SMART CONTRACT AUDIT REPORT

for

dYdX StarkProxy & Merkle Distributor

Prepared By: Yiqun Chen

PeckShield
June 25, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	June 25, 2021	Xuxian Jiang	Final Release
1.0-rc	June 25, 2021	Xuxian Jiang	Release Candidate
0.2	June 20, 2021	Xuxian Jiang	Add More Findings #1
0.1	June 15, 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

Given the opportunity to review the design document and related smart contract source code of the **dYdX StarkProxy & Merkle Distributor** 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 StarkProxy & Merkle Distributor module allows to borrow staked funds for use only on the L2 exchange as well as publish DYDX token rewards for participating users.

The basic information of the dYdX StarkProxy & Merkle Distributor protocol is as follows:

Table 1.1: Basic Information of dYdX StarkProxy & Merkle Distributor

Item	Description
Name	dYdX
Website	https://dydx.exchange/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 25, 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 `stark-proxy` and `merkle-distributor` sub-directories.

- https://github.com/dydxfoundation/governance-contracts.git (3fd209c)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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.

Table 1.3: The Full List of Check Items

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

To 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) [5], 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 1.4: 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 design and implementation of the support on `dYdX StarkProxy & Merkle Distributor`. 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	1	■
Informational	1	■
Total	2	

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 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key dYdX StarkProxy & Merkle Distributor Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant nonReentrant Modifier Removal	Coding Practices	Resolved
PVE-002	Low	Trust Issue of Admin Keys	Security Features	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 nonReentrant Modifier Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The support on the dYdX StarkProxy & Merkle Distributor has extensive use of the nonReentrant modifier, as a precaution to defend against possible reentrancy risks. However, our analysis shows its use in a number of contracts may not be necessary.

For example, if we examine closely the MD1Pausable contract, there are two external functions, i.e., pauseRootUpdates() and unpauseRootUpdates(). These two functions allow the authorized entities to either prevent (or resume) proposed Merkle roots from becoming (to become) active. Moreover, both functions are protected with the nonReentrant, which is unnecessary and can be safely removed.

```
/**
 * @dev Called by PAUSER_ROLE to prevent proposed Merkle roots from becoming active.
 */
function pauseRootUpdates()
  onlyRole(PAUSER_ROLE)
  whenNotPaused
  nonReentrant
  external
{
  _ARE_ROOT_UPDATES_PAUSED_ = true;
  emit RootUpdatesPaused();
}

/**
 * @dev Called by UNPAUSER_ROLE to resume allowing proposed Merkle roots to become
    active.
```

```
  */
function unpauseRootUpdates()
  onlyRole(UNPAUSER_ROLE)
  whenPaused
  nonReentrant
  external
{
  _ARE_ROOT_UPDATES_PAUSED_ = false;
  emit RootUpdatesUnpaused();
}
```

Listing 3.1: `MD1Pausable::pauseRootUpdates()/unpauseRootUpdates()`

The same issue is also present in other contracts, including `MD1Logic`, `SP1Owner`, `SP1Guardian`, and `SP1FundsAdmin`.

Recommendation Consider the removal of the redundant `nonReentrant` modifier in the above contracts.

Status This issue has been resolved as the team intends to be more strict for all possible external interactions.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

The `dYdX StarkProxy & Merkle Distributor` 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 with the privileged roles.

```
/**
 * @notice Set the parameters defining the function from timestamp to epoch number.
 *  Note that the epoch number is made available externally but is not used internally
    .
 *
 * @param  interval  The length of an epoch, in seconds.
 * @param  offset    The start of epoch zero, in seconds.
 */
```

```
function setEpochParameters(uint256 interval, uint256 offset)
  external
  onlyRole(OWNER_ROLE)
  nonReentrant
{
  _setEpochParameters(interval, offset);
}

/**
 * @notice Set the address of the oracle which provides Merkle root updates.
 *
 * @param  rewardsOracle  The new oracle address.
 */
function setRewardsOracle(address rewardsOracle)
  external
  onlyRole(OWNER_ROLE)
  nonReentrant
{
  _setRewardsOracle(rewardsOracle);
}

// ============ Internal Functions ============

function _setRewardsOracle(address rewardsOracle)
  internal
{
  _REWARDS_ORACLE_ = IRewardsOracle(rewardsOracle);
  emit RewardsOracleChanged(rewardsOracle);
}
```

Listing 3.2: Example Privileged Operations in `MD1Owner`

We emphasize that the privilege assignment is necessary and consistent with the intended 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 owned by the governance DAO. 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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the **dYdX StarkProxy & Merkle Distributor** support in the dYdX protocol. The system presents a unique, robust offering as a decentralized non-custodial platform allowing users to earn rewards for staking USDC. The audited module allows to borrow staked funds for use only on the L2 exchange as well as publish DYDX token rewards for participating users. 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.

The audited StarkProxy & Merkle Distributor module allows to borrow staked funds for use only on the L2 exchange as well as publish DYDX token rewards for participating users.

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 CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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