SECURITY AUDIT OF IICO SMART CONTRACT

AUDIT REPORT

MAY 13, 2018

✓erichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology >> Forward

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on May 13, 2018. We would like to thank Kleros to trust Verichains Lab to audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the smart contracts. The scope of the audit is limited to the source code files provided to Verichains Lab on May 09, 2018. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques in 03 days.

The assessment identified some issues in IICO smart contracts code. Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

CONTENTS

ACRONYMS AND ABBREVIATIONS

Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
ETH (Ether)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
EVM	Ethereum Virtual Machine.

AUDIT OVERVIEW

ABOUT INTERACTIVE COIN OFFERING

Kleros is a blockchain Dispute Resolution Layer that provides fast, secure and affordable arbitration for virtually everything.

Kleros implements the Interactive Coin Offering token sale smart contract following paper from published by Jason Teutsch, Vitalik Buteri and Christopher Brown at https://people.cs.uchicago.edu/~teutsch/papers/ico.pdf

SCOPE OF THE AUDIT

This audit focused on identifying security flaws in code and the design of the smart contracts. It was conducted at commit `e67e647` of branch `master` from GitHub repository of OpenIICO Contract.

Repository URL: https://github.com/kleros/openiico-contract/tree/e67e64753d934c464356558848a61db20c7892b6

Source File	SHA256 Hash
IICO.sol	`9a95a5d9f68b14a70c43af1d7702d0e25889ba06653a13c0d98bfc57a459c433`
LevelWhitelistedIICO.sol	`3c17a3be4437018967b2303fbcec0a372481a13a4aafc8dcc89587fea96d93f6`

AUDIT METHODOLOGY

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- TimeStamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- Dos with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories, depending on their criticality:

LOW	An issue that does not have a significant impact, can be considered as less important
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.

AUDIT RESULT

VULNERABILITIES FINDINGS

HIGH
INCORRECT ORDERING OF BIDS FOR SAME VALUATION VALUE

Bids are ordered in ascending order of maxValuation and then id, which is decided in submitBid method by the ordering requirement:

```
require(_maxValuation >= prevBid.maxValuation && _maxValuation < nextBid.maxValuation); //
The new bid maxValuation is higher than the previous one and strictly lower than the next
one.
```

At final stages, the bid's cut point is calculated by traversing from last (highest bid) to first (lowest bid), this scheme convince users to "spam" bids by resubmit last user's maxValuation because the later bid has more chance to be selected at final stage.

Recommended fixes

- Change bids list's ordering to ascending order of maxValuation and then in descending order of id to ensure earlier bids are selected for bids with same maxValuation:
    - Change the ordering check in submitBid from:

```
require(_maxValuation >= prevBid.maxValuation && _maxValuation < nextBid.maxValuation);
```

to

```
require(
    ((_maxValuation > prevBid.maxValuation) || (prev == HEAD))
    && (_maxValuation <= nextBid.maxValuation)
);
```

    - Change the accepted bid condition check in redeem:

```
    function redeem(uint _bidID) public {
        Bid storage bid = bids[_bidID];
        Bid storage cutOffBid = bids[cutOffBidID];
        require(finalized);
```

```
        require(!bid.redeemed);

        bid.redeemed=true;
        if (bid.maxValuation > cutOffBid.maxValuation || (bid.maxValuation ==
cutOffBid.maxValuation && _bidID >= cutOffBidID)) // Give tokens if the bid is accepted.
            token.transfer(bid.contributor, (tokensForSale * (bid.contrib + (bid.contrib *
bid.bonus) / BONUS_DIVISOR)) / sumAcceptedVirtualContrib);
        else
// Reimburse ETH otherwise.
            bid.contributor.transfer(bid.contrib);
    }
```

to

```
if (bid.maxValuation > cutOffBid.maxValuation || (bid.maxValuation == cutOffBid.maxValuation
&& _bidID <= cutOffBidID))
```

- And disable submitBid in default method or adding an infinityTail which updates every time state variable to be used instead of TAIL.

LOW

POSSIBLE INTEGER OVERFLOW IN METHOD BONUS

Bid's bonus amount is calculated depend on time using bonus function and might lead to unexpected behaviors by malicious creation parameters.

```
function bonus() public view returns(uint b) {
    if (now < endFullBonusTime) // Full bonus.
        return maxBonus;
    else if (now > endTime)     // Assume no bonus after end.
        return 0;
    else                        // Compute the bonus decreasing linearly from
endFullBonusTime to endTime.
        return (maxBonus * (endTime - now)) / (endTime - endFullBonusTime);
}
```

In the above code from IICO contract, the multiplication maxBonus * (endTime - now) could be overflowed to return not-intended values.

