Rhinestone (Smart Sessions) Audit Report

Version 2.0

Audited by:

MiloTruck

bytes032

December 25, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Smart Sessions

Smart Sessions are a framework for creating on-chain permissions using ERC-7579 modules. Each permission consists of two components: Policies and Session Validators.

A Policy is a submodule that enforces restrictions on the kinds of ERC-4337 userOps or ERC-1271 data that can be signed. Policies can be a UserOp Policy (allowing for a broader context of policy enforcement over the entire UserOp) or an ActionPolicy (enforcing limitations that apply to a specific action a UserOp executes).

Session Validators verify the signature issued by the signer of the ERC-4337 UserOperation or ERC-1271 data. The Smart Session module orchestrates using these submodules in accordance with a PermissionID to enable on-chain permissions.

2.2 Overview

Project	Rhinestone (Smart Sessions)
Repository	smartsessions
Commit Hash	821a4f51a65a...
Mitigation Hash	fa4a4d787bd8...
Date	16 December 2024 - 25 December 2024

2.3 Issues Found

Severity	Count
High Risk	0
Medium Risk	2
Low Risk	0
Informational	1
Total Issues	**3**

3 Findings Summary

ID	Description	Status
M-1	Missing checks in `ExecutionLib.decodeUserOpCallData()`	Resolved
M-2	`ExecutionLib.decodeBatch()` does not revert when `executionData` is malformed if `pointers.length = 0`	Resolved
I-1	Session digest does not change when called with `USE` or `ENABLE` modes	Acknowledged

4 Findings

Medium Risk

[M-1] Missing checks in `ExecutionLib.decodeUserOpCallData()`

Context: ExecutionLib.sol#L17-L31

Description: The code in `ExecutionLib.decodeUserOpCallData()` is as shown:

```
assembly {
    let baseOffset := add(userOpCallData.offset, 0x24) //skip 4 bytes of selector and
32 bytes of execution mode
    let calldataLoadOffset := calldataload(baseOffset)
    // check for potential overflow in calldataLoadOffset
    if gt(calldataLoadOffset, 0xffffffffffffffff) { revert(0, 0) }
    erc7579ExecutionCalldata.offset := add(baseOffset, calldataLoadOffset)
    erc7579ExecutionCalldata.length :=
calldataload(sub(erc7579ExecutionCalldata.offset, 0x20))

    let calldataBound := add(userOpCallData.offset, userOpCallData.length)
    // revert if erc7579ExecutionCalldata starts after userOp finishes and if
erc7579ExecutionCalldata ends
    // after userOp finishes
    if gt(erc7579ExecutionCalldata.offset, calldataBound) { revert(0, 0) }
    if gt(add(erc7579ExecutionCalldata.offset, erc7579ExecutionCalldata.length),
calldataBound) { revert(0, 0) }
}
```

In comparison, for a function with `bytes32 mode` and `bytes calldata data` as parameters, `solc` extracts `data` as such (written in psuedo-code):

```
headStart = userOpCallData.offset + 4
dataEnd = userOpCallData.offset + userOpCalldata.length

if (dataEnd - headStart < 64) revert

calldataLoadOffset = calldataload(headStart + 32)
if (calldataLoadOffset > 0xffffffffffffffff) revert

lengthOffset = headStart + calldataLoadOffset
if (lengthOffset + 32 > dataEnd) revert

length = calldataload(lengthOffset)
if (length > 0xffffffffffffffff) revert

dataOffset = lengthOffset + 32
if (dataOffset + length > dataEnd) revert
```

When the code generated by `solc` is compared to `decodeUserOpCallData()`, there are two checks are missing:

1. `dataEnd - headStart < 64` - This check should be added as it's possible that the length of `userOpCallData` is less than intended. Should this occur, the function might end up decoding data from the subsequent fields in `PackedUserOperation`.

