QuillAudits

Audit Report
April, 2023

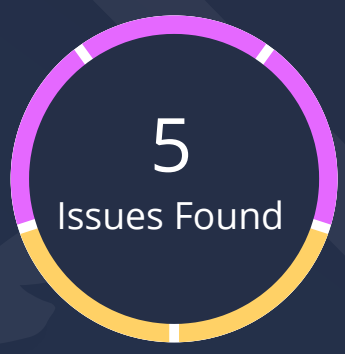
For

UrDEX

Table of Content

Executive Summary

Project Name	urDex
Overview	urDexTimelock is a smart contract that implements a timelock mechanism for executing transactions, whose ownership will be transferred to dao contract.
Timeline	15 April, 2023 to 17 April, 2023
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse urDex codebase for quality, security, and correctness. *https://github.com/Urdex-finance/Urdex-contract/tree/main/contracts* Commit hash: a7020ae279d4fb7e9bfe3be28cdf6cbe9386a320 Contracts under Audit Scope: 1]UrdexTimelock.sol 2]UrdToken.sol
Fixed in	c427000ac533fabb6621034d0017c3c678c5d17c



High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	3
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	2	0	0

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - urDexTimelock.sol

High Severity Issues

No issues were found

Medium Severity Issues

A1. Creating invalid calldata

When a signature is passed while executing transaction with executeTransaction() , else block on L120 gets executed it is using abi.encodeWithSignature which takes first parameter as signature passed to the function and second it takes bytes data. encodeWithSignature() takes function arguments in their data types but currently, values are getting converted to bytes data and getting passed to it. the calldata that it creates is different that what it should create when function argument are passed to it.

```
116
117             bytes memory callData;
118             if (bytes(signature).length == 0) {
119                 callData = data;
120             } else {
121                 callData = abi.encodeWithSignature(signature, data);
122             }
```

Recommendation

when taking signature a function can contain a multiple number of parameters with any supported data types; it is difficult to actually support multiple parameters in this logic when it comes to passing multiple parameters to encodeWithSignature(). we recommend taking calldata directly while executing a transaction and or use different logic which will allow to specify bytes data with to create calldata.

Status

Resolved

A2. Incorrect variable is getting checked

On L26 and L34, in constructor and in setDelay() function, delay is getting checked intead of _delay while checking argument is greater than MAXIMUM_DELAY delay range.

```
25          constructor(address _admin, uint256 _delay) {
26              if (_delay < MINIMUM_DELAY || delay > MAXIMUM_DELAY) {
27                  revert ValueNotInRange(_delay, MINIMUM_DELAY, MAXIMUM_DELAY);
28              }
29              admin = _admin;
30              delay = _delay;
31          }
32
33          function setDelay(uint256 _delay) public onlyTimelock {
34              if (_delay < MINIMUM_DELAY || delay > MAXIMUM_DELAY) {
35                  revert ValueNotInRange(_delay, MINIMUM_DELAY, MAXIMUM_DELAY);
36              }
```

Recommendation

Use _delay instead of delay on highlighted places.

Status

Resolved

Low Severity Issues

No issues were found

Informational Issues

A3. Unlocked pragma (pragma solidity ^0.8.17)

Contracts are using floating pragma (pragma solidity ^0.8.17) Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version gets selected while deploying a contract which has higher chance of having bugs in it.

Recommendation

Remove floating pragma and use a specific compiler version with which contracts have been tested.

Status

Acknowledged

A4. Care needs to be taken while designing the DAO contract

The Dao contract wasn't in the scope for this audit, We recommend designing the DAO in a way which won't break any functionality of the timelock contract as the DAO would be handling the timelock.

Status

Acknowledged

B. Contract - urdToken.sol

High Severity Issues

No issues were found

Medium Severity Issues

No issues were found

Low Severity Issues

No issues were found

Informational Issues

A3. Unlocked pragma (pragma solidity ^0.8.17)

Contracts are using floating pragma (pragma solidity ^0.8.17) Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version gets selected while deploying a contract which has higher chance of having bugs in it.

Recommendation

Remove floating pragma and use a specific compiler version with which contracts have been tested.

Status

Acknowledged

Functional Testing

Some of the tests performed are mentioned below:

urdToken.sol

- Should get decimals of the contract.
- Should get totalSupply of the contract.
- Should get Name of the Contract.
- Should be able to Increase Allowance.
- Should be able to Derease Allowance.
- Should be able to transfer tokens.

UrdexTimelock.sol

- Should be able to queue the transaction
- Should be able to cancel the queued transaction
- Should be able to execute the transaction without signature
- Only pending admin should be able to accept ownership
- Reverts when transaction executed before eta
- Reverts when transaction executed after eta plus GRACE_PERIOD
- Reverts when unauthorized address calls queueTransaction
- Reverts when unauthorized address calls cancelTransaction
- Reverts when unauthorized address calls executeTransaction
- Should be able to execute the transaction when signature is passed

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the urDex Contracts. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the urDex Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the urDex Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+
Audits Completed

$16B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
April, 2023

For

UrDEX

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com