Rhinestone (Smart Sessions) Audit Report

Version 1.0

Audited by:

MiloTruck

bytes032

November 9, 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 Action Policy (enforcing limitations that apply to a specific action a UserOp executes).

Session Validators verify the signature issued by the signer of the ERC-4337 User Operation 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	Smart Sessions
Repository	smartsessions
Commit Hash	b17010a5e18a...
Mitigation Hash	1dadde95dd7c...
Date	8 October 2024 - 15 October 2024

2.3 Issues Found

Severity	**Count**
High Risk	2
Medium Risk	3
Low Risk	3
Informational	2
Total Issues	**10**

3 Findings Summary

ID	Description	Status
H-1	Enabling action policies is susceptible to reentrancy	Resolved
H-2	Fallback action policies can be used by sessions to escalate their privileges	Resolved
M-1	`SmartSession._erc1271IsValidSignatureNowCalldata()` wrongly returns `false` when no ERC-1271 policies are added	Resolved
M-2	Reentrancy risk while enabling policies could corrupt session storage	Resolved
M-3	Incorrect implementation in `AssociatedArrayLib` violates EIP-7562 associated storage rules	Resolved
L-1	Smart sessions are incompatible with enable mode in Biconomy's Nexus accounts	Resolved
L-2	ERC-165 checks in `ConfigLib.requirePolicyType()` might incorrectly pass	Resolved
L-3	`FlatBytesLib.store()` wrongly overwrites an extra storage slot when storing 320 bytes of data	Resolved
I-1	Having too many policies could DOS due to OOG	Resolved
I-2	Code improvements	Resolved

4 Findings

High Risk

[H-1] Enabling action policies is susceptible to reentrancy

Context:

- ConfigLib.sol#L128-L139
- ConfigLib.sol#L81-L87

Description: When enabling actions through `ConfigLib.enable()`, `actionId` is added to `enableActionIds` before each action policy is enabled and added to the `actionPolicies` mapping:

```
// Record the enabled action ID
$self.enabledActionIds[permissionId].add(smartAccount, ActionId.unwrap(actionId));

// Record the enabled action ID
$self.actionPolicies[actionId].enable({
    policyType: PolicyType.ACTION,
    permissionId: permissionId,
    configId: permissionId.toConfigId(actionId),
    policyDatas: actionPolicyData.actionPolicies,
    smartAccount: smartAccount,
    useRegistry: useRegistry
});
```

However, this makes enabling actions susceptible to reentrancy as calling `enable()` for policies has an external call to the policy:

```
// Initialize the policy with the provided configuration
// overwrites the config
IPolicy(policy).initializeWithMultiplexer({
    account: smartAccount,
    configId: configId,
    initData: policyDatas[i].initData
});
```

If an untrusted contract gains control flow in the call to `initializeWithMultiplexer()`, they could abuse this to (1) bypass action policy validation or (2) corrupt action policy storage in the contract.

An example of 1:

- `ConfigLib.enable()` is called to add an `actionId` with two policies, referred to as A and B:
  - `actionId` is added to `enabledActionIds`.
  - `enable()` is called for policy A, which adds policy A to the `actionPolicies` mapping.
  - `initializeWithMultiplexer()` is called for policy A, which gives control flow to an untrusted contract.
  - Now, if `validateUserOp()` is called for the account with calldata for the same `actionId`, only policy A is validated. This is because policy B has not been added yet.

- Therefore, policy B is bypassed.

An example of 2:

- `ConfigLib.enable()` is called to add an `actionId` with two policies, referred to as A and B:
  - `actionId` is added to `enabledActionIds()`.
  - `enable()` is called for policy A, which adds policy A to the `actionPolicies` mapping.
  - `initializeWithMultiplexer()` is called for policy A, which gives control flow to an untrusted contract.
  - The untrusted contract calls `disableActionId()` for the same `actionId`:
    * `actionId` is removed from `enabledActionIds`.
    * Policy A is removed from the `actionPolicies` mapping.
  - `enable()` is called for policy B, which adds policy B to the `actionPolicies` mapping.
- Therefore, policy B is in the `actionPolicies` mapping although `actionId` is not enabled.

