LOOPRING PROTOCOL AUDIT

IN PARTNERSHIP WITH

ARGUS AUDITS

Observe. Report. Defend.

AUTHORED BY:

ALI MOUSA
ALI@ARGUS.OBSERVER

PRANAV GADDAMADUGU
PRANAV@ARGUS.OBSERVER

HOWARD WU
HOWARD@ARGUS.OBSERVER

RAYMOND CHU
RAY@ARGUS.OBSERVER

COLLIN CHIN
COLLIN@ARGUS.OBSERVER

LOOPRING AUDIT

The Argus team was contracted by Loopring to perform a security audit of the Loopring protocol on October 29, 2017. The team conducted a three week security audit of the Loopring protocol at tag v1.0-beta.2. Our findings were reported to the Loopring team on November 22, 2017 and necessary updates were made to the protocol implementation between November 22-24, 2017. The final audit report was submitted to the Loopring team on November 25, 2017.

The review process comprised of individual inspections of the Loopring protocol by members of the audit team, followed by a group review of the codebase, tests, documentation, and supporting information. Proprietary and open-source tools were employed to determine the safety and correctness of the protocol implementation. Following discussions addressing our findings with the Loopring team, no major smart contract code vulnerabilities remain to be found that put funds at immediate risk.

This is the final audit report of the Loopring protocol, completed on November 25, 2017. The final audit report includes documentation of protocol functions, general findings from our security audit, specific findings from our analysis, and reports from code coverage and code analysis tools.

SCOPE

The team reviewed the code with the following metrics in mind:

Security
Identification of potential security flaws/attacks including but not limited to:

Data compliance
Integer overflow and underflow
Race conditions
DoS attacks
Timestamp dependence
Code quality
Reentrancy attacks
Sybil attacks
Front running
Transaction-ordering dependence
Call depth attacks

CODE QUALITY

Review of code to follow general best practices and Consensys Best Practices with a focus on:

Syntax
Readability
Consistency
Complexity
Scalability

Reivew of the smart contract architecture with primary focus on:

Game theory
Incentives
Pain points
Usability

ANALYSIS

LoopringProtocolImpl.sol

Description of contract system

This contract implements the functionality of the Loopring Protocol as outlined in the whitepaper. Methods for creating, assembling, and submitting ring orders are implemented as outlined in the LoopringProtocol interface. Various calculations and checks are completed during each step of the process by private helper functions.

Functions

submitRing

This function submits an order-ring for validation and settlement. First, a ringIndex mutex is checked to prevent ring re-submission, the ring size is checked for validity, the data input parameters are checked for integrity with a call to the verifyInputDataIntegrity() function, and the token addresses are checked for registry with a call to the verifyTokensRegistered() function. Next, a call to the RinghashRegistry contract's function computeAndGetRinghashInfo() is made to obtain the ringhash and ring attributes for the order ring. A function call to verifySignature() looks at the last entry in the v, r, s signature arrays to verify the hash of the computed ring. After, a call to the assembleOrders() function assembles input data into Order structs so they can be passed to other functions. The miner is set as the fee recipient. Lastly, the function handleRing() is called which processes some final checks on the ring and then proceeds to submit payments for the ring. In addition, the mutex is reset, indicating that a submission has been completed.

cancelOrder

This function first reconstructs an order given the parameters for an Order struct and verifies that the reconstructed order has a valid hash and signature. Next it adds the Order struct to a mapping of canceled or filled orders using the order hash as the key. Lastly, an OrderCancelled event is fired to indicate that the order has been cancelled.

setCutoff

This function sets a cutoff timestamp to invalidate all orders whose timestamp is smaller than or equal to the new value of the address's cutoff timestamp.

verifyRingHasNoSubRing

This function ensures that the ring does not have two orders with the same selling token by iterating through all orders in a ring.

