Reya Network Security Review

Pashov Audit Group

Conducted by: Peakbolt, SpicyMeatball, 0xbepresent, rvierdiiev, Shaka, 0xCiphky

June 29th 2024 - July 9th 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 - 0dee13ca39660bcfae64184163d7cf84cf67c722

fixes review commit hash - 56d589baa31fdcdbf6f8bb15618327a1f731d37b

Scope

The following smart contracts were in scope of the audit:

- `CollateralPoolPermissions`
- `SignatureHelpers`
- `AccountLiquidation`
- `AutoExchange`
- `AccountModule`
- `AutoExchangeConfigurationModule`
- `CollateralModule`
- `CollateralPoolModule`
- `ExecutionModule`
- `InsuranceFundConfigurationModule`
- `RiskConfigurationModule`
- `Account`
- `AccountRBAC`
- `ConfigurationModule`
- `PassivePerpInformationModule`
- `Market`
- `CoreAccountPermission`
- `Signature`
- `/orders-gateway`

7. Executive Summary

Over the course of the security review, Peakbolt, SpicyMeatball, 0xbepresent, rvierdiiev, Shaka, 0xCiphky engaged with Reya Network to review Reya Network. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Reya Network
Repository	https://github.com/Reya-Labs/reya-network
Date	June 29th 2024 - July 9th 2024
Protocol Type	Perpetuals Trading L2

Findings Count

Severity	Amount
High	1
Medium	3
Low	3
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect increment of signature nonce	High	Resolved
[M-01]	Admin role evasion risk	Medium	Resolved
[M-02]	Inability to cancel signed commands	Medium	Acknowledged
[M-03]	Nonce reset vulnerability on Owner change	Medium	Resolved
[L-01]	validatePermission() can be bypassed	Low	Resolved
[L-02]	Signature malleability attack	Low	Resolved
[L-03]	No ability to invalidate nonce	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Incorrect increment of signature nonce

Severity

Impact: Medium

Likelihood: High

Description

`ExecutionModule.executeBySig()` will `incrementSigNonce(accountOwner)` to increment the nonce for `accountOwner` to prevent replay attacks by ensuring each signature can only be used once.

However, it incorrectly increments the nonce for `accountOwner` even when the signer is not the `acccountOwner`.

This allows anyone to DoS the transactions of `accountOwner` or other valid signers, by incrementing the nonce by calling `executeBySig()`. When the nonce for `accountOwner` is incremented, it will reject all transactions with a signature generated using the previous nonces.

As Reya Network does not require a gas fee, the DoS attack could easily be executed by spamming `executeBySig()` continuously. Furthermore, this can be performed by anyone by calling `executeBySig()` with zero commands to bypass any permission check on the signer.

In the worst case, the attacker could DoS withdrawal, causing funds to be locked within the contract.

The same issue applies to `AccountModule.grantAccountPermissionBySig()` and `AccountModule.revokeAccountPermissionBySig()`.

```
function executeBySig(
        uint128 accountId,
        Command[] calldata commands,
        EIP712Signature memory sig,
        bytes memory extraSignatureData
    )
        external
        override
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        Account.Data storage account = Account.exists(accountId);
        address accountOwner = AccountRBAC.load(account.id).owner;
        //@audit this should increment nonce for signer instead of accountOwner
>>>     uint256 incrementedNonce = Signature.incrementSigNonce(accountOwner);
        address recoveredAddress = Signature.generateRecoveredAddress(
            Signature.calculateDigest(
                hashExecuteBySig(
                  accountId,
                  commands,
                  incrementedNonce,
                  sig.deadline,
                  extraSignatureData
                )
            ),
            sig
        );

        return _execute(accountId, commands, recoveredAddress);
    }
```

Recommendations

Increment the signature nonce of the signer (`recoveredAddress`) instead of `accountOwner`.

8.2. Medium Findings

[M-01] Admin role evasion risk

Severity

Impact: High

Likelihood: Low

Description

The function `AccountModule::revokeAccountPermission` is designed to remove permissions from a user. However, a malicious admin could preemptively execute a front-run by calling `AccountModule::grantAccountPermission` to assign admin permissions to another controlled account before their own permissions are revoked. Consider this scenario:

1. Permissions are being revoked for an admin using the function `AccountModule::revokeAccountPermission`.
2. The malicious admin performs a front-run, executing the function `AccountModule::grantAccountPermission` to assign admin privileges to another controlled user.
3. The transaction from step 1 completes, but the malicious admin retains access through the newly empowered user.

Recommendations

It is advisable to restrict the capability to assign admin roles exclusively to the system owner.