Recommendation: Add `actionId` to `enabledActionIds` after all action policies have been enabled:

```
- // Record the enabled action ID
- $self.enabledActionIds[permissionId].add(smartAccount, ActionId.unwrap(actionId));

  // Record the enabled action ID
  $self.actionPolicies[actionId].enable({
      // ..
  });

+ // Record the enabled action ID
+ $self.enabledActionIds[permissionId].add(smartAccount, ActionId.unwrap(actionId));
```

Rhinestone: Fixed in PR #126.

Renascence: Verified, the recommended fix was implemented.

[H-2] Fallback action policies can be used by sessions to escalate their privileges

Context:

- ConfigLib.sol#L120-L124
- PolicyLib.sol#L201-L207
- PolicyLib.sol#L221-L234

Description: When enabling action policies, `ConfigLib.enable()` checks that the target of an action (ie. `actionTarget`) is not the `SmartSession` contract:

```
// disallow actions to be set for address(0) or to the smartsession module itself
// sessionkeys that have access to smartsessions, may use this access to elevate
their privileges
if (actionPolicyData.actionTarget == address(0) || actionPolicyData.actionTarget ==
address(this)) {
    revert ISmartSession.InvalidActionId();
}
```

This is meant to prevent sessions from escalating their own privileges. If an action policy could call the `SmartSession` contract, it could enable a new session with any privileges on behalf of the account.

However, when enforcing action policies, `PolicyLib.checkSingle7579Exec()` does not check that the `target` address is not the `SmartSession` contract:

```
// Prevent potential bypass of policy checks through nested self executions
if (targetSig == IERC7579Account.execute.selector && target == userOp.sender) {
    revert ISmartSession.InvalidSelfCall();
}

// Prevent fallback action from being used directly
if (target == FALLBACK_TARGET_FLAG) revert ISmartSession.InvalidTarget();
```

This allows an action with `target == address(this)` to pass validation through fallback action policies:

```
// If tryCheck returns RETRY_WITH_FALLBACK magic value, that means not enough
policies were configured
// for the actionId. Proceed with checking fallback action policies
($policies[FALLBACK_ACTIONID]).
if (vd == RETRY_WITH_FALLBACK) {
    // If no policies were configured for FALLBACK_ACTIONID for this PermissionId,
this will revert
    vd = $policies[FALLBACK_ACTIONID].check({
        userOp: userOp,
        permissionId: permissionId,
        callOnIPolicy: abi.encodeCall(
            IActionPolicy.checkAction,
            (permissionId.toConfigId(FALLBACK_ACTIONID), userOp.sender, target,
value, callData)
        ),
        minPolicies: minPolicies
    });
}
```

When `checkSingle7579Exec()` is called with `target == address(this)`, since it is not possible to add an action policy with `address(this)` as the target address, `tryCheck()` will always return `RETRY_WITH_FALLBACK`. Subsequently, if all fallback action policies pass for `target == address(this)`, it becomes possible for a session to call the `SmartSession` contract on behalf of the account.

Recommendation: Consider checking if `target` is the `SmartSession` contract address, and reverting if so:

```
  // Prevent fallback action from being used directly
- if (target == FALLBACK_TARGET_FLAG) revert ISmartSession.InvalidTarget();
+ if (target == FALLBACK_TARGET_FLAG || target == address(this)) revert
ISmartSession.InvalidTarget();
```

Rhinestone: Fixed in PR #127.

Renascence: Verified, the recommended fix was implemented.