ANALYSIS

verifyTokensRegistered

This function loads an array of length ringSize with the token addresses in addressList. Every address in this array is tested and enforced by requiring the TokenRegistry's areAllTokensRegistered() function to return true.

handleRing

This function makes final checks before payments to orders in an order ring. It is called by the submitRing() function. First, calls to verifyRingHasNoSubRing() and verifyMinerSuppliedFillRates() are made to validate inputs and exchange rates. Next, orders are scaled down by calling scaleRingBasedOnHistoricalRecords() and calculateRingFillAmount() which determine the maximum flow of value possible for the entire ring. After, ring fees are calculated by a call to calculateRingFees(). Lastly, the function settleRing() is called to process payments and a RingMined event is fired.

createTransferBatch

This function uses the ringSize and list of orders to create a batch (bytes32 array) with all the relevant information and format needed for TokenTransferDelegate's batchTransferToken().

settleRing

This function iterates through all orders in a ring, adds them to the canceledOrFilled mapping, and fires an OrderFilled event for each. Then, the createTransferBatch() method is called to assemble a batch order. After, a call to the TokenTransferDelagate.sol contract's batchTransferToken() method is made under the specified token transfer delegate to submit the orders and pay fees to the ring miner.

verifyMinerSuppliedFillRates

This function checks iterates through all the orders a miner provides and calculates the rate ratio of the orders. This rate ratio is then checked to enforce that it is less than or equal to the rateRatioCVSThreshold constant that the Loopring team determined.

calculateRingFees

This function calculates the fees for submitting a ring based on a miner's choice to take LRC or take the margin split.

ANALYSIS

calculateRingFillAmount

This function iterates through the list of orders and calls the calculateOrderFillAmount() function on every iteration to determine the limiting factor of the list of orders (similar to a max-flow algorithm). The second for-loop updates all the orders in the order book up to the limiting factor order with the minimum order filled to accurately reflect a correct flow between all orders.

calculateOrderFillAmount

This function returns the smallest order's index given two OrderState's and the current smallest index. This function is called multiple times by calculateRingFillAmount()

scaleRingBasedOnHistoricalRecords

This function iterates through a list of orders and scales down all orders based on historical fill or cancellation statistics, but keeps the order's original exchange rate.

getSpendable

This function returns the amount of ERC20 token that can be spent by this contract.

verifyInputDataIntegrity

This function verifies the input data of an order ring. The length of the order ring is checked against the length of all order data arrays to ensure that there are no missing or extra orders. In addition, all ring-mining related arguments are iterated through and validated.

assembleOrders

This function takes in parameters for a set of orders as input, assembles a corresponding Order struct, calls calculateOrderHash(), verifySignature(), validateOrder(), and populates an array of OrderState structs for each order. In addition, it checks that each order has a nonzero available amount of tokens.

validateOrder

This function checks that the Order's addresses, amounts, ttl, and salt are nonzero; that the timestamp is greater than the specified cutoff, less than the current block number, and within the specified ttl range; and that the Order's marginSplitPercentage is less than or equal to the base percentage. This function reverts state should any of the checks fail.

ANALYSIS

calculateOrderHash

This function computes a hash on the data contained by an Order struct and the other specified inputs.

verifySignature

This function checks that the signer matches the address that computed a valid signature on the hash and reverts the state if the addresses do not match.

TokenTransferDelegate.sol

Description of contract system

The TokenTransferDelegate is interfaced within the implementation of the Loopring Protocol. It serves as a decentralized, trustless intermediary such that orders can be placed asynchronously, without having to know who the next participant in the ring would be.

Modifiers

onlyAuthorized

This modifier intends to verify that the sender of the message is an authorized address. It makes a function call to verify the statement and can instead use an in-line check in the body of the function. This achieves the same effect with potentially less gas cost.

Functions

authorizeAddress

This function adds the address of a LoopringProtocol contract. It can only be called by the owner of the contract.

isAddressAuthorized

This function traverses mapping using known AddressInfo structs. This while loop will always fail however, since one condition of the while loop is unsatisfiable with the initial condition. getLatestAuthorizedAddresses(uint max) This function traverses mapping using known AddressInfo structs. It attempts to aggregate the last max number of addresses registered and return them in an array. Finding: This while loop will always fail however, since one condition of the while loop is unsatisfiable with the initial condition. Since max > 0 by the definition of a uint the loop will never execute, so this function will always return an empty array.

ANALYSIS

transferToken

This function is critical to the use of the TokenTransferDelegate. When called, it transfers an amount of some ERC20 token from one party to another using a pre approved allowance.

batchTransferToken

This function takes a batch of orders and sends tokens to their intended recipients through the ring as specified by the batch. Again, this can only be called by the LoopringProtocol contract. The function loops through the orders and executes by transferring tokens as specified by orders in the ring.

TokenRegistry.sol

Description of contract system

This contract maintains an array of token contract addresses that are able to be used by an exchange using the Loopring protocol. A mapping of token contract addresses to booleans to keep track of registered and unregistered tokens. In addition, a mapping of token symbols to token addresses is stored. The contract's functions provide the logic to register, unregister, lookup, and return any token that the contract has interacted with.

