



SMART CONTRACT AUDIT REPORT

for

dYdX Liquidity Staking



Prepared By: Yiqun Chen

PeckShield
June 4, 2021

Document Properties

Client	dYdX
Title	Smart Contract Audit Report
Target	Liquidity Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Jian Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 4, 2021	Xuxian Jiang	Final Release
1.0-rc	May 22, 2021	Xuxian Jiang	Release Candidate
0.2	May 9, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 4, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About dYdX	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Redundant State/Code Removal	11
3.2	Trust Issue of Admin Keys	12
3.3	Accommodation of Non-ERC20-Compliant Airdrop Tokens	13
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the **Liquidity Staking** support in the `dYdX` 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. This document outlines our audit results.

1.1 About dYdX

`dYdX` is a leading decentralized exchange that currently supports perpetual, margin trading, and spot trading, as well as lending, and borrowing. `dYdX` runs on smart contracts on the `Ethereum` blockchain, and allows users to trade with no intermediaries. `dYdX` is designed to bring trading tools from the traditional world of finance to the blockchain with similar user experience. When using `dYdX`, users deposit their collateral to off-chain order books. These are non-custodial, but they offer faster trade execution and users only have to pay gas fees when depositing or withdrawing assets from the platform. The audited `liquidity staking` module allows users to earn rewards for staking `USDC`. The staked funds may be borrowed by certain pre-approved partners, on a reputational basis, without collateral. The funds may only be used on the L2 exchange.

The basic information of the Liquidity Staking protocol is as follows:

Table 1.1: Basic Information of Liquidity Staking

Item	Description
Name	dYdX
Website	https://dydx.exchange/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 4, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audited repository contains a number of sub-directories (e.g., `staking`, `stark-proxy`, and `liquidity`) and this audit covers the `liquidity` and `cumulative-merkle-distributor` sub-directories.

- <https://github.com/dydxfoundation/governance-contracts.git> (1088116)

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

- <https://github.com/dydxfoundation/governance-contracts.git> (59b4ab0)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], 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), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices



Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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 Liquidity Staking protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	2	
Informational	1	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Liquidity Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Unused State/Code Removal	Coding Practices	Resolved
PVE-002	Low	Trust Issue of Admin Keys	Security Features	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Airdrop Token	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 Redundant State/Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The liquidity staking module makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, and ReentrancyGuard, to facilitate its code implementation and organization. For example, the LS1Borrowing smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the LS1Storage contract, there is a specific storage variable, i.e., `_DEBT_OPERATORS_`, that is defined, but not used. The intended debt-related operators have been replaced with the access control role `DEBT_OPERATOR_ROLE`.

```

17 abstract contract LS1Storage is AccessControlUpgradeable, ReentrancyGuard,
    VersionedInitializable {
18     // ===== Access Control =====
19
20     /// @dev Addresses which are allowed to modify debt balances.
21     mapping(address => bool) internal _DEBT_OPERATORS_;
22
23     // ===== Epoch Schedule =====
24
25     /// @dev The parameters specifying the function from timestamp to epoch number.
26     LS1Types.EpochParameters internal _EPOCH_PARAMETERS_;
27     ...
28 }

```

Listing 3.1: The LS1Storage Contract

Moreover, we notice the function `_settleGlobalIndexUpToEpoch()` in the `LS1Rewards` contract contains unused local variable `previouslySettledTimestamp` (line 211), which can also be safely removed.

```

206  function _settleGlobalIndexUpToEpoch(uint256 totalStaked, uint256 epochNumber)
207      internal
208      returns (uint256)
209  {
210      uint256 settleUpToTimestamp = getStartOfEpoch(epochNumber.add(1));
211      uint256 previouslySettledTimestamp = _GLOBAL_INDEX_TIMESTAMP_;
212
213      uint256 globalIndex = _settleGlobalIndexUpToTimestamp(totalStaked,
214                                                             settleUpToTimestamp);
215      _EPOCH_INDEXES_[epochNumber] = globalIndex;
216      return globalIndex;
  
```

Listing 3.2: `LS1Rewards::_settleGlobalIndexUpToEpoch()`

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

Status The issue has been fixed by this commit: `f87429b`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LS10operators`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The `liquidity staking` module supports a number of roles that can be regulated and managed by the one with the designated `OWNER_ROLE`. As the name indicates, this is a privileged account that plays a critical role in governing and regulating the token-related operations (e.g., assigning other roles). In the following, we show representative privileged operations in the `liquidity staking` module.

```

116  function claimRewardsFor(address staker, address recipient)
117      external
118      onlyRole(CLAIM_OPERATOR_ROLE)
119      nonReentrant
120      returns (uint256)
121  {
122      uint256 rewards = _settleAndClaimRewards(staker, recipient); // Emits an event
123                          internally.
124      emit OperatorClaimedRewardsFor(staker, recipient, rewards, msg.sender);
  