2. `length > 0xffffffffffffffff` - This check is needed, otherwise `add(erc7579ExecutionCalldata.offset, erc7579ExecutionCalldata.length)` could overflow and incorrectly pass the bounds check at the end of the function.

Recommendation: Consider adding the following checks to `decodeUserOpCallData()`:

```
  assembly {
+     if lt(userOpCallData.length, 68) { revert(0, 0) }

      let baseOffset := add(userOpCallData.offset, 0x24) //skip 4 bytes of selector
and 32 bytes of execution mode
      let calldataLoadOffset := calldataload(baseOffset)
      // check for potential overflow in calldataLoadOffset
      if gt(calldataLoadOffset, 0xffffffffffffffff) { revert(0, 0) }
      erc7579ExecutionCalldata.offset := add(baseOffset, calldataLoadOffset)
      erc7579ExecutionCalldata.length :=
calldataload(sub(erc7579ExecutionCalldata.offset, 0x20))
+     if gt(erc7579ExecutionCalldata.length, 0xffffffffffffffff) { revert(0, 0) }

      let calldataBound := add(userOpCallData.offset, userOpCallData.length)
      // revert if erc7579ExecutionCalldata starts after userOp finishes and if
erc7579ExecutionCalldata ends
      // after userOp finishes
      if gt(erc7579ExecutionCalldata.offset, calldataBound) { revert(0, 0) }
      if gt(add(erc7579ExecutionCalldata.offset, erc7579ExecutionCalldata.length),
calldataBound) { revert(0, 0) }
  }
```

Alternatively, the function could be refactored to implement the psuedo-code from `solc` in yul.

Rhinestone: Fixed in PR 156.

Renascence: Verified, the recommendation was implemented.

[M-2] `ExecutionLib.decodeBatch()` does not revert when `executionData` is malformed if `pointers.length = 0`

Context: ExecutionLib.sol#L57-L87

Description: The code in `ExecutionLib.decodeBatch()` is as shown:

```
let u := calldataload(executionData.offset)
if or(shr(64, u), gt(0x20, executionData.length)) {
    mstore(0x00, 0xba597e7e) // `DecodingError()`.
    revert(0x1c, 0x04)
}
pointers.offset := add(add(executionData.offset, u), 0x20)
pointers.length := calldataload(add(executionData.offset, u))
if pointers.length {
    // Some logic and checks here...
}
```

As seen from above, there is no check that `pointers.offset` is within the bounds of `executionData` (ie. `pointers.offset < executionData.offset + executionData.length`).

When `pointers.length` is non-zero, this is not an issue as the `if` block contains checks that implicitly ensures `pointers.offset` is within `executionData`.

However, when `pointers.length == 0`, nothing ensures that `pointers.offset` is within `executionData`. This means that `pointers.length` can be stored in any arbitrary offset, which becomes problematic if calldata contains other data after `executionData`.

For example, assume a function has two parameters:

1. `bytes calldata executionData`
2. `bytes calldata garbage`

The function is called with the following arguments:

1. `abi.encodePacked(uint256(0x40))`
2. `abi.encodePacked(bytes32(0))`

The function's calldata would be as follows (excluding the function selector):

```
0000000000000000000000000000000000000000000000000000000000000040 (offset of
executionData)
0000000000000000000000000000000000000000000000000000000000000080 (offset of garbage)

executionData:
0000000000000000000000000000000000000000000000000000000000000020
(executionData.length)
0000000000000000000000000000000000000000000000000000000000000040 (executionData)

garbage:
0000000000000000000000000000000000000000000000000000000000000020 (garbage.length)
0000000000000000000000000000000000000000000000000000000000000000 (garbage)
```

If `executionData` was passed to `decodeBatch()`:

```
executionData.offset = 0x60
u = calldataload(0x60) = 0x40
pointers.offset = 0x60 + 0x40 + 0x20 = 0xc0
pointers.length = calldataload(0x60 + 0x40) = calldataload(0xa0) = 0
```

Since `pointers.length = 0`, the `if` block is skipped and the function returns. However, the function should revert instead as `executionData` is malformed - `pointers.length` was read from the data in `garbage`, which is outside of `executionData`.

