SMART CONTRACT AUDIT REPORT

for

OTSea & OTSeaERC20

Prepared By: Xiaomi Huang

PeckShield

January 30, 2024

Document Properties

Client	OTSea
Title	Smart Contract Audit Report
Target	OTSea & OTSeaERC20
Version	1.0.1
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
1.0.1	January 30, 2024	Xuxian Jiang	Post Final Release #1
1.0	January 22, 2024	Xuxian Jiang	Final Release
1.0-rc	January 21, 2024	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 smart contract source code of the `OTSea` protocol and related token contracts, 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-documented and well-engineered, and it can benefit from addressing the reported issues. This document outlines our audit results.

1.1 About OTSea

`OTSea` allows users to create over-the-counter buy/sell orders for any token for a specified amount of `ETH`. The protocol charges a fee on the amount of `ETH` traded. The fee in question is dependent on the sellers `$OTSea` holdings which is calculated off-chain. It protects `DeFi` traders from excessive market volatility while offering a simplified peer-to-peer trading experience. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of OTSea & OTSeaERC20

Item	Description
Issuer	OTSea
Website	https://www.otsea.xyz/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 30, 2024

In the following, we show the (private) Git repository name of reviewed files and the commit hash value used in this audit. This audit covers the following contracts: `OTSea.sol`, `OTSeaERC20.sol`, `OTSeaMigration.sol`, `OTSeaStaking.sol`, and `OTSeaRevenueDistributor.sol`.

- otsea-smart-contracts.git (a97865e)

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

- otsea-smart-contracts.git (f3f83f0)

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 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	DeltaPrimeLabs 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 checklist of 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 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 implementation of the `OTSea` protocol and related token 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 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	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 and 1 informational recommendation. We point out that the given repo has included extensive test cases and achieves 100% coverage.

Table 2.1: Key OTSea & OTSeaERC20 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Reward Calculation Logic in OTSeaStaking	Coding Practices	Resolved
PVE-002	Informational	Revisited Initial Liquidity Addition in OTSeaERC20	Business Logic	Resolved
PVE-003	Low	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 Reward Calculation Logic in OTSeaStaking

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OTSeaStaking
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The OTSea protocol has a key OTSeaStaking contract that enables users to stake tokens and earn rewards from token fees and platform revenue. The rewards are calculated pro-rata based on the token amount staked in each epoch. While examining the reward calculation, we notice the current approach can be improved.

To elaborate, we show below the related _calculateRewards() routine that calculates the accumulated rewards by the given _account from the specific deposit _index. It comes to our attention that the reward calculation has an if condition, i.e., if (startingEpoch <= _currentEpoch), and this condition can be revised as if (startingEpoch < _currentEpoch) for improved gas efficiency.

```
    function _calculateRewards(address _account, uint256 _index) private view returns (
        uint256) {
        if (_deposits[_account][_index].lastEpoch != 0) {
            return 0;
        }
        uint32 startingEpoch = _deposits[_account][_index].claimedEpoch != 0
            ? _deposits[_account][_index].claimedEpoch
            : _deposits[_account][_index].firstEpoch;
        if (startingEpoch <= _currentEpoch) {
            return
                (_deposits[_account][_index].amount *
                    (_epochs[_currentEpoch - 1].sharePerToken -
                        _epochs[startingEpoch - 1].sharePerToken)) / REWARD_PRECISION;
        }
        return 0;
```

```
470     }
```

Listing 3.1: OTSeaStaking::_calculateRewards()

In addition, the related `_createDeposit()` helper adds a new `Deposit` element into the user-specific `_deposits` array. And the new `Deposit` can be better initialized as `Deposit(nextEpoch, nextEpoch, 0, _amount)` (line 358). With that, the above `_calculateRewards()` routine can be further improved by simply computing the `startingEpoch` variable as `uint32 startingEpoch = _deposits[_account][_index].claimedEpoch` (lines 460 – 462).

```
356     function _createDeposit(uint256 _amount) private returns (Deposit memory deposit) {
357         uint32 nextEpoch = _currentEpoch + 1;
358         deposit = Deposit(nextEpoch, 0, 0, _amount);
359         _deposits[_msgSender()].push(deposit);
360         _epochs[nextEpoch].totalStake += uint88(_amount);
361         return deposit;
362     }
```

Listing 3.2: OTSeaStaking::_createDeposit()

Recommendation Revise the above-mentioned routines to improve the reward calculation.

Status The issue has been resolved by following the above suggestion.

3.2 Revisited Initial Liquidity Addition in OTSeaERC20

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: OTSeaERC20
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `OTSeaERC20` contract implements an `ERC20`-compliant token that charges specific fees for buy/sell/transfer operations. It also provides a privileged function to add initial liquidity into a `Uniswap V2` pool. Our analysis shows the initial liquidity was provided by the calling owner, which may be revisited by sourcing the `OTSeaERC20` token from the `_migrationContract` contract.

