# OTSea

Smart Contract Security Assessment

Dec 18, 2023

# ABSTRACT

Dedaub was commissioned to perform a security audit of the OTSea platform contracts. The OTSea Protocol is a peer-to-peer exchange where users can buy ERC20 tokens for ETH or sell ERC20 tokens for ETH without going through a liquidity pool, while enjoying extra benefits such as order discounts and private exchange opportunities.

The audit covered the new version of the platform's main contract, OTSea, which acts as the trusted intermediary between sellers and buyers, and the FeeSplitter contract.

The code and accompanying artifacts (e.g., test suite, documentation) have been developed with high professional standards. No security issues/threats that could lead to theft were identified by the audit. No issues leading to loss of funds resulting from the intended use of the system were identified by the audit.

## SETTING & CAVEATS

The audit report covers commit hash `9288e4e10f32bf730ecc0a630c6fbf66007b6dca` of the at the time private repository otsea-smart-contracts. Audited suggested fixes were also reviewed up to commit hash `b1515946527a35f50881fb832700340447862593`

Two auditors worked on the codebase for 5 days.

The test suite was consulted during the audit but was not part of it. The full list of audited files is:

```
contracts/
├─ fees/
│  └─ FeeSplitter.sol
├─ helpers/
│  ├─ ListHelper.sol
│  ├─ SignatureHelper.sol
│  ├─ TransferHelper.sol
```

```
│   └── WhitelistHelper.sol
├── libraries/
│   ├── OTSeaErrors.sol
│   └── OTSeaLibrary.sol
└── otc/
    └── OTSea.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

## PROTOCOL-LEVEL CONSIDERATIONS

| ID | Description | STATUS |
|---|---|---|
| P1 | Any ERC20 token, including malicious/scam tokens, can be exchanged through the protocol | **INFO** |

The OTSea protocol allows the exchange of any token that implements the ERC20 standard. The OTSea contracts do not impose any other restriction or check on the tokens one could sell or buy. Users of the protocol should be aware of this fact and should not trust but verify that the tokens they are exchanging are legitimate and not malicious or scam tokens, as such a token could arbitrarily change users' balances and its total supply, disallow transfers, transfer incorrect amounts, report incorrect balances, etc. The OTSea team is implementing a warning system on their front-end app to protect users from interacting with such tokens, but as this is an ever changing landscape users should be aware that such a system might not always successfully identify a malicious token. The audit results suggest that an adversary would not be able to use a malicious token to extract funds without the users' consent.

## VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

| Category | Description |
|---|---|
| CRITICAL | Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result. |
| HIGH | Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated. |
| MEDIUM | Examples:<br>• User or system funds can be lost when third-party systems misbehave.<br>• DoS, under specific conditions.<br>• Part of the functionality becomes unusable due to a programming error. |
| LOW | Examples:<br>• Breaking important system invariants but without apparent consequences.<br>• Buggy functionality for trusted users where a workaround exists.<br>• Security issues which may manifest when the system evolves. |

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

## CRITICAL SEVERITY:

[NO CRITICAL SEVERITY ISSUES]

## HIGH SEVERITY:

[NO HIGH SEVERITY ISSUES]

## MEDIUM SEVERITY:

[NO MEDIUM SEVERITY ISSUES]

## LOW SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| L1 | Trade execution can be DOSed | **ACKNOWLEDGED** |

In the case of partial orders it is trivial for an attacker to spend minimal denominations of a currency in order to delay a specific purchase of an order. This is made possible due to the following check in `_executeTrade` function:

```
if (
    order.isAON
        ? _trade.amountToSwap != remainingOutput
        : remainingOutput < _trade.amountToSwap
) revert OTSeaErrors.InvalidPurchase();
```

The attacker would be able to frontrun the user's transaction and decrease the `remainingOutput`, making the user's transaction revert. Although such an attack would be expensive on mainnet and could also be avoided if the user would submit their transaction using a MEV resistant RPC, which is advised in general, it is advisable that this scenario is thought of further on cheaper chains.

| L2 | TransferHelper assumptions | ACKNOWLEDGED |
|---|---|---|

TransferHelper has a `MaroonedETH` mechanism that allows for the salvaging of funds in the case of an incorrect implementation of the receiver, i.e., the receiver function has not implemented a fallback or receive function.

`TransferHelper::_safeETHTransfer:65`

```
function _safeETHTransfer(address _account, uint256 _amount) internal {
    (bool success, ) = _account.call{value: _amount}("");
    if (!success) {
        _maroonedETH[_account] += _amount;
        emit MaroonedETH(_account, _amount);
    }
}
```

Instead of reverting when the low-level call `_account.call{value: _amount}("")` returns false, i.e., when the ETH transfer from the OTSea contract to the `_account` has failed, the ETH amount is stored in the `_maroonedETH` mapping for `_account` to claim later via the `claimMaroonedETH` function. This assumes that the `_account` has implemented functionality that calls `claimMaroonedETH` or is at least able to upgrade their code to do so. However, if this is the case the contract should get upgraded to be able to receive ETH and then re-execute the order, instead of resorting to more complex (and possible error prone) solutions.

At the same time, such an implementation could be considered bad practice as it broadens the attack surface to protect for advanced user errors. A simple issue that can be found is in the FeeSplitter contract where a malicious/compromised owner could set both the teamWallet and dividendWallet to non payable wallets, call `distribute` multiple times inflating the `maroonedEth` value of the team wallet and then switch back to normal wallets, where the protocol would be able to pull a different ratio of funds to the team wallet depending on the fee percentage by calling `claimMaroonedEth`. Considering the current design of the protocol this is a non-issue,

| | | |
|---|---|---|
| however this feature adds unnecessary complexity and it would be advised to just let the contract revert. | | |
| L3 | Certain fee on transfer and rebase tokens might reduce OTSea contract's balance and affect other orders | **ACKNOWLEDGED** |
| The OTSea protocol allows any token, including fee on transfer and rebasing tokens, to be exchanged using its contracts as the trusted intermediary. The protocol handles inclusive fee on transfer tokens appropriately. Exclusive fee on transfer tokens on the other hand might prove problematic as they work by sending an additional transfer from the sending address after the primary transfer. This second transfer would be eating from other orders' funds stored in the contract, potentially leading to a situation where the remaining amount is not sufficient to cover an order's whole input amount. The remaining amount would be available to trade If the order creator has allowed partial order fulfillment. A similar situation could be exhibited for negative rebasing tokens, which would usually deflate the contract's balance at pre-specified intervals. For positive rebasing tokens, token surpluses would be trapped in the contract. | | |

## CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

| ID | Description | STATUS |
|---|---|---|
| N1 | Off-chain signer could get compromised | **ACKNOWLEDGED** |
| The main OTSea contract uses signatures produced by an off-chain signer operated by the team to reliably set a user's fee type and also pass in any seller's discount and referral details. If the signer key were to be compromised by an attacker they would be able to appear as an OTSea whale and enjoy a 0.3% instead of a 1% fee while attributing most of it to themselves by exploiting the referral system. All in all, an attacker would be able to use the exchange and pay less fees (to the protocol) for their trades, but they would not be able to affect other users' orders/funds. | | |
| N2 | Owner's control over the protocol | **ACKNOWLEDGED** |
| Below we describe in detail the control that the owner bears over the protocol:<br>• Pause/unpause order creation, execution, updating but not cancellation. The owner can also pause/unpause updates to whitelists.<br>• Set the fish and whale fees to values only less than the initial, which are 1% and 0.3% respectively.<br>• Set the team fee and the team and dividends wallet. | | |

- Set the maximum amount of trades that are allowed to be executed during a single transaction.

## OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

| ID | Description | STATUS |
|---|---|---|
| A1 | _checkSequence checks can be simplified | **INFO** |

The checks implemented by function `_checkSequence` can be simplified from:

```
if (_start == 0 || _start > _total) revert InvalidStartID();
if (_end == 0) revert InvalidEndID();
if (_start > _end) revert InvalidStartID();
if (_end - _start + 1 > LOOP_LIMIT) revert InvalidSequence();
if (_end > _total) revert InvalidEndID();
```

to:

```
if (_start == 0) revert InvalidStartID();
if (_start > _end) revert InvalidStartID();
if (_end - _start + 1 > LOOP_LIMIT) revert InvalidSequence();
if (_end > _total) revert InvalidEndID();
```

as requiring `_end <= _total` and `_start <= _end` implies `_start <= _total`, while `start <= _end` and `_start != 0` implies `_end != 0`.

| ID | Description | STATUS |
|---|---|---|
| A2 | cancelOrders::totalTokensOwed is used as an accumulator of amounts of different ERC20 tokens | **FIXED** |

In the `cancelOrders` function the variable `totalTokensOwed` takes account of all ERC20 tokens being transferred back to the order creator. This is misleading in the logs as it does not accurately tell the log consumer which tokens have been transferred

| | | |
|---|---|---|
| back, since the variable aggregates all different token orders being canceled. The more consistent approach would be to emit separate events for each order canceled. | | |
| A3 | Whitelists' size can exceed LOOP_LIMIT | **INFO** |
| The abstract contract ListHelper defines the constant `LOOP_LIMIT` that is used by the WhitelistHelper contract to enforce an upper limit on the loop iterations performed by its functions. Thus, one could assume that the maximum size of a whitelist cannot exceed `LOOP_LIMIT`, which is not true as the function `_updateWhitelist` can be called an arbitrary amount of times and each time increase the size of a whitelist by `LOOP_LIMIT`. Nevertheless, there is no function in the current codebase that will iterate over a whole whitelist, meaning that arbitrarily increasing the size of whitelists cannot be leveraged to cause a DOS attack. Still, we think developers of the protocol should be aware of this scenario when working on future versions of the protocol. | | |
| A4 | The ListHelper contract could be made abstract | **FIXED** |
| The ListHelper contract could be made abstract, as its purpose is to serve as a parent contract. | | |
| A5 | BuyOrderCreated events convey less information compared to SellOrderCreated events | **INFO** |
| The `BuyOrderCreated` events do not store the order's `totalInput` and `totalOutput` amounts. On the other hand, the `SellOrderCreated` events store the `totalInput` and `totalOutput` amounts after the amounts have been updated to account for any transfer fees applied by the token. Thus, `SellOrderCreated` events will store `totalInput` and `totalOutput` amounts after fees if any, while `BuyOrderCreated` events completely ignore this info (due to the fact that no tokens fee are applied during a buy order creation) when there could be event consumers that would be interested in them. | | |
| A6 | Further strengthening the OTSea contract against reentrancy attacks | **INFO** |

The current version of the OTSea contract is not vulnerable to reentrancy attacks. Nevertheless, several defensive oriented improvements could be made to minimize the possibility of a reentrancy vulnerability getting introduced in future versions of the protocol.

1. In functions `_executeBuy` and `_executeSell`, the instructions updating `totalAmountToSwap` and `totalAmountToReceive` should be moved before the calls `_handleETHPayment` and `safeTransferFrom`.
2. Function updatePrice could be made non-reentrant even though only an order's creator can call it for their order and they cannot change the `totalInput` amount or extract any of the input, they can only change the `totalOutput` (or in other words requested) amount. This means that at the moment an order creator can make their order cheaper or more expensive but they cannot gain anything by fulfilling it.
3. Function `TransferHelper::claimMaroonedETH` should be made internal. The OTSea contract could then implement a non-reentrant `claimMaroonedETH` function that calls `TransferHelper::_claimMaroonedETH`.
4. Function `createBuyOrder` could be made non-reentrant.

| A7 | Canceling an order does not update the inputTransacted amount | **INFO** |
|---|---|---|

Canceling an order sets its `state` to `Canceled` but does not update the `inputTransacted` field. Technically, no input is transacted but `totalInput - inputTransacted` tokens are transferred out of the contract and are no longer available to be traded for this order. Thus, we would advise to set the `inputTransacted` field to `totalInput`.

# DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

# ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.