QuillAudits

Audit Report, April, 2024

For

TEGRO

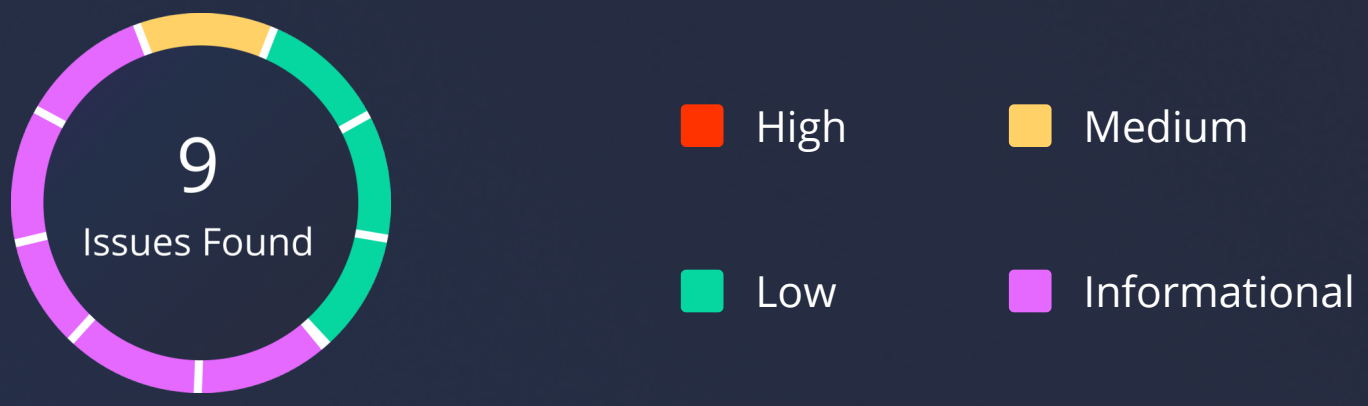
Table of Content

Table of Content

Executive Summary

Project Name	Tegro Dex
Overview	The project contains two contracts in scope, TegroDex is the contract that facilitates the actual order settlements. The seller can create and sign the order to sell any ERC20 tokens. The buyer can create and sign the order to buy tokens using the created sell order. The contract allows partial order settlements. TegroDEXSettlement allows functionality for settling orders in the batch.
Timeline	18 March 2024 - 29 March 2024
Update code Received	1st April 2024
Second Review	1st April 2024 - 4th April 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this audit was to analyse the **Tegro Dex Contrac**t for quality, security, and correctness.
Source code	*https://github.com/ashtegro/tegroDEX/tree/main/contracts*
Contracts In-Scope:	1. Settler.sol 2. TegroDEX.sol
Branch	main
Fixed in	f5f3b673e82054d126d1ebabb6440038fccf3b1d

Number of Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	1	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	1	2	4

Checked Vulnerabilities

We scanned the application for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- Transaction-Ordering Dependence
- Redundant fallback function
- ERC20 API violation
- ERC721 API violation
- Malicious libraries
- Compiler version not fixed
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

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

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC's 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 analyzed 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

A 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 analyzed, 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 behavior 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

Hardhat, Foundry.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

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 Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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 four 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.

Issues Found

High Severity Issues

No issues were found.

Medium Severity Issues

1. Incorrect totalPrice calculation

Path

TegroDEX.sol

Function

_calculateTotalPrice()

Description

When **baseTokenDecimals** would be greater than **quoteTokenDecimals** the function is unable to calculate the correct totalPrice.

Let's say the price is 1e6 (i.e. using **quoteTokenDecimals**) and the quantity is 2e18 (i.e. using baseTokenDecimals which is greater than the quoteTokenDecimals).

In **_calculateTotalPrice()** L261 would be executed where it will calculate the price with 1e6 * 10 ** (18 - 6) which would be 10000000000000000

Now, on L265 it will set totalPrice = 10000000000000000 * 2e18 = 20000000000000000000000000000000000

Then if{} on L268 will execute where it will scale down the totalPrice to remove extra decimals: totalPrice = 20000000000000000000000000000000000 / 10 ** (18 + 6) = 2000000000000

So the calculated totalPrice would be 2000000000000 which is not equal to 2e6.

The calculated totalPrice should be 2e6 because if the buyer is buying two tokens with the price 1e6 then the buyer needs to pay only 2e6 for two tokens.

Recommendation

Consider multiplying the price with quantity and scaling down the answer so that the answer would be in the **quoteTokenDecimals** i.e. the decimals that the price is using. So the **_calculateTotalPrice()** will only contain the following calculation which will take care of scaling down the numerator.

E.g : (price * quantity)/ (10 ** **baseTokenDecimals**)
= (1e6 * 2e18)/(10** 18)
= 2000000000000000000000000 / 1000000000000000000
= 2000000 (i.e 2e6)

Status

Resolved

Low Severity Issues

2. Unfixed pragma

Path

TegroDEX.sol#L2, Settler.sol#L2

Function

-

Description

TegroDEXSettlement and TegroDEX are not using fixed solidity versions. Contracts are using unfixed pragma (^0.8.18 and ^0.8.19), 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. The most recent compiler version may get selected while deploying a contract which has a higher chance of having bugs in it.

Recommendation

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

Status

Resolved

3. Add expiry timestamp

Path

TegroDEX.sol#L81

Function

settleOrders()

Description

The expiry timestamp can be added as a part of the buy and sell order, which can be used while creating an order hash. And it can be checked in the **settleOrders()** that while executing the orders, **buyOrder.expiry >= block.timestamp && sellOrder.expiry >= block.timestamp.**

This will ensure that the order can't be executed after a specific timeframe.

