SMART CONTRACT AUDIT REPORT

for

Lido on Polygon

Prepared By: Xiaomi Huang

PeckShield

December 25, 2022

Document Properties

Client	Lido Finance
Title	Smart Contract Audit Report
Target	Lido on Polygon (v2.0)
Report Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 25, 2022	Xuxian Jiang	Final Release
1.0-rc	December 23, 2022	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the PRs to the `Lido on Polygon` 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 Lido

The `Lido` is a liquid staking solution for `proof-of-stake (PoS)` cryptocurrencies that supports `Ethereum 2.0 (The Merge)` staking and a growing ecosystem of other `Layer 1 PoS` blockchains. Users can stake their `PoS` tokens on `Lido` and receive a tokenized version of their staked assets. They can use this tokenized version of their staked assets to earn additional yield from other `DeFi` protocols while receiving staking rewards from their tokens deposited on `Lido`. The audited PRs support the `MATIC` staking on `Polygon` to earn daily staking rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Lido on Polygon

Item	Description
Name	Lido Finance
Website	https://lido.fi
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 25, 2022

In the following, we show the two PRs of reviewed files in the audit.

- https://github.com/lidofinance/polygon-contracts/pull/2
- https://github.com/lidofinance/polygon-contracts/pull/3

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

- https://github.com/lidofinance/polygon-contracts.git (7845610e763fbbab2152b8c2873da795d94ea148)

1.2 About PeckShield

PeckShield Inc. [7] 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 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.

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

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) [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) 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 implementation of the two PRs to the `Lido on Polygon` protocol. 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	0	
Informational	2	■■
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

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

Table 2.1: Key Audit Findings of Lido on Polygon Protocol

ID	Severity	Title	Category	Status
PVE-001	Informational	Redundant State/Code Removal	Coding Practices	Resolved
PVE-002	Informational	Improved Gas Usage in PoLidoNFT::_beforeTokenTransfer()/_removeApproval()	Business Logic	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Redundant State And Code Removal

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: NodeOperatorRegistry
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

Description

The Lido (Polygon) protocol makes good use of a number of reference contracts, such as ERC20Upgradeable, AccessControlUpgradeable, and PausableUpgradeable, to facilitate its code implementation and organization. For example, the NodeOperatorRegistry smart contract has so far imported at least three 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 listWithdrawNodeOperators() function, there is an internal variable validator that is computed twice. The first one is calculated in the _getOperatorStatusAndValidator() call (line 389) while the second one is explicitly generated in the stakeManager.validators() call (line 394). And there is no update between the two calls. As a result, the second call can be safely skipped.

```
375     function listWithdrawNodeOperators()
376         external
377         view
378         override
379         returns (ValidatorData[] memory, uint256)
380     {
381         uint256 totalNodeOperators = 0;
382         uint256[] memory memValidatorIds = validatorIds;
383         uint256 length = memValidatorIds.length;
384         IStakeManager.Validator memory validator;
385         NodeOperatorRegistryStatus operatorStatus;
386         ValidatorData[] memory withdrawValidators = new ValidatorData[](length);
```

```

        for (uint256 i = 0; i < length; i++) {
            (operatorStatus, validator) = _getOperatorStatusAndValidator(
                memValidatorIds[i]
            );
            if (operatorStatus == NodeOperatorRegistryStatus.INACTIVE) continue;

            validator = stakeManager.validators(memValidatorIds[i]);
            withdrawValidators[totalNodeOperators] = ValidatorData(
                validator.contractAddress,
                validatorIdToRewardAddress[memValidatorIds[i]]
            );
            totalNodeOperators++;
        }

        return (withdrawValidators, totalNodeOperators);
    }
```

Listing 3.1: NodeOperatorRegistry::listWithdrawNodeOperators

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been resolved by following the above suggestion in the PR 6.

3.2 Improved Gas Usage in PoLidoNFT::_beforeTokenTransfer()/_removeApproval()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PoLidoNFT
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

To facilitate the withdraw management and reward distribution, the Lido protocol has a PoLidoNFT contract. As the name indicates, it is an NFT implementation to encapsulate the withdraw requests. Our analysis shows this contract can be improved for gas efficiency.

In particular, we show below the related helper for token transfer – _beforeTokenTransfer(). Its logic differentiates three cases: mint, burn, and transfer. During our analysis in the second case, the current implementation reads the same storage state ownerTokens[lastOwnerTokensIndex] twice: the first time occurs at line 132 while the second time is performed at line 140. Note that the storage state is not updated in between and therefore the second read can be safely replaced with the `lastOwnerTokenId`, which is the result from the first read.

```
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    )
        internal
        virtual
        override(ERC721Upgradeable, ERC721PausableUpgradeable)
        whenNotPaused
    {
        require(from != to, "Invalid operation");

        super._beforeTokenTransfer(from, to, tokenId);

        // Minting
        if (from == address(0)) {
            uint256[] storage ownerTokens = owner2Tokens[to];

            ownerTokens.push(tokenId);
            token2Index[tokenId] = ownerTokens.length - 1;
        }
        // Burning
        else if (to == address(0)) {
            uint256[] storage ownerTokens = owner2Tokens[from];
            uint256 ownerTokensLength = ownerTokens.length;
            uint256 burnedTokenIndexInOwnerTokens = token2Index[tokenId];
            uint256 lastOwnerTokensIndex = ownerTokensLength - 1;

            if (
                burnedTokenIndexInOwnerTokens != lastOwnerTokensIndex &&
                ownerTokensLength != 1
            ) {
                uint256 lastOwnerTokenId = ownerTokens[lastOwnerTokensIndex];
                // Make the last token have an index of a token we want to burn.
                // So when we request index of token with id that is currently last in ownerTokens it does not point
                // to the last slot in ownerTokens, but to a burned token's slot (we will update the slot at the next line)
                token2Index[lastOwnerTokenId] = burnedTokenIndexInOwnerTokens;
                // Copy currently last token to the place of a token we want to burn.
                // So updated pointer in token2Index points to a slot with the correct value.
                ownerTokens[burnedTokenIndexInOwnerTokens] = ownerTokens[
                    lastOwnerTokensIndex
                ];
            }
            ownerTokens.pop();
            delete token2Index[tokenId];

```

```
            if (getApproved(tokenId) != address(0)) {
                _removeApproval(tokenId);
            }
        }
        // Transferring
        else if (from != to) {
            ...
        }
    }
```

Listing 3.2: `PoLidoNFT::_beforeTokenTransfer()`

Recommendation Revise the above logic to avoid repeated reads from the same storage state. Note the same suggestion is also applicable to the `_removeApproval()` routine.

Status The issue has been fixed in the following `PR 6`.

4 Conclusion

In this audit, we have analyzed the design and implementation of two PRs to the `Lido on Polygon` protocol, which allows users to stake their `MATIC` to earn daily staking rewards and interact with other DeFi protocols across the `Polygon` ecosystems. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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