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 a 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:

1. OTC Contract: Facilitates the creation, management, and execution of OTC orders between various tokens.
2. 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

- bistro
- bistro

Findings Summary

ID	Title	Severity	Status
[C-01]	Cross function reentrancy	Critical	
[H-01]	Potential Fee Evasion in `executeOrder` Function for Native Token Transactions	High	
[H-02]	Order Front-Running Vulnerability in updateOrderInfo Function	High	
[M-01]	Rebasing Tokens Can Get Stuck in the Contract	Medium	
[M-02]	Unchecked Fee Parameters Allow Arbitrary Fee Setting by Owner	Medium	
[L-01]	Unsafe NFT Transfer in Staking Function	Low	
[L-02]	PUSH0 opcode Is Not Supported on Pulse Chain	Low	
[L-03]	Non-Adherence to ERC20 Standard Functions and Missing Decimals Implementation	Low	
[L-04]	Lack of Self-Trading Prevention Enables Wash Trading	Low	
[I-01]	Unintended Inclusion of Hardhat Console Library in Production Code	Info	
[I-02]	Caching Storage Variable in cancelOrder Function	Info	

[C-1] - Cross function reentrancy

Description

There's a cross-function reentrancy vulnerability present in the `updateOrderInfo` function as it does an external call to the `sendNative()` function which hands over execution control to the receiver contract. This would allow a malicious contract to reenter the contract and call `cancelOrder()` before the previous invocation of `updateOrderInfo()` is finished.

You can see on line 235 in the `updateOrderInfo()` function, the contract is minting additional tokens to the user based on the difference between the new sell amount and the initial sell amount.

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

Right after this line, the contract checks if the sell token is the native token and makes an external call to the `sendNative()` function which hands over execution to the receiver contract.

```
   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 `cancelOrder()` via the fallback function of a contract they control to cancel the order and get back the original funds that were transferred to the contract when placing an order. This would technically set their orderDetails to 0 at this point in the execution.

Let's recap what our amounts currently are after we reenter the contract:

```
orderDetailsWithId.orderDetails.sellAmount = 0;
orderDetailsWithId.orderDetails.buyAmount = 0;

token1.balanceOf(user) = initialSellAmount;
token2.balanceOf(user) = initialBuyAmount;
```

When in reality they should be:

```
orderDetailsWithId.orderDetails.sellAmount = initialSellAmount;
orderDetailsWithId.orderDetails.buyAmount = initialBuyAmount;

token1.balanceOf(user) = 0;
token2.balanceOf(user) = 0;
```

The function execution then resumes and since we're still in the `updateOrderInfo()` function, the function then sets the orderDetails.sellAmount to the newSellAmount and the orderDetails.buyAmount to the newBuyAmount that were passed into the function by the user.

This leaves a user with an orderId that has a sellAmount of x and buyAmount of x however they were able to withdraw their tokens essentially allowing them to create an order without having to transfer tokens. The additional amount that was minted will allow them to redeem the order amount after executing the order and withdraw tokens from the contract.

Impact

Users can create orders of arbitrary amounts and buy/sell amounts without transferring any tokens to the contract essentially allowing them to steal tokens from the contract.

Resolution

While a lot of the functions in the contract havde a nonReentrant modifier, this doesn't save the contract from cross function reentrancy. And to be fair I don't think that just adding the nonReentrant modifier to the functions will fix this issue. Ideally every state changing function should be follow the checks effects interactions pattern. This will ensure that state changes are completed before making external calls. Considering your team is using the `sendNative()` function a lot in this contract you should be strictly folllowing CEI and make sure that all state changes are completed before making external calls.

[H-1] Potential Fee Evasion in `executeOrder` Function for Native Token Transactions

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L323-L333

Description

In the `executeOrder` function, there appears to be a logical error in the handling of native token (ETH) transactions. When the buy token is the native token (ETH), the function may inadvertently refund the entire msg.value to the user, including the fees that should be sent to the admin wallet. This could potentially allow users to execute trades without paying the required fees.

Impact

This issue could lead to:

1. Loss of revenue for the protocol, as fees are not properly collected for native token trades.
2. Potential exploitation by malicious users to perform free trades.

Proof of Concept:

Consider the following scenario:

1. Alice wants to execute an order with the following parameters: `_buyAmount: 1 ETH` and `_fees: 0.01 ETH`
2. Alice calls `executeOrder` with msg.value = 1.01 ETH. The function processes as follows:
3. extraTokens is calculated as 1.01 ETH - 1 ETH = 0.01 ETH
4. 0.01 ETH is sent to the admin wallet from Alice's `msg.value`
5. 0.01 ETH (`extraTokens`) is sent back to Alice
6. Result: Alice effectively pays no fees, as the entire `msg.value` is either used for the trade or refunded.

Recommendation

To fix this issue, modify the calculation of `extraTokens` to account for the fees:

```
if (_buyToken == NATIVE_ADDRESS) {
    require(msg.value >= _buyAmount + _fees, "otc:not enough tokens
provided"); //@note ->update the require statement

    uint256 extraTokens = msg.value - (_buyAmount + _fees); //@note ->
add the `_fees` value in `extraTokens` calculation
    if (_fees > 0) {
        _sendNativeTokenToAdmin(_fees);
    }
    if (extraTokens > 0) {
        _sendNativeToken(extraTokens);
    }
}
```

This change ensures that the fees are properly deducted from the `msg.value` before calculating the extra tokens to be refunded. Additionally, update the require statement to check that the msg.value covers both the buy amount and the fees.

[H-2] Order Front-Running Vulnerability in updateOrderInfo Function

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/OTC.sol#L219-L280

Description

The `updateOrderInfo` function in the OTC contract allows users to modify their existing orders without any time delay or restrictions. This creates a vulnerability where a malicious seller can front-run an incoming `executeOrder` transaction by calling `updateOrderInfo` to increase the `sellAmount`, forcing the buyer to pay more than initially agreed upon.

The current implementation lacks safeguards against such manipulations, potentially leading to unfair trades and undermining the integrity of the order execution process.

Impact

This vulnerability can result in:

1. Buyers paying more than intended for their orders.
2. Loss of trust in the platform due to unpredictable and manipulable order execution.
3. Potential for griefing attacks, where a malicious seller repeatedly updates their order to prevent execution.
4. Financial losses for buyers who may be forced to overpay for assets.

Proof of Concept

Consider the following scenario:

1. Alice creates an order to sell 100 tokens for 1 ETH.
2. Bob sees this order and decides to execute it by calling `executeOrder`.
3. Alice notices Bob's pending transaction in the mempool.
4. Alice quickly submits a transaction with higher gas to call `updateOrderInfo`, changing the sell amount to 150 tokens for 1 ETH.
5. Alice's transaction is processed first, updating the order.
6. Bob's transaction is then processed, forcing him to pay 1 ETH for 150 tokens instead of the original 100.

Recommendation

Implement a cooldown period between order updates to prevent rapid manipulations. This gives buyers time to notice changes and decide whether to proceed with the trade. Also, consider adding events for order updates to increase transparency. Below is an example implementation however you should implement a solution that suits your contract and find a cooldown period that works for your usecase.

```
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 the OTC contract does not account for rebasing tokens. Rebasing tokens are designed to automatically adjust their supply based on various factors, which can lead to discrepancies between the amount of tokens deposited and the amount available for withdrawal.

When users deposit rebasing tokens into the contract (e.g., when placing an order), the contract records the deposited amount. However, if the token undergoes a rebase event while held in the contract, the actual balance may increase or decrease. This can lead to two primary issues:

1. If the balance increases due to a positive rebase, the extra tokens will be stuck in the contract as the recorded amount will be less than the actual balance.
2. If the balance decreases due to a negative rebase, the contract may not have sufficient funds to fulfill withdrawal requests based on the originally recorded amounts.

Impact

This vulnerability can lead to:

1. Loss of funds for users if positive rebases occur, as excess tokens remain locked in the contract.
2. Potential contract insolvency or failed withdrawals if negative rebases occur, as the contract may not have sufficient funds to honor all withdrawal requests.
3. Inconsistencies in order execution and cancellation, as the actual token amounts may differ from the recorded amounts.

Proof of Concept

Consider the following scenario with a hypothetical rebasing token:

1. User A deposits 100 tokens to create an order.
2. The token undergoes a positive 10% rebase, increasing the contract's balance to 110 tokens.
3. User A cancels the order, but only receives back 100 tokens.
4. The extra 10 tokens remain stuck in the contract.

Recommendation

To mitigate this issue, consider the following approach:

1. Add clear warnings and documentation:
    - Inform users about the risks associated with using rebasing tokens in the contract.

- Consider maintaining a whitelist of supported tokens, excluding known rebasing tokens if the above solutions are not implemented.

[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] PUSH0 opcode Is Not Supported on Pulse Chain

Lines

https://github.com/ShintoSan/bistro-contract/blob/96c9bbec9cca809993ab88355328867c5a144f1f/contracts/utils/whitelist.sol#L2

Description:

`whitelist.sol` is using Solidity version `0.8.20`, which introduces the `PUSH0` opcode. This opcode is part of the Shanghai upgrade for Ethereum, which occurred in April 2023. However, if PulseChain has not implemented this upgrade or does not support this opcode, it could lead to compatibility issues.

The PUSH0 opcode is an optimization that pushes the constant value 0 onto the stack. It's more gas-efficient than the previous method of using PUSH1 0x00. Solidity 0.8.20 automatically uses this opcode when compiling, which means any contract compiled with this version will include PUSH0 instructions in its bytecode.

Impact

Assuming PulseChain does not support the PUSH0 opcode, the impact could be severe like Deployment Failure, Execution Errors, Ecosystem Incompatibility etc.,

Recommendation:

To address this issue, consider the following options:

1. Downgrade Solidity Version: Use an earlier version of Solidity (0.8.19 or lower) that doesn't use the PUSH0 opcode. This ensures compatibility with PulseChain but may miss out on other optimizations or features introduced in 0.8.20.
2. PulseChain Compatibility Check: Verify PulseChain's current EVM version and supported opcodes. If PUSH0 is supported or planned to be supported soon, document this clearly and plan for a safe transition.

[Low-3] 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-4] 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);
}
```