The following PoC demonstrates in the example above, `decodeBatch()` does not revert whereas `abi.decode()` does:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import "forge-std/Test.sol";
import {Execution, ExecutionLib} from "contracts/lib/ExecutionLib.sol";
```

```solidity
contract ExecutionLibTest is Test {
    struct S {
        bytes executionData;
        bytes garbage;
    }

    function test_decodeBatch() public {
        /*
        Calldata is as follows when S is passed to a function:

        9988592b (function selector)
        0000000000000000000000000000000000000000000000000000000000000020 (offset of s)
        0000000000000000000000000000000000000000000000000000000000000040 (offset of
s.executionData)
        0000000000000000000000000000000000000000000000000000000000000080 (offset of
s.garbage)
        0000000000000000000000000000000000000000000000000000000000000020
(s.executionData.length)
        0000000000000000000000000000000000000000000000000000000000000040
(s.executionData)
        0000000000000000000000000000000000000000000000000000000000000020
(s.garbage.length)
        0000000000000000000000000000000000000000000000000000000000000000 (s.garbage)
        */
        S memory s = S({
            executionData: abi.encode(uint256(0x40)),
            garbage: abi.encode(uint256(0))
        });

        // ExecutionLib.decodeBatch() doesn't revert
        this.decodeBatch(s);

        // abi.decode() reverts
        vm.expectRevert();
        this.abi_decodeBatch(s);
    }

    function decodeBatch(S calldata s) external {
        Execution[] calldata pointers = ExecutionLib.decodeBatch(s.executionData);
        assertEq(pointers.length, 0);
    }

    function abi_decodeBatch(S calldata s) external {
        Execution[] memory pointers = abi.decode(s.executionData, (Execution[]));
    }
}
```

Recommendation: When `pointers.length == 0`, check that `executionData.offset + u + 32` does not exceed the bounds of `executionData`.

Rhinestone: Fixed in PR 156.

Renascence: Verified, `decodeBatch()` has been fixed alongside Solady's `LibERC7579.decodeBatch()`. The function now ensures `executionData.offset + u + 32 + pointers.length * 32` does not exceed the bounds of `executionData`.

Informational

[I-1] Session digest does not change when called with `USE` or `ENABLE` modes

Context:

- HashLib.sol#L180-L195
- HashLib.sol#L26-L60

Description: The `HashLib._sessionDigest()` function is used to compute the digest of a session alongside other data (ie. `account`, `nonce`, `SmartSession` address, `mode`). The session digest is computed as such:

```
    _hash = keccak256(
        abi.encode(
            SESSION_TYPEHASH,
            account,
            hashPermissions({
                session: session,
                ignoreSecurityAttestations: mode == SmartSessionMode.UNSAFE_ENABLE
            }),
            address(session.sessionValidator),
            keccak256(session.sessionValidatorInitData),
            session.salt,
            smartSession,
            nonce
        )
    );
}
```

As seen from above, `mode` is no longer encoded in the session digest. Instead, the session digest now contains a `ignoreSecurityAttestations` boolean, which specifies if `mode` is equal to `SmartSessionMode.UNSAFE_ENABLE`. This can also be inferred by looking at the `SESSION_TYPEHASH`.

This allows `mode` to be changed from `SmartSessionMode.USE` to `SmartSessionMode.ENABLE` (and vice versa) without altering the session digest (ie. calling `_sessionDigest()` with `mode = USE` and `mode = ENABLE` will result in the same digest, assuming all other parameters remain the same).

However, this should not be an issue in the current codebase as `_sessionDigest()` is never called with `mode = USE`.

Rhinestone: Acknowledged.

Renascence: Acknowledged.