SECURITY AUDIT OF REQUEST NETWORK SMART CONTRACT

AUDIT REPORT

MARCH 22, 2018

VeriChains Lab

info@verichains.io

https://www.verichains.io

Driving Technology >> Forward

EXECUTIVE SUMMARY

This private Security Audit Report prepared by VeriChains Lab on March 22, 2018. We would like to thank Request Network 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. It was conducted on commit `4f10411553cafda2a8910872adb180066c849e0a` of branch `audit-mainnet-0.0.3-verichains` from GitHub repository of Request Network.

Overall, the audited code demonstrates high code quality standards adopted and effective use of modularity and security best practices. No major vulnerabilities were discovered during the audit.

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 REQUEST NETWORK

Request is a decentralized network that allows anyone to request a payment (a Request Invoice) for which the recipient can pay in a secure way. All of the information is stored in a decentralized authentic ledger. This results in cheaper, easier, and more secure payments, and it allows for a wide range of automation possibilities.

The website of Request Network is at https://request.network/
White paper (EN) is at https://request.network/assets/pdf/request_whitepaper.pdf

SCOPE OF THE AUDIT

This audit focused on identifying security flaws in code and the design of the smart contracts. It was conducted on commit *4f10411553cafda2a8910872adb180066c849e0a* of branch *audit-mainnet-0.0.3-verichains* from GitHub repository of Request Network.

Repository URL: https://github.com/RequestNetwork/requestNetwork/tree/audit-mainnet-0.0.3-verichains/packages/requestNetworkSmartContracts

The scope of the audit is limited to the following 7 source code files received on March 12, 2018:

Source File	SHA256 Hash
Administrable.sol	cda81645c53596336327bb498120368dbdcefd7c767087cb865002c5d6848c49
RequestCore.sol	04fe4172a2c15143b1907e068092366d54eaedf39781e842fb71f2fe8c39ba98
RequestEthereumCollect.sol	8ef55d169fedb272a7897ceca4cbb0f1ea3ddbbc65289b6d47c1bd1978a8bf6a
RequestEthereum.sol	d6ef33a9b87f69dd58684a46269296e821c8319cf7749434d9288ffb11eea6e3
SafeMathInt.sol	571e36a65ccd5bfb622fbd925c3ad3caf5a91f61a6ac5c3354984dd3e27ff8b0
SafeMathUint8.sol	b78833dd1917741e26298a4cc329ad22fbb3f8f81f855326a7afaff761099c8c
SafeMathUint96.sol	3d37e519ee5942875e9c3628cbee084a321fe3f01596b16dbe3297349e688966

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

No security vulnerabilities were discovered during the audit.

RECOMMENDATIONS / SUGGESTIONS

- Consider removing several redundant condition checks of uint8 < 256 (always returns true) in the code to reduce gas burning, for examples

File: RequestCore.sol

```
358:    function getSubPayeesCount(bytes32 _requestId)
        public
        constant
        returns(uint8)
    {
        for (uint8 i = 0; i < 256 && subPayees[_requestId][i].addr != address(0); i =
i.add(1)) {
```

```
407:        for (uint8 i = 0; i < 256 && subPayees[_requestId][i].addr != address(0);
i = i.add(1))
```

```
428:        for (uint8 i = 0; isNull && i < 256 && subPayees[_requestId][i].addr !=
address(0); i = i.add(1))
```

```
448:        for (uint8 i = 0; i < 256 && subPayees[_requestId][i].addr != address(0);
i = i.add(1))
```

```
482:       for (uint8 i = 0; i < 256 && subPayees[_requestId][i].addr != address(0); i
= i.add(1))
```

- In several for loop with has condition uint8 < array.length, if array.length is greater than 255 then the loop will go through the entire 256 values before throwing an error. You could consider stopping gas burning early by having a require(array.length < 256) before the loop. For examples:

File: RequestEthereum.sol

```
208:    // set payment addresses for payees
        for (uint8 j = 0; j < _payeesPaymentAddress.length; j = j.add(1)) {
```

```
            payeesPaymentAddress[requestId][j] = _payeesPaymentAddress[j];
        }
```

```
245:    for (uint8 i = 0; i < _expectedAmounts.length; i = i.add(1))
        {
            // all expected amount must be positive
            require(_expectedAmounts[i]>=0);
            // compute the total expected amount of the request
            totalExpectedAmounts = totalExpectedAmounts.add(_expectedAmounts[i]);
        }
```

- Consider changing the function name insertBytes20inBytes to updateBytes20inBytes to avoid confusing as the function actually **updates** 20 bytes in data bytes, **not insert**.

