at(uint _tokenId, uint _timestamp) external view returns (uint) {
136         address ve = voting_escrow;
137         uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
138         uint epoch = _find_timestamp_user_epoch(ve, _tokenId, _timestamp, max_user_epoch
                );
139         IVotingEscrow.Point memory pt = IVotingEscrow(ve).user_point_history(_tokenId,
                epoch);
140         return Math.max(uint(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.
                ts))))), 0);
141     }
```

Listing 3.14: `RewardsDistributor::ve_for_at()`

Recommendation Revise the above three routines to properly compute the user's voting power.

Status This issue has been fixed in the following commit: `69598e2`.

3.13 Timely Reward Resume Upon Reviving Gauges

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Voter
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Easyswap protocol creates a gauge for the supported pool and the created gauge can be killed or revived based on the community needs. While reviewing the current gauge-reviving logic, we notice a revived gauge needs to be properly re-initialized!

To elaborate, we show below the related reviveGauge() routine. While it properly marks the gauge alive (line 257), it does not properly set the associated supplyIndex, i.e., supplyIndex[_gauge] = index, making it still eligible for rewards even before its revive.

```
254     function reviveGauge(address _gauge) external {
255         require(msg.sender == emergencyCouncil, "not emergency council");
256         require(!isAlive[_gauge], "gauge already alive");
257         isAlive[_gauge] = true;
258         emit GaugeRevived(_gauge);
259     }
```

Listing 3.15: Voter::reviveGauge()

Recommendation Revise the above logic to properly revive a current gauge.

Status This issue has been fixed in the following commit: 69598e2.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Easyswap` protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from `Solidly/Velodrome` with a unique `AMM`. The DEX is compatible with all the standard features as popularized by `UniswapV2` with a number of novel improvements, including price oracles without upkeeps, a new curve $(x^3y+xy^3 = k)$ for efficient stable swaps, as well as a built-in `NFT`-based voting mechanism and associated token emissions. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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