SMART CONTRACT AUDIT REPORT

for

Lens V2

Prepared By: Xiaomi Huang

PeckShield

May 16, 2023

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	Lens V2
Version	1.0
Author	Stephen Bie
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 16, 2023	Stephen Bie	Final Release
1.0-rc	April 27, 2023	Stephen Bie	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 smart contract source code of the Lens V2, 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 Lens V2

Lens V2 is a composable social graph protocol built to be community-owned and ever-evolving. It empowers its users by allowing them to decide how they want their social graph to be built, and how they want it to be monetized, if at all. Furthermore, the protocol is engineered with the concept of modularity at its core, allowing for an infinitely expanding amount of use cases. This, from the user's perspective, translates to a new paradigm of ownership and customization that just isn't possible (or financially feasible) in traditional Web2. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Lens V2

Item	Description
Target	Lens V2
Website	https://lens.dev/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 16, 2023

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

- https://github.com/lens-protocol/core-private.git (194b464)

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

- https://github.com/lens-protocol/core-private.git (6d506e3)

1.2 About PeckShield

PeckShield Inc. [9] 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 [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.

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) [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) 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 `Lens V2` 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 logic, 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	3	
Low	0	
Informational	2	
Total	5	

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 3 medium-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key Lens V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic of LensHandles::burn()	Business Logic	Fixed
PVE-002	Medium	Enhanced Sanity Check in TokenHandleRegistry::_unlinkHandleFromToken()	Business Logic	Fixed
PVE-003	Informational	Improved Logic of PublicationLib::_initPubActionModules()	Coding Practices	Fixed
PVE-004	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Revisited Logic of LensHandles::burn()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: LensHandles
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Lens V2 protocol implements a decentralized social media, which is achieved by allowing users to create profiles and interact with each other via these profiles. A user needs to create a profile, for which he will receive a Lens Profile NFT as the unique profile identification. Additionally, the privileged owner of the protocol will mint a Lens Handles NFT to the user, which points to a local name (e.g., John). The user can link his Lens Handles NFT with Lens Profile NFT (or unlink his Lens Handles NFT from Lens Profile NFT). In particular, the LensHandles::burn() routine is used by the user to burn his own Lens Handles NFT. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the LensHandles contract. Inside the burn() routine, we observe there is a lack of msg.sender validation, which allows the malicious actor to burn someone else's Lens Handles NFT, which directly undermines the assumption of the protocol design. Moreover, we notice it does not clear the localName of the given Lens Handles NFT after the token is burnt.

```
    function burn(uint256 tokenId) external {
        _burn(tokenId);
    }
```

Listing 3.1: LensHandles::burn()

Recommendation Correct the logic errors mentioned above accordingly.

Status The issue has been addressed in the following commit: `1aa2271`.

3.2 Enhanced Sanity Check in TokenHandleRegistry::_unlinkHandleFromToken()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TokenHandleRegistry`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, in the `Lens V2` protocol, a user needs to create a profile, for which he will receive a `Lens Profile NFT` as the unique profile identification. Additionally, the privileged `owner` of the protocol will mint a `Lens Handles NFT` to the user, which points to a local name (e.g., `John`). The user can link his `Lens Handles NFT` with `Lens Profile NFT` (or unlink his `Lens Handles NFT` from `Lens Profile NFT`). In particular, the `TokenHandleRegistry::_unlinkHandleFromToken()` routine is designed to unlink the `Lens Handles NFT` from `Lens Profile NFT`. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `TokenHandleRegistry` contract. Inside the `_unlinkHandleFromToken()` routine, we observe it directly clears the `handleToToken` and `tokenToHandle` storage variables (lines 137/138) (which store the pointed `Lens Profile NFT` and `Lens Handles NFT`) to clean up the link status of the given `Lens Handles NFT` and `Lens Profile NFT`. Apparently, it does not verify their current link status and is therefore vulnerable to break someone else's link status.

```
136     function _unlinkHandleFromToken(RegistryTypes.Handle memory handle,RegistryTypes.
            Token memory token) internal onlyHandleOrTokenOwner(handle, token, msg.sender) {
137         delete handleToToken[_handleHash(handle)];
138         delete tokenToHandle[_tokenHash(token)];
139         emit RegistryEvents.HandleUnlinked(handle, token);
140     }
```

Listing 3.2: `TokenHandleRegistry::_unlinkHandleFromToken()`

Recommendation Apply necessary sanity checks to prevent someone else's link status from being broken.