(It should be noted that the functionality of having expiry depends on the business logic.)

Recommendation

Add the expiry timestamp as suggested.

Status

Resolved

4. A lower limit should be set for matchedAmount

Path

TegroDEX.sol#L207

Function

onMessageRecalled()

Description

The transaction executor can be anyone who can enter the matchedAmount. The matchedAmount can be entered in the way where the calculated **baseFeeAmount** would be rounded down to 0.

E.g While calculating fee on L207 if the feeAmount is 20 and matchedAmount is 499 then the calculation would be: 499 * 20 / 10000 = 0 (In solidity). The same can happen while calculating quoteFeeAmount using totalPrice.

The above example shows that if the numerator is less than the denominator then the answer can be rounded down to 0 which will allow an executor to bypass the fees on a small portion of the amount.

The successful attack depends on the amount of tokens that are getting sold/bought, but to perform this attack recursively to bypass the fee, it might not be feasible as every transaction would cost the gas (also depends on who pays the fee or who is executing the signed orders). But it would be good practice to add a lower limit for the amount so that if the **matchedAmount** is less than that specified limit then the transaction will revert.

Recommendation

Add a lower limit for **matchedAmount** in the **settleOrders()** so that it will check if the entered **matchedAmount** is greater than or equal to a certain limit.

The code to add in **settleOrders()** would look like this:

require((matchedAmount * feeAmount) >= 10000, “Enter bigger matchedAmount”);

Status

Acknowledged

Informational Issues

5. _disableInitializers() should be called in the constructor

Path

TegroDEX.sol , Settler.sol

Function

-

Description

In the proxy implementations **_disableInitializers()** should be called in the constructors. If not added, malicious users/addresses can call initialize() on the implementation smart contract where they would be able to set certain values that initialize() allowed to set.

The severity of impact depends on the contract logic, but it's a good practice to add **_disableInitializers()** call in the implementation logic.

Recommendation

Consider adding _disableInitializers() in constructor as suggested here: https://docs.openzeppelin.com/contracts/4.x/api/proxy#Initializable-_disableInitializers–

Status

Resolved

6. Fix the comment

Path

TegroDEX.sol

Function

-

Description

settleOrders() contains the comment on the L97 //**Require the buy price to always be higher than the sell price**. Currently, the required check allows **buyOrder.price** to be equal to **sellOrder.price**. So the comment can be changed to //**Require the buy price to always be higher than or equal to the sell price.**

Recommendation

Change the comment to match the functionality.

Status

Resolved

7. Discrepancy while checking the price

Path

TegroDEX.sol

Function

settleOrders(), _transferTokens()

Description

In **settleOrders()** on **L98** it checks/requires that **buyOrder.price >= sellOrder.price** after that in the **_transferTokens()** on the **L192** it sets the smallest value to **matchedPrice** by using this expression **buyOrder.price < sellOrder.price ? buyOrder.price : sellOrder.price;** but because it is already checked on **L98** the **buyOrder.price** should be greater than or equal to **sellOrder.price, buyOrder.price** would be always greater than or equal to **sellOrder.price** so on **L192** the condition will always fail and **sellOrder.price** will get assigned to the **matchedPrice**.

Recommendation

Consider using Ownable2StepUpgradable instead of OwnableUpgradable.

Status

Resolved

8. Note regarding fee-on-transfer tokens

Path

TegroDEX.sol

Function

-

Description

While using fee-on-transfer tokens (as baseToken or quoteToken), the amount that will get actually transferred would be less than what is getting transferred because of fee deductions.

This doesn't create any security issues but can create confusion on the receiver side.

Recommendation

Care should be taken while dealing with fee-on-transfer tokens in the orders.

Status

Acknowledged

9. Unused import

Path

Settler.sol#L5

Function

-

Description

OwnableUpgradeable is getting imported and is getting inherited by the **TegroDEXSettlement** contract. But no functionality is getting used from the **OwnableUpgradeable** and it can be removed if it remains unused.

Recommendation

Care should be taken while dealing with fee-on-transfer tokens in the orders.

Status

Resolved

Functional Tests Cases

Some of the tests performed are mentioned below:

TegroDEX:

- ✓ Should be able to fully settle the order.
- ✓ Should be able to partially settle the order.
- ✓ Should be able to execute the order correctly when baseDecimals are less than quoteDecimals.
- ✓ Should be able to execute the order correctly when baseDecimals are equal to quoteDecimals.
- ✓ Should be able to execute the order correctly when baseDecimals are greater than quoteDecimals.
- ✓ Should be able to cancel the order.
- ✓ Owner should be able to set the fee recipient.
- ✓ Owner should be able to set the fee amount.
- ✓ Reverts if the order signature is invalid.
- ✓ Reverts if the order is already filled.
- ✓ Reverts if the order is already canceled.
- ✓ Reverts if tries to execute the order after the expiry.
- ✓ Should not be able to cancel the already filled order.
- ✗ Should revert when the amount is less than a certain limit which allows to bypass the fees

TegroDEXSettlement:

- ✓ Should be able to execute the orders in the batch.
- ✓ Should be able to execute the partial order in the batch.
- ✓ emits the event when any order settlement execution fails.

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 **Tegro Dex** codebase. 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. Ind the End, the Tegro team resolved 1 medium,1 low and 4 informational issues and Acknowledged other issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in **Tegro Dex** smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of **Tegro Dex** smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the **Tegro Dex** to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

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.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
April, 2024

For

TEGRO

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com