QuillAudits

Audit Report
November, 2022

For

ZENLAND

Table of Content

Executive Summary

Project Name	Zenland Escrow Contract
Overview	Zenland escrow contract is a blockchain platform where buyer and seller can create deals which after fulfilling the payments will be done. This is done through a smart contract, a simple programmable agreement between two parties. Such agreement is written in the form of computer code and stored on the blockchain. Smart contracts are simpler and faster alternatives to formal agreements. They can easily be created and deployed through the Zenland platform by anyone
Timeline	28 October,2022 - 3 November,2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Zenland Escrow Contract codebase for quality, security, and correctness. *https://github.com/zenland-dao/contracts/blob/main/escrow/Escrow_v1.0.0.sol* Commit hash: 6bde30158fe7783a292ad2661a106f8f4c013f9d
Fixed In	*https://github.com/zenland-dao/contracts/blob/main/escrow/Escrow_v1.0.0.sol* **Commit hash:** 91a9b907803926959b645a83b91f13ecf779dd54

High
Medium
Low
Informational

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

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

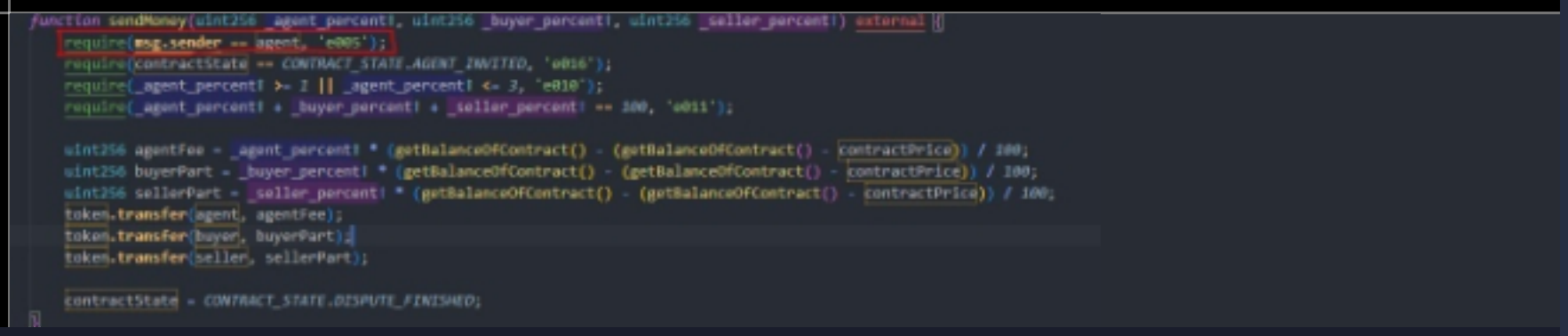
A. Contract - Zenland Escrow

High Severity Issues

No issues found

Medium Severity Issues

A.1: sendMoney() controlled by malicious agent causing issues in payments

Line	sendMoney()
637-651	

```
function sendMoney(uint256 _agent_percent!, uint256 _buyer_percent!, uint256 _seller_percent!) external {
    require(msg.sender == agent, 'e005');
    require(contractState == CONTRACT_STATE.AGENT_INVITED, 'e016');
    require(_agent_percent! >= 1 || _agent_percent! <= 3, 'e010');
    require(_agent_percent! + _buyer_percent! + _seller_percent! == 100, 'e011');

    uint256 agentFee = _agent_percent! * (getBalanceOfContract() - (getBalanceOfContract() - contractPrice)) / 100;
    uint256 buyerPart = _buyer_percent! * (getBalanceOfContract() - (getBalanceOfContract() - contractPrice)) / 100;
    uint256 sellerPart = _seller_percent! * (getBalanceOfContract() - (getBalanceOfContract() - contractPrice)) / 100;
    token.transfer(agent, agentFee);
    token.transfer(buyer, buyerPart);
    token.transfer(seller, sellerPart);

    contractState = CONTRACT_STATE.DISPUTE_FINISHED;
}
```

Description

In function sendMoney()if say buyer open dispute so status changed to DISPUTED. So as you know when a dispute is created means an agent is required so status changed to AGENT_INVITED but somehow those guys agree to resolve. So now the agent calls the function sendMoney() setting the percentage for each 3.
say agent = 2
buyer = 49
seller = 49
assuming there is no other way for buyer and seller to communicate other than chat it goes correct.
But is it possible even after agreeing on those values agent can make differ in percentage and calls sendMoney() with
agent = 2
buyer = 0
seller = 98

Remediation

It is not quite possible how agents will act but to resolve the issue the team need to make sure that things like this won’t happen and if it ever happens then they take care of the issue immediately.