[M-02] Inability to cancel signed commands

Severity

Impact: Medium

Likelihood: Medium

Description

Currently, the `nonce` for a signed message can only be incremented through the execution of the `ExecutionModule::executeBySig` function:

```
function executeBySig(
        uint128 accountId,
        Command[] calldata commands,
        EIP712Signature memory sig,
        bytes memory extraSignatureData
    )
        external
        override
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        Account.Data storage account = Account.exists(accountId);
        address accountOwner = AccountRBAC.load(account.id).owner;

>>      uint256 incrementedNonce = Signature.incrementSigNonce(accountOwner);
        address recoveredAddress = Signature.generateRecoveredAddress(
            Signature.calculateDigest(
                hashExecuteBySig(
                  accountId,
                  commands,
                  incrementedNonce,
                  sig.deadline,
                  extraSignatureData
                )
            ),
            sig
        );

        return _execute(accountId, commands, recoveredAddress);
    }
```

This design does not provide a mechanism for an owner or admin to cancel a command, which can be problematic, especially in situations where urgent cancellation of orders, such as withdrawals or transfer accounts, or in transactions reverts like Out of gas error, etc.

Recommendations

To address this issue, it is recommended to implement a function that allows the owner or a designated administrator to manually increment the `nonce`.

[M-03] `Nonce` reset vulnerability on Owner change

Severity

Impact: High

Likelihood: Low

Description

The nonce, which is tied to the `accountOwner`, is reset to zero when the owner is changed using the `AccountModule::notifyAccountTransfer` function. This resetting allows for the potential replay of previously processed signed messages, posing a significant security risk.

```
// ExecutionModule.sol
    function executeBySig(
        uint128 accountId,
        Command[] calldata commands,
        EIP712Signature memory sig,
        bytes memory extraSignatureData
    )
        external
        override
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        Account.Data storage account = Account.exists(accountId);
        address accountOwner = AccountRBAC.load(account.id).owner;

>>      uint256 incrementedNonce = Signature.incrementSigNonce(accountOwner);
        address recoveredAddress = Signature.generateRecoveredAddress(
            Signature.calculateDigest(
                hashExecuteBySig(
                  accountId,
                  commands,
                  incrementedNonce,
                  sig.deadline,
                  extraSignatureData
                )
            ),
            sig
        );

        return _execute(accountId, commands, recoveredAddress);
    }
```

Recommendations

It is recommended not to reset the `nonce` when an account owner is changed.

8.3. Low Findings

[L-01] `validatePermission()` can be bypassed

`executeBySig()` will check the permission of the signer within `executeCommand()`, which will trigger `validatePermission()`.

However, the signer can bypass the permission check by calling `executeBySig()` with zero commands, as it will not execute the for-loop and call `executeCommand()`.

Currently, this will allow the signer to increment the signature nonce of `accountOwner` (reported in a separate issue).

```
function _execute(
        uint128 accountId,
        Command[] calldata commands,
        address signer
    )
        internal
        returns (bytes[] memory outputs, MarginInfo memory usdNodeMarginInfo)
    {
        Account.Data storage account = Account.exists(accountId);
        account.ensureAccess();

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


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

        // post-execution checks
        if (AccountCollateral.hasPool(account.id)) {
            usdNodeMarginInfo = account.getUsdNodeMarginInfo();
            AccountChecks.checkAboveIm(account.id, usdNodeMarginInfo);
        }
    }
```

This issue can be resolved by requiring `commands.length` to be greater than zero.

[L-02] Signature malleability attack

`Signature.generateRecoveredAddress()` uses `ecrecover()` to obtain the signer address from the signature to validate the

However, it is susceptible to signature malleability attacks as it does not reject non-unique (malleable) signatures, which are allowed by `ecrecover()`.

That means an attacker can obtain the signature from a pending transaction and alter the v, r, s values to create another valid signature for that particular hash. A frontrunning attack could then be conducted with the new signature to DoS the legitimate transaction as the nonce would be incremented by the attack.

Suppose the scenario,

1. Victim submits tx1 with a signature.
2. Attacker obtains signature from tx1, alter the v,r,s values to create a new valid signature.
3. Attacker frontrun tx1 with tx2 using the new signature.
4. When tx2 is executed, it will increment the nonce via `Signature.incrementSigNonce()`.
5. Now when Victim's tx1 is executed, it will fail, as the nonce has been incremented.

Note that the frontrunning attack is only possible for chains with mempool and is not possible for Reya Network chain as transactions are executed based on FIFO.

```
function generateRecoveredAddress(
        bytes32 digest,
        EIP712Signature memory sig
    ) internal view returns (address recoveredAddress) {
        if (sig.deadline < block.timestamp) revert SignatureExpired();
        //@audit this is susceptible to signature malleability attack
>>>     recoveredAddress = ecrecover(digest, sig.v, sig.r, sig.s);
        if (recoveredAddress == address(0)) revert SignatureInvalid();
    }
```

This issue can be resolved by using OZ ECDSA library, which checks the r and s values and rejects any non-unique (malleable) signatures.

[L-03] No ability to invalidate nonce

When the user signs a new stop loss order he encodes the nonce that will be used with it. Nonces can be used in an unordered way and once the order is signed anyone can execute it on behalf of the signer, and there is no ability for the signer to invalidate the order, the only thing he can do is to create a new order with the same nonce and execute it. But it can be inconvenient or even impossible to do.

We believe that it will be useful for the protocol to have a function that allows invalidating nonce for accounts.