Three Sigma Labs

Code Audit

Ordiswap

Ordiswap Bitcoin AMM

Disclaimer

Code Audit

Ordiswap Bitcoin AMM

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Ordiswap Bitcoin AMM

Table of Contents

Summary

Code Audit

Ordiswap Bitcoin AMM

Summary

Three Sigma audited the Ordiswap token solidity smart contract in a 2 person week engagement. The audit was conducted from 25-12-2023 to 29-12-2023.

Protocol Description

Ordiswaps is a Bitcoin exchange which uses an off-chain server to manage the logic and parameters of the Automated Market Maker (AMM). This includes handling trades, liquidity provision, and setting dynamic AMM parameters.

Scope

Code Audit

Ordiswap Bitcoin AMM

Scope

The scope of the engagement includes the ORDS token solidity smart contract.

Assumptions

It is an assumption that the Ordiswap team is a trusted actor that will not act maliciously.

Methodology

Code Audit

Ordiswap Bitcoin AMM

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Ordiswap Bitcoin AMM

Application Summary

Name	Ordiswap Token
Commit	0c7e1081cb20a6cb69671e55028642cd4d6c444d
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	25-12-2023 to 29-12-2023
Nº of Auditors	2
Review Time	1 week

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	2	0	2
Medium	2	0	2
Low	2	0	2
None	2	0	2

Category Breakdown

Suggestion	4
Documentation	0
Bug	4
Optimization	4
Good Code Practices	0

Code Maturity Evaluation

Code Audit

Ordiswap Bitcoin AMM

Code Maturity Evaluation

Code Maturity Evaluation Guidelines

Category	Evaluation
Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system.
Arithmetic	The proper use of mathematical operations and semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Code Stability	The extent to which the code was altered during the audit.
Upgradeability	The presence of parameterizations of the system that allow modifications after deployment.
Function Composition	The functions are generally small and have clear purposes.
Front-Running	The system's resistance to front-running attacks.
Monitoring	All operations that change the state of the system emit events, making it simple to monitor the state of the system. These events need to be correctly emitted.
Specification	The presence of comprehensive and readable codebase documentation.
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage.

Code Maturity Evaluation Results

Category	Evaluation
Access Controls	Satisfactory. All onlyOwner functions were correctly set.
Arithmetic	Satisfactory. No arithmetic errors were found.
Centralization	Moderate. The code is immutable but the owner can censor users and change fees anytime.
Code Stability	Satisfactory. The code was stable during the audit.
Upgradeability	Weak. The contract is not upgradeable.
Function Composition	Weak. The code could be split into helper functions.
Front-Running	Moderate. Frontrunning is possible on fee swaps.
Monitoring	Satisfactory. Events are correctly emitted for state changes.
Specification	Moderate. The codebase follows most of the specifications correctly, but not everything was correctly implemented.
Testing and Verification	Weak. Some issues should have been caught by a bigger testing suite.

Findings

Code Audit

Ordiswap Bitcoin AMM

Findings

3S-ORDS-H01

`.transfer()` fails for non standard ERC20s such as USDT and gets stuck

Id	3S-ORDS-H01
Classification	High
Severity	Critical
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

`IERC20(tkn).transfer(msg.sender, amount);` fails for tokens that don't return any value due to the fact that the EVM adds a return length check, which will revert for USDT as it returns nothing (see the code at line 340).

Recommendation

Use **`safeTransfer()`** from Openzeppelin's `SafeERC20`.

3S-ORDS-H02

`addLiquidityEth()` and `swapExactTokensForETHSupportingFeeOnTransferTokens` are vulnerable to MEV

Id	3S-ORDS-H02
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Acknowledged

Description

`addLiquidityEth()` should specify `amountTokenMin` and `amountETHMin` to prevent transactions frontrunning it, changing the price and getting less lp tokens in return. Additionally, the deadline argument should be sent as an argument to `launch()` instead of using `block.timestamp`. When `block.timestamp` is used, validators may include the transaction whenever they want, such that the price could have changed significantly by then.

`swapExactTokensForETHSupportingFeeOnTransferTokens()` also specifies the minimum ETH amount out as argument, but currently it is set to 0. The deadline is also `block.timestamp`.

Recommendation

When adding liquidity, `amountTokenMin` and `amountETHmin` should be sent as a percentage of the total supplied ETH and Ordiswap tokens, respectively, so the price can't flactuate more than the defined percentage.

When swapping, the price should ideally be fetched off chain and sent as argument to a `swap()` call in the `OrdiswapToken` contract. In fact, the current design choice of swapping when users transfer funds and the fees reach the threshold may lead to poor UX as users can't fully predict if their transfer will trigger the swap, possibly leading to reverts. Create a

separate **`onlyOwner swap`** function that will swap the **`OrdiswapToken`** for ETH when the owner wants.

A good rule of thumb for the deadline argument is using 30 minutes after the **`block.timestamp`**, calculated off chain and sent to **`launch()`** and the new **`swap() onlyOwner`**.

3S-ORDS-M01

Tokens will get stuck if sent to a bot wallet

Id	3S-ORDS-M01
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

`OrdiswapToken` allows transfers to a bot address, but not from, such that it would be possible to send tokens to a bot, but the bot can't transfer the tokens out anymore.

3S-ORDS-M02

`setAutomatedMarketMakerPairV2()` and `setAutomatedMarketMakerPairV3()` should be external

Id	3S-ORDS-M02
Classification	Medium
Severity	Medium
Likelihood	High
Category	Bug
Status	Acknowledged

Description

`setAutomatedMarketMakerPairV2()` and `setAutomatedMarketMakerPairV3()` are internal functions but should likely be external.