Functions

registerToken

This function pushes a new token to the token array, stores the token symbol, and sets a boolean corresponding with the token address to true if the token has not been registered before or has its boolean set to false. This function can only be called by the contract owner.

unregisterToken

This function removes a token from the token array, deletes the token symbol, and flips the boolean corresponding with the token address to false if the supplied token address and symbol are currently stored. This function can only be called by the contract owner.

isTokenRegisteredBySymbol

This function looks up a token address corresponding to the symbol parameter and returns the value of comparing the returned address to the zero address.

ANALYSIS

isTokenRegistered

This function looks up a token boolean corresponding to the address parameter in the registered tokens mapping and returns it.

areAllTokensRegistered

This function iterates through the array of stored tokens and looks up each token's corresponding boolean value. If any token's boolean is false (it is unregistered) the function outputs false. Otherwise, it returns true.

getAddressBySymbol

This function looks up a token address corresponding to the symbol parameter and returns the value of the returned address.

RinghashRegistry.sol

Description of contract system

This contract maintains a mapping of valid ringhashes to submissions. The two public functions submitRinghash() and batchSubmitRinghash() implement logic for calculating and registering a new ring or batch of rings and their hashes to the contract.

Functions

submitRinghash

This function stores a submission in the mapping of all submissions at a certain block number and logs an event with the same info.

batchSubmitRinghash

This function submits a ringhash using the submitRinghash() function for every submission pairing in ringminerList and ringHashList.

calculateRinghash

This function calculates the ringhash by calling hashing the xorReduce() between all the ECSDA signature parameter lists and the ringSize. xorReduce() is implemented in the Loopring defined library.

ANALYSIS

computeAndGetRinghashInfo

This function calculates the ringhash with the calculateRingHash() function and returns if it can be submitted and if it is reserved.

canSubmit

This function checks if the ringhash can be submitted given a ringhash and ringminer pair.

isReserved

This function checks if the ringhash was submitted and still valid.

Library Contracts

ERC20.sol
MathBytes32.sol
MathUint.sol
MathUint8.sol
Ownable.sol

FINDINGS

General Findings

Testing

The Loopring protocol handles nontrivial logic for managing user finance and would benefit from more unit tests. The testing suite features 21 tests, which provide good coverage for completeness of the Loopring protocol, but should include more checks to be sufficient in considering all edge cases.

Order Struct

The Order struct in LoopringProtocol.sol does not implement, but has documentation for, timestamp, ttl, salt, v, r, and s.

var Usage

The `var` keyword is used in several places and is not necessary. The following are references we found:

TokenTransferDelegate.sol: Line 178
RinghashRegistry.sol: Lines 149, 166
LoopringProtocolImpl.sol: Lines 254, 278, 514-515, 543-545, 585, 621-622, 776-777, 820, 878

Potential Gas Optimization

For child functions that are used only once to validate inputs for a parent function, it is possible to reduce total gas costs for a parent function by moving child function logic into the parent function.

The following is an example of a potential gas optimization for input validation logic:
Line 438: handleRing() is the parent function. Line 450: verifyRingHasNoSubRing() is the child function.
Making a call to verifyRingHasNoSubRing() and then computing input validation is more expensive than having the input validation within handleRing().

Documentation

Both Rate and Order structs contain amountS and amountB for different purposes and the use case of these amount values in Rate should be documented.

Minor typo concerns were found in codebase documentation.

Implicit Typing

An ERC20 address is implicitly configured as an ERC20 address and type inference is not necessary. LoopringProtocolImpl.sol Line: 820 TokenDelegateTransfer.sol Lines: 178, 192 Some comments are misspelled. LoopringProtocolImpl.sol Lines: 577, 858

Gas Limit

There is a hard upper bound, based on the gas limit, on the number of tokens one can add to the token registry and subsequently remove.

FINDINGS

Significant Findings

High

RinghashRegistry.sol: calculateRingHash()

Line 94: The ring hash is calculated - using the ECDSA signature parameter lists - as part of the "commitment" phase in the process of submitting an order. However, because the fee recipient is not included in the hash, we are able to initiate a race condition to exploit submitRing() (LoopringProtocolImpl.sol: Line 219).

Description of the Exploit: Wait for someone to send a "submitRing" transaction. Send a "submitRing" transaction with the same parameters except a fee recipient of your choice. One can hope that their transaction is included in a block before the first one, spend more gas, or collude with miners.

Recommendation: Exchanges can ensure that their transactions are included with high enough gas fees or confirming transactions. Include fee recipient in the ring hash.

Medium

LoopringProtocolImpl.sol: submitRing()

Line 258: There should be an additional check ringminer is not address(0).

LoopringProtocolImpl.sol: verifyInputDataIntegrity()

Line 843: There should be an additional check on the elements of the addressList argument. Specifically, the function should check that the two addresses are not address(0).

RinghashRegistry.sol: canSubmit()

Lines 151 - 153: Function logic is unclear. Anyone can submit a ringhash with a ringminer of 0, which will pass the require statement and push events that are false. Change the first check with: submission.ringminer == address(0) && submission.block == 0

FINDINGS

Significant Findings

Low

LoopringProtocolImpl.sol: verifyRingHasNoSubRing()

Lines 411-415: The performance can be improved by checking when tokenS appears in the ring more than once. In addition, there will be an upper bound on the maximum number of times one can run a nested for loop, thus one should limit the maximum ring size to prevent the function from costing too much gas.

TokenTransferDelegate.sol: authorizeAddress()

Line 94: This line can be removed as it is repeated on line 100. Only line 100 is neccessary.

TokenTransferDelegate.sol: onlyAuthorized()

Lines 56-61 and 119-125: This modifier goes through more logic than is needed. It can be replaced with require(addressInfos[msg.sender].authorized).

TokenRegistry.sol: getAddressBySymbol()

Line 94: The function modifier constant is deprecated and should be view.

TokenRegistry.sol: areAllTokensRegistered()

Lines 84-89: This function is called every time submitRing() is invoked and loops through all tokens to ensure they are verified. This becomes more expensive with every token added to the registry and every time an order is submitted. In the event that there are exists too many tokens, submitRing() can reach the gas limit before running to completion.

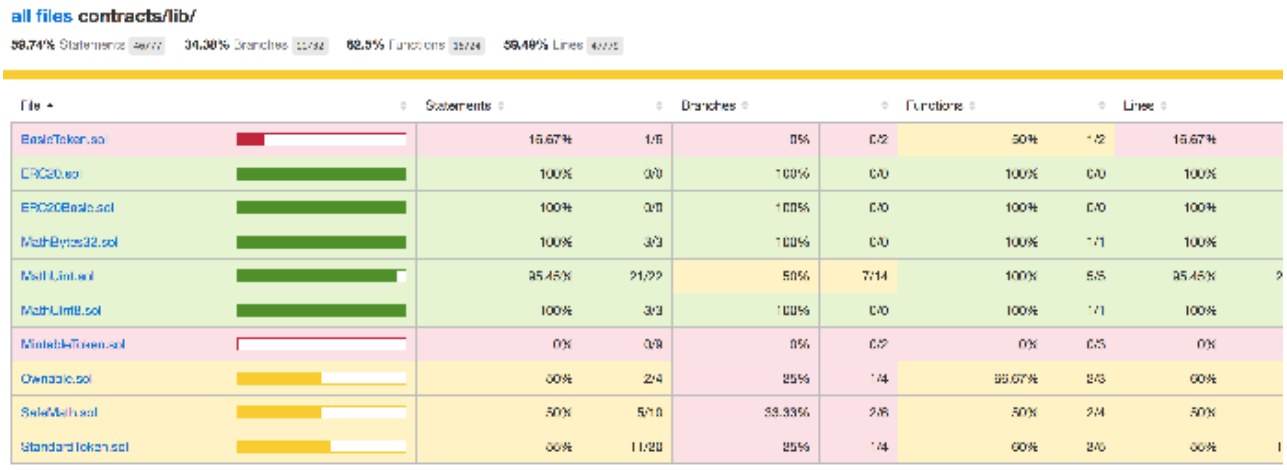
RinghashRegistry.sol: calculateRinghash()

Line 94: This function is “public” and will leak information if people use it to compute their ring hashes, should they call it in a transaction.

CODE COVERAGE

Code coverage using the Solidity-Coverage tool was used to measure the what portions of the codebase was run with the given test suite.

