Reya Network Security Review

Pashov Audit Group

Conducted by: 0xbepresent, Shaka, pontifex

August 3rd - August 8th

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 - eccefaaa52c204b244ebf46b0b8c65f1bafb9ea5

fixes review commit hash - 75fb7305127161490c8ea33654a978d77b4a61de

Scope

The following smart contracts were in scope of the audit:

- `Pyth`
- `Stork`
- `OracleUpdateModule`
- `PythPriceInformationModule`
- `StorkERC7412WrapperModule`
- `StorkPriceInformationModule`
- `Configuration`
- `NodeModule`
- `PythOffchainLookupNode`
- `StorkOffchainLookupNode`
- `NodeDefinition`
- `AccountPermissions`
- `ConditionChecks`
- `DataTypes`
- `ConditionChecks`
- `DataTypes`
- `Errors`
- `Events`
- `ExecuteConditionalOrder`
- `ExecuteMatchOrder`
- `FeatureFlagSupport`
- `SessionOrdersHashing`
- `SessionUpdateHashing`
- `ExecuteLimitOrder`
- `ExecuteSlOrder`
- `ExecuteTpOrder`
- `Utils`
- `ExecutionModule`
- `SessionOrderModule`
- `SessionPermissionModule`
- `NonceMap`
- `SessionPermissions`

7. Executive Summary

Over the course of the security review, 0xbepresent, Shaka, pontifex engaged with Reya Network to review Reya Network. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Reya Network
Repository	https://github.com/Reya-Labs/reya-network
Date	August 3rd - August 8th
Protocol Type	Perpetuals Trading L2

Findings Count

Severity	**Amount**
Medium	9
Low	3
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Stork and Pyth prices can be arbitrated	Medium	Acknowledged
[M-02]	InitiateOrUpdateSession typehash is encoded incorrectly	Medium	Resolved
[M-03]	Confidence interval of Pyth price is not validated	Medium	Acknowledged
[M-04]	Incorrect parameters length check	Medium	Resolved
[M-05]	Lack of signature cancellation functionality	Medium	Acknowledged
[M-06]	Potential loss of Ether due to overpayment	Medium	Acknowledged
[M-07]	Misconfiguration of pythAddress causes failure in dependent contracts	Medium	Resolved
[M-08]	Unauthorized signature cancellation in ExecutionModule	Medium	Acknowledged
[M-09]	Nonce front-running vulnerability	Medium	Acknowledged
[L-01]	Check for the trusted executor on Pyth price update is not enforced	Low	Acknowledged
[L-02]	Potential loss of native tokens	Low	Resolved
[L-03]	The account owners have to initiate themselves for sessions	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Stork and Pyth prices can be arbitrated

Severity

Impact: Medium

Likelihood: Medium

Description

`fullfillOracleQueryStork` and `fullfillOracleQueryPyth` functions are used to update the price of the Stork and the Pyth oracles, respectively.

Given that both oracles work with a pull model, this opens the opportunity to arbitrage a price update by performing a trade, setting the new price, and closing the position, all in the same transaction.

What is more, the price can be updated multiple times in the same block, so the attacker can choose both the initial and final price, as long as they satisfy the following conditions:

- The initial price timestamp is greater than the previous price timestamp.
- The final price timestamp is greater than the initial price timestamp.

This increases the chances to perform a profitable arbitrage, especially in moments of high volatility.

Note that while `FeatureFlagSupport.ensureExecutorAccess()` ensures that the caller is allowed to perform the price update, and the protocol will start restricting the update of the price to trusted entities, it is planned to allow anyone to perform this action in the future, which can be done by setting `allowAll` to `true` for the executor feature flag. Also, in the case of the Pyth oracle, it is possible to update the price by calling directly the `Pyth` contract.

Recommendations

Ensure that the Stork and Pyth prices are not updated more than once per block and that the new price is not stale.

