Cyberscope

# Audit Report

# OTSea

October 2023

SHA256 8cd5e5b7ea98ae0586e916efe8a87bbdfd9e473973fa9aa4d06963106d6158ab

Audited by © cyberscope

# Table of Contents

# Review

| | |
|---|---|
| **Testing Deploy** | https://testnet.bscscan.com/address/0x85500390bc38892fcf5825754e0410bd71eb8442 |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 05 Oct 2023 |

## Source Files

| Filename | SHA256 |
|---|---|
| **contracts/console.sol** | 3a470828b181406daeb23a44c353c09d270172686f4e0813f94a51c15a549cfa |
| **contracts/OTSea.sol** | 8cd5e5b7ea98ae0586e916efe8a87bbdfd9e473973fa9aa4d06963106d6158ab |
| **@openzeppelin/contracts/utils/Context.sol** | 1458c260d010a08e4c20a4a517882259a23a4baa0b5bd9add9fb6d6a1549814a |
| **@openzeppelin/contracts/token/ERC20/IERC20.sol** | 7ebde70853ccafcf1876900dad458f46eb9444d591d39bfc58e952e2582f5587 |
| **@openzeppelin/contracts/token/ERC20/ERC20.sol** | d20d52b4be98738b8aa52b5bb0f88943f62128969b33d654fbca731539a7fe0a |
| **@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol** | af5c8a77965cc82c33b7ff844deb9826166689e55dc037a7f2f790d057811990 |
| **@openzeppelin/contracts/security/ReentrancyGuard.sol** | fa97ea556c990ee44f2ef4c80d4ef7d0af3f5f9b33a02142911140688106f5a9 |
| **@openzeppelin/contracts/access/Ownable.sol** | a8e4e1ae19d9bd3e8b0a6d46577eec098c01fbaffd3ec1252fd20d799e73393b |

# Overview

The OTSea smart contract is a decentralized exchange (DEX) designed to facilitate the trading of ERC-20 tokens for Ether (ETH). It includes various features to support token swaps, fee distribution, and order management. Users can create orders by specifying the token they want to trade, the amount of tokens they offer, and the amount of ETH they request in return. Orders can be partially fillable, and the contract calculates the price per token based on these parameters. Users can fulfil orders by sending ETH to the contract, and tokens are transferred in exchange based on the specified price. The contract also handles fee distribution, with fees going to different wallets, including marketing and operational wallets. Orders can be updated, and the contract can be paused or resumed by the owner. It provides a flexible and customizable DEX solution for Ethereum-based tokens.

## Roles

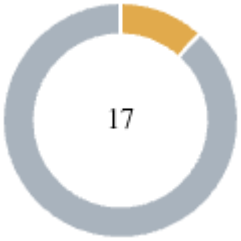
### Owner

- Can pause and resume the contract, which can prevent trading activities.
- Can set the fees for fish (max 1%) and whale (max 0.3%).
- Can set the whale threshold, which determines the fee percentage for users based on their token holdings.
- Can update the operational wallets and wallets for marketing and dividends.

### User

- Can create new orders by specifying token, ETH, and other order details.
- Can update the price per token for their orders (if partially fillable or all-or-nothing).
- Can fulfill open orders by sending the required ETH.
- Can receive tokens in exchange for their ETH based on the specified price per token.

# Findings Breakdown




| | |
|---|---|
| Critical | 0 |
| Medium | 2 |
| Minor / Informative | 15 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|
| Critical | 0 | 0 | 0 | 0 |
| Medium | 2 | 0 | 0 | 0 |
| Minor / Informative | 15 | 0 | 0 | 0 |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| Medium | PUO | Possible Unfulfilled Order | Unresolved |
| Medium | WTCI | Whale Threshold Cap Inconsistency | Unresolved |
| Minor / Informative | CI | Calculation Inconsistency | Unresolved |
| Minor / Informative | PZP | Possible Zero Price | Unresolved |
| Minor / Informative | GO | Gas Optimization | Unresolved |
| Minor / Informative | OO | Operator Optimization | Unresolved |
| Minor / Informative | VTO | Variable Type Optimization | Unresolved |
| Minor / Informative | MU | Modifiers Usage | Unresolved |
| Minor / Informative | MEE | Missing Events Emission | Unresolved |
| Minor / Informative | PTRP | Potential Transfer Revert Propagation | Unresolved |
| Minor / Informative | RSW | Redundant Storage Writes | Unresolved |
| Minor / Informative | IDI | Immutable Declaration Improvement | Unresolved |
| Minor / Informative | L04 | Conformance to Solidity Naming Conventions | Unresolved |
| Minor / Informative | L11 | Unnecessary Boolean equality | Unresolved |

| | | | |
|---|---|---|---|
| ● | L13 | Divide before Multiply Operation | Unresolved |
| ● | L16 | Validate Variable Setters | Unresolved |
| ● | L19 | Stable Compiler Version | Unresolved |

# PUO - Possible Unfulfilled Order

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | contracts/OTSea.sol#L197,205 |
| **Status** | Unresolved |

## Description

The contract exhibits a potential problem involving partially the fillable orders that might remain unfulfilled. This situation arises when the `pricePerToken` exceeds the result of the expression `msg.value * 10 ** ERC20(order.tokenAddress).decimals()`, resulting in a division outcome of zero. Furthermore, if the `tokensToFulfill` exceeds the available tokens in the order, the transaction will fail. As a result the order will not be fulfilled.

```
tokensToFulfill =
    (msg.value * 10 ** ERC20(order.tokenAddress).decimals()) /
    order.pricePerToken;

require(tokensToFulfill > 0, "Token amount must be greater than 0");
require(tokensToFulfill <= order.availableTokens, "Exceeds available
tokens to fulfill");
```

## Recommendation

To prevent partially fillable orders from remaining unfulfilled, the team should consider implementing additional checks and logic to handle these scenarios. By adding these additional checks, the team will ensure that partially fillable orders are handled gracefully, and users can fulfil these orders as expected.

## WTCI - Whale Threshold Cap Inconsistency

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | contracts/OTSea.sol#L81,88 |
| **Status** | Unresolved |

## Description

In the contract's constructor, the `whaleThreshold` variable is initialized with a value that is significantly higher than the intended threshold of 1% of the total supply of the OTSea ERC20 token. The expression `(2 * otseaERC20.totalSupply()) / 1000` already contains the decimal points, hence multiplying the value with `1e18` the contract effectively doubles the decimal places. This deviation from the desired threshold could potentially lead to unintended consequences, such as failing to identify certain addresses as whales when they should be.

```
whaleThreshold = ((2 * otseaERC20.totalSupply()) / 1000) * 1e18;
require(
    _threshold <= otseaERC20.totalSupply() / 100,
    "Whale threshold can't be higher than 1%"
);
```

## Recommendation

To align with the intended functionality of capping the whale threshold at 1% of the total supply, the initialization of the `whaleThreshold` variable in the constructor should be adjusted accordingly. Ensuring that the `whaleThreshold` is set accurately from the start will help the contract correctly identify and manage whale addresses as intended.

## CI - Calculation Inconsistency

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L306 |
| **Status** | Unresolved |

### Description

The contract exhibits inconsistency in the calculation of the `requestedETH` variable. In the requestOrder function, requestedETH is equal to `pricePerToken * formattedTransferredTokenAmount`, while in the updatePrice function, it is equal `order.fulfilledETH + (formattedAvailableTokens * newPrice)`. As a result, the calculation in the updatePrice function does not align with the one in the requestOrder function.

```
// Calculate the adjusted requestedETH by multiplying it by the net %
order.pricePerToken = order.requestedETH /
formattedTransferredTokenAmount;

order.requestedETH = order.fulfilledETH + (formattedAvailableTokens *
newPrice);
```

### Recommendation

To ensure consistency and clarity in the calculation of the `requestedETH` variable, the team is advised to use a consistent approach throughout the contract. The team should choose one calculation method and apply it consistently across all relevant functions. This will help prevent confusion and potential errors in the contract's logic.

## PZP - Possible Zero Price

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L170 |
| **Status** | Unresolved |

### Description

The contract calculates the `pricePerToken` based on the `requestedETH` value and the `formattedTransferredTokenAmount`. However, if the formattedTransferredTokenAmount is greater than the requestedETH value, the division operation will result in zero. This can lead to division by zero errors when a user tries to fulfill an order, potentially causing issues with order fulfillment.

```
order.pricePerToken = order.requestedETH /
formattedTransferredTokenAmount;
```

### Recommendation

To prevent division by zero errors and ensure that order fulfillment can proceed smoothly, the team is advised to check if the `formattedTransferredTokenAmount` is greater than the `requestedETH`, and if so, handle the case appropriately.

# GO - Gas Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L119 |
| **Status** | Unresolved |

## Description

Gas optimization refers to the process of reducing the amount of gas required to execute a transaction. Gas is the unit of measurement used to calculate the fees paid to miners for including a transaction in a block on the blockchain.

In the `requestOrder` function, the contract generates a unique order identifier using both the "OTSea" string and a nonce. However, since the hashed value is only used as an identifier, the contract can optimize gas usage by using only the nonce as the identifier, eliminating the need to generate a hash.

```
bytes32 orderId = keccak256(abi.encodePacked("OTSea", ++nonce));
```

## Recommendation

To optimize gas usage in the `requestOrder` function, the team shoudl consider using only the nonce as the unique order identifier instead of generating a hash with the "OTSea" string. This change can help reduce gas consumption for generating order identifiers, as hashing operations can consume significant gas. By simplifying the identifier generation process, the contract can operate more efficiently, resulting in lower transaction costs for users.

# OO - Operator Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L116,117,189,205 |
| **Status** | Unresolved |

## Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

In the contract, there is a requirement check on certain variables, which are declared as unsigned integers. The condition checks if these variables are greater than zero. However, since the variables are unsigned integers (uint), their value is always greater than or equal to zero by default. Therefore, using the ">" operator can be optimised, by replacing it with the "!=" operator.

```
require(requestedETHAmount > 0, "Requested ETH amount must be greater than
0");
require(requesterTokenAmount > 0, "Token amount must be greater than 0");
require(msg.value > 0, "ETH amount must be greater than 0");
require(tokensToFulfill > 0, "Token amount must be greater than 0");
```

## Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

# VTO - Variable Type Optimization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L159,60,62,59,160,255 |
| **Status** | Unresolved |

## Description

The contract currently declares certain variables as uint256, even though their maximum value is capped at a much lower amount than the `type(uint256).max`. These variables can be more efficiently stored with smaller integer types to save storage space and reduce gas consumption. As a result, the contracts reserves unnecessary storage space and consumes more gas when using these variables.

For instance, the `netTransferPercent` and `transferTax` variables are declared as uint256, while their value varies between 0 and 10000.

```
uint256 public fishFee = 100; // 1%
uint256 public whaleFee = 30; // 0.3%
uint256 netTransferPercent = (transferredTokenAmount * 10000) /
requesterTokenAmount;
uint256 transferTax = 10000 - netTransferPercent;
uint256 feePercentage = otseaERC20.balanceOf(order.requester) >=
whaleThreshold
    ? whaleFee
    : fishFee;
```

## Recommendation

To optimize the smart contract and improve resource utilization, it is strongly recommended to review and update the variable types used. Specifically, consider changing variables currently declared as `uint256` to smaller integer types wherever applicable, given that the maximum value they store does not need a uint256. This adjustment aligns the variable types more closely with the actual data requirements, reducing unnecessary gas costs and optimizing storage efficiency.

# MU - Modifiers Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L232,299 |
| **Status** | Unresolved |

## Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(order.requester == msg.sender, "Not authorized");
require(msg.sender == order.requester, "Not authorized");
```

## Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

## MEE - Missing Events Emission

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L65,66,91,318,323,328,333,338,343 |
| **Status** | Unresolved |

### Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
fishFee = _fishFee;
whaleFee = _whaleFee;
whaleThreshold = _threshold;
contractState = ContractState.Paused;
contractState = ContractState.Active;
opWallet1 = _opWallet1;
opWallet2 = _opWallet2;
dividendsWallet = _dividendsWallet;
marketingWallet = _marketingWallet;
```

### Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

# PTRP - Potential Transfer Revert Propagation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L272,275,278,281 |
| **Status** | Unresolved |

## Description

The contract sends funds to certain wallets as part of the order settling flow. These addresses can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the contract and revert the transaction.

```
(bool successMarketing, ) = marketingWallet.call{value: marketingFee}("");
require(successMarketing, "Marketing Wallet - ETH transfer failed");

(bool success1, ) = opWallet1.call{value: op1Fee}("");
require(success1, "Operations Wallet 1 - ETH transfer failed");

(bool success2, ) = opWallet2.call{value: op2Fee}("");
require(success2, "Operation Wallet 2 - ETH transfer failed");

(bool success3, ) = dividendsWallet.call{value: dividendsFee}("");
require(success3, "Dividends wallet - ETH transfer failed");
```

## Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

## RSW - Redundant Storage Writes

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L125,318,323 |
| **Status** | Unresolved |

## Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The `whitelistedAddress` is optional. However, the contract modifies its state even when the provided value is the zero address.

```
order.whitelistedAddress = whitelistedAddress;

// If there's a whitelisted address, ensure it is the sender
if (order.whitelistedAddress != address(0)) {
    require(msg.sender == order.whitelistedAddress, "Not authorized");
}
```

Additionally, the contract modifies the state of certain variables even when their current value matches the provided argument. As a result, the contract performs redundant storage writes.

```
contractState = ContractState.Paused;
contractState = ContractState.Active;
```

## Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

## IDI - Immutable Declaration Improvement

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L80 |
| **Status** | Unresolved |

### Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
otseaERC20
```

### Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L62,86,326,331,336,341 |
| **Status** | Unresolved |

### Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _whaleFee
uint256 _fishFee
uint256 _threshold
address payable _opWallet1
address payable _opWallet2
address payable _dividendsWallet
address payable _marketingWallet
```

## Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

## L11 - Unnecessary Boolean equality

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L192 |
| **Status** | Unresolved |

### Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
order.partiallyFillable == false
```

### Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

## L13 - Divide before Multiply Operation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L81,159,164,301,306 |
| **Status** | Unresolved |

### Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 formattedAvailableTokens = order.availableTokens /
            10 ** ERC20(order.tokenAddress).decimals()
order.requestedETH = order.fulfilledETH + (formattedAvailableTokens *
newPrice)
```

### Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

## L16 - Validate Variable Setters

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L76,77,78,79,328,333,338,343 |
| **Status** | Unresolved |

### Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
opWallet1 = _opWallet1
opWallet2 = _opWallet2
dividendsWallet = _dividendsWallet
marketingWallet = _marketingWallet
```

### Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

## L19 - Stable Compiler Version

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/OTSea.sol#L2 |
| **Status** | Unresolved |

### Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.8;
```

### Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

# Functions Analysis

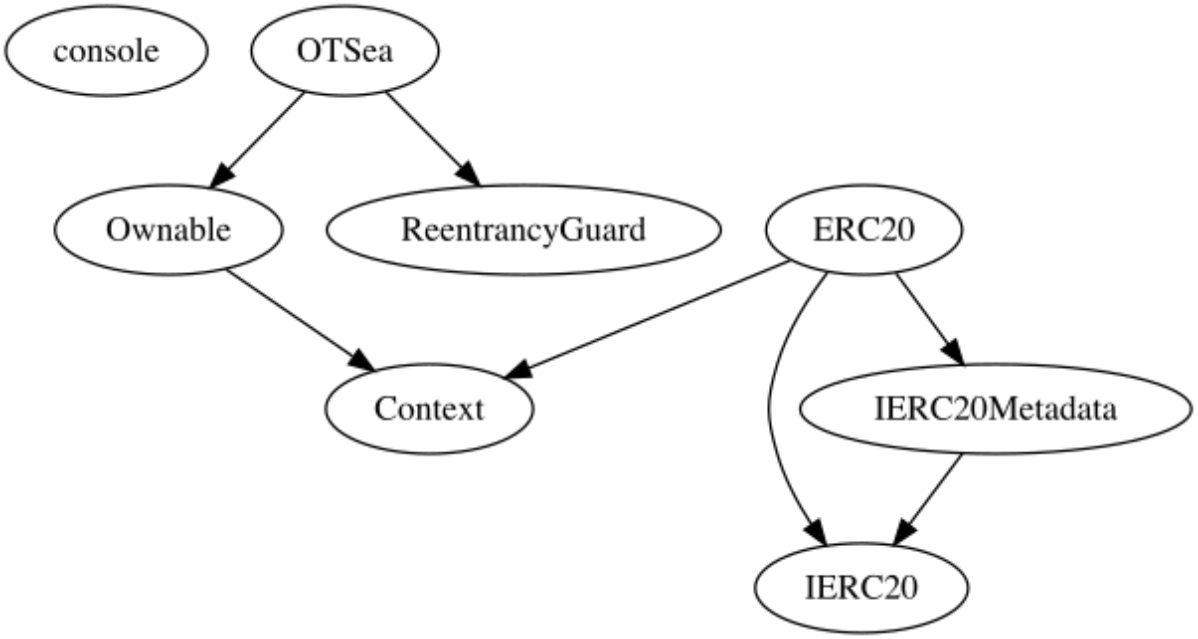
| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **OTSea** | Implementation | Ownable, ReentrancyGuard | | |
| | setFees | External | ✓ | onlyOwner |
| | | Public | ✓ | - |
| | setWhaleThreshold | External | ✓ | onlyOwner |
| | requestOrder | External | ✓ | nonReentrant whenNotPaused |
| | fulfillOrder | External | Payable | nonReentrant whenNotPaused |
| | settleOrder | External | ✓ | nonReentrant |
| | updatePrice | External | ✓ | nonReentrant whenNotPaused |
| | pauseContract | External | ✓ | onlyOwner |
| | unpauseContract | External | ✓ | onlyOwner |
| | setOpWallet1 | External | ✓ | - |
| | setOpWallet2 | External | ✓ | - |
| | setDividendsWallet | External | ✓ | - |
| | setMarketingWallet | External | ✓ | - |
| | | | | |
| **Context** | Implementation | | | |

| | | | | |
|---|---|---|---|---|
| | _msgSender | Internal | | |
| | _msgData | Internal | | |
| | | | | |
| **IERC20** | Interface | | | |
| | totalSupply | External | | - |
| | balanceOf | External | | - |
| | transfer | External | ✓ | - |
| | allowance | External | | - |
| | approve | External | ✓ | - |
| | transferFrom | External | ✓ | - |
| | | | | |
| **ERC20** | Implementation | Context, IERC20, IERC20Metadata | | |
| | | Public | ✓ | - |
| | name | Public | | - |
| | symbol | Public | | - |
| | decimals | Public | | - |
| | totalSupply | Public | | - |
| | balanceOf | Public | | - |
| | transfer | Public | ✓ | - |
| | allowance | Public | | - |
| | approve | Public | ✓ | - |
| | transferFrom | Public | ✓ | - |

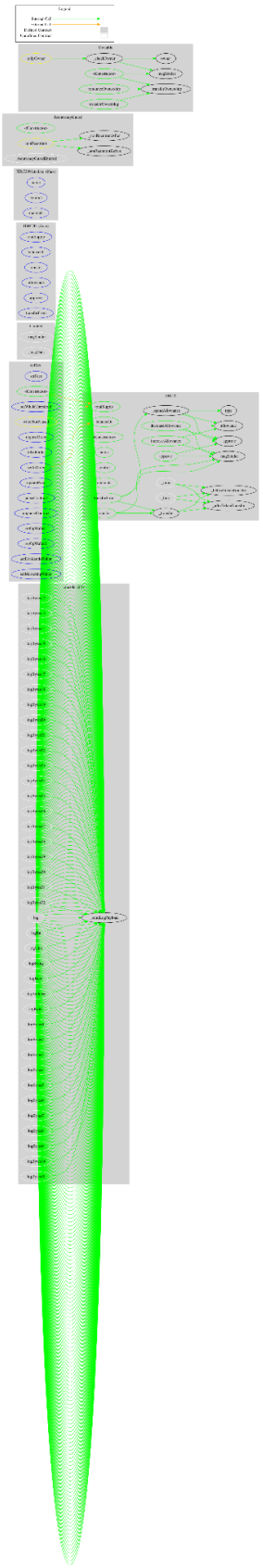
| | | | | |
|---|---|---|---|---|
| | increaseAllowance | Public | ✓ | - |
| | decreaseAllowance | Public | ✓ | - |
| | _transfer | Internal | ✓ | |
| | _mint | Internal | ✓ | |
| | _burn | Internal | ✓ | |
| | _approve | Internal | ✓ | |
| | _spendAllowance | Internal | ✓ | |
| | _beforeTokenTransfer | Internal | ✓ | |
| | _afterTokenTransfer | Internal | ✓ | |
| | | | | |
| **IERC20Metadata** | Interface | IERC20 | | |
| | name | External | | - |
| | symbol | External | | - |
| | decimals | External | | - |
| | | | | |
| **ReentrancyGuard** | Implementation | | | |
| | | Public | ✓ | - |
| | _nonReentrantBefore | Private | ✓ | |
| | _nonReentrantAfter | Private | ✓ | |
| | _reentrancyGuardEntered | Internal | | |
| | | | | |
| **Ownable** | Implementation | Context | | |
| | | Public | ✓ | - |

| | | | | |
|---|---|---|---|---|
| | owner | Public | | - |
| | _checkOwner | Internal | | |
| | renounceOwnership | Public | ✓ | onlyOwner |
| | transferOwnership | Public | ✓ | onlyOwner |
| | _transferOwnership | Internal | ✓ | |

# Inheritance Graph



# Flow Graph



# Summary

OTSea contract implements a financial mechanism. This audit investigates security issues, business logic concerns and potential improvements.

# Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

# About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

**The Cyberscope team**

https://www.cyberscope.io