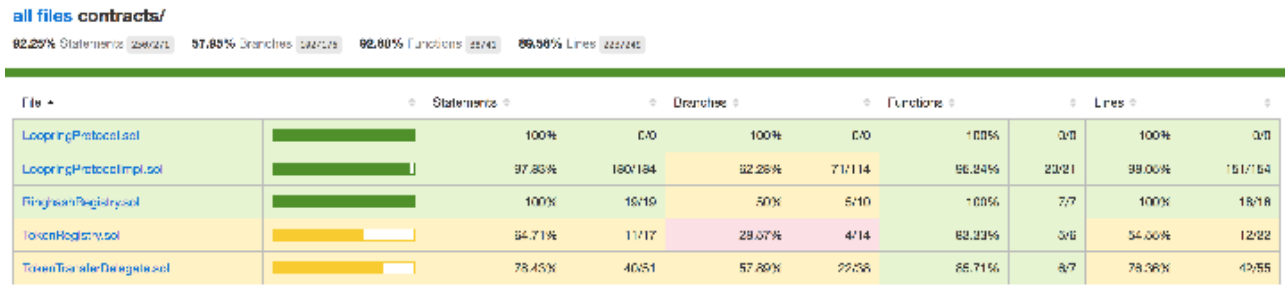
The test coverage results are located in the code coverage directory and can be viewed below. Coverage results for MintableToken.sol can be ignored, as this was included to run our test coverage tool and is not reflective of any defects in the testing of the Loopring protocol:



all files contracts/lib/

59.74% Statements 34.38% Branches 62.5% Functions 59.49% Lines

File	Statements		Branches		Functions		Lines	
BasicToken.sol	16.67%	1/6	0%	0/2	50%	1/2	16.67%	
ERC20.sol	100%	0/0	100%	0/0	100%	0/0	100%	
ERC20Basic.sol	100%	0/0	100%	0/0	100%	0/0	100%	
MathBytes32.sol	100%	3/3	100%	0/0	100%	1/1	100%	
MathUint.sol	85.45%	21/22	50%	7/14	100%	5/5	85.45%	2
MathUint8.sol	100%	3/3	100%	0/0	100%	1/1	100%	
MintableToken.sol	0%	0/8	0%	0/2	0%	0/5	0%	
Ownable.sol	50%	2/4	25%	1/4	66.67%	2/3	50%	
SafeMath.sol	50%	5/10	33.33%	2/6	50%	2/4	50%	
StandardToken.sol	55%	11/20	25%	1/4	60%	3/5	55%	1



all files contracts/

82.29% Statements 57.85% Branches 82.86% Functions 89.58% Lines

File	Statements		Branches		Functions		Lines	
LoopringProtocol.sol	100%	0/0	100%	0/0	100%	0/0	100%	0/0
LoopringProtocolImpl.sol	97.83%	180/184	82.28%	71/114	95.24%	20/21	98.06%	151/154
RinghashRegistry.sol	100%	19/19	50%	5/10	100%	7/7	100%	18/18
TokenRegistry.sol	84.71%	11/17	28.57%	4/14	83.33%	5/6	54.55%	12/22
TokenTransferDelegate.sol	78.45%	40/51	57.89%	22/38	85.71%	6/7	78.38%	43/55

OYENTE REPORT

Below are the Oyente reports for all contracts:

browser/LoopringProtocolImpl.sol

browser/LoopringProtocolImpl.sol:LoopringProtocolImpl

EVM Code Coverage
15.2%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

browser/TokenRegistry.sol

browser/TokenRegistry.sol:TokenRegistry

EVM Code Coverage
59.4%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

OYENTE REPORT

browser/RinghashRegistry.sol

browser/RinghashRegistry.sol:RinghashRegistry

EVM Code Coverage
56.4%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

browser/MathUint8.sol

browser/MathUint8.sol:MathUint8

EVM Code Coverage
100.0%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

OYENTE REPORT

browser/MathUint.sol

browser/MathUint.sol:MathUint

EVM Code Coverage
100.0%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

browser/TokenTransferDelegate.sol

browser/TokenTransferDelegate.sol:TokenTransferDelegate

EVM Code Coverage
89.4%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

OYENTE REPORT

browser/Ownable.sol

browser/Ownable.sol:Ownable

EVM Code Coverage
99.4%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

browser/MathBytes32.sol

browser/MathBytes32.sol:MathBytes32

EVM Code Coverage
100.0%
Callstack Depth Attack Vulnerability:
False
Re-Entrancy Vulnerability:
False
Assertion Failure:
False
Timestamp Dependency:
False
:
False
Transaction-Ordering Dependence (TOD):
False

ARGUS AUDITS

Contact

Argus Audits Team
audit@argus.observer

IN PARTNERSHIP WITH

Loopring

BLOCKCHAIN AT BERKELEY