[M-02] `InitiateOrUpdateSession` typehash is encoded incorrectly

Severity

Impact: Low

Likelihood: High

Description

According to the ERC-712 standard:

> The type of a struct is encoded as `name ‖ "(" ‖ member`$_1$` ‖ "," ‖ member`$_2$` ‖ "," ‖ … ‖ member`$_n$` ")"` where each member is written as `type ‖ " " ‖ name`.

In `SessionUpdateHashing.sol`, the `_INITIATE_OR_UPDATE_SESSION_TYPEHASH` constant does not follow the standard, as it includes an extra space between the `deadline` and `session` members.

As a result, `SessionPermissionModule.verifySignature` will fail to verify a correct signature for the given session data.

Recommendations

```
bytes32 constant _INITIATE_OR_UPDATE_SESSION_TYPEHASH = keccak256(
        //solhint-disable-next-line max-line-length
-       "InitiateOrUpdateSession
- (uint256 verifyingChainId,uint256 deadline, InitiateOrUpdateSessionDetails session)I
+       "InitiateOrUpdateSession
+ (uint256 verifyingChainId,uint256 deadline,InitiateOrUpdateSessionDetails session)In
    );
```

[M-03] Confidence interval of Pyth price is not validated

Severity

Impact: High

Likelihood: Low

Description

`PythOffchainLookupNode.process` does not validate the confidence interval of the Pyth price.

As stated in the Pyth documentation, it is important to check this value to prevent the contract from accepting untrusted prices.

Recommendations

```
+       if (latestPrice.conf > 0 && (latestPrice.price / int64
+ (latestPrice.conf) < minConfidenceRatio)) {
+           revert LowConfidencePyth
+ (latestPrice.price, latestPrice.conf, oracleAdaptersProxy);
+       }
```

The `minConfidenceRatio` value could be an additional parameter for Pyth nodes, a global configuration or a constant in the contract.

Note that a confidence interval of 0 means no spread in price, so should be considered as a valid price.

[M-04] Incorrect parameters length check

Severity

Impact: Low

Likelihood: High

Description

`PythOffchainLookupNode.isValid` check on the length of the parameters data is incorrect.

```
// Must have correct length of parameters data
    if (nodeDefinition.parameters.length != 32 * 4) {
        return false;
    }
```

The parameters data for Pyth nodes is composed of the oracle adapter address and the pair id (of type bytes32) ABI encoded, which is 64 bytes long, not 128 bytes long.

As a result, registering a node with the correct parameters will fail. On the other hand, padding the parameters data with 64 bytes after the correct parameters will enable users to register the same node data multiple times, by changing the padding data.

Proof of concept

```
function test_pythParams() public {

    bytes32 pairId = bytes32(uint256(2));
    bytes memory parameters = abi.encode(oracleAdaptersProxy, pairId);
    assertEq(parameters.length, 32 * 2);
}
```

Recommendations

```
// Must have correct length of parameters data
-       if (nodeDefinition.parameters.length != 32 * 4) {
+       if (nodeDefinition.parameters.length != 32 * 2) {
            return false;
        }
```

[M-05] Lack of signature cancellation functionality

Severity

Impact: Medium

Likelihood: High

Description

The `SessionOrderModule.sol` contract allows users to execute orders by signing messages using the `executeSessionOrdersBySig` function. However, no functionality is provided to cancel a previously signed message. This omission poses a security risk, as users may want to revoke or cancel their signed orders in case of a change in intention or if they suspect their private key might have been compromised.

Currently, the only way to cancel the execution of an order is to revoke a user's permissions, but this only addresses situations where a private key is compromised. This option is not viable when a user simply wants to cancel already signed orders.

```
function _executeSessionOrders(
        address signer,
        uint128 accountId,
        MatchOrderDetails[] memory orders
    )
        internal
        returns (bytes[] memory)
    {
>>>     if (!SessionPermissions.isPermissionActive(accountId, signer)) {
>>>         revert Errors.Unauthorized(signer);
        }

        return executeMatchOrders
          (accountId, orders, Configuration.getCoreProxyAddress());
    }
```

Recommendations

To mitigate this vulnerability, it is recommended that a function that allows users to cancel their signed messages be implemented. This function should mark the nonce of the signed message as used, preventing any future execution of the same message. It is important to note that cancellation should only be possible by the creator of the signed message nonce or by the owner of the `accountId` as it mentioned in another report.