In the following, we show the logic to add initial liquidity via the `addInitialLiquidity()` function. The liquidity was added with two tokens `OTSeaERC20` and `Ether`. Our analysis shows the calling owner directly provides the two tokens, which may be revisited to source `OTSeaERC20` from the migration contract (and the calling owner only provides `Ether`). The reason is that the migration contract holds all `TOTAL_SUPPLY` when the `OTSeaERC20` token contract is instantiated.

```
    function addInitialLiquidity(uint256 _amount) external payable onlyOwner {
        if (_pair != address(0)) revert OTSeaErrors.NotAvailable();
        if (_amount == 0  msg.value == 0) revert OTSeaErrors.InvalidAmount();
        super._update(_msgSender(), address(this), _amount);
        _approve(address(this), address(_router), _amount);
        /// @dev multi-sig admin receives initial LP
        _router.addLiquidityETH{value: msg.value}(
            address(this),
            _amount,
            0,
            0,
            owner(),
            block.timestamp
        );
        address uniswapV2Pair = IUniswapV2Factory(_router.factory()).getPair(
            address(this),
            _router.WETH()
        );
        _pair = uniswapV2Pair;
        emit AddedLiquidity(_pair, _amount, msg.value);
    }
```

Listing 3.3: `OTSeaERC20::addInitialLiquidity()`

Recommendation Revisit the liquidity provider when the initial liquidity is added.

Status The issue has been resolved as it is part of the intended design.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OTSea`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `OTSea` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and contract pause). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contracts.

Specifically, the `owner` account in `OTSea` contract can pause/unpause the contract, decrease the fish and whale fees, update partners, manage blacklisted accounts, as well as set the lock time. The `owner` account in `OTSeaERC20` can add the initial liquidity, adjust swap threshold, and decrease total fee charge. Also, `owner` in `OTSeaStaking`, `OTSeaRevenueDistributor`, and `OTSeaMigration` is rather limited in performing only protocol-necessary functions.

```
    function pauseContract() external onlyOwner {
        _pause();
    }

    /// @notice Unpause the contract
    function unpauseContract() external onlyOwner {
        _unpause();
    }

    /**
     * @notice Set the fish and whale fees
     * @param _newFishFee Fish fee
     * @param _newWhaleFee Whale fee
     * @param _newPartnerFee Partner fee relative to the revenue
     */
    function setFees(
        uint8 _newFishFee,
        uint8 _newWhaleFee,
        uint16 _newPartnerFee
    ) external onlyOwner {
        if (
            _fishFee < _newFishFee
            _whaleFee < _newWhaleFee
            _newFishFee < _newWhaleFee
            _newPartnerFee < MIN_PARTNER_FEE
            MAX_PARTNER_FEE < _newPartnerFee
        ) revert OTSeaErrors.InvalidFee();
        _fishFee = _newFishFee;
        _whaleFee = _newWhaleFee;
        _partnerFee = _newPartnerFee;
        emit FeesUpdated(_newFishFee, _newWhaleFee, _newPartnerFee);
    }

    /**
     * @notice Set the maximum number of trades that can occur in a single TX
     * @param maxTrades_ Max trades
     */
    function setMaxTrades(uint8 maxTrades_) external onlyOwner {
        if (maxTrades_ == 0  MAX_TRADES_UPPER_LIMIT < maxTrades_)
            revert OTSeaErrors.InvalidAmount();
        _maxTrades = maxTrades_;
        emit MaxTradesUpdated(maxTrades_);
    }

    /**
     * @notice Add, remove or update a partner's details
     * @param _token Token address
     * @param _partner Partner details
```

```
     */
    function updatePartner(address _token, Partner calldata _partner) external onlyOwner
         {
        if (_token == address(0)) revert OTSeaErrors.InvalidAddress();
        if (
            _partners[_token].account == _partner.account &&
            _partners[_token].isLockUpOverrideEnabled == _partner.
                isLockUpOverrideEnabled
        ) revert OTSeaErrors.Unchanged();
        if (_partner.account == address(0) && _partner.isLockUpOverrideEnabled)
            revert OTSeaErrors.NotAvailable();
        _partners[_token] = Partner(_partner.account, _partner.isLockUpOverrideEnabled);
        emit PartnerUpdated(_token, _partner);
    }
```

Listing 3.4: Example Privileged Operations in `OTSea`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 addressed as the team clarifies the use of a multisig. Also, it is worth mentioning that if there are tokens that remain to be migrated, the owner can claim them after 90 days.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `OTSea` protocol and related token contracts. The protocol allows users to create over-the-counter buy/sell orders for any token for a specified amount of `ETH`. It protects `DeFi` traders from excessive market volatility while offering a simplified peer-to-peer trading experience. The current code base is well structured and neatly organized. 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-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-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.