Medium Risk

[M-1] `SmartSession._erc1271IsValidSignatureNowCalldata()` wrongly returns `false` when no ERC-1271 policies are added

Context:

- SmartSession.sol#L432-L444
- PolicyLib.sol#L318-L333

Description: In `SmartSession._erc1271IsValidSignatureNowCalldata()`, to check if a signature and hash is allowed according to ERC-1271 policies, the function calls `PolicyLib.checkERC1271()`:

```
// check the ERC-1271 policy
bool valid = $erc1271Policies.checkERC1271({
    // ...
    minPoliciesToEnforce: 0
});

// if the erc1271 policy check failed, return false
if (!valid) return false;
```

As seen from above, if `checkERC1271()` returns `valid = false`, the signature is invalid and `false` will be returned from `_erc1271IsValidSignatureNowCalldata()`.

`checkERC1271()` loops over all ERC-1271 policies added for `permissionId` and `account` to check that `check1271SignedAction()` never returns `false`. Note that `valid` is returned by the function and is initialized as `false` by default before the logic shown below:

```
address[] memory policies = $self.policyList[permissionId].values({ account: account
});
uint256 length = policies.length;
if (minPoliciesToEnforce > length) revert ISmartSession.NoPoliciesSet(permissionId);

// iterate over all policies and intersect the validation data
for (uint256 i; i < length; i++) {
    valid = I1271Policy(policies[i]).check1271SignedAction({
        // ..
    });
    // If any policy check fails, return false immediately
    if (!valid) return false;
}
```

If there are no ERC-1271 policies added for `permissionId` and `account` (ie. `policies.length = 0`), the for-loop will be skipped and `valid` will be returned as `false`.

As a result, the signature is considered invalid and `_erc1271IsValidSignatureNowCalldata()` returns `false`. This is incorrect as `checkERC1271()` is called with `minPoliciesToEnforce = 0`, which means that no policies are needed for a signature to be considered valid.

Recommendation: Set `valid` to `true` before iterating over all ERC-1271 policies:

```
+ valid = true;
  // iterate over all policies and intersect the validation data
  for (uint256 i; i < length; i++) {
      valid = I1271Policy(policies[i]).check1271SignedAction({
          // ..
      });
      // If any policy check fails, return false immediately
      if (!valid) return false;
  }
```

This ensures the signature will be considered valid if `length` is `0`.

Rhinestone: Fixed in PR #130. We decided that at least one ERC-1271 policy has to be added for ERC-1271 verification to pass.

Renascence: Verified, the case where `policies.length = 0` can no longer occur as `checkERC1271()` is now called with `minPolicyToCheck = 1`.

[M-2] Reentrancy risk while enabling policies could corrupt session storage

Context:

- ConfigLib.sol#L66-L87
- SmartSessionBase.sol#L48-L63

Description: In the protocol, policies are enabled with `ConfigLib.enable()`, which iterates over all policy addresses as such:

```
// iterage over all policyData
uint256 lengthConfigs = policyDatas.length;
for (uint256 i; i < lengthConfigs; i++) {
    address policy = policyDatas[i].policy;

    // ...

    // Add the policy to the list for the given permission and smart account
    $policy.policyList[permissionId].add({ account: smartAccount, value: policy });

    // Initialize the policy with the provided configuration
    // overwrites the config
    IPolicy(policy).initializeWithMultiplexer({
        account: smartAccount,
        configId: configId,
        initData: policyDatas[i].initData
    });
```

As seen from above, each policy is added to the `policyList`, followed by an external `initializeWithMultiplexer()` call to the added policy address.

However, such an implementation is susceptible to reentrancy since the external call to `initializeWithMultiplexer()` could give control flow to an untrusted contract, or the policy could be malicious. This could be used to corrupt the session's storage.