[M-06] Potential loss of Ether due to overpayment

Severity

Impact: Medium

Likelihood: Medium

Description

In the `fullfillOracleQueryPyth` function within `Pyth.sol`, there is a potential issue where users can inadvertently send more Ether (`msg.value`) than required to the `pyth.updatePriceFeeds` function. This overpayment has not been refunded, resulting in a loss of Ether.

The function implementation is as follows:

```
function fullfillOracleQueryPyth
  (address pythAddress, bytes calldata signedOffchainData) {
    IPyth pyth = IPyth(pythAddress);

    (bytes[] memory updateData) = abi.decode(signedOffchainData, (bytes[]));

>>  try pyth.updatePriceFeeds{ value: msg.value }(updateData) { }
    catch (bytes memory reason) {
        if (_isFeeRequired(reason)) {
            revert Errors.FeeRequiredPyth(pyth.getUpdateFee(updateData));
        } else {
            uint256 len = reason.length;
            assembly {
                revert(add(reason, 0x20), len)
            }
        }
    }
}
```

The `msg.value` is transferred without validating the exact fee required by `pyth.updatePriceFeeds`. Based on the `Pyth::updatePriceFeeds` function at the address `0xdd24f84d36bf92c65f92307595335bdfab5bbd21`, there is no mechanism that returns any excess ETH sent:

```
function updatePriceFeeds(
        bytes[] calldata updateData
    ) public payable override {
        uint totalNumUpdates = 0;
        for (uint i = 0; i < updateData.length; ) {
            if (
                updateData[i].length > 4 &&
                UnsafeCalldataBytesLib.toUint32(updateData[i], 0) ==
                ACCUMULATOR_MAGIC
            ) {
                totalNumUpdates += updatePriceInfosFromAccumulatorUpdate(
                    updateData[i]
                );
            } else {
                updatePriceBatchFromVm(updateData[i]);
                totalNumUpdates += 1;
            }

            unchecked {
                i++;
            }
        }
        uint requiredFee = getTotalFee(totalNumUpdates);
        if (msg.value < requiredFee) revert PythErrors.InsufficientFee();
    }
```

Therefore, two scenarios could occur:

- A user sends more ETH than expected, resulting in the extra ETH not being returned to the user by `Pyth`.
- Since the `Pyth` contract is upgradeable, the fees in `pyth` could change before the function `fullfillOracleQueryPyth` is executed, causing them to differ from what was expected.

Recommendations

Before calling `pyth.updatePriceFeeds`, calculate and validate the exact fee required. This can be achieved by calling `pyth.getUpdateFee(updateData)` and comparing it with `msg.value`.

```
function fullfillOracleQueryPyth
  (address pythAddress, bytes calldata signedOffchainData) {
    IPyth pyth = IPyth(pythAddress);

    (bytes[] memory updateData) = abi.decode(signedOffchainData, (bytes[]));
+   uint fee = pyth.getUpdateFee(updateData);
+   require(msg.value == fee);
    try pyth.updatePriceFeeds{ value: msg.value }(updateData) { }
    catch (bytes memory reason) {
        if (_isFeeRequired(reason)) {
            revert Errors.FeeRequiredPyth(pyth.getUpdateFee(updateData));
        } else {
            uint256 len = reason.length;
            assembly {
                revert(add(reason, 0x20), len)
            }
        }
    }
}
```

[M-07] Misconfiguration of `pythAddress` causes failure in dependent contracts

Severity

Impact: High

Likelihood: Low

Description

The `oracle-adapters/src/storage/Configuration.sol` library is responsible for storing and retrieving configuration data, including the address for the Pyth oracle (`pythAddress`). However, the current implementation does not set the `pythAddress`, leading to potential issues in contracts that rely on this address. Specifically, if the `pythAddress` is not set, any calls to retrieve or use this address will fail.

