Reya Network Security Review

Pashov Audit Group

Conducted by: ubermensch, Dan Ogurtsov, pontifex

June 2nd 2024 - June 3rd 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-Labs/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 for perpetuals. 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 - 87ee315c30023bae6ef1af364193189f1e05e7f3

fixes review commit hash - f522b5f203975d2bb9daaa397a52fe853d26fab5

Scope

The following smart contracts were in scope of the audit:

- `Deposits`
- `DepositsModule`
- `Withdrawals`
- `WithdrawalsModule`
- `DataTypes`
- `BridgingUtils`
- `IDepositsModule`
- `IWithdrawalsModule`

7. Executive Summary

Over the course of the security review, ubermensch, Dan Ogurtsov, pontifex engaged with Reya Network to review Reya Network. In this period of time a total of **2** issues were uncovered.

Protocol Summary

Protocol Name	Reya Network
Repository	https://github.com/Reya-Labs/reya-network
Date	June 2nd 2024 - June 3rd 2024
Protocol Type	Perpetuals Trading L2

Findings Count

Severity	**Amount**
Low	2
Total Findings	**2**

Summary of Findings

ID	**Title**	**Severity**	**Status**
[L-01]	Lack of zero amount check	Low	Resolved
[L-02]	Lack of verification for bridging cost	Low	Acknowledged

8. Findings

8.1. Low Findings

[L-01] Lack of zero amount check

The protocol logic restricts withdrawal from accounts and bridging out zero amounts:

```
function withdrawMA(WithdrawMAInputs memory inputs) internal {
        address rUSDProxy = GlobalConfiguration.getRUSDProxyAddress();

        bytes memory extraSignatureData = abi.encode
          (inputs.receiver, inputs.chainId, inputs.socketMsgGasLimit);

>>      CoreUtils.coreWithdraw(
  inputs.accountId,
  inputs.token,
  inputs.tokenAmount,
  inputs.sig,
  extraSignatureData
);
<...>
    function coreWithdraw(
        uint128 accountId,
        address token,
        uint256 tokenAmount,
        EIP712Signature memory sig,
        bytes memory extraSignatureData
    )
        internal
    {
        IExecutionModule(GlobalConfiguration.getCoreProxyAddress
          ()).executeBySig({
            accountId: accountId,
>>          commands: singleCoreCommand
  (CommandType.Withdraw, token, tokenAmount),
            sig: sig,
            extraSignatureData: extraSignatureData
        });
    }
<...>
```

```
<...>
    function executeBySig(
        uint128 accountId,
        Command[] calldata commands,
        EIP712Signature memory sig,
        bytes memory extraSignatureData
    )
        external
        override
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        // note, signature is only used for accountId check, if there are batch
        // match orders then counterparty
        // permissions are checked directly

        Account.Data storage account = Account.exists(accountId);
        address accountOwner = AccountRBAC.load(account.id).owner;

        uint256 incrementedNonce = Signature.incrementSigNonce(accountOwner);

        Signature.validateRecoveredAddress(
            Signature.calculateDigest(
                hashExecuteBySig(
                  accountId,
                  commands,
                  incrementedNonce,
                  sig.deadline,
                  extraSignatureData
                )
            ),
            accountOwner,
            sig
        );

>>      return _execute(account, commands);
<...>
```

```
<...>
    function _execute(
        Account.Data storage account,
        Command[] calldata commands
    )
        internal
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        account.ensureAccess();

        // execution
        outputs = new bytes[](commands.length);

        for (uint256 i = 0; i < commands.length; i++) {
>>         outputs[i] = executeCommand(account, commands[i]);
<...>
```

```
<...>
        if (command.commandType == CommandType.Withdraw) {
            (address collateral, uint256 collateralAmount) = abi.decode
              (command.inputs, (address, uint256));
>>          if (collateralAmount == 0) {
                revert Errors.ZeroWithdraw(account.id, collateral);
```

But it can be done for addresses via `WithdrawalsModule.withdraw` function because of the zero amount check absence in the `Withdrawals.withdraw`:

```
function withdraw(WithdrawInputs memory inputs) internal {
        address rUSDProxy = GlobalConfiguration.getRUSDProxyAddress();

        inputs.token.safeTransferFrom(msg.sender, address
          (this), inputs.tokenAmount);

        address withdrawToken = inputs.token;

        if (inputs.token == rUSDProxy) {
            IWrappedERC20(rUSDProxy).withdraw(inputs.tokenAmount);
            withdrawToken = IWrappedERC20(rUSDProxy).getUnderlyingAsset();
        }

        BridgingUtils.executeDirectBridging({
            withdrawToken: withdrawToken,
            chainId: inputs.chainId,
            socketMsgGasLimit: inputs.socketMsgGasLimit,
            tokenAmount: inputs.tokenAmount,
            receiver: inputs.receiver
        });
    }
```

Consider checking that the `inputs.tokenAmount` parameter is greater than zero.

[L-02] Lack of verification for bridging cost

The `executeDirectBridging` function is used for withdrawals and relies on `msg.value` expecting the withdrawer to cover the bridging cost. However, there is no verification on this amount. If the provided amount is lower than needed, the transaction will simply revert on the Socket bridge side. However, if the amount is significantly higher, the Socket bridge considers any extra fees beyond the required amount as an execution fee:

Source: https://github.com/SocketDotTech/socket-DL/blob/42f1703c427246661d84d94e5fcb77e050c1a784/contracts/ExecutionManager.sol#L249-L250

```
// any extra fee is considered as executionFee
executionFee = msgValue - transmissionFees - switchboardFees_;
```

This lack of restriction can lead to users overspending during withdrawals due to the excessive bridging cost.

Introduce a verification mechanism to ensure that `msg.value` is within an acceptable range for the expected bridging cost. This can be achieved by calculating the min fees (just like the other `withdrawMA` and `withdrawPassivePool`) and checking if `msg.value` is not excessively higher than the required amount before proceeding with the bridging operation. This will prevent users from inadvertently overpaying for withdrawals.