Using `enableUserOpPolicies()` as an example, as seen below, `enable()` is called after `permissionId` is checked to be in `enabledSessions`:

```
function enableUserOpPolicies(PermissionId permissionId, PolicyData[] memory
userOpPolicies) public {
    // Check if the session is enabled for the caller and the given permission
    if ($enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId)) ==
false) {
        revert InvalidSession(permissionId);
    }

    // Enable the specified user operation policies
    $userOpPolicies.enable({
        // ...
    });
}
```

Assume `enableUserOpPolicies()` is called to add two `userOp` policies, referred to as A and B:

- `permissionId` is checked to be in `enabledSessions`.
- `ConfigLib.enable()` is called to add policies A and B:
  - In the first iteration of the for-loop:
    - Policy A is added to the `policyList` for `userOp` policies.
    - `initializeWithMultiplexer()` is called for policy A
    - Assume control flow is given to an untrusted contract, which calls `removeSession()` for the same `permissionId`:
      - Policy A is removed from the `policyList`.
      - `permissionId` is removed from `enabledSessions`.
  - In the second iteration of the for-loop:
    - Policy B is added to the `policyList` for `userOp` policies.

Even though `permissionId` is no longer an enabled session, policy B is still added to the `policyList` as an enabled `userOp` policy. If `permissionId` is ever enabled again, policy B will be carried over from the old session and will be enabled without being added again.

Note that this attack can be performed when adding any type of policy.

Recommendation: A possible mitigation would be to check that `permissionId` is still in `enabledSessions` after policies have been enabled:

```
  function enableUserOpPolicies(PermissionId permissionId, PolicyData[] memory
userOpPolicies) public {
      // Check if the session is enabled for the caller and the given permission
      if ($enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId)) ==
false) {
          revert InvalidSession(permissionId);
      }

      // Enable the specified user operation policies
      $userOpPolicies.enable({
          // ...
      });

+     if ($enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId)) ==
false) {
+         revert InvalidSession(permissionId);
+     }
  }
```

This ensures that a policy did not disable the session during initialization.

Rhinestone: Fixed in PR #124.

Renascence: Verified, a `enabledSessions.contains()` check was added at the end of all functions that enable policies.

[M-3] Incorrect implementation in `AssociatedArrayLib` violates EIP-7562 associated storage rules

Context:

- AssociatedArrayLib.sol#L77-L87

Description: In `AssociatedArrayLib`, elements in the array are stored at multiples of `0x20`:

```
function _set(bytes32 slot, uint256 index, bytes32 value) private {
    assembly {
        //if (index >= _length(s, account)) revert AssociatedArray_OutOfBounds(index);
        if iszero(lt(index, sload(slot))) {
            mstore(0, 0x8277484f) // `AssociatedArray_OutOfBounds(uint256)`
            mstore(0x20, index)
            revert(0x1c, 0x24)
        }
        sstore(add(slot, mul(0x20, add(index, 1))), value)
    }
}
```

According to associated storage rules in EIP-7562, storage slots can only be `keccak(A || x) + n`, where `n` is within the range `[0, 128]`:

> Associated storage: a storage slot of any smart contract is considered to be "associated" with address `A` if:
>
> 1. The slot value is `A`

2. The slot value was calculated as `keccak(A||x)+n`, where `x` is a `bytes32` value, and `n` is a value in the range 0..128

Assumine that `_slot()` returns `p`, where `p = keccak256(account || s.slot)`. In the current implementation, elements in the array are stored at slot `p + (index + 1) * 32` instead of `p + (index + 1)`, which will easily exceed the `[0, 128]` limit from associated storage rules. For example, an element at index 5 in the array would be stored at slot `keccak(A || x) + 160`, which violates this rule.

Storage slots were most likely mixed up with memory offsets - `sload`/`sstore` uses slots and each slot has 32 bytes in storage. Instead of having array elements next to each other in storage as so:

```
[p]: array length
[p+1]: element 1
[p+2]: element 2
```

The current implementation has 31 empty slots in between each element as such:

```
[p]: array length
[p+1]: empty
...
[p+31]: empty
[p+32]: element 1
[p+33]: empty
...
[p+63]: empty
[p+64]: element 2
```

It is also worth noting that even if elements were stored at `p + (index + 1)`, an array with more than 128 elements would exceed the `[0, 128]` range allowed by associated storage rules. This is possible if a session has more than 128 policies added for a single type of policy (eg. more than 128 `userOp` policies).