The relevant code section from `Configuration.sol` shows that the `pythAddress` is defined but not set:

```
...
...
    struct Data {
        /**
         * @dev Stork Verify Contract Address
         */
        address storkVerifyContract;
        /**
         * @dev Pyth Address
         */
>>>     address pythAddress;
    }
...
...
    function set(Data memory config) internal {
        Data storage storedConfig = load();

        storedConfig.storkVerifyContract = config.storkVerifyContract;

        emit Events.ConfigurationUpdated(config, block.timestamp);
    }
...
...
    function getPythAddress() internal view returns (address pythAddress) {
>>>     pythAddress = load().pythAddress;
        if (pythAddress == address(0)) {
>>>         revert Errors.PythAddressNotSet();
        }
    }
...
...
```

Failure to properly set the `pythAddress` can lead to:

- Inability to interact with the Pyth oracle, resulting in failed oracle queries.
- Potential denial of service in contracts that depend on Pyth for critical data feeds.

Test:

```
// File: OracleAdapter.t.sol
    function test_setConfiguration_Pyth() public {
        Configuration.Data memory existingConfig = module.getConfiguration();
        assertEq(address(0), existingConfig.storkVerifyContract);
        assertEq(address(0), existingConfig.pythAddress);
        //
        // 1. `pythAddress` set to address(123)
        config = Configuration.Data({
            storkVerifyContract: mockStorkVerifyContract,
            pythAddress: address(123)
         });
        module.setConfiguration(config);
        existingConfig = module.getConfiguration();
        assertEq
          (config.storkVerifyContract, existingConfig.storkVerifyContract);
        //
        // 2. `pythAddress` is zero address
        assertEq(address(0), existingConfig.pythAddress);
    }
```

Recommendations

Ensure that the `pythAddress` is set correctly in the `Configuration::set` function:

```
function set(Data memory config) internal {
        Data storage storedConfig = load();

        storedConfig.storkVerifyContract = config.storkVerifyContract;
+       storedConfig.pythAddress = config.pythAddress;
        emit Events.ConfigurationUpdated(config, block.timestamp);
    }
```

[M-08] Unauthorized signature cancellation in `ExecutionModule`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `ExecutionModule.sol` contract, the `cancelSignature` function allows an authorized user to cancel a signature by marking a nonce as used. This functionality is intended to invalidate any signed messages associated with that nonce, preventing further execution. However, the current implementation permits any user authorized under the same `accountId` to cancel signatures, which can lead to potential misuse.

```
function cancelSignature
      (uint128 accountId, uint256 nonce) external override {
        FeatureFlagSupport.ensureGlobalAccess();

>>>     if (!isAuthorizedToTrade
  (accountId, msg.sender, Configuration.getCoreProxyAddress())) {
            revert Errors.Unauthorized(msg.sender);
        }

>>>     NonceMap.useUnorderedNonce(accountId, nonce);
    }
}
```

In the above code, the `ExecutionModule::cancelSignature` function checks if the caller (`msg.sender`) is authorized to trade for the specified `accountId`. If authorized, it then proceeds to use the nonce, effectively canceling the signature associated with that nonce.

The issue arises when multiple users are authorized (has the `MATCH_ORDER` permission) for the same `accountId`. Any of these users can call `cancelSignature` to invalidate signatures from other authorized users, which could potentially lead to disruption of legitimate transactions. Consider the next scenario:

1. `AccountId.Owner` assigns `MATCH_ORDER` permission to `alice` and `bob`.
2. `alice` signs a message using `nonce=1` in order to be executed using the `ExecutionModule::execute` function.
3. `bob` maliciously calls `ExecutionModule::cancelSignature` to cancel `nonce=1` before the alice's message is executed and invalidates the `alice` message `nonce=1`.

```
// File: ExecutionModule.t.sol
    function test_cancelSignature_ByOtherUserWithPermissionInTheSameAccountId
      () public {
        (address bob,) = makeAddrAndKey("bob");
        //
        // 1. Bob has permission (`MATCH_ORDER`) in `accountId=1`
        vm.mockCall(
            mockCoreProxy,
            abi.encodeCall(
                IAccountModule.isAuthorizedForAccount,
                (
                  1,
                  0x5d3e646d0f5446da275659f3f1489b5d0bf7a151df31e98bb653f21267dd0fad,
                  bob
                )
            ),
            abi.encode(true)
        );
        //
        // 2. Bob cancels alice's order.
        vm.prank(bob);
        module.cancelSignature(1, 1);
    }
```

Recommendations

Ensure that only the user who created a nonce (or account's owner) can cancel it, thus preventing unauthorized signature cancellations by other users authorized for the same `accountId`.

[M-09] Nonce front-running vulnerability

Severity

Impact: High

Likelihood: Low

Description

In the current implementation of the `NonceMap`, the `nonce` is used to prevent double-spending and replay attacks. However, it is possible for a malicious user with active permissions to the account to preemptively use the same nonce as a legitimate user, causing the legitimate user's order to become invalid. The vulnerability lies in the lack of proper isolation or protection of nonces between different users who have permission to interact with the same `account`.

The error occurs in `SessionOrderModule.sol` where a malicious user only needs to have permissions within the account to cancel orders of other users from the same account. Consider the following scenario:

1. `account's owner` assings permissions using the `SessionPermissionModule::initiateOrUpdateSession` to `alice` and `bob`.
2. `alice` creates a signed message using `nonce=100`.
3. `bob` maliciously creates a signed message using the same `nonce=100` and submits it before Alice's transaction.
4. As a result, Alice's order becomes unexecutable due to the nonce already being used by `bob`.

```
// File: SessionOrderModule.sol
...
...
    function executeSessionOrdersBySig(
        ExecuteSessionOrdersDetails memory orders,
        EIP712Signature calldata signature
    )
        external
        override
        returns (bytes[] memory)
    {
        FeatureFlagSupport.ensureGlobalAccess();

>>>     if (NonceMap.load(orders.accountId).hasNonceBeenUsed(orders.nonce)) {
            revert Errors.NonceAlreadyUsed(orders.accountId, orders.nonce);
        }

        if (!verifySignature(orders, signature)) {
            revert Errors.InvalidSignature();
        }

>>>     NonceMap.useUnorderedNonce(orders.accountId, orders.nonce);

        return _executeSessionOrders
          (orders.signer, orders.accountId, orders.orders);
    }
...
...
    function _executeSessionOrders(
        address signer,
        uint128 accountId,
        MatchOrderDetails[] memory orders
    )
        internal
        returns (bytes[] memory)
    {
>>>     if (!SessionPermissions.isPermissionActive(accountId, signer)) {
            revert Errors.Unauthorized(signer);
        }

        return executeMatchOrders
          (accountId, orders, Configuration.getCoreProxyAddress());
    }
...
...
```

The same behavior occurs in `ExecutionModule.sol` where orders from other users within the same account can be canceled. The attack would proceed as follows:

1. The `account's owner` assigns `MATCH_ORDER` permissions to `alice` and `bob`.
2. `alice` creates a signed message using `nonce=100`.
3. Maliciously, `bob` creates an order using `nonce=100` and sends the transaction before `alice`, preventing `alice`'s legitimate transaction from executing.

It should be noted that this attack vector is quite different from the one described in the report where orders from other users of the same `accountId` can be canceled using the `ExecutionModule::cancelSignature` function.

Similarly, I would also say that the `SessionPermissionModule.sol` would be affected since it uses the same `NonceMap` mechanism, allowing a user to potentially block the assignment of permissions within an account. Consider the following scenario:

1. `alice` obtains a signed message off-chain from the `account's owner` with `nonce=100`. The signed message assigns permissions to user X.
2. `bob` obtains a signed message off-chain from the `account's owner` using the same `nonce=100`. The signed message assigns permissions to user Z.
3. `bob` maliciously sends the transaction before alice (frontrunning), preventing `alice`'s transaction from being processed.