Recommended fixes

- Add check to constructor to ensure no integer overflow of bonus calculation:

```
require(
     (_maxBonus == 0)
  || (endTime == endFullBonusTime)
  || (_maxBonus * (endTime - endFullBonusTime)) / (endTime - endFullBonusTime) == _maxBonus
);
```

OTHER RECOMMENDATIONS / SUGGESTIONS

- Add require(_maxValuation <= INFINITY) to search function to ensure early termination to prevent possible infinity loop in search

  From the way every variables are setup, all bid's maxValuation must be less than INFINITY, which is MAX_UINT - 1, so if search function is called with MAX_UINT, the loop will run forever and consume all supplied gas.

```
function search(uint _maxValuation, uint _nextStart) view public returns(uint nextInsert) {
    uint next = _nextStart;
    bool found;

    while(!found) { // While we aren't at the insertion point.
        Bid storage nextBid = bids[next];
        uint prev = nextBid.prev;
        Bid storage prevBid = bids[prev];

        if (_maxValuation < prevBid.maxValuation)       // It should be inserted before.
            next = prev;
        else if (_maxValuation >= nextBid.maxValuation) // It should be inserted after. The
second value we sort by is bidID. Those are increasing, thus if the next bid is of the same
maxValuation, we should insert after it.
            next = nextBid.next;
        else                                    // We found the insertion point.
            found = true;
    }
```

```
    return next;
}
```

In the above code from IICO contract, if _maxValuation is MAX_UINT ($2^{256} - 1$), the second **if** will always be executed and perform a cyclic traversal over all bids until all gas is spent.

- While the code below in finalize function has no security issue, please consider using claim pattern to let contributors and beneficiary receive funds instead of using send and ignore any transfer error.

```
    function finalize(uint _maxIt) public {
        require(now >= endTime);
        require(!finalized);
        // Make local copies of the finalization variables in order to avoid modifying
storage in order to save gas.
        uint localCutOffBidID = cutOffBidID;
        uint localSumAcceptedContrib = sumAcceptedContrib;
        uint localSumAcceptedVirtualContrib = sumAcceptedVirtualContrib;
        // Search for the cut-off bid while adding the contributions.
        for (uint it = 0; it < _maxIt && !finalized; ++it) {
            Bid storage bid = bids[localCutOffBidID];
            if (bid.contrib+localSumAcceptedContrib < bid.maxValuation) { // We haven't found
the cut-off yet.
                localSumAcceptedContrib        += bid.contrib;
                localSumAcceptedVirtualContrib += bid.contrib + (bid.contrib * bid.bonus) /
BONUS_DIVISOR;
                localCutOffBidID = bid.prev; // Go to the previous bid.
            } else { // We found the cut-off. This bid will be taken partially.
                finalized = true;
                uint contribCutOff = bid.maxValuation >= localSumAcceptedContrib ?
bid.maxValuation - localSumAcceptedContrib : 0; // The amount of the contribution of the cut-
off bid that can stay in the sale without spilling over the maxValuation.
                contribCutOff = contribCutOff < bid.contrib ? contribCutOff : bid.contrib; //
The amount that stays in the sale should not be more than the original contribution. This
line is not required but it is added as an extra security measure.
                bid.contributor.send(bid.contrib-contribCutOff); // Send the non-accepted
part. Use send in order to not block if the contributor's fallback reverts.
                bid.contrib = contribCutOff; // Update the contribution value.
                localSumAcceptedContrib += bid.contrib;
                localSumAcceptedVirtualContrib += bid.contrib + (bid.contrib * bid.bonus) /
BONUS_DIVISOR;
```

```
                beneficiary.send(localSumAcceptedContrib); // Use send in order to not block
if the beneficiary's fallback reverts.
            }
        }
        // Update storage.
        cutOffBidID = localCutOffBidID;
        sumAcceptedContrib = localSumAcceptedContrib;
        sumAcceptedVirtualContrib = localSumAcceptedVirtualContrib;
    }
```

CONCLUSION

IICO smart contracts have been audited by Verichains Lab using various public and in-house analysis tools and intensively manual code review. The assessment identified some issues in IICO smart contracts code. Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

LIMITATIONS

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

APPENDIX I

Figure 1 Call graph of IICO.sol