Bestcoins

Smart Contract Audit Report for UCoin

The document contains a thorough analysis of UCoin Smart Contract based on guidelines set forth by Bestcoins

Crafted by – Bestcoins.co

5/25/2018

Table of Contents

Overview

Overview

This report presents the findings of the security of the UCoin smart contract project. The scope of our task is to review and report the security issues in the Smart Contracts of the platform in line with the industry practice as at the date of this report.

Disclaimer

Disclaimer

This audit documentation reflects the evaluation of security flaws and vulnerabilities as identified by the BestCoins team, and is purely based on the result of the analysis. This document makes no statements on the viability of the project, or the safety of its contracts. This audit is not meant to represent any investment advice.

Procedure & Findings

Procedure & Findings

For our understanding, we considered UCoin Smart Contracts code provided on 22nd May 2018. The audit used a variety of techniques that involved both automated analysis and manual analysis. The audit team checked each code line and compared the logic with the contract specification document.

3.1 Automated Analysis

We used a static code analyzer which helped us to run the analysis in the Solidity source code. This helped us to automatically detect the potential vulnerabilities and security flaws.

Here are the combined results of their analysis.

Tool	Issue type	Number of issues
smartdec	Reentrancy external call	10
smartdec	Using the *approve* function of the ERC-20 standard	2
smartdec	Functions *transfer* and *transferFrom* of ERC-20 Token should throw	2
smartdec	No payable fallback function	10
smartdec	Compiler version not fixed	1
smartdec	Implicit visibility level	2
securify	*Transaction Reordering*	2
securify	*Use of Untrusted Inputs in Security Operations*	1

3.2 Manual Analysis

Each and every code line of the Contracts was manually reviewed with a keen focus on the logic of the code. Care was also taken to check whether the codes are in conjunction with the contract specification document. In addition to checking the codes manually, the process also involved manual verification of the results produced by the automated analysis tools. The analysis declared that the project does not contain any serious vulnerability. Besides few minor issues, there were no discrepancies found between the Smart Contracts and the specification document.

Terminology

This audit categorizes vulnerabilities using the Bestcoins Risk Rating Method based on Severity. Every vulnerability is given severity rating, which are used to give a more accurate estimation of vulnerability's overall severity.

3.2.1 High Severity

1). Reentrancy

Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed.

This pattern is experimental and can report false issues. This pattern might also be triggered when

- accessing struct's field
- using enum's element

Example:

To give an example, the following code contains a bug (it is just a snippet and not a complete contract):

```
pragma solidity ^0.4.0;
// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract Fund {
    /// Mapping of ether shares of the contract.
    mapping(address => uint) shares;
    /// Withdraw your share.
    function withdraw() {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

The problem is not too serious here because of the limited gas as part of send, but it still exposes a weakness: Ether transfer always includes code execution, so the recipient could be

a contract that calls back into withdraw. This would let it get multiple refunds and basically retrieve all the Ether in the contract.

3.2.2 Medium Severity

1). Compiler version not fixed

Solidity source files indicate the versions of the compiler they can be compiled with.

pragma solidity ^0.4.17; // bad: compiles w 0.4.17 and above

pragma solidity 0.4.17; // good : compiles w 0.4.17 only

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Example:

```
pragma solidity ^0.4.5;
contract StandardToken is ERC20, BasicToken {
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
```

```
    }
}
```

2). The approve function of ERC-20 might lead to vulnerabilities.

Example:

In the following contract the function approve is used

```
pragma solidity ^0.4.5;
contract StandardToken is ERC20, BasicToken {
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }
}
```

3.2.3 Low Severity

1). HardCoded Address

The contract contains unknown address. This address might be used for some malicious activity. Please check hardcoded address and its usage.

2). DoS by external function call in require

The contract that calls functions of other contracts should not rely on results of these functions. Be careful when verifying the result of external calls with require as called contract can always return false and prevent correct execution. Especially if the contract relies on state changes made by this function.

This pattern is experimental and can report false issues. This pattern might be also triggered when

- accessing struct's field.
- using enum's element.

3). Function *transfer* **and** *transferFrom* **should throw**

Functions of ERC-20 Token Standard should throw in special cases:

- *transfer* should throw if the _from account balance does not have enough tokens to spend.
- *transferFrom* should throw unless the _from account has deliberately authorized the sender of the message via some mechanism.

Conclusion

Conclusion

Although no major security issues were found, the audit recommends few changes that can reduce the potential attack surface. Overall, the code is well written, modularized logically and the quality is appreciated. Also, the audit team did not identify any unnecessary complex functions. Most of the code is very well commented and aligns well with the high technical capabilities.