```

```

124     return rewards;
125 }

127 function decreaseStakerDebt(address staker, uint256 amount)
128     external
129     onlyRole(DEBT_OPERATOR_ROLE)
130     nonReentrant
131     returns (uint256)
132 {
133     uint256 oldDebtBalance = _settleStakerDebtBalance(staker);
134     uint256 newDebtBalance = oldDebtBalance.sub(amount);
135     _STAKER_DEBT_BALANCES_[staker] = newDebtBalance;
136     emit OperatorDecreasedStakerDebt(staker, amount, newDebtBalance, msg.sender);
137     return newDebtBalance;
138 }

```

Listing 3.3: Example Privileged Operations in LS1Operators

We emphasize that the privilege assignment is necessary and consistent with the staking design. However, it is worrisome if the `owner` is not governed by a DAO-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. It should be noted that a compromised `owner` account would allow the attacker to mess up internal records and claim rewards for others, which directly undermines the assumption of the staking support.

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 resolved and the team confirms that the contract will be owned by the governance DAO, not by a multisig.

3.3 Accommodation of Non-ERC20-Compliant Airdrop Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CumulativeMerkleDistributor`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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 the following, we examine the `transfer()` routine and related 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 `transfer()`, there is a check, i.e., `if (balances[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 to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller’s account balance does not have enough tokens to spend.”*

```

64     function transfer(address _to, uint _value) returns (bool) {
65         //Default assumes totalSupply can't be over max (2^256 - 1).
66         if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67             balances[msg.sender] -= _value;
68             balances[_to] += _value;
69             Transfer(msg.sender, _to, _value);
70             return true;
71         } else { return false; }
72     }

74     function transferFrom(address _from, address _to, uint _value) returns (bool) {
75         if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
76             balances[_to] + _value >= balances[_to]) {
77             balances[_to] += _value;
78             balances[_from] -= _value;
79             allowed[_from][msg.sender] -= _value;
80             Transfer(_from, _to, _value);
81             return true;
82         } else { return false; }
83     }

```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. 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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `claim()` routine in the `CumulativeMerkleDistributor` contract. If the USDT token is supported as the claimable token, the unsafe version of `IERC20(token).transfer(account, amount)` (line 75) may revert as there is no return value in the USDT token contract’s `transfer()` implementation (but the `IERC20` interface expects a return value)!

```

63     function claim(uint256 index, address account, uint256 cumulativeAmount, uint256
64         epoch, bytes32[] calldata merkleProof
65     ) external override {
66         bytes32 merkleRoot = _merkleRoot;
67         require(merkleRoot != bytes32(0), "MERKLE_DISTRIBUTOR_ROOT_NOT_SET");

```

```
67
68 // Verify the merkle proof
69 bytes32 node = keccak256(abi.encodePacked(index, account, cumulativeAmount,
70 epoch));
71 require(MerkleProof.verify(merkleProof, merkleRoot, node), '
72     MERKLE_DISTRIBUTOR_INVALID_PROOF');
73
74 // Mark user address as having claimed 'cumulativeAmount'
75 uint256 claimable = cumulativeAmount.sub(claimed[account]);
76 require(claimable > 0, "MERKLE_DISTRIBUTOR_NOTHING_TO_CLAIM");
77 claimed[account] = cumulativeAmount;
78
79 // Send the user the remaining amount they haven't claimed yet
80 require(IERC20(TOKEN).transfer(account, claimable), '
81     MERKLE_DISTRIBUTOR_TRANSFER_FAILED');
82
83 emit Claimed(index, account, claimable, epoch);
84 }
```

Listing 3.5: CumulativeMerkleDistributor :: claim()

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

Status The issue has been fixed by this commit: 59b4ab0.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the **Liquidity Staking** support in the **avax** protocol. The system presents a unique, robust offering as a decentralized non-custodial platform allowing users to earn rewards for staking **USDC**. (The staked funds may be borrowed for use on the L2 exchange by certain pre-approved partners, on a reputational basis, without collateral.) The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Furthermore, 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [2] MITRE. CWE-563: Assignment to Variable without Use. <https://cwe.mitre.org/data/definitions/563.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.