3S-ORDS-L01

renounceOwnership() should be disabled, as it may be triggered unintentionally

Id	3S-ORDS-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Status	Acknowledged

Description

OrdiswapToken has several **`onlyOwner`** methods that would be unreachable if the **`owner`** renounced ownership.

3S-ORDS-L02

Ownable2Step should be used instead of Ownable

Id	3S-ORDS-L02
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Status	Acknowledged

Description

`Ownable` is vulnerable to setting the wrong address as admin. `Ownable2Step` has an extra verification step as it requires the **`pendingOwner`** to accept being the new **`owner`**, reducing the chance of a wrong owner being set.

3S-ORDS-N01

`_transfer()` could be simplified

Id	3S-ORDS-N01
Classification	None
Category	Suggestion
Status	Acknowledged

Description

Function `_transfer()` has significant complexity and could be simplified to increase readability.

As mentioned in #7, the swap mechanism if the fees inside the contract hit the threshold could be placed in a separate function that the owner would call when it wanted to withdraw the fees to ETH.

Whis this in mind and the fact that if the **`from`** or **`to`** addresses have been excluded from fees, the conditions are always skipped, the function can be reduced to:

```
function _transfer(address from, address to, uint256 amount) internal
virtual override {
    require(from != ZERO_ADDRESS, "ERC20: transfer from the zero address");
    require(to != ZERO_ADDRESS, "ERC20: transfer to the zero address");
    if (_isExcludedFromFees[from] || _isExcludedFromFees[to]) {
        super._transfer(from, to, amount);
        return;
    }
    require(!_isBot[from], "ERC20: bot detected");
    require(msg.sender == from || !_isBot[msg.sender], "ERC20: bot
detected");
    require(tx.origin == from || tx.origin == msg.sender ||
!_isBot[tx.origin], "ERC20: bot detected");
    if (amount == 0) {
        super._transfer(from, to, amount);
        return;
    }
    require(!_v3LPProtectionEnabled || (!_automatedMarketMakerPairsV3[from]
&& !_automatedMarketMakerPairsV3[to]),
```

```
        "ERC20: Not authorized to add LP to Uniswap V3 Pool"
    );

    address owner = owner();
    require(launched || from == owner || to == owner || to == DEAD_ADDRESS,
"ERC20: Not launched.");

    if (taxesEnabled) {
        uint256 fees;
        if (_automatedMarketMakerPairsV2[to] && sellFees > 0) {
            fees = amount.mul(sellFees).div(100);
        } else if (_automatedMarketMakerPairsV2[from] && buyFees > 0) {
            fees = amount.mul(buyFees).div(100);
        }
        if (fees > 0) {
            super._transfer(from, address(this), fees);
            amount -= fees;
        }
    }
    super._transfer(from, to, amount);
}
```

NOTE: this function should be thoroughly tested, this is just an example implementation.

3S-ORDS-N02

`swapExactTokensForETHSupportingFeeOnTransferTokens()` could send the ETH directly to the `operationsWallet`

Id	3S-ORDS-N02
Classification	None
Category	Optimization
Status	Acknowledged

Description

`swapExactTokensForETHSupportingFeeOnTransferTokens()` specifies the `to` address as `address(this)` and then sends the funds to the owner, but it could send the funds directly to the owner.

3S-ORDS-N03

Useless token approvals

Id	3S-ORDS-N03
Classification	None
Category	Optimization
Status	Acknowledged

Description

`OrdiswapToken` sets approvals of itself to the pools and , but this approval is never used as the token contract never makes calls to the pools that require approval. The only approval it needs is in **`addLiquidityETH()`**, but this approval is set to the router already in the constructor.

The approval to the router of the **`uniswapV2Pair`** is also not required as it does not ever remove liquidity.

This approval is not necessary, as it is set to the max in the constructor.

3S-ORDS-N04

Storage variables should be cached to memory

Id	3S-ORDS-N04
Classification	None
Category	Optimization
Status	Acknowledged

Occurences

`operationsWallet = owner();` and `_excludeFromFees(owner(), true);` should be **`msg.sender`**. **`_excludeFromFees(operationsWallet, true);`** is not necessary as the **`operationsWallet`** is the **`owner`** and it has already been excluded.

`uniswapV2Pair`, `uniswapV3Pool10000`, `uniswapV3Pool3000`, `uniswapV3Pool500`. `uniswapV3Pool100`

`addLiquidityEth()` **`msg.sender`** instead of **`owner()`**

`balanceOf(address(this))` should be cached

`swapEnabled`

`taxesEnabled`

`swapTokensAtAmount`

`buyFees`

`sellFees`

`operationsWallet` 1 2

`IERC20(tkn).balanceOf(address(this))`

`_isBot[msg.sender]` should be checked only if **`msg.sender != from`** and `_isBot[tx.origin]` only if **`tx.origin != from && tx.origin != msg.sender`**

`owner()` should be cached.

3S-ORDS-N05

Lock modifiers should use a non null variable to save gas

Id	3S-ORDS-N05
Classification	None
Category	Optimization
Status	Acknowledged

Description

It's more expensive setting a storage variable from null to non null than from non null to non null.

The `swapping` variable should follow the same pattern as in Openzeppelin's `ReentrancyGuard`.

3S-ORDS-N06

Hardcoded values should be placed as constants for better readability

Id	3S-ORDS-N06
Classification	None
Category	Suggestion
Status	Acknowledged

Occurences

`_swapping`

`totalSupply`

`uniswapV2Router`

`uniswapV3Router`

`swapTokensAtAmount`

minimum `swapTokensAtAmount`

maximum `swapTokensAtAmount`

maximum `buyFeeds`

maximum `sellFees`

percent denominator