File: RequestEthereum.sol

```
712:        function insertBytes20inBytes(bytes data, uint offset, bytes20 b) internal
pure returns(bytes) {
        for(uint8 j = 0; j <20; j++) {
            data[offset+j] = b[j];
        }
        return data;
    }
```

Furthermore, in order to optimize gas usage, this function can be rewritten into

```
    function updateBytes20inBytes(bytes data, uint offset, bytes20 b) internal pure {
       require(offset >=0 && offset + 20 <= data.length);
       assembly {
           let m := mload(add(data, add(20, offset)))
           m := and(m,
0xFFFFFFFFFFFFFFFFFFFFFFFF0000000000000000000000000000000000000000)
           m := or(m, div(b, 0x1000000000000000000000000))
           mstore(add(data, add(20, offset)), m)
       }
    }
```

This implementation of updateBytes20inBytes code **reduces significantly 75% gas cost** from the original insertBytes20inBytes function. It is a bit hard to read compared to the original version though.

- Consider removing unnecessary safemath usage if gas optimization is required

File: RequestEthereum.sol

```
712:    for(uint8 i = 0; i < payeesCount; i++) {
            // extract the expectedAmount for the payee[i]
            int256 expectedAmountTemp = int256(extractBytes32(_requestData,
uint256(i).mul(52).add(61)));
            // compute the total expected amount of the request
            totalExpectedAmounts = totalExpectedAmounts.add(expectedAmountTemp);
            // all expected amount must be positibe
            require(expectedAmountTemp>0);
        }
```

The expression uint256(i).mul(52).add(61) can be simplified into 61 + 52 * uint256(i) as the value of i is only within range 0 to 255 (uint8) so using SafeMath will cause unnecessary gas overhead.

This could be further optimized using constants and also improving code readability as the following:

```
uint256 constant kSizeOfAddress = 20;
uint256 constant kSizeOfUint256 = 32;

uint256 constant kPayeeOffset = 41;
uint256 constant kPayeeExpectedAmountBaseOffset = kPayeeOffset + kSizeOfAddress;
uint256 constant kPayeeRecordSize = kSizeOfAddress + kSizeOfUint256;

...

==> kPayeeExpectedAmountBaseOffset + kPayeeRecordSize * uint256(i)
```

- Gas optimization of extractAddress function in both RequestCore.sol and RequestEthereum.sol

  This extractAddress function

```
    function extractAddress(bytes _data, uint offset) internal pure returns (address)
{
        // for pattern to reduce contract size
        uint160 m = uint160(_data[offset]);
        for(uint8 i = 1; i < 20; i++) {
            m = m*256 + uint160(_data[offset+i]);
        }
        return address(m);
    }
```

  can be rewritten into

```
function extractAddress(bytes _data, uint offset) internal pure returns (address m) {
       require(offset >=0 && offset + 20 <= _data.length);
       assembly {
           m := and(mload(add(_data, add(20,
offset))),0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
       }
    }
```

  This implementation is valid because _data contain 1 word (32 bytes) prefix, so we can borrow 12 bytes from it and read the whole word at offset 20, then remove borrowed bytes using *and*

  This function costs ~650 gas, **reduces significantly 80-85% gas cost** from the original extractAddress function (~4000 gas).

- Gas optimization of extractBytes32 function in both RequestCore.sol and RequestEthereum.sol

  This extractBytes32 function

```
   function extractBytes32(bytes _data, uint _offset) public pure returns (bytes32) {
        // no "for" pattern to optimise gas cost
        uint256 m = uint256(_data[_offset]); // 3930 gas
        m = m*256 + uint256(_data[_offset+1]);
        m = m*256 + uint256(_data[_offset+2]);
        m = m*256 + uint256(_data[_offset+3]);
        m = m*256 + uint256(_data[_offset+4]);
        m = m*256 + uint256(_data[_offset+5]);
```

```
        m = m*256 + uint256(_data[_offset+6]);
        m = m*256 + uint256(_data[_offset+7]);
        m = m*256 + uint256(_data[_offset+8]);
        m = m*256 + uint256(_data[_offset+9]);
        m = m*256 + uint256(_data[_offset+10]);
        m = m*256 + uint256(_data[_offset+11]);
        m = m*256 + uint256(_data[_offset+12]);
        m = m*256 + uint256(_data[_offset+13]);
        m = m*256 + uint256(_data[_offset+14]);
        m = m*256 + uint256(_data[_offset+15]);
        m = m*256 + uint256(_data[_offset+16]);
        m = m*256 + uint256(_data[_offset+17]);
        m = m*256 + uint256(_data[_offset+18]);
        m = m*256 + uint256(_data[_offset+19]);
        m = m*256 + uint256(_data[_offset+20]);
        m = m*256 + uint256(_data[_offset+21]);
        m = m*256 + uint256(_data[_offset+22]);
        m = m*256 + uint256(_data[_offset+23]);
        m = m*256 + uint256(_data[_offset+24]);
        m = m*256 + uint256(_data[_offset+25]);
        m = m*256 + uint256(_data[_offset+26]);
        m = m*256 + uint256(_data[_offset+27]);
        m = m*256 + uint256(_data[_offset+28]);
        m = m*256 + uint256(_data[_offset+29]);
        m = m*256 + uint256(_data[_offset+30]);
        m = m*256 + uint256(_data[_offset+31]);
        return bytes32(m);
    }
```

can be rewritten into

```
    function extractBytes32(bytes _data, uint offset) public pure returns (bytes32 bs)
{
       require(offset >=0 && offset + 32 <= _data.length);
       assembly {
           bs := mload(add(_data, add(32, offset)))
       }
```

This function costs ~650 gas, **reduces significantly 80-85% gas cost** from the original extractBytes32 function (~4000 gas).