Recommendations

Implement a mechanism to ensure that nonces are unique per user, not just per account.

8.2. Low Findings

[L-01] Check for the trusted executor on Pyth price update is not enforced

`OracleUpdateModule` allows trusted executors to update the Stork and Pyth oracles price.

For the Stork oracle, the price is verified and then saved in storage. However, for the Pyth oracle, the price is updated in the `Pyth` contract. Given that anyone can interact directly with the `Pyth` contract, the check for the caller to be a trusted executor can be easily bypassed.

[L-02] Potential loss of native tokens

The `fulfillOracleQuery` function within the `OracleUpdateModule` contract is marked as `payable`, which means it can accept Ether (`msg.value`). However, when the `oracleProvider` is set to `STORK`, the function does not handle the `msg.value` in any particular way. This can lead to a situation where a user inadvertently sends Ether along with their call to `fulfillOracleQuery` for a `STORK` oracle provider, resulting in the loss of native tokens. The Ether sent in such a transaction is not utilized or refunded, causing users to lose the sent amount.

```
function fulfillOracleQuery(
        OracleProvider oracleProvider,
        bytes calldata signedOffchainData
    )
        external
>>>     payable
        override
    {
        FeatureFlagSupport.ensureGlobalAccess();

        // note, if an executor is trusted, they are allowed to execute a
        // fullfill oracle query operation
        // on top of any oracle provider type (e.g. stork, pyth, etc)
        FeatureFlagSupport.ensureExecutorAccess();

>>>     if (oracleProvider == OracleProvider.STORK) {

            fullfillOracleQueryStork(storkVerifyContract, signedOffchainData);
        } else if (oracleProvider == OracleProvider.PYTH) {
            address pythAddress = Configuration.getPythAddress();
            fullfillOracleQueryPyth(pythAddress, signedOffchainData);
        }
    }
```

In the above code, there is no handling of `msg.value` when `oracleProvider` is `STORK`, which can lead to native token loss.

It is recommended to implement a check to accept Ether only when the `oracleProvider` is `PYTH`

[L-03] The account owners have to initiate themselves for sessions

The `accountId` owners can not invoke the `SessionOrderModule.executeSessionOrdersBySig` function and the `SessionOrderModule.executeSessionOrders` function while they do not initiate themselves via the `SessionPermissionModule._initiateOrUpdateSession` function though the `accountId` owners are authorized to trade by default.

```
function _executeSessionOrders(
        address signer,
        uint128 accountId,
        MatchOrderDetails[] memory orders
    )
        internal
        returns (bytes[] memory)
    {
>>      if (!SessionPermissions.isPermissionActive(accountId, signer)) {
            revert Errors.Unauthorized(signer);
        }

        return executeMatchOrders
          (accountId, orders, Configuration.getCoreProxyAddress());
    }
<...>

library SessionPermissions {
<...>
    function isPermissionActive
      (uint128 accountId, address target) internal view returns (bool) {
        uint256 activeUntil = load(accountId).permissions[target];

        return activeUntil >= block.timestamp;
    }
```

Consider checking if the `msg.sender` is the `accountId` owner in the `_executeSessionOrders` function:

```
+import { isOwner } from "../libraries/AccountPermissions.sol";
<...>
    function _executeSessionOrders(
        address signer,
        uint128 accountId,
        MatchOrderDetails[] memory orders
    )
        internal
        returns (bytes[] memory)
    {
-       if (!SessionPermissions.isPermissionActive(accountId, signer)) {
+       if (!SessionPermissions.isPermissionActive(accountId, signer) &&
+           !isOwner(accountId, msg.sender, Configuration.getCoreProxyAddress
+ ())) {
            revert Errors.Unauthorized(signer);
        }

        return executeMatchOrders
          (accountId, orders, Configuration.getCoreProxyAddress());
    }
```