33AUDITS & CO.

Bistro Audit Report

Introduction

A time-boxed security review of the protocol was done by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits and Zuhaib.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities.

Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @solidityauditor.

About Bistro

This audit is being performed on the core protocol contracts for Bistro's decentralized OTC trading platform and staking system. Bistro provides a flexible toolkit for creating and executing over-the-counter trades between ERC20 tokens.

The main components audited include:

- OTC Contract: Facilitates the creation, management, and execution of OTC orders between various tokens.
- Bistro Staking Contract: Allows users to stake platform tokens or NFTs for rewards.

The Bistro platform aims to provide a decentralized alternative to traditional OTC trading, enabling users to conduct peer-to-peer trades with increased flexibility and reduced counterparty risk.

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

Scope

The following smart contracts were in the scope of the audit:

- bistro/src/OTC.sol
- bistro/src/BistroStaking.sol
- bistro/src/Whitelist.sol

Commit hash: 96c9bbec9cca809993ab88355328867c5a144f1f

Findings Summary

ID	Title	Severity	Status
[C-01]	Cross function reentrancy	Critical	Fixed
[H-01]	Order Front-Running Vulnerability in updateOrderInfo Function	High	Fixed
[M-01]	Rebasing Tokens Can Get Stuck in the Contract	Medium	Acknowledged
[L-01]	Unsafe NFT Transfer in Staking Function	Low	Fixed
[L-02]	Non-Adherence to ERC20 Standard Functions and Missing Decimals Implementation	Low	Fixed
[L-03]	Lack of Self-Trading Prevention Enables Wash Trading	Low	Acknowledged
[I-01]	Unintended Inclusion of Hardhat Console Library in Production Code	Info	Fixed
[I-02]	Caching Storage Variable in cancelOrder Function	Info	Fixed

Detailed Findings

[C-01] Cross function reentrancy

Description

There is a cross-function reentrancy in vulnerbility present in the `updateOrderInfo` function as it does an extenral call to the `sendNative()` function before updating the `sellAmount` and `buyAmount` variables.

You can see on line 235 in the `updateOrderInfo` function, the contrat is mindint addtional tokens to the user based on the different between the new sellAmount and the old sellAmount.

```
uint256 additionalAmount = _newSellAmount -
orderDetailsWithId.orderDetails.sellAmount;
_mint(msg.sender, additionalAmount);
```

After minting the additonal tokens to the user the contract checkins if th esell toke is the native token and if so it makes an external call through the `sendNative()` function to send the native token back to the user.

```
if (orderDetailsWithId.orderDetails.sellToken == NATIVE_ADDRESS) {
    require(msg.value >= additionalAmount, "otc:provide correct amount");
    uint256 extraTokens = msg.value - additionalAmount;
    if (extraTokens > 0) {
        _sendNativeToken(extraTokens);
    }
}
```

Its here that a user can call the `cancelOrder()` function and in the execution of that function the `updateOrderInfo()` is called leading to a reentrancy. This would cause the original funds sent by the user to be sent back to the user. `updateOrderInfo()` would then continue to execute updating the `sellAmount` and `buyAmount` to the new values. A user would now have essentially created a phantom order without having sent any tokens to the contract.

Resolution

Implement proper reentrancy guards and ensure that state changes are made before external calls. Following CEI in all state changing functions should be strictly followed.

[H-1] Order Front-Running Vulnerability in updateOrderInfo Function

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328cl867c5a144f1f/contracts/OTC.sol#L201-L217

Description

The `updateOrderInfo` function allows users to modify the details of their orders. However, there is a potential front-running vulnerability in the function. When a user places an order, the order details are stored in the `orders` mapping. If the user then calls `updateOrderInfo` to modify the order, the new details are written back to the `orders` mapping.

A malicious user could exploit this by quickly calling `updateOrderInfo` right before another user calls `executeOrder`. The user would be able to decrease the `sellAmount` of the order causing the buyer to buy an order that is more expensive than the originally intended.

Impact

This vulnerability can result in:

1. Buyers paying more than intended for their orders.

Proof of Concept

Consider the following scenario:

1. User A places an order to sell 100 tokens at a price of 1 ETH per token.
2. User B decides to execute that order.
3. User A is monitoring the mempool and sees that User B is about to execute the order.
4. User A calls `updateOrderInfo` to modify the order, decreasing the `sellAmount` to 50 tokens.
5. User B will now execute the order at the new price.