Recommendation: Consider storing elements in the array at slot `p + index + 1` instead:

```
function _set(bytes32 slot, uint256 index, bytes32 value) private {
    assembly {
        //if (index >= _length(s, account)) revert AssociatedArray_OutOfBounds(index);
        if iszero(lt(index, sload(slot))) {
            mstore(0, 0x8277484f) // `AssociatedArray_OutOfBounds(uint256)`
            mstore(0x20, index)
            revert(0x1c, 0x24)
        }
-       sstore(add(slot, mul(0x20, add(index, 1))), value)
+       sstore(add(slot, add(index, 1)), value)
    }
}
```

Note that this has change has to be made for all functions that access array elements, namely `_set()`, `_get()`, `_push()` and `_remove()`.

Additionally, to avoid violating associated storage rules, consider enforcing that only a maximum number of 128 elements can be added to the array. Trying to add more elements to the array should revert.

Rhinestone: Fixed in PR #129.

Renascence: Verified, the recommended fix was implemented.

Low Risk

[L-1] Smart sessions are incompatible with enable mode in Biconomys Nexus accounts

Context:

- SmartSessionBase.sol#L220-L228
- SmartSessionBase.sol#L325

Description: When policies are added in `SmartSessionbase.enableSessions()`, they are first checked against the registry according to EIP-7484. As an example, `userOp` policies are enabled with the `useRegistry` parameter specified as `true`:

```
// Enable UserOp policies
$userOpPolicies.enable({
    policyType: PolicyType.USER_OP,
    permissionId: permissionId,
    configId: permissionId.toUserOpPolicyId().toConfigId(),
    policyDatas: session.userOpPolicies,
    smartAccount: msg.sender,
    useRegistry: true
});
```

Checking the registry violates the associated storage rules specified in EIP-7562, as such, `enableSession()` is not allowed be called in `validateUserOp()`. Since `onInstall()` directly calls `enableSessions()`, it cannot be called in `validateUserOp()` as well.

However, when using enable mode in Biconomy's Nexus accounts, `onInstall()` is called to install validator modules when they are used for the first time in `validateUserOp()`. As such, smart sessions are incompatible with enable mode in Nexus accounts - any `userOp` that uses enable mode with smart sessions have to be sent to alternate mempools.

Recommendation: Similar to unsafe enable mode in `SmartSession.validateUserOp()`, consider allowing users to specify if the registry should be used in `enableSessions()` and `onInstall()`. This can be done by encoding an additional `useRegistry` parameter to the `data` passed to `onInstall()`, which specifies if the registry should be used.

Otherwise, document this limitation.

Rhinestone: Fixed in commit d9ad2cd.

Renascence: Verified, `onInstall()` now allows the user to specify if the registry should be checked when adding policies.

[L-2] ERC-165 checks in `ConfigLib.requirePolicyType()` might incorrectly pass

Context:

- ConfigLib.sol#L22-L38

Description: In `ConfigLib.requirePolicyType()`, to check if a policy supports an interface, the function directly calls `supportsInterface()` and reverts if `false` is returned:

```
function requirePolicyType(address policy, PolicyType policyType) internal view {
    bool supportsInterface;
    if (policyType == PolicyType.USER_OP) {
        supportsInterface =
IPolicy(policy).supportsInterface(type(IUserOpPolicy).interfaceId);
    } else if (policyType == PolicyType.ACTION) {
        supportsInterface =
IPolicy(policy).supportsInterface(type(IActionPolicy).interfaceId);
    } else if (policyType == PolicyType.ERC1271) {
        supportsInterface =
IPolicy(policy).supportsInterface(type(I1271Policy).interfaceId);
    } else {
        revert UnsupportedPolicy(policy);
    }

    // Revert if the policy does not support the required interface
    if (!supportsInterface) {
        revert UnsupportedPolicy(policy);
    }
}
```