- Consider double-checking the visibility of functions and state variables to make sure visibility like external, internal, private and public is used and defined properly. While we have done our best to review the visibility of functions and state variables based on our understanding of the code, it is highly recommended you as the contract developer to double check it.

  Call graphs of RequestCore and RequestEthereum contracts can be found in Appendix I

CONCLUSION

Request Network smart contracts have been audited by VeriChains Lab using various public and in-house analysis tools and intensively manual code review. Overall, the audited code demonstrates high code quality standards adopted and effective use of modularity and security best practices. No major vulnerabilities were discovered during the audit.

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 RequestEthereum.sol

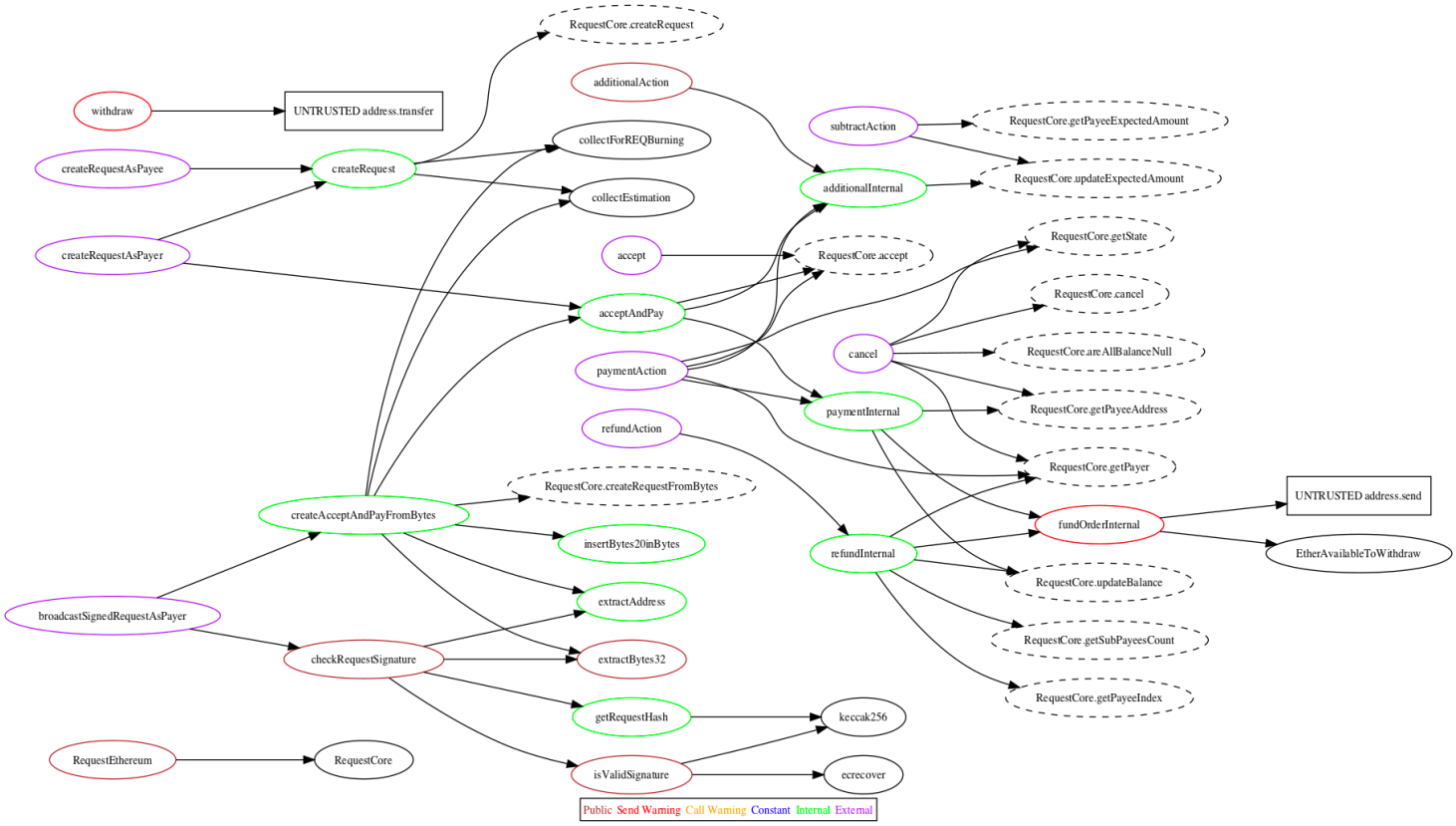

Figure 2 Call graph of RequestCore.sol