# SMART CONTRACT AUDIT REPORT

for

# Omnipool Protocol

Prepared By: Xiaomi Huang

PeckShield

February 3, 2024

## Document Properties

| | |
|---|---|
| Client | Omnipool |
| Title | Smart Contract Audit Report |
| Target | Omnipool |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jason Shen, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | February 3, 2024 | Xuxian Jiang | Final Release |
| 1.0-rc | February 2, 2024 | 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 `Omnipool` 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About Omnipool

`Omnipool` aims to offer a liquid staking service for `ZETA` tokens and users will be distributed with respective staking rewards. It functions as a native `Liquid-Staking Protocol` comprising two key components: `liquid staking` and `launchpad`. The staking allows users to stake their `ZETA` tokens in return for a liquid stake pool token, known as `omZeta`. Holding `omZeta` offers the dual benefit of staking rewards, providing liquidity, and earning rewards. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Omnipool

| Item | Description |
|---:|---|
| Name | Omnipool |
| Website | https://omnipool.app/ |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | February 3, 2024 |

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

- https://github.com/OmnipoolApp/omnipool-contract.git (5836a36)

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

- https://github.com/OmnipoolApp/omnipool-contract.git (15fdaa3)

## 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 accordingly classified into four categories, 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. 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 `Omnipool` 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 | 0 | |
| Medium | 1 | ■ |
| Low | 3 | ■ ■ ■ |
| Informational | 0 | |
| Total | 4 | |

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 may involve 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 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Improved Proxy Initialization Logic in OmniPoolNft | Coding Practices | Resolved |
| PVE-002 | Low | Possible Frontrunning-Assisted Slash Avoidance | Time and State | Confirmed |
| PVE-003 | Low | Improved Ether Transfer For DoS Avoidance | Coding Practices | Resolved |
| PVE-004 | 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 Improved Proxy Initialization Logic in OmniPoolNft

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OmniPoolNft
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

The Omnipool protocol provides users with the ability to stake supported tokens and instantly receive a representation of their stake in the form of omZeta tokens. While examining the construction and initialization logic of associated OmniPoolNft contract, we notice current implementation can be improved.

In the following, we show the related code snippet from the OmniPoolNft contract, with the highlighted initialization logic. It comes to our attention that the initialize() routine directly calls a few parent contracts' initialization routines, including __Ownable_init() (line 41). However, it does not call the initialization routine of another parent contract (ERC2981Upgradeable). In other words, it is suggested to add the following initialization, i.e., __ERC2981_init(). By doing so, we can ensure the coding practice follows the intended call convention when initializining an upgradeable proxy contract.

```
12 contract OmniPoolNft is
13     UUPSUpgradeableWithDelay,
14     ERC721EnumerableUpgradeable,
15     ERC2981Upgradeable,
16     OwnableUpgradeable
17 {
18     OmniPool public omniPool;
19     uint256 public nextTokenId;

21     event UpdateRoyalty(address indexed _royaltyReceiver, uint96 _royaltyFee);
```

```
    modifier onlyOmnipool() {
        require(msg.sender == address(omniPool), "Caller is not Omnipool");
        _;
    }

    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }

    /**
     * @param _omniPoolAddress Address of OmniPool
     * @param _upgradeDelay time to wait before new upgrade implementation
     */
    function initialize(address _omniPoolAddress, uint256 _upgradeDelay) public initializer {
        require(address(_omniPoolAddress) != address(0), "_omniPoolAddress addresss zero");
        require(_upgradeDelay > 0, "upgradeDelay is zero");

        __Ownable_init();
        __ERC721_init("OmniPool Unstake", "OmniPool NFT");
        __ERC721Enumerable_init();
        __UUPSUpgradeableWithDelay_init(_upgradeDelay);

        omniPool = OmniPool(_omniPoolAddress);
    }
```

Listing 3.1: `OmniPoolNft::initialize()`

**Recommendation** Improve the above-mentioned initialization routine by all all parent contracts' initialization routines. Also, the current `OmniPool` contract can be improved by adding the following constructor logic:

```
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

Listing 3.2: `OmniPool::constructor()`

**Status** This issue has been fixed by the following commit: `15fdaa3`.

## 3.2 Possible Frontrunning-Assisted Slash Avoidance

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OmniPool`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [3]

### Description

The `Omnipool` protocol has a core `OmniPool` contract for user interactions. Note operating nodes may be slashed due to poor performance and the slash will be reflected in pool rewards. While examining the related slashing logic, we notice the possibility of avoiding slashing by preemptive unbonding.

To elaborate, we show below the related `slash()` routine. Our analysis shows the slashing in essence adjusts the global state `totalPooledZeta`, which will be used to compute `omZetaToZetaExchangeRate` when making an unbonding request. As a result, it is possible that staking users may monitor any pending transactions in the mempool (before they are mined) and immediately frontrun it to unbond before slashing is applied.

```
    function slash(
        string calldata _validatorAddress,
        uint256 _amount,
        uint256 _time
    ) external override onlyRole(ROLE_SLASHER) {
        require(validator2Time2AmountSlashed[_validatorAddress][_time] == 0, "SLASH_RECORDED");
        require(_amount > 0, "ZERO_AMOUNT");
        // totalPooledzeta cannot go to 0, otherwise convertToShare will not mint the correct share for new stakers
        require(_amount < totalPooledZeta, "amount must be less than totalPooledzeta");

        validator2Time2AmountSlashed[_validatorAddress][_time] = _amount;
        totalPooledZeta -= _amount;

        emit Slash(_validatorAddress, _amount, _time);
    }
```

Listing 3.3: `OmniPool::slash()`

**Recommendation** Apply necessary unbonding fee to discoverage the above-mentioned frontrunning-based unbonding request.

**Status** This issue has been confirmed.

## 3.3 Improved Ether Transfer For DoS Avoidance

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OmniPool
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

### Description

As mentioned in Section 3.1, `Omnipool` provides users with the ability to stake supported tokens and instantly receive a representation of their stake in the form of `omZeta` tokens. It also allows users to unstake and receive their staked funds. The returned funds can be retrieved by calling the `unbond()` routine. While reviewing the implementation of this routine, we notice that the native coin transfer may fail because of a potential `Out-of-Gas` issue.

To elaborate, we show below the code snippet of the `unbond()` routine, which allows to transfer native coins (or `ETH` in `Ethereum`) to the contract owner. We notice that the `unbond()` routine directly calls `transfer()` (line 208) to transfer native coins. However, this `transfer()` method is not recommended to use any more since the `EIP-1884` may increase the gas cost and the 2300 gas limit may be exceeded. There is a helpful blog stop-using-soliditys-transfer-now that explains why the `transfer()` is not recommended any more.

As a result, the `transfer()` may revert and native coins could be locked in the contract. Based on this, we suggest to use the low-level `call()` directly with value attached to transfer native coins.

```
182     function unbond(
183         uint256 _tokenId,
184         address _receiver
185     ) public override nonReentrant whenNotPaused returns (uint256) {
186         require(_receiver != address(0), "ZERO_ADDRESS");
187         require(omniPoolNft.isApprovedOrOwner(msg.sender, _tokenId), "NOT_OWNER");

189         UnbondRequest storage unbondRequest = token2UnbondRequest[_tokenId];
190         require(unbondRequest.unlockEndTime <= block.timestamp, "NOT_UNLOCK_YET");

192         UnbondingStatus status = batch2UnbondingStatus[unbondRequest.batchNo];
193         require(status == UnbondingStatus.UNBONDED, "NOT_UNBONDED_YET");

195         // Burn NFT
196         omniPoolNft.burn(_tokenId);
197         unbondRequests.remove(_tokenId);

199         uint256 totalZetaAmount = (unbondRequest.omZeta * unbondRequest.
                omZetaToZetaExchangeRate) /
200             EXCHANGE_RATE_PRECISION;
```

```
        // Send zeta fee amount to treasury
        uint256 zetaFeeAmount = (totalZetaAmount * unbondingFee) /
            UNBONDING_FEE_DENOMINATOR;
        payable(treasury).transfer(zetaFeeAmount);

        // Send zeta amount to user
        uint256 zetaAmount = totalZetaAmount - zetaFeeAmount;
        payable(_receiver).transfer(zetaAmount);

        emit Unbond(_receiver, _tokenId, zetaAmount, zetaFeeAmount);
        return zetaAmount;
    }
```

Listing 3.4: `OmniPool::unbond()`

**Recommendation** Revisit the above `unbond()` routine to transfer native coins using `call()`.

**Status** This issue has been fixed by the following commit: `d14cbdf`.

## 3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

### Description

The `Omnipool` protocol has a privileged account (with the `DEFAULT_ADMIN_ROLE` role) that plays a critical responsibility in governing and regulating the protocol-wide operations (e.g., configure parameters, assign roles, and execute privileged operations). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setUnbondingFee(uint256 _unbondingFee) external onlyRole(DEFAULT_ADMIN_ROLE
        ) {
        require(_unbondingFee <= 1000, "Fee must be 1% or lower");

        uint256 oldUnbondingFee = unbondingFee;
        unbondingFee = _unbondingFee;
        emit SetUnbondingFee(oldUnbondingFee, unbondingFee);
    }

    function setTreasury(address _treasury) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_treasury != address(0), "EMPTY_ADDRESS");
```

```
        address oldTreasury = treasury;
        treasury = _treasury;
        emit SetTreasury(oldTreasury, treasury);
    }


    function setNFTAddress(OmniPoolNft _poolNFT) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(address(_poolNFT) != address(0), "EMPTY_ADDRESS");
        address oldNFT = address(omniPoolNft);
        omniPoolNft = _poolNFT;
        emit SetNFT(oldNFT, address(_poolNFT));
    }

    /**
     * @dev only called if Crypto org has a new proposal which changes the unbonding duration
     */
    function setUnbondingDuration(uint256 _unbondingDuration) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_unbondingDuration <= 30 days, "_unbondingDuration is too high");

        uint256 oldUnbondingDuration = unbondingDuration;
        unbondingDuration = _unbondingDuration;

        emit SetUnbondingDuration(oldUnbondingDuration, _unbondingDuration);
    }
```

Listing 3.5: Example Privileged Operations in `OmniPool`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

**Recommendation** Promptly transfer the privileged admin role to the intended DAO-like governance contract. And 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 with a multi-sig account to manage the admin role.

# 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Omnipool` protocol, which aims to offer a liquid staking service for `ZETA` tokens and users will be distributed with respective staking rewards. It functions as a native `Liquid-Staking Protocol` comprising two key components: `liquid staking` and `launchpad`. The staking allows users to stake their `ZETA` tokens in return for a liquid stake pool token, known as `omZeta`. Holding `omZeta` offers the dual benefit of staking rewards, providing liquidity, and earning rewards. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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