Zenland team Comment: Unfortunately it's impossible to make sure that both parties agree on consensus in a contract. Therefore, sometimes the agent will release money from the contract even if the seller or a buyer doesn't agree. Agent will try to find an agreement between them, but if an agent can clearly see that one of the parties is trying to scam the other - the agent will decide based on the evidence.

Currently all Zenland agents are formal employees of Zenland, therefore all cases would be discussed and supervised by the whole team to form the right guide for future decentralized agents to work with.

When we form a DAO, buyer and seller will have a chance to report an agent to the entire DAO, and the DAO will vote if the agent did the right thing.

Status

Acknowledged

Low Severity Issues

A.1: sendMoney() controlled by malicious agent causing issues in payments

Line	release()
57-71	

```
function confirmFulfillment() external {
    require(msg.sender == seller, 'e004');
    require(ContractState == ContractStateChoices.DEPLOYED, 'e013');
    require(getBalanceOfContract() >= contractPrice, 'e002');
    ContractState = ContractStateChoices.FULFILLED;
    executionTimestamp = getCurrentTimestamp();
}

ftrace | funcSig
function release() external {
    require(msg.sender == buyer || msg.sender == seller, 'e012');
    require(
        ContractState == ContractStateChoices.DEPLOYED ||
        ContractState == ContractStateChoices.FULFILLED ||
        ContractState == ContractStateChoices.DISPUTED,
    'e017');
    require(getBalanceOfContract() >= contractPrice, 'e002');
    if (msg.sender == seller) {
        require(getCurrentTimestamp() > executionTimestamp + buyerProtectionTime, 'e009');
        require(ContractState == ContractStateChoices.FULFILLED, 'e014');
    }
    token.transfer(seller, contractPrice);
    ContractState = ContractStateChoices.EXECUTED;
}
```

Description

So if you look at the above code, confirmFulfillment() is called by the seller(by the user who has done the work). status changed to FULFILLED
but the buyer(the one who posted the work) didn't like the work. If you see the function release it can be called by the seller or buyer. As the buyer didn't like the work he won't call release() function. He is going to call the openDispute() function to get a middle man to solve the issue.
but say the seller realizes that and he calls the function release() with high gas and front run buyers function call in release() function seller can only call it after (buyerProtectiontime + executionTimestamp) and when status is FULFILLED can transfer the amount to the seller's address.

Remediation

The above situation is only possible in extreme conditions say the buyer for some reason forgets to check the work in protection time. By that time he calls the openDispute() the seller realizes and front-run the buyers call. One remediation would be having longer buyerProtection time or letting the buyer know(sending notification) about the work done at (execution time by the seller)

Status

Resolved

Informational Issues

A.3: Unlocked pragma (pragma solidity ^0.8.14)

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Here all the in-scope contracts have an unlocked pragma, it is recommended to lock the same.

Status

Resolved

A.4 General Recommendation

Assuming token address will be selected from a set of available tokens on the front-end, it won't allow zero address as token. Otherwise add a check for zero address for token in constructor.
In contract Escrow_V100, buyer, seller, agent, contractPrice, buyerProtectionTime, version can be set as immutable to save gas.

Status

Resolved

Functional Testing

- [PASS] testConfirmFulfillment() (gas: 110862)
- [PASS] testDispute() (gas: 115994)
- [PASS] testFailReleaseBySellerIfNotFulfilled() (gas: 122453)
- [PASS] testInviteAgent() (gas: 117786)
- [PASS] testRelease() (gas: 124196)
- [PASS] testReturnMoney() (gas: 129784)
- [PASS] testSendMoneyThroughAgent() (gas: 167081)
- [PASS] testValues() (gas: 25819)

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 Zenland Escrow Contract. 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 ZenlandDAO 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 Zenland DAO 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.

600+
Audits Completed

$15B
Secured

600K
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2022

For

ZENLAND

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com