Recommendation

Implement a cooldown period after an order is updated before it can be executed. This would prevent users from getting front-runned by a malicious user changing the order details in their favor.

```
function updateOrderInfo(
    uint256 _orderId,
    uint256 _newSellAmount,
    uint256[] calldata _newBuyTokensIndex,
    uint256[] calldata _newBuyAmount
) external payable nonReentrant validOrderId(_orderId) {
    // ... existing code ...

    // Add a time delay or cooldown period
    require(
        block.timestamp >= orderDetailsWithId.lastUpdateTime +
UPDATE_COOLDOWN,
        "Too soon to update"
    );

    // ... rest of the function ...

    // Update the last update time
    orderDetailsWithId.lastUpdateTime = block.timestamp;
}

// Add this constant at the contract level
uint256 constant UPDATE_COOLDOWN = 1 hours;
```

[M-1] Rebasing tokens can get stuck in the contract

Description

The current implementation of OTC contract does not account for rebasing tokens. Rebaisng tokens are designed to a automatically adjust their supply base don various facors, which can lead to discrepancies between the amount of tokens deposied and the amount available for withdrwasl.

When users deposit rebsaicng tokens into the contract the contract reocrds the deposited amount. However if the token undergoese a rebasing event prior to the order being executed the amount of tokens the user has available will be less/greater than the amount they deposited.

1. If the balance increased due to a postive rebase the extra tokens will be stuck in the contract as the recoxerd amount will be less thant ht eactula user balance.

2. If the balance decreases due to a negative rebase, the contract may no have sufficent tokens to be able to execute the order.

Impact

This vulnerability can lead to:

1. Loss of funds for users if positive rebases occur as excess tokens qill remains locked in the contract.
2. Potential contract insolvency or failed withdrawals if negative rebases occur and there are no sufficient tokens in the contract to execute orders.

Recommendation

Add clear warning and documentation about the risks of depositing rebasing tokens and the possibility of excess tokens being locked in the contract.

[M-2] Unchecked Fee Parameters Allow Arbitrary Fee Setting by Owner

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L720-L731

Description

The functions `_updateListingFees()`, `_updateRedeemFees()`, and `_updateDiscountInRedeemFees()` lack proper input validation for their parameters. This allows the contract owner to set fees to any value, including excessively high amounts.

For example, in `_updateRedeemFees()`:

```
function _updateRedeemFees(uint256 _newRedeemFees) internal {
    require(_newRedeemFees < PECENTAGE_DIVISOR, "OTC: Redeem fees cannot
exceed 100%");
    redeemFees = _newRedeemFees;
}
```

While there's a check to prevent fees exceeding 100%, the owner could still set fees to 99%, which would be economically unfeasible for users.

Similarly, `_updateListingFees()` and `_updateDiscountInRedeemFees()` have no upper bounds at all:

```
function _updateListingFees(uint256 _amountInUSD) internal {
    listingFeesInUSD = _amountInUSD;
}

function _updateDiscountInRedeemFees(uint256 _newDiscountInRedeemFees)
internal {
    discountInRedeemFees = _newDiscountInRedeemFees;
}
```

This centralized control over critical economic parameters introduces significant trust assumptions and risks for users.

Impact

The contract owner can set arbitrarily high fees, potentially draining user funds or rendering the protocol unusable.

Recommendation

1. Implement reasonable upper bounds for all fee parameters.
2. Consider using a tiered admin system or time-locked updates for sensitive parameters.
3. Emit events for all parameter changes to ensure transparency.

Example for `_updateRedeemFees()`:

```
uint256 public constant MAX_REDEEM_FEE = 500; // 5%

function _updateRedeemFees(uint256 _newRedeemFees) internal {
    require(_newRedeemFees <= MAX_REDEEM_FEE, "OTC: Redeem fees cannot
exceed 5%");
    redeemFees = _newRedeemFees;
    emit RedeemFeesUpdated(_newRedeemFees);
}
```

[L-1] Unsafe NFT Transfer in Staking Function

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/bistroStaking.sol#L90

Description

In the `stake` function of the `BistroStaking` contract, when staking an NFT, the contract uses `transferFrom` instead of `safeTransferFrom` for transferring NFTs. While this doesn't necessarily create an immediate issue, it's considered a best practice to use `safeTransferFrom` when dealing with NFT transfers.

Impact

Using `transferFrom` instead of `safeTransferFrom` for NFT transfers can lead to several issues:

1. If the receiving contract (e.g., `BistroStaking`) is not properly configured to handle incoming NFTs, the NFT could become stuck in the contract without a way to retrieve it.
2. Some NFT implementations require `safeTransferFrom` for proper functionality and may not work correctly with `transferFrom`.
3. `safeTransferFrom` includes additional checks to ensure that the receiving address can accept NFTs, which are bypassed when using `transferFrom`."

Recommendation

Follow the mentioned steps below to fix the issue.

1. Replace `transferFrom` with `safeTransferFrom` in the `stake` function:

```
function stake(bool _nftStake, uint256 _nftTokenId) external nonReentrant
{
    // ... existing code ...
    if (_nftStake) {
        // ... existing code ...
        IERC721(nftToken).safeTransferFrom(msg.sender, address(this),
_nftTokenId); // Use safeTransferFrom
        // ... existing code ...
    }
    // ... existing code ...
}
```

2. Implement the `IERC721Receiver` interface in the `BistroStaking` contract to properly handle NFT receipts:

```
import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

contract BistroStaking is ReentrancyGuard, Ownable2Step, IERC721Receiver {
    // ... existing code ...

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external override returns (bytes4) {
        // Add any necessary logic here
        return this.onERC721Received.selector;
    }

    // ... existing code ...
}
```

[L-2] Non-Adherence to ERC20 Standard Functions and Missing Decimals Implementation

Links

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L733-L741

Description

The `OTC` contract inherits from `ERC20` but does not properly implement several standard ERC20 functions. Specifically:

1. The `transfer`, `approve`, `transferFrom`, and `allowance` functions are left empty without any revert messages.
2. The `decimals` function is empty and will return 0 instead of the standard 18 for most ERC20 tokens.

Impact

1. Empty ERC20 functions: While the intention seems to be creating a non-transferable token, the lack of revert messages in these functions could lead to confusion for users and integrations. Calls to these functions will silently fail, potentially causing issues in systems that expect standard ERC20 behavior.

2. Missing decimals implementation: Returning 0 for decimals instead of 18 could cause calculation errors in systems interacting with this token, as they may assume the standard 18 decimals for ERC20 tokens.

Recommendation

By implementing these changes, the contract will provide clear feedback when unsupported operations are attempted and will correctly report its decimal places, reducing the risk of integration issues and calculation errors.

1. For the empty ERC20 functions, add explicit revert messages to clearly indicate that these operations are not supported:

```
function transfer(address to, uint256 value) public override returns
(bool) {
    revert("OTC: transfer not supported");
}

function approve(address spender, uint256 value) public override returns
(bool) {
    revert("OTC: approve not supported");
}

function transferFrom(address from, address to, uint256 value) public
override returns (bool) {
    revert("OTC: transferFrom not supported");
}

function allowance(address owner, address spender) public view override
returns (uint256) {
    revert("OTC: allowance not supported");
}
```

2. Implement the `decimals` function to return the correct number of decimals (usually 18 for most ERC20 tokens):

```
function decimals() public view override returns (uint8) {
    return 18;
}
```

[L-3] Lack of Self-Trading Prevention Enables Wash Trading

Description

The current implementation of the `placeOrder` and `updateOrderInfo` functions in the OTC contract allows users to create or modify orders where the `sellToken` is the same as one of the `buyTokens`. This oversight can be exploited to conduct wash trading, a form of market manipulation where an investor simultaneously sells and buys the same asset to create artificial activity in the marketplace.

The potential consequences of this vulnerability include:

1. Artificial inflation of trading volumes, misleading other users about the actual liquidity and demand for certain tokens.
2. Manipulation of market perceptions, potentially influencing other traders' decisions based on false signals.
3. Exploitation of reward systems or rankings that are based on trading volume, undermining the integrity of the platform.
4. Potential regulatory issues, as wash trading is often considered a form of market manipulation and is illegal in many jurisdictions.

This vulnerability could significantly impact the trustworthiness and reliability of the trading platform, potentially leading to reputational damage and loss of user confidence.

Recommendation

To mitigate this vulnerability, implement a check in both the `placeOrder` and `updateOrderInfo` functions to ensure that the `sellToken` is not present in the `buyTokensIndex` array. This can be achieved by adding a helper function and calling it in both places:

```
// Add this helper function
function _checkNoSelfTrading(address sellToken, uint256[] memory
buyTokensIndex) internal view {
    for (uint i = 0; i < buyTokensIndex.length; i++) {
        (address buyToken, ) = getTokenInfoAt(buyTokensIndex[i]);
        require(buyToken != sellToken, "OTC: Sell token cannot be the same
as any buy token");
    }
}

function placeOrder(OrderDetails calldata _orderDetails) external payable
nonReentrant {
    _checkNoSelfTrading(_orderDetails.sellToken,
_orderDetails.buyTokensIndex);
    // Rest of the function remains unchanged
    ...
}

function updateOrderInfo(
    uint256 _orderId,
    uint256 _newSellAmount,
    uint256[] calldata _newBuyTokensIndex,
    uint256[] calldata _newBuyAmount
) external payable nonReentrant validOrderId(_orderId) {
    UserOrderDetails memory userOrderDetails = _orderPreCheck(_orderId);
    OrderDetailsWithId storage orderDetailsWithId = orders[msg.sender]
[userOrderDetails.orderIndex];

    // Add this check when updating buy tokens
    if (_newBuyTokensIndex.length > 0) {
        _checkNoSelfTrading(orderDetailsWithId.orderDetails.sellToken,
_newBuyTokensIndex);
    }

    // Rest of the function remains unchanged
    ...
}
```

This implementation ensures that the wash trading prevention check is consistently applied both when creating new orders and when updating existing ones. The helper function `_checkNoSelfTrading` encapsulates the logic for this check, promoting code reuse and maintainability.

[I-1] Unintended Inclusion of Hardhat Console Library in Production Code

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L11

Description

The contract imports and potentially uses the Hardhat Console library (import "hardhat/console.sol";`). This library is typically used for debugging during development and testing phases. Its presence in production code can lead to increased gas costs and potentially expose sensitive information if logging statements are left in the deployed contract.

Recommendation

Remove the import statement for the Hardhat Console library and any associated console.log() calls before deploying to production. If logging is necessary for the production environment, consider implementing a more gas-efficient and secure logging mechanism that doesn't expose sensitive data.

Remove the following line from the contract: Also, ensure that any console.log() statements within the contract body are removed or commented out before deployment.

[I-2] Caching Storage Variable in cancelOrder Function

###Lines https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L201-L217

Description

In the `cancelOrder` function, the `orderDetailsWithId.remainingExecutionPercentage` storage variable is read multiple times. This can lead to unnecessary gas costs, as each storage read operation is expensive in terms of gas consumption.

Recommendation

To optimize gas usage, it's recommended to cache the `remainingExecutionPercentage` value in a memory variable at the beginning of the function. This cached value can then be used for subsequent operations and comparisons. Here's how you can modify the `cancelOrder` function:

By implementing this change, you reduce the number of storage reads and potentially save gas costs, especially for contracts with high usage.

```
function cancelOrder(uint256 _orderId) external validOrderId(_orderId) {
    UserOrderDetails memory userOrderDetails =
userDetailsByOrderId[_orderId];
    require(userOrderDetails.orderOwner == msg.sender,
"otc:Unauthorized");

    redeemOrder(_orderId);

    OrderDetailsWithId storage orderDetailsWithId = orders[msg.sender]
[userOrderDetails.orderIndex];
```

```
    OrderDetails memory orderDetails = orderDetailsWithId.orderDetails;

    // Cache the remainingExecutionPercentage
    uint256 remainingPercentage =
orderDetailsWithId.remainingExecutionPercentage;

    orderDetailsWithId.status = OrderStatus.Cancelled;
    require(remainingPercentage > 0, "otc:order is already completed");

    uint256 remainingAmount = (orderDetails.sellAmount *
remainingPercentage) / DIVISOR;
    _burn(msg.sender, remainingAmount);

    if (orderDetails.sellToken == NATIVE_ADDRESS) {
        _sendNativeToken(remainingAmount);
    } else {
        IERC20(orderDetails.sellToken).safeTransfer(msg.sender,
remainingAmount);
    }

    // Update the storage variable only once at the end
    orderDetailsWithId.remainingExecutionPercentage = 0;

    emit OrderCancelled(msg.sender, _orderId);
}
```