Reya Network Security Review

Pashov Audit Group

Conducted by: T1MOH, Dan Ogurtsov, ubermensch

March 30th 2024 - April 5th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **reya-network** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Reya Network

Reya Network is a trading-optimised modular L2. The chain layer is powered by Arbitrum Orbit and is gas-free, with transactions ordered on a FIFO basis. The protocol layer directly tackles the vertical integration of DeFi applications by breaking the chain into modular components to support trading, such as PnL settlements, margin requirements, liquidations.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 39b31762f0fe5836397c87ba78fd6cd11f147a9f

fixes review commit hash - 29b5286569b08c63b2c94365b04434a5d24a7a03

Scope

The following smart contracts were in scope of the audit:

- `IDepositModule`
- `IExecutionModule`
- `SignatureHelpers`
- `DepositModule`
- `ExecutionModule`
- `ISharesModule`
- `SharesModule`
- `IConfigurationModule`
- `ConfigurationModule`
- `NodeModule`
- `DivReducerNode`
- `NodeDefinition`
- `IConfigurationModule`
- `ITransfersModule`
- `ISocketControllerWithPayload`
- `ISocketExecutionHelper`
- `BridgingUtils`
- `CoreUtils`
- `DataTypes`
- `Deposits`
- `Errors`
- `Events`
- `Transfers`
- `Withdrawals`
- `ConfigurationModule`
- `DepositsFallbackModule`
- `TransfersModule`
- `Configuration`
- `TokenProxy`
- `IERC20TokenModule`
- `FeatureFlagSupport`
- `FeatureFlagModule`
- `ERC20TokenModule`
- `OwnerUpgradeModule`
- `IOrderModule`
- `OrderModule`

7. Executive Summary

Over the course of the security review, T1MOH, Dan Ogurtsov, ubermensch engaged with Reya Network to review Reya Network. In this period of time a total of **10** issues were uncovered.

Protocol Summary

Protocol Name	Reya Network
Repository	https://github.com/Reya-Labs/reya-network
Date	March 30th 2024 - April 5th 2024
Protocol Type	Trading-optimised modular L2

Findings Count

Severity	**Amount**
Critical	3
High	1
Medium	4
Low	2
Total Findings	**10**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Users can't bridge funds back from the app chain	Critical	Resolved
[C-02]	Attacker can drain Periphery by specifying big socketPayloadSize	Critical	Resolved
[C-03]	Decimals are incorrectly handled in DivReducerNode	Critical	Resolved
[H-01]	User can lose tokens during deposit fallback bridging	High	Resolved
[M-01]	Inadequate Verification of tokenAmount Leads to Potential Dust Theft	Medium	Resolved
[M-02]	Stale Price Data in DivReducer Due to Average Timestamp Calculation	Medium	Resolved
[M-03]	Lack of Price Freshness Verification in Oracle Price Data	Medium	Resolved
[M-04]	Invalid Nodes can be registered due to an incorrect check	Medium	Resolved
[L-01]	Non-Compliance with EIP-712 Specification in Signature Functions	Low	Acknowledged
[L-02]	Signature Malleability in ecrecover Precompile Usage	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Users can't bridge funds back from the app chain

Severity

Impact: High

Likelihood: High

Description

The protocol must pay a fee in native coin to bridge funds back from the app chain:

```
(uint256 tokenFees, uint256 nativeFees) =
            getFees(
              withdrawToken,
              socketController,
              socketConnector,
              socketMsgGasLimit,
              socketPayloadSize
            );
        if (tokenAmount > tokenFees) {
            uint256 tokensToWithdraw = tokenAmount - tokenFees;
@>          socketController.bridge{ value: nativeFees }({
                receiver_: receiver,
                amount_: tokensToWithdraw,
                msgGasLimit_: socketMsgGasLimit,
                connector_: socketConnector,
                execPayload_: abi.encode(),
                options_: abi.encode()
            });
```

