pepper sec

PUBLIC REPORT

Aira's Robonomic

Smart Contract Security Audit

Clarity is a rare commodity. That is why for the convenience of both the client and the reader, we have introduced a system of marking vulnerabilities and security issues we discover during our security audits.

Let's start with an ideal case. If an identified security imperfection bears no impact on the security of our client, we mark it with the No issue label.

The fixed security issues get the ✓ Fixed label that informs those reading our public report that the flaws in question should no longer be worried about.

In case a client addresses an issue in another way (e.g., by updating the information in the technical papers and specification) we put a nice Addressed tag right in front of it.

If an issue is planned to be addressed in the future, it gets the Acknowledged tag, and a client clearly sees what is yet to be done.

Although the issues marker with ✓ Fixed and Acknowledged are no threat, we still list them to provide the most detailed and up-to-date information for the client and the reader.

TABLE OF CONTENTS

08. CONGRESS (MULTISIG)

Dangerous function Addressed

Solidity version is too old Addressed

APPENDIX 1. TERMINOLOGY

Severity

Source code

Object	Location
Robonomics_contracts	**#cc35a91de187072214d215262d8371f0159c2498**

Audit methodology

The code of a smart contract has been automatically and manually scanned for known vulnerabilities and logic errors that may cause security threats. The conformity of the requirements (e.g., White Paper) and practical implementation has also been reviewed. See more information on the methodology **here**.

Auditors

Alexey Pertsev. **PepperSec**.

Discovered vulnerabilities

Below, you can find a table with all the discovered bugs and security issues listed.

Vulnerability description	Severity	Paragraph
Token stealing	Critical	**Lighthouse**
Possible keccak256 collisions	Major	**RobotLiability**
Gas improvement	Medium	**Ambix**
Dangerous function		**Congress (MultiSig)**
Validation after storing	Minor	**Ambix**
Possible re-entrancy at withdraw		**Lighthouse**
The use of tx.origin		
Possible re-entrancy at finalize		**RobotLiability**
Possible integer overflow		**LiabilityFactory**
Outdated Openzeppelin lib version		**XRT**
Solidity version is too old		**Congress (MultiSig)**
Address colliding around zero index in indexOf	None	**Lighthouse**
Possible integer overflow		

Gas improvement

► **Severity:** Medium

Ambix.sol#L35. The `appendSource` function can be more Gas effective. It uses the **for loop** to validate the input and add elements to arrays. The separation of these actions can save sufficient amount of Gas.

Recomendations:

1. `appendSource` may look as follows:

```
function appendSource(address[] _a, uint256[] _n) public onlyOwner {

    require(_a.length == _n.length);



    for (uint256 i = 0; i < _a.length; ++i) {

        require(_a[i] != 0);

    }



    A.push(_a);

    N.push(_n);

}
```

This aproach takes 131191 Gas less per 10 elements than the original function. (or $1.26 with GasPrice of 20 Gwei)

Status:

Fixed – **#818a90313e32a74dbdd32164281c5a733d49fe76**

Validation after storing

► **Severity:** Minor

Ambix.sol#L82. The `run` function makes its decision, according to the first item of the `N` array (token value coefficients), if it is equal to zero than a function starts `Dynamic conversion` and it is presumed that `A[ix]` and `B[ix]` have the length of == 1. However, there are no guarantees that the length actually equals 1. So, the execution would be halted at **line 109** in negative case.

With this type of flow, it is still possible to submit the values that will never be processed by `run` (length of `A[ix]` and `B[ix]` exceed 1 and `N[_ix][0] == 0`).

Recommendations:

1. Consider validating `A`, `B`, and `N` before calling `run` (within the `appendSource` function, strictly speaking).

Status:

Fixed – **#6173a0270b844376366a5a54f76134891a6e8a53**

Possible re-entrancy at withdraw

► **Severity: Minor**

LighthouseLib.sol#L24. The `withdraw` function sends tokens before changing the internal `balance`. The exploitation of this behavior can result in token stealing if the actual Token meets the following circumstance: it should have the `onTokenTransfer` method (or similar) that calls the fallback function of the token reciever when an actual `transfer` has happened (e.g., ERC223 and ERC667).

Recommendations:

1. Consider swapping lines **23 and 24** and also **28 and 29**, in order to prevent potential accidents.

Status:

Fixed – **#5fdd39e7cc2c189fd44bc35bdd977c6ae4577096**

Token stealing

► **Severity: Critical**

LighthouseLib.sol#L83. The `to` function can be used to make arbitrary calls on behalf of the `Lighthouse` contract. In other words, the function can be utilized to steal tokens after someone has `approved` some amount of tokens to become a `member`.

In addition to this, `to` can be used to increase `quota` (by calling `refill` and `to`).

Recommendations:

1. Consider implementing wrappers for external calls to specific contracts instead of the `to` function.

Status:

Fixed – **#eddf51b9948e3c15ab1c70bc74438c608a0d6e6b**

Address colliding around zero index in indexOf

► **Severity:** None

LighthouseLib.sol#L83. Due to using mapping to store member index, the index is equal `0` for all unknown addresses and the first member in the `member` array. So, this collision just should be taken into account for the future development or appropriate countermeasures should be taken to prevent potential security incidents.