This does not followed the recommended approach for checking if an interface is implemented, as outlined by ERC-165 below:

> **How to Detect if a Contract Implements ERC-165**
>
> 1. The source contract makes a `STATICCALL` to the destination address with input data: `0x01ffc9a701ffc9a700000000000000000000000000000000000000000000000000000000` and gas 30,000. This corresponds to `contract.supportsInterface(0x01ffc9a7)`.
> 2. If the call fails or return false, the destination contract does not implement ERC-165.
> 3. If the call returns true, a second call is made with input data `0x01ffc9a7ffffffff00000000000000000000000000000000000000000000000000000000`.
> 4. If the second call fails or returns true, the destination contract does not implement ERC-165.
> 5. Otherwise it implements ERC-165.
>
> **How to Detect if a Contract Implements any Given Interface**
>
> 1. If you are not sure if the contract implements ERC-165, use the above procedure to confirm.
> 2. If it does not implement ERC-165, then you will have to see what methods it uses the old-fashioned way.
> 3. If it implements ERC-165 then just call `supportsInterface(interfaceID)` to determine if it implements an interface you can use.

With the current implementation, `supportsInterface()` could incorrectly return `true`. For example, the policy could contain a function with a selector that clashes with `0x01ffc9a` and always returns `true`.

Recommendation: Consider using OpenZeppelin's `ERC165Checker` library to check if interfaces are supported.

Rhinestone: Fixed in PR #128.

Renascence: Verified, the recommended fix was implemented.

[L-3] `FlatBytesLib.store()` wrongly overwrites an extra storage slot when storing 320 bytes of data

Context:

- BytesLib.sol#L19-L22
- BytesLib.sol#L100-L117
- BytesLib.sol#L43-L54

Description: In `FlatBytesLib`, data is stored as an array `bytes32` with length 10:

```
// Data structure to store bytes in consecutive slots using an array
struct Data {
    bytes32[10] slot1;
}
```

Therefore, the maximum number of bytes that can be stored in `Data` is 320. When `toArray()` is used to convert data from `bytes` to `bytes32[10]`, it correctly checks that the data length is not more than 320:

```
// Find 32 bytes segments nb
totalLength = data.length;
if (totalLength > 32 * 10) revert();
uint256 dataNb = totalLength / 32 + 1;
```

Afterwards, the number of `bytes32` needed to store the data is calculated as `totalLength / 32 + 1`. The number of `bytes32` needed is increased by `1` to account for Solidity rounding down.

However, when `totalLength` is exactly divisible by 32, adding `1` causes `dataNb` to be larger than it should be. For example, data that is 320 bytes long can fit into 10 `bytes32`, but `dataNb` will be calculated as `11`.

This causes an extra storage slot to be written to when `store()` is called when `data.length` is exactly divisible by 32:

```
(self.totalLength, entries) = data.toArray();

uint256 length = entries.length;

Data storage _data = self.data;

for (uint256 i; i < length; i++) {
    bytes32 value = entries[i];
    assembly {
        sstore(add(_data.slot, i), value)
    }
}
```

More importantly, when `data.length = 320`, `store()` will overwrite 11 slots, which is 1 more than the 10 storage slots belonging to the `Data` struct. As a result, when `store()` is called, it will incorrectly overwrite the next storage slot after the `Bytes` struct, which could corrupt storage.

The following PoC demonstrates how `store()` overwrites the storage slot after `Bytes`:

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

import "forge-std/Test.sol";
import { FlatBytesLib } from "flatbytes/BytesLib.sol";

contract FlatBytesTