Cyberscope

Audit Report

Trait Exchange

October 2023

SHA256 2fae38acf7444f7b78efb99ed94573a5c2525536e7af1f03943cf0696333d793

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x5191cc15fe7500f7aae3d4cd67716ef367a230c8

Audit Updates

Initial Audit	03 Oct 2023

Source Files

Filename	SHA256
contracts/Trait.sol	2fae38acf7444f7b78efb99ed94573a5c2525536e7af1f03943cf0696333d793
@openzeppelin/contracts/utils/Strings.sol	cb2df477077a5963ab50a52768cb74ec6f32177177a78611ddbbe2c07e2d36de
@openzeppelin/contracts/utils/Context.sol	1458c260d010a08e4c20a4a517882259a23a4baa0b5bd9add9fb6d6a1549814a
@openzeppelin/contracts/utils/Address.sol	8b85a2463eda119c2f42c34fa3d942b61aee65df381f48ed436fe8edb3a7d602
@openzeppelin/contracts/utils/math/SignedMath.sol	420a5a5d8d94611a04b39d6cf5f02492552ed4257ea82aba3c765b1ad52f77f6
@openzeppelin/contracts/utils/math/Math.sol	85a2caf3bd06579fb55236398c1321e15fd524a8fe140dff748c0f73d7a52345
@openzeppelin/contracts/utils/introspection/IERC165.sol	701e025d13ec6be09ae892eb029cd83b3064325801d73654847a5fb11c58b1e5
@openzeppelin/contracts/utils/introspection/ERC165.sol	8806a632d7b656cadb8133ff8f2acae4405b3a64d8709d93b0fa6a216a8a6154

@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol	77f0f7340c2da6bb9edbc90ab6e7d3eb8e2ae18194791b827a3e8c0b11a09b43
@openzeppelin/contracts/token/ERC721/IERC721.sol	c8d867eda0fd764890040a3644f5ccf5db92f852779879f321ab3ad8b799bf97
@openzeppelin/contracts/token/ERC721/ERC721.sol	7af3ff063370acb5e1f1a2aab125ceca457cd1fa60ff8afa37aabc366349d286
@openzeppelin/contracts/token/ERC721/extensions/IERC721Metadata.sol	f16b861aa1f623ccc5e173f1a82d8cf45b678a7fb81e05478fd17eb2ccb7b37e
@openzeppelin/contracts/token/ERC20/IERC20.sol	7ebde70853ccafcf1876900dad458f46eb9444d591d39bfc58e952e2582f5587
@openzeppelin/contracts/access/Ownable.sol	a8e4e1ae19d9bd3e8b0a6d46577eec098c01fbaffd3ec1252fd20d799e73393b

Overview

The `Trait` contract facilitates a decentralized marketplace for trading assets on the blockchain. Users can create offers, specifying the assets they wish to exchange, including native token (i.e. ETH), ERC20 tokens, and ERC721 tokens (NFTs). The contract handles the transfer of these assets between parties, ensuring that the terms of the offer are met. Key features include the ability to offer multiple types of assets in a single transaction, validation checks to ensure the correct amount of ETH is sent, and mechanisms for offer management, such as rejection and withdrawal. The contract also incorporates fee handling, with certain users who are NFT holders of a specific collection being excluded from these fees. The design aims to provide a secure and efficient platform for peer-to-peer asset trading.

Create Offer Functionality

The `createOffer` function allows users to create offers on the Trait contract. This function is designed to facilitate the exchange of assets between users. Here's a breakdown of its functionality:

Parameters Specification: Users specify the parameters of the offer they want to create. These parameters include:

- The recipient of the offer (_receiver).
- The amount of Ethereum they are offering (_offeredETH).
- The amount of Ethereum they are requesting in return (_requestedETH).
- The ERC20 tokens they are offering (_offeredERC20).
- The ERC20 tokens they are requesting in return (_requestedERC20).
- The NFTs (Non-Fungible Tokens) they are offering (_offeredERC721).
- The NFTs they are requesting in return (_requestedERC721).
- The duration for which the offer is valid (_offerValidDuration).

The `createOffer` function authorizes users to create offers within the smart contract. When invoking this function, users specify the parameters of their offer, including the desired NFT, ERC20 tokens, and ETH amount. Concurrently, they also specify the assets they are willing to offer in exchange, which could be an NFT, ERC20 tokens, or a certain ETH value. These offered assets are then transferred from the user's account (msgSender)

to the contract. If a user isn't a holder of an NFT that is listed in the excluded from fees contracts, they are obligated to pay an additional fee ammount. Upon successful validation and fee payment, the offer is officially registered in the contract, with both the proposed and requested assets recorded.

Accept Offer Functionality

The `acceptOffer` function of the Trait contract, enables users to accept offers made by other participants. Upon invoking the `acceptOffer` function, users are required to provide a valid `_offerId` which is then checked against the existing offers to ensure its validity. The function ensures that the individual accepting the offer (`msgSender`) is indeed the intended receiver of the offer. Additionally, it verifies that the ETH sent by the receiver matches or exceeds the requested ETH amount in the offer. A time check confirms that the offer hasn't expired based on its creation timestamp and valid duration.

Exchange of assets:

The function facilitates the transfer of offered ERC721 tokens from the contract to the receiver. Conversely, the requested ERC721 tokens are transferred from the receiver to the offer creator. If there's any ETH offered, it's directly transferred to the receiver's account. Similarly, any requested ETH is sent to the offer creator's account. For ERC20 tokens, if any are requested in the offer, they are transferred from the receiver to the offer creator. Conversely, if any ERC20 tokens are offered, they are transferred to the receiver.

Once all asset transfers are successfully executed, the status of the offer is updated to `accepted` , and an event is emitted to log this change in offer status. Throughout the process, it's imperative for users to ensure they've set adequate allowances, to avoid any transaction failures.

Reject Offer Functionality

The `rejectOffer` function give the ability to users to decline offers.

When a user decides to invoke the `rejectOffer` function, they must provide a valid `_offerId`. This ID is then cross-referenced with the existing offers in the contract to confirm its authenticity. The function strictly ensures that the individual rejecting the offer (msgSender) is the intended receiver of that particular offer in order to prevent unauthorized rejections.

Upon successful validation, all the offered ERC721 tokens are returned from the contract back to the original offer creator. If there's any ETH that was part of the offer, it's transferred back to the offer creator's account. For ERC20 tokens, if any were included in the offer, they are also returned to the offer creator. The transfer is executed only if the ERC20 token contract address is valid and the offered ERC20 token value is greater than zero. Once all assets are reverted back to the offer creator, the status of the offer is updated to `rejected`.

Withdraw Offer Functionality

The `withdrawOffer` function, grants users the ability to retract offers they've previously made. Its operation mirrors the `rejectOffer` function, but with a distinct difference in the user's role who invoking the function.

When a user wishes to withdraw an offer, they initiate the `withdrawOffer` function by providing the relevant `_offerId`. This ID is then matched with the existing offers in the contract to ascertain its validity. A pivotal aspect of this function is that only the original creator of the offer (msgSender) can execute the withdrawal. This safeguard ensures that only authorized users can retract offers, preventing potential misuse.

The method of returning assets to the offer creator's account mirrors the procedures in the `rejectOffer` function. Essentially, it conducts identical asset restitution actions but is executed from the standpoint of the offer creator instead of the receiver.

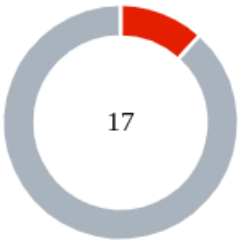
Owner Functionalities

The owner of the contract has the ability to set specific NFT contract addresses that are exempted from incurring exchange fees. This is achieved through the `excludeFromExchangeFees` function, where the owner can input a contract address to be exempted, ensuring that holders of the NFTs from that particular contract are not burdened with additional fees.

Additionally, the owner can invoke the `setFees` function to modify the fee amount, ensuring flexibility in fee management based on evolving requirements or market conditions.

When the owner decides to claim the accumulated fees, the `claimFees` function facilitates this process. It calculates the pending fees, updates the claimed fees record, and transfers the pending amount directly to the owner's account. This action is then transparently recorded through the `FeesClaimedByAdmin` event, ensuring traceability and transparency in the contract's financial operations.

Findings Breakdown



Critical	2
Medium	0
Minor / Informative	15

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	PRE	Potential Reentrance Exploit	Unresolved
●	MVV	Missing Value Validation	Unresolved
●	RMU	Redundant Mapping Usage	Unresolved
●	RFGO	Reject Functionality Gas Optimization	Unresolved
●	MRM	Missing Revert Messages	Unresolved
●	CR	Code Repetition	Unresolved
●	MU	Modifiers Usage	Unresolved
●	CO	Code Optimization	Unresolved
●	DTI	Data Type Inconsistency	Unresolved
●	MC	Missing Check	Unresolved
●	RM	Redundant Modifier	Unresolved
●	RED	Redudant Event Declaration	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved

●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

PRE - Potential Reentrance Exploit

Criticality	Critical
Location	contracts/Trait.sol#L205,256,294
Status	Unresolved

Description

The contract makes an external call to transfer funds to recipients using the payable transfer method. The recipient could be a malicious contract that has an untrusted code in its fallback function that makes a recursive call back to the original contract. The re-entrance exploit could be used by a malicious user to drain the contract’s funds or to perform unauthorized actions. This could happen because the original contract does not update the state before sending funds.

```
    function acceptOffer(uint256 _offerId)
        external
        payable
        isValidOffer(_offerId)
    {
    ...
    if (offerAccount.offeredETH > 0) {

payable(offerAccount.receiver).transfer(offerAccount.offeredETH
);
    }

    if (offerAccount.requestedETH > 0) {

payable(offerAccount.sender).transfer(offerAccount.requestedETH
);
    }
    ...
    }

    function rejectOffer(uint256 _offerId) external
isValidOffer(_offerId) {
    ...
    if (offerAccount.offeredETH > 0) {

payable(offerAccount.sender).transfer(offerAccount.offeredETH);
    }
    ...
    }

    function withdrawOffer(uint256 _offerId) external
isValidOffer(_offerId) {
    ...
    if (offerAccount.offeredETH > 0) {

payable(offerAccount.sender).transfer(offerAccount.offeredETH);
    }
    ...
    }
```

Recommendation

The team is advised to prevent the potential re-entrance exploit as part of the solidity best practices. A suggested implementation is to proceed with the external call as the last statement of the method, so that the state will have been updated properly during the re-entrance phase.

Some additional suggestions are:

- Add lockers/mutexes in the method scope. It is important to note that mutexes do not prevent cross-function reentrancy attacks.
- Do Not allow contract addresses to receive funds.

MVV - Missing Value Validation

Criticality	Critical
Location	contracts/Trait.sol#L77,131
Status	Unresolved

Description

The contract is designed to handle the creation of offers, where users can specify an amount of ETH (`_offeredETH`) they wish to offer. Within the contract, there's an if statement that checks whether the `msgSender` is excluded from additional fees. If they are not `ExcludedFromFees` , the contract requires the `msgValue` (the amount of ETH sent with the transaction) to be greater than or equal to the sum of `_fees` and `_offeredETH` . However, if the `msgSender` is excluded from fees, the contract does not check to ensure that the `msgValue` is at least equal to `_offeredETH` . This oversight means that a user could potentially create an offer without actually sending the specified `_offeredETH` amount to the contract.

Additionally, the contract is using the `isValidOffer` modifier to check that the `receiver` address is not the zero address. However, the `createOffer` function does not prevent users from passing the `_receiver` address as the zero address (address(0)). As a result, if a zero address is passed through the `createOffer` function, then that offer will be stuck and will be unable to be accepted, rejected, or withdrawn since all these functions require the `isValidOffer` modifier.

```
function createOffer(
        address _receiver,
        uint256 _offeredETH,
        uint256 _requestedETH,
        StructERC20Value memory _offeredERC20,
        StructERC20Value memory _requestedERC20,
        StructERC721Value[] memory _offeredERC721,
        StructERC721Value[] memory _requestedERC721,
        uint256 _offerValidDuration
    ) external payable returns (uint256 offerId) {
        uint256 msgValue = msg.value;
        address msgSender = msg.sender;
        uint256 currentTime = block.timestamp;

        ...

        if (!_isBalanceExcludedFromFees(msgSender)) {
            require(msgValue >= _fees + _offeredETH);
            _feesCollected += _fees;
        }
    ...
    }

    modifier isValidOffer(uint256 _offerId) {
        StructOffer memory offerAccount = _mappingOffer[_offerId];
        require(
            offerAccount.sender != address(0),
            "Address zero cannot make offer."
        );
        require(
            offerAccount.receiver != address(0),
            "Cannot make offer to address zero."
        );
        require(
            offerAccount.status == OfferStatus.pending,
            "Offer already used."
        );
        _;
    }
```

Recommendation

It is recommended to enhance the contract's validation checks. Specifically, irrespective of whether the `msgSender` is paying additional fees or not, the contract should always verify that the `msgValue` is greater than or equal to `_offeredETH`. This will ensure that the correct amount of ETH is sent to the contract in line with the offer being created.

Additionally, the team is advised to add an additional check inside the `createOffer` function to ensure that the `_receiver` is not the zero address. This will prevent offers from being created with a zero address as the receiver, which would render them unusable.

RMU - Redundant Mapping Usage

Criticality	Minor / Informative
Location	contracts/Trait.sol#L63,359,382
Status	Unresolved

Description

The contract contains the `isExemptedFromFees` mapping, which is redundant. Throughout the code, this mapping isn't used in a manner that enhances the contract's functionality or its gas optimization. The contract already utilizes the `_excludedFeesContracts` array, which serves a similar purpose and is actively used within the contract's logic.

```
mapping(address => bool) public isExemptedFromFees;

 function includeInFees(address _contractAddress) external onlyOwner {
        require(
            isExemptedFromFees[_contractAddress],
            "Already included in exchange fees."
        );

        isExemptedFromFees[_contractAddress] = false;
...
}
    function excludeFromExchangeFees(address _contractAddress)
        external
        onlyOwner
    {
        require(
            !isExemptedFromFees[_contractAddress],
            "Already excluded from exchange fees."
        );

        isExemptedFromFees[_contractAddress] = true;
        _excludedFeesContracts.push(_contractAddress);
    }
....
 }
```

Recommendation

It is recommended to remove the `isExemptedFromFees` mapping from the contract. Its usage does not contribute any additional functionality and might lead to unnecessary complexity and potential gas inefficiencies. Simplifying the contract by removing redundant component can increase the gas efficiency.

RFGO - Reject Functionality Gas Optimization

Criticality	Minor / Informative
Location	contracts/Trait.sol#L238
Status	Unresolved

Description

The contract is designed to allow recipients of offers to reject them through the `rejectOffer` function. This function, when invoked, transfers the assets back to the sender of the offer. However, the rejection process requires the recipient to pay the amount of gas needed for the transaction. This gas cost can be a deterrent for users, discouraging them from rejecting offers even if they do not wish to accept them.

```
function rejectOffer(uint256 _offerId) external isValidOffer(_offerId)
{
        address msgSender = msg.sender;
        StructOffer storage offerAccount = _mappingOffer[_offerId];

        require(
            msgSender == offerAccount.receiver,
            "Offer is not made to you."
        );
        ...

        offerAccount.status = OfferStatus.rejected;
        emit Status(offerAccount, OfferStatus.rejected);
  }
```

Recommendation

It is recommended to introduce an additional status, which allows the recipient to indicate their intention to reject without incurring the gas costs of the actual asset transfers. The sender can then invoke a separate function to finalize the rejection and bear the gas costs associated with the asset transfers. This approach ensures that the receiver is not discouraged from rejecting offers due to high gas costs, and the sender, who benefits from the return of their assets, bears the associated costs.

MRM - Missing Revert Messages

Criticality	Minor / Informative
Location	contracts/Trait.sol#L132
Status	Unresolved

Description

The contract is missing error messages. These missing error messages are making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(msgValue >= _fees + _offeredETH);
```

Recommendation

The team is advised to carefully review the source code in order to address these issues. To accelerate the debugging process and mitigate these issues, the team should use more specific and descriptive error messages.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/Trait.sol#L238,273
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specificaaly, the `rejectOffer` and `withdrawOffer` functions share identical code segments.

```
    function rejectOffer(uint256 _offerId) external
isValidOffer(_offerId) {
        address msgSender = msg.sender;
        StructOffer storage offerAccount =
_mappingOffer[_offerId];
        ...
        for (uint8 i; i < offerAccount.offeredERC721.length;
i++) {

ERC721(offerAccount.offeredERC721[i].erc721Contract).transferFr
om(
                address(this),
                offerAccount.sender,
                offerAccount.offeredERC721[i].erc721Id
            );
        }

        if (offerAccount.offeredETH > 0) {

payable(offerAccount.sender).transfer(offerAccount.offeredETH);
        }

        if (
            offerAccount.offeredERC20.erc20Contract !=
address(0) &&
            offerAccount.offeredERC20.erc20Value > 0
        ) {

IERC20(offerAccount.offeredERC20.erc20Contract).transfer(
                offerAccount.sender,
                offerAccount.offeredERC20.erc20Value
            );
        }
        ...
    }
```

```
   function withdrawOffer(uint256 _offerId) external
isValidOffer(_offerId) {
        address msgSender = msg.sender;

        StructOffer storage offerAccount =
_mappingOffer[_offerId];
        ...

        for (uint8 i; i < offerAccount.offeredERC721.length;
i++) {

ERC721(offerAccount.offeredERC721[i].erc721Contract).transferFr
om(
                address(this),
                offerAccount.sender,
                offerAccount.offeredERC721[i].erc721Id
            );
        }

        if (offerAccount.offeredETH > 0) {

payable(offerAccount.sender).transfer(offerAccount.offeredETH);
        }

        if (
            offerAccount.offeredERC20.erc20Contract !=
address(0) &&
            offerAccount.offeredERC20.erc20Value > 0
        ) {

IERC20(offerAccount.offeredERC20.erc20Contract).transfer(
                offerAccount.sender,
                offerAccount.offeredERC20.erc20Value
            );
        }
        ...
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/Trait.sol#L175,243
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(msgSender == offerAccount.receiver, "You are not
receiver.");
require(msgSender == offerAccount.receiver,"Offer is not made
to you.");
```

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.

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/Trait.sol#L110,146
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 contract is using two separate for loops to iterate over the same `_offeredERC721` array. The first loop transfers the ERC721 tokens from the sender to the contract, while the second loop pushes these tokens into the `offerAccount.offeredERC721` array. This approach results in redundant code and unnecessary gas consumption, as the contract iterates over the same array twice.

```
  function createOffer(
        address _receiver,
        uint256 _offeredETH,
        uint256 _requestedETH,
        StructERC20Value memory _offeredERC20,
        StructERC20Value memory _requestedERC20,
        StructERC721Value[] memory _offeredERC721,
        StructERC721Value[] memory _requestedERC721,
        uint256 _offerValidDuration
    ) external payable returns (uint256 offerId) {
        uint256 msgValue = msg.value;
        address msgSender = msg.sender;
        uint256 currentTime = block.timestamp;
        ...
        for (uint8 i; i < _offeredERC721.length; i++) {

ERC721(_offeredERC721[i].erc721Contract).transferFrom(
                msgSender,
                address(this),
                _offeredERC721[i].erc721Id
            );
        }
        ...

        for (uint256 i; i < _offeredERC721.length; ++i) {
            offerAccount.offeredERC721.push(_offeredERC721[i]);
        }
    ...
}
```

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. It is recommended to consolidate the two for loops into a single loop. Within this loop, both the transfer of the ERC721 tokens and the pushing of these tokens into the `offerAccount.offeredERC721` array can be performed. This will optimize the contract's efficiency by reducing the number of iterations and consequently saving gas.

DTI - Data Type Inconsistency

Criticality	Minor / Informative
Location	contracts/Trait.sol#L111,146,150
Status	Unresolved

Description

The contract contains an inconsistency in the data types used for iterating through the `_offeredERC721` array. Specifically, within the `createOffer` function, a `uint8` is employed to loop through the `_offeredERC721` array for transferring ERC721 tokens. However, a `uint256` is utilized in the function to iterate through the same `_offeredERC721` array to populate the `offerAccount.offeredERC721 list`. This discrepancy in data types can lead to potential issues, especially if the length of the `_offeredERC721` array surpasses the maximum value of `uint8`. Additionally, the `_requestedERC721` array consistently uses a `uint256` for iteration, further highlighting the inconsistency in the contract's design.

```
function createOffer(
        ...
        StructERC721Value[] memory _offeredERC721,
        StructERC721Value[] memory _requestedERC721,
        uint256 _offerValidDuration
    ) external payable returns (uint256 offerId) {
        ...

        for (uint8 i; i < _offeredERC721.length; i++) {

ERC721(_offeredERC721[i].erc721Contract).transferFrom(
                msgSender,
                address(this),
                _offeredERC721[i].erc721Id
            );
        }
    ...
    for (uint256 i; i < _offeredERC721.length; ++i) {
        offerAccount.offeredERC721.push(_offeredERC721[i]);
    }

    for (uint256 i; i < _requestedERC721.length; ++i) {
        offerAccount.requestedERC721.push(_requestedERC721[i]);
    }
    ...
```

Recommendation

It is recommended to harmonize the data types used for iterating through array parameters to prevent potential inconsistencies and vulnerabilities. If the intent is for the `_offeredERC721` and `_requestedERC721` arrays to have a maximum length of `255` elements, then a `uint8` should be consistently used for iteration. Conversely, if there's a possibility that these arrays could contain more than 255 elements, then a larger uint type should be used throughout the contract. Ensuring consistent use of data types will enhance the contract's reliability and reduce the risk of unexpected behaviors.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/Trait.sol#L111,146,150
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The contract is utilizing a `uint8` data type to iterate through the `_offeredERC721` array in the `createOffer` function. This inherently limits the maximum length of the `_offeredERC721` array to `2**8-1` which is 255. However, there is no checks in the contract to prevent the addition of entries beyond this limit. If an array with a length greater than 255 is passed to `_offeredERC721`, the contract does not return a coherent error message, potentially leading to unexpected behaviors and vulnerabilities.

```
function createOffer(
        ...
        StructERC721Value[] memory _offeredERC721,
        StructERC721Value[] memory _requestedERC721,
        uint256 _offerValidDuration
    ) external payable returns (uint256 offerId) {
        ...

        for (uint8 i; i < _offeredERC721.length; i++) {

ERC721(_offeredERC721[i].erc721Contract).transferFrom(
                msgSender,
                address(this),
                _offeredERC721[i].erc721Id
            );
        }
    ...
    for (uint256 i; i < _offeredERC721.length; ++i) {
        offerAccount.offeredERC721.push(_offeredERC721[i]);
    }

    for (uint256 i; i < _requestedERC721.length; ++i) {
        offerAccount.requestedERC721.push(_requestedERC721[i]);
    }
    ...
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to introduce a safeguard within the `createOffer` function to ensure that the length of the `_offeredERC721` array does not exceed the maximum permissible value of 255. If the length surpasses 255, the contract should revert the transaction and provide a descriptive error message to inform the user of the limitation.

RM - Redundant Modifier

Criticality	Minor / Informative
Location	contracts/Trait.sol#L69
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 contract declares the `tokensTransferable` modifier. The modifier is not being used by the contract. As a result, the modifier is redundant.

```
modifier tokensTransferable(address _token, uint256 _tokenId) {
        require(
            ERC721(_token).getApproved(_tokenId) ==
address(this),
            "The HTLC must have been designated an approved
spender for the tokenId"
        );
        _;
    }
```

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. If the intended purpose of the code is to utilize the `tokensTransferable` modifier, then the contract should incorporate it. Otherwise, since the modifier is not used within the contract, it can be safely removed to streamline the code.

RED - Redudant Event Declaration

Criticality	Minor / Informative
Location	contracts/Trait.sol#L15,16
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 events `ExcludedFromFees` and `IncludedFromFees` are declared and not being used in the contract. As a result, they are redundant.

```
event ExcludedFromFees(address contractAddress);
event IncludedFromFees(address contractAddress);
```

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Trait.sol#L95,96,97,98,99,100,101,102,166,238,273,310,318,359,382,399
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.

```
address _receiver
uint256 _offeredETH
uint256 _requestedETH
StructERC20Value memory _offeredERC20
StructERC20Value memory _requestedERC20
StructERC721Value[] memory _offeredERC721
StructERC721Value[] memory _requestedERC721
uint256 _offerValidDuration
uint256 _offerId
address _userAddress
address _contractAddress
uint256 _feesInWei
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/Trait.sol#L400
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
_fees = _feesInWei
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/Trait.sol#L111,146,150,187,196,247,283,344,369
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint8 i
uint256 i
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/Trait.sol#L3
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 ^0.8.19;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/Trait.sol#L124,216,228,263,299
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(_offeredERC20.erc20Contract).transferFrom(
                msgSender,
                address(this),
                _offeredERC20.erc20Value
            )

...
                offerAccount.requestedERC20.erc20Value
            )

IERC20(offerAccount.offeredERC20.erc20Contract).transfer(
                offerAccount.receiver,
                offerAccount.offeredERC20.erc20Value
            )

...
```

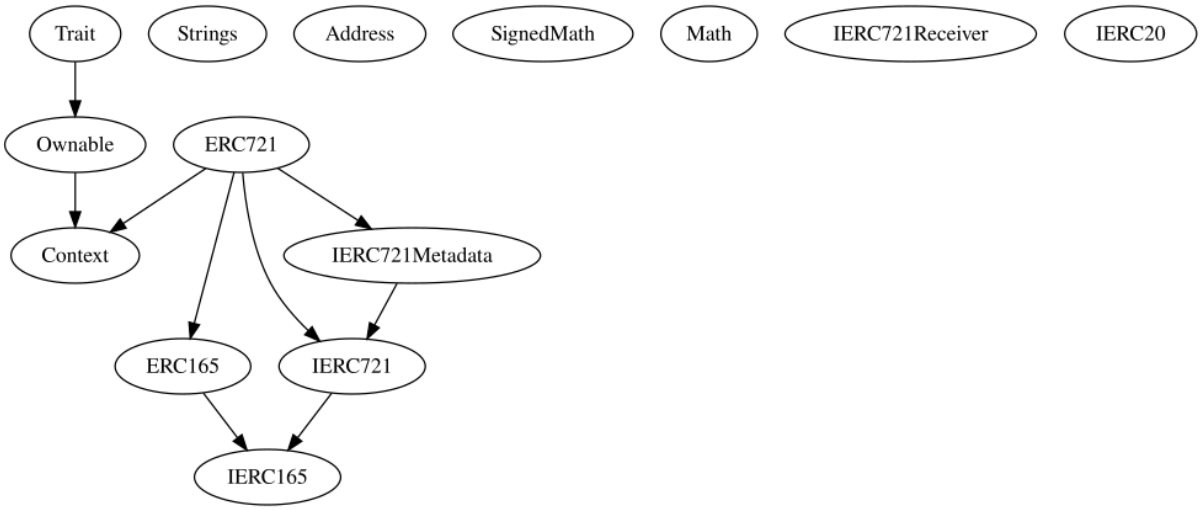
Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

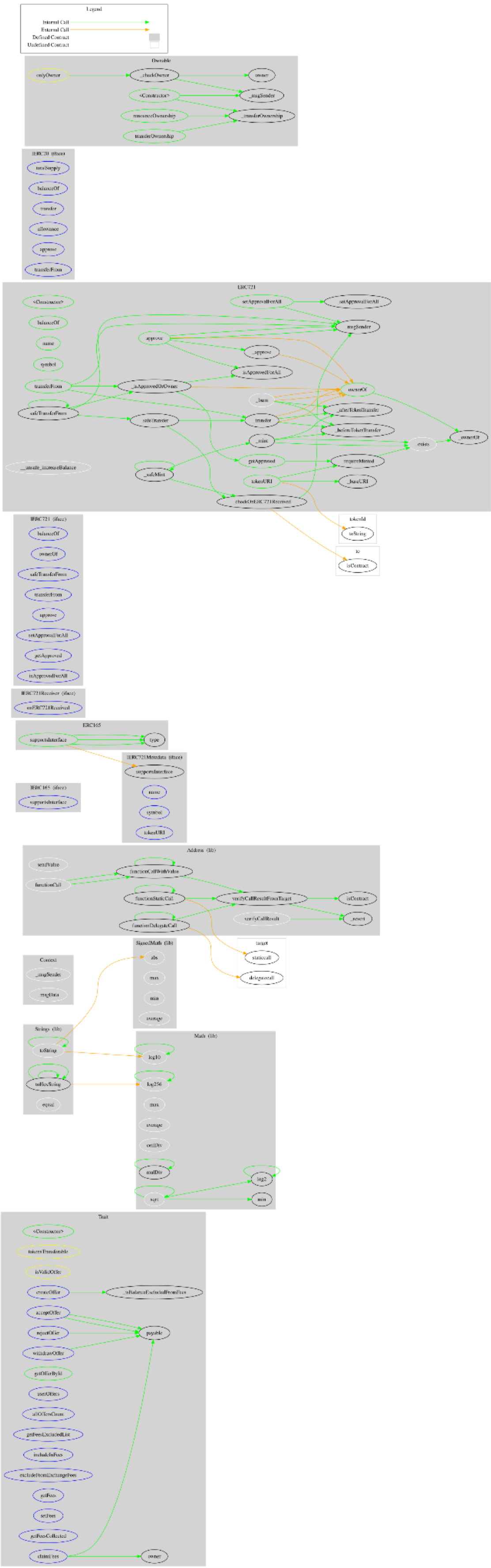
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Trait	Implementation	Ownable		
		Public	✓	-
	createOffer	External	Payable	-
	acceptOffer	External	Payable	isValidOffer
	rejectOffer	External	✓	isValidOffer
	withdrawOffer	External	✓	isValidOffer
	getOfferById	Public		-
	userOffers	External		-
	allOffersCount	External		-
	_isBalanceExcludedFromFees	Private		
	getFeesExcludedList	External		-
	includeInFees	External	✓	onlyOwner
	excludeFromExchangeFees	External	✓	onlyOwner
	getFees	External		-
	setFees	External	✓	onlyOwner
	getFeesCollected	External		-
	claimFees	External	✓	onlyOwner

Inheritance Graph



Flow Graph



Summary

Trait Exchange contract facilitates a decentralized trading platform for digital assets on the blockchain. 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