Recommendations:

1. Consider shifting all indexes to `+1` in the `indexOf` mapping.

Status:

Taken into account

Possible integer overflow

► **Severity:** None

LighthouseLib.sol#L52. The `quoted` modifier decreases the `quota` variable via `-=`. So, if `nextMember` has the *balance* of `< minimalFreeze`, **quota** will be equal to zero and after that underflowed (become 2**256). Current `withdraw` does not allow this, but it is worth considering to use `assert(quota != 0);` before **line 52** to avoid possible security incidents.

Status:

Fixed – **#5fdd39e7cc2c189fd44bc35bdd977c6ae4577096**

The use of tx.origin

► **Severity:** Minor

Taking into account the `Lighthouse` contract controls `LiabilityFactory`, it has the special fallback function to proxy all calls it. The interactions of the kind may lead to the use of `tx.origin` to determine the actual caller by the `LiabilityFactory` contract. However, the use of `tx.origin` is considered to be **dangerous** and is not recommended.

Recomendations:

1. Consider passing the actual caller to a call to LiabilityFactory (as an additional parameter) or check that

```
require(msg.sender == tx.origin);
```

at least in the **fallback function**.

Team's comment:

tx.origin is used for bounty transfer only

Status:

The team decided to leave it as it is.

Possible keccak256 collisions

► **Severity: Major**

RobotLiabilityLib.sol#L73. The `bid` function checks `model` and `objective` that are sent like this:

```
require(keccak256(abi.encodePacked(model, objective)) ==
        keccak256(abi.encodePacked(_model, _objective)));
```

An obvious disadvantage of this approach is that it is vulnerable to collisions. Therefore, it cannot be considered reliable.

Example:

```
keccak256(abi.encodePacked("\x60\x8b","\x00\x29")) ==
keccak256(abi.encodePacked("\x60","\x8b\x00\x29")) // true
```

Recommendations:

1. Use `abi.encode` instead of `abi.encodePacked`, so that the information about `length` is included into the hash.

Status:

Fixed – **#eddf51b9948e3c15ab1c70bc74438c608a0d6e6b**

Possible re-entrancy at finalize

► **Severity: Minor**

RobotLiabilityLib.sol#L102. The `finalize` function sends tokens before changing `isFinalized` to `true` (see the explanation **above**).

Recommendations:

1. Consider moving **line 135** to **line 112**.

Status:

Fixed – **#5fdd39e7cc2c189fd44bc35bdd977c6ae4577096**

Possible integer overflow

► **Severity:** Minor

LiabilityFactory.sol#L222. The `liabilityFinalized` function does not check the input arg. In case of `_gas` being less than `gasleft()`, LiabilityFactory mints a huge amount of tokens (because of the integer overflow).

Recomendations:

1. Due to the current workflow, there is no appropriate way to exploit the function. Still, it is worth considering adding `assert(_gas >= gasLeft())` to avoid security incidents in the future.

Status:

Fixed – **#1e30dfe6182b46e03e70d05a44c986ca9d47bd88**

Outdated Openzeppelin lib version

► **Severity:** Minor

The **project** uses old version OpenZeppelin lib.

Recommendations:

1. Consider updating the lib to get up-to-date improvements and patches.

Status:

Fixed – **23b4227a8eab214f2abb1b19913d5f295c25c71c**

Dangerous function

► **Severity: Medium**

Congress.sol#L133. The `receiveApproval` function is used to receive the tokens that have been approved by someone. The function can be called by anyone with an arbitrary Token address. After that, the contract just calls the `transferFrom` function of it.

The described approach is dangerous because there is no guarantee *Token address* is the address of a real Token. In case of getting control over certain smart contract by Congress multisig, an attacker can use the `receiveApproval` function to call the contract on behalf of Congress multisig, which may lead to unexpected consequences.

Recommendations:

1. Consider using `executeProposal` for this purpose or at least adding an access control modifier for `receiveApproval`.

Status:

Taken into account

Solidity version is too old

► **Severity: Minor**

The Solidity version (v0.4.9+commit.364da425) used in the contract is too old. The latest version has a bunch of improvements that can be extremely useful:

- keywords: `constructor`, `require`, `emit` – to keep code more readable;
- the `abi.encode()` function to encode args. That could be used to prepare args before hashing (lines 368, 393, 444);
- fixed compiler bugs (e.g., of **zero string literal** used in the contract).

Recommendations:

1. Consider improving the code in case of redeploying.

Status:

Taken into account.

Appendix 1. Terminology

Severity

Severity is the category that described the magnitude of an issue.

		Severity		
Impact	Major	Medium	Major	Critical
	Medium	Minor	Medium	Major
	Minor	None	Minor	Medium
		Minor	Medium	Major
		Likelihood		

Minor

Minor issues are generally subjective in their nature or potentially associated with the topics like “best practices” or “readability”. As a rule, minor issues do not indicate an actual problem or bug in the code.

The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

Medium

Medium issues are generally objective in their nature but do not represent any actual bugs or security problems.

These issues should be addressed unless there is an apparent reason not to.

Major

Major issues are things like bugs or vulnerabilities. These issues may be unexploitable directly or may require a certain condition to arise to be exploited.

If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations which make the system exploitable.

Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

If unaddressed, these issues are likely or guaranteed to cause major problems and ultimately a full failure in the operations of the contract.

About Us

Worried about the security of your project? You're on the right way! The second step is to find a team of seasoned cybersecurity experts who will make it impenetrable. And you've just come to the right place.

PepperSec is a group of whitehat hackers seasoned by many-year experience and have a deep understanding of the modern Internet technologies. We're ready to battle for the security of your project.

LET'S KEEP IN TOUCH


peppersec.com


hello@peppersec.com


Github