Periphery is the module that interacts with the bridge. The problem is that none of these contracts has `payable` function to receive ETH

```
contract PeripheryRouter is
    ConfigurationModule,
    DepositsModule,
    DepositsFallbackModule,
    OrderModule,
    TransfersModule,
    WithdrawalsModule,
    OwnerUpgradeModule,
    ERC721ReceiverModule,
    FeatureFlagModule
{ }

contract PeripheryProxy is UUPSProxyWithOwner, PeripheryRouter {
    constructor(
        address firstImplementation,
        address initialOwner
    )
        UUPSProxyWithOwner(firstImplementation, initialOwner)
    { }
}
```

Recommendations

Make sure that PeripheryRouter.sol inherits the module with the function `receive() payable`

[C-02] Attacker can drain Periphery by specifying big `socketPayloadSize`

Severity

Impact: High

Likelihood: High

Description

When a user withdraws funds from protocol, tokens are bridged to another chain to address `receiver`. The fee to pay for bridging is based on `gasLimit` and `payloadSize`:

```
function getFees(...)
        internal
        view
        returns (uint256 feeInToken, uint256 nativeFees)
    {
@>      nativeFees = controller.getMinFees(connector, gasLimit, payloadSize);
        feeInToken = Configuration.getStaticWithdrawFee(token, connector);
    }
```

User can just set very high `payloadSize` and protocol will pay high fee:

```
function executeBridging(...)
        internal
    {
        ISocketControllerWithPayload socketController =
            ISocketControllerWithPayload(Configuration.getController
              (withdrawToken));

        (uint256 tokenFees, uint256 nativeFees) =
            getFees(
              withdrawToken,
              socketController,
              socketConnector,
              socketMsgGasLimit,
              socketPayloadSize
            );
        if (tokenAmount > tokenFees) {
            uint256 tokensToWithdraw = tokenAmount - tokenFees;
@>          socketController.bridge{ value: nativeFees }({
                receiver_: receiver,
                amount_: tokensToWithdraw,
                msgGasLimit_: socketMsgGasLimit,
                connector_: socketConnector,
                execPayload_: abi.encode(),
                options_: abi.encode()
            });
            withdrawToken.safeTransfer(OwnableStorage.getOwner(), tokenFees);
        } else {
            revert Errors.NotEnoughFees(tokenAmount, tokenFees);
        }
    }
```

Note that Socket which is used for bridging doesn't send back excessive `msg.value`. It treats excessive `msg.value` as `executionFee`: Link

Another note is that currently `payloadSize` is not used in fee calculation, but will be in a future version link

Recommendations

Remove argument `socketPayloadSize` and use 0 instead

[C-03] Decimals are incorrectly handled in DivReducerNode

Severity

Impact: High

Likelihood: High

Description

Node DivReducer is supposed to have 2 parents which are Redstone oracles and combine 2 prices. For example, to price ETH/USDC it will fetch 2 prices and divide `(ETH/USD) / (USDC/USD)`.

The problem is that Redstone oracles have 8 decimals by default, but the code uses 1e18 arithmetic:

```
function process
      (NodeOutput.Data[] memory parentNodeOutputs) internal pure returns (NodeOutput.D
        if (parentNodeOutputs[1].price == 0) {
            revert InvalidPrice();
        }

@>      uint256 price = divUintUint
  (parentNodeOutputs[0].price, parentNodeOutputs[1].price).unwrap();
        uint256 timestamp =
          (parentNodeOutputs[0].timestamp + parentNodeOutputs[1].timestamp) / 2;

        return NodeOutput.Data({ price: price, timestamp: timestamp });
    }


function divUintUint(uint256 a, uint256 b) pure returns (UD60x18) {
    return UD60x18.wrap(a).div(UD60x18.wrap(b));
}
```

Here you can see the default decimals is 8: link

Recommendations

Normalize the price from RedstoneOracle by decimals of that oracle. Only after using it in internal calculations

8.2. High Findings

[H-01] User can lose tokens during deposit fallback bridging

Severity

Impact: High

Likelihood: Medium

Description

DepositsFallbackModule handles situations where a deposit reverts and initiates bridging back of users' funds. Note that it uses the address `receiver` of the deposit on the Reya chain to bridge back funds on the source chain:

```
try DepositsModule(address(this)).depositPassivePool(inputs) { }
        catch {
            BridgingUtils.executeBridging({
                withdrawToken: usdc,
                socketConnector: fallbackData.socketConnector,
                socketMsgGasLimit: fallbackData.socketMsgGasLimit,
                tokenAmount: inputs.amount,
@>              receiver: inputs.owner,
                socketPayloadSize: fallbackData.socketPayloadSize
            });
        }
```

It incorrectly assumes that the address `inputs.owner` on the source chain is owned by the same person on Reya chain. There are 2 cases when the assumption is not guaranteed:

1. Account Abstraction wallet implementations
2. old version of Safe multisigs https://rekt.news/wintermute-rekt/

Recommendations

Add argument `receiver` to FallbackData struct and use it instead of `inputs.accountOwner` in DepositsFallbackModule.sol

8.3. Medium Findings

[M-01] Inadequate Verification of `tokenAmount` Leads to Potential Dust Theft

Severity

Impact: Medium

Likelihood: Medium

Description

The Periphery's functionality allows bridging of funds between the source chain and the protocol, encompassing integration with the deposit, withdrawal, and transfer functionalities of the Core and Passive Pool. An issue arises when the deposit action fails on the destination chain; the `DepositsFallbackModule` is designed to catch this failure and refund the user on the source chain via the Socket bridge. The problem occurs when the `tokenAmount` is lower than the `tokenFees` (a static fee), leading to a transaction revert due to insufficient fees, consequently trapping the `tokenAmount` in the periphery. This scenario becomes exploitable due to the absence of verification between the user-input `tokenAmount` and the `bridgeAmount` in the `BridgingUtils::executeBridging` function. Attackers can exploit this by calling `DepositsFallbackModule::depositPassivePool` with a `tokenAmount` equating to the Periphery's balance (accumulated from previous users' dust) and a different `bridgeAmount`, causing a revert in the `DepositsModule::depositPassivePool` that triggers the `BridgingUtils::executeBridging` function, thereby bridging the Periphery's balance back to the attacker in the other chain. This issue allows attackers to siphon accumulated dust amounts from the Periphery.

Recommendations

To mitigate this vulnerability and safeguard against potential dust theft, it is recommended to:

1. Implement a mechanism to allow users to reclaim their tokens on the destination chain in case of a bridging failure.

2. Enhance the verification within the `BridgingUtils::executeBridging` function to ensure that the `tokenAmount` matches the `bridgeAmount` exactly. This would prevent the discrepancy that allows the attack to occur.

[M-02] Stale Price Data in `DivReducer` Due to Average Timestamp Calculation

Severity

Impact: High

Likelihood: Low

Description

The `DivReducer` function within the system is designed to calculate the quotient of the prices from two input nodes, typically used for deriving asset prices in alternative currency terms when direct feeds are not available. A critical part of this functionality is the calculation of the `updated_at` timestamp for the output, which currently averages the timestamps of the two input nodes. This approach introduces a significant risk; if one input node provides a very recent timestamp and the other is significantly stale, the averaged timestamp could misleadingly pass staleness checks, thus presenting the output as more current than it actually is. This can lead to the use of outdated price data in critical financial calculations, potentially affecting all dependent systems relying on the accuracy of this feed for timely decision-making.

Recommendations

To mitigate the risk of using stale data and enhance the reliability of the `DivReducer` node's output, amend the logic for determining the `updated_at` timestamp of the `DivReducerNode` output. Instead of averaging the timestamps of the input nodes, use the minimum of the two timestamps. This approach ensures that the output timestamp accurately reflects the freshness of the data, prioritizing the most conservative estimate of data recency.

[M-03] Lack of Price Freshness Verification in Oracle Price Data

Severity

Impact: Medium

Likelihood: Medium

Description

The `getOraclePrice` and `getCollateralExchangeInfo` functions retrieve `NodeOutput.Data` containing price information and a timestamp indicating the freshness of this price. An issue has been identified wherein these functions use the price data directly without verifying the freshness of the data based on the timestamp. This oversight could lead to scenarios where stale or outdated price data is used in significant financial calculations or decision-making processes.

Recommendations

To address this vulnerability and ensure the reliability of price data used throughout the system by introducing logic in both `getOraclePrice` and `getCollateralExchangeInfo` functions to check the timestamp of the `NodeOutput.Data` against a predefined freshness threshold.

[M-04] Invalid Nodes can be registered due to an incorrect check

Severity

Impact: Medium

Likelihood: Medium

Description

DivReducer node is supposed to have 2 parent nodes. During registration node is checked to be valid, however it does nothing if parents are invalid:

```
function _isValidNodeDefinition
      (NodeDefinition.Data memory nodeDefinition) internal view returns (bool valid) {
        if (nodeDefinition.nodeType == NodeDefinition.NodeType.DIV_REDUCER) {
            //check if parents are processable
@>          _hasValidParentNodeDefinitions(nodeDefinition);
        }

        ...
    }

    function _hasValidParentNodeDefinitions
      (NodeDefinition.Data memory nodeDefinition)
        internal
        view
        returns (bool valid)
    {
        for (uint256 i = 0; i < nodeDefinition.parents.length; i++) {
            NodeDefinition.Data memory nodeDef = _getNode
              (nodeDefinition.parents[i]);
            if (!_isValidNodeDefinition(nodeDef)) {
                return false;
            }
        }
        return true;
    }
```

Recommendations

```
function _isValidNodeDefinition
      (NodeDefinition.Data memory nodeDefinition) internal view returns (bool valid) {
        if (nodeDefinition.nodeType == NodeDefinition.NodeType.DIV_REDUCER) {
            //check if parents are processable
-           _hasValidParentNodeDefinitions(nodeDefinition);
+          if( !_hasValidParentNodeDefinitions(nodeDefinition)) return false;
        }
        ...
    }
```

8.4. Low Findings

[L-01] Non-Compliance with EIP-712 Specification in Signature Functions

The `calculateDigest` and `hashExecuteBySig` functions currently do not adhere to the `EIP-712` specification regarding the encoding and hashing of messages for signature verification. `EIP-712` aims to standardize typed data signing with Ethereum, providing a secure and compliant way to generate verifiable and understandable messages. According to the specification, the correct encoding format is `"\x19\x01" ‖ domainSeparator ‖ hashStruct(message)`, with the `domainSeparator` being the result of `hashStruct(eip712Domain)`, where `eip712Domain` is a struct containing fields like `name`, `version`, `chainId`, `verifyingContract`, and `salt`. These fields are essential for ensuring the integrity and domain specificity of signatures, enhancing security against certain attacks.

The deviation from this standard in the current implementation could potentially lead to unexpected integration failures with EIP712-compliant wallets or tooling that perform the encoding in the appropriate way, where users will be requested to sign random bytes instead of a clear message that they can verify.

It is recommended to adopt the OpenZeppelin library's `EIP712.sol` implementation, which is fully compliant with the EIP-712 standard and widely recognized for its security and reliability. This change would ensure consistency with Ethereum's best practices for signing and verifying typed data, enhancing the protocol's overall security posture with minimal impact on functionality.

[L-02] Signature Malleability in ecrecover Precompile Usage

The protocol's current use of the `ecrecover` precompile introduces a security concern due to signature malleability. Specifically, the vulnerability arises from the possibility of altering the `s` and `v` components of a signature, thereby generating a different yet valid signature that corresponds to the same hash and signer. This issue does not presently pose a direct threat to the protocol's security due to the implementation of nonces within the system's signature scheme, which mitigates the risk of replay attacks.

However, addressing this form of signature malleability is considered best practice to fortify the protocol against potential future vulnerabilities or exploits that may arise from unforeseen interactions or changes within the system. OpenZeppelin's ECDSA library provides a solution to this issue ensuring that signatures are both standard and strictly non-malleable.