Status The issue has been addressed in the following commit: `1aa2271`.

3.3 Improved Logic of PublicationLib::_initPubActionModules()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PublicationLib`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The `Lens V2` protocol implements a decentralized social media, which is achieved by allowing users to create profiles and interact with each other via these profiles. Moreover, the protocol has a modular design with three types of modules supported: `follow` (taking effect when a profile is followed), `action` (taking effect when a publication is performed action, e.g., `collect`), and `reference` (taking effect when a publication is referred). When the user creates a new publication, he can select a series of whitelisted `action` modules. In particular, the `PublicationLib::_initPubActionModules()` routine is designed to initialize the selected `action` modules. While examining its logic, we observe its current implementation needs to be improved.

To elaborate, we show below the related code snippet of the `PublicationLib` contract. Inside the `_initPubActionModules()` routine, we notice it has the inherent assumption on the same length of the given two arrays, i.e., `actionModules` and `actionModulesInitDatas`. However, this is not enforced in the `_initPubActionModules()` routine. Given this, we suggest to properly validate the given arrays to have the same length. Moreover, it turns out it does not perform necessary sanity checks in preventing the duplicate `action` module from being selected. Thus we suggest to improve the implementation as below: `if (actionModuleBitmap & (1 << (actionModuleWhitelistData.id - 1)))revert('duplicate action module')` (line 450).

```
431     function _initPubActionModules(
432         uint256 profileId,
433         address transactionExecutor,
434         uint256 pubId,
435         address[] calldata actionModules,
436         bytes[] calldata actionModulesInitDatas
437     ) private returns (bytes[] memory) {
438         bytes[] memory actionModuleInitResults = new bytes[](actionModules.length);
439         uint256 actionModuleBitmap;
440 
441         uint256 i;
442         while (i < actionModules.length) {
443             Types.ActionModuleWhitelistData memory actionModuleWhitelistData =
                StorageLib.actionModuleWhitelistData()[
```

```
                actionModules[i]
            ];

            if (!actionModuleWhitelistData.isWhitelisted) {
                revert Errors.NotWhitelisted();
            }

            ...
        }

        StorageLib.getPublication(profileId, pubId).actionModulesBitmap =
            actionModuleBitmap;

        return actionModuleInitResults;
    }
```

Listing 3.3: `PublicationLib::initPubActionModules()`

Recommendation Improve the implementation of the `_initPubActionModules()` routine as above-mentioned.

Status The issue has been addressed in the following commit: `178fd34`.

3.4 Redundant State/Code Removal

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `FollowNFT/DegreesOfSeparationReferenceModule`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

While reviewing the implementation of `Lens V2` protocol, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. Using `FollowNFT::_followTokenExists()` as an example, it is designed to check whether the given `NFT` token exists. However, we observe it is not used anywhere. Given this, we suggest to remove the redundant code safely.

```
    function _followTokenExists(uint256 followTokenId) internal view returns (bool) {
        return _followDataByFollowTokenId[followTokenId].followerProfileId != 0
            _isFollowTokenWrapped(followTokenId);
    }
```

Listing 3.4: `FollowNFT::_followTokenExists()`

Moreover, we observe the `mapping(address signer => uint256 nonce)public nonces` storage variable defined in the `DegreesOfSeparationReferenceModule` contract can be safely removed as well.

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

Status The issue has been addressed in the following commit: `178fd34`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Lens V2` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    // This allows the governance to call anything on behalf of itself.
    // And also allows the Upgradable contract to call anything, except the LensHub with
        Governance permissions.
    function executeAsGovernance(
        address target,
        bytes calldata data
    ) external payable onlyOwnerOrControllerContract returns (bytes memory) {
        if (msg.sender == controllerContract && target == address(LENS_HUB)) {
            revert Unauthorized();
        }
        (bool success, bytes memory returnData) = target.call{gas: gasleft(), value: msg
            .value}(data);
        require(success, string(returnData));
        return returnData;
    }
```

Listing 3.5: `Governance::executeAsGovernance()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team. The team intends to introduce multi-sig mechanism to mitigate this issue.

4 Conclusion

In this audit, we have analyzed the `Lens V2` design and implementation. `Lens V2` is a fully composable, monetizable and decentralized social graph, which aims to empower creators to own the links between them and their community. Furthermore, the protocol is engineered with the concept of modularity at its core, allowing for an infinitely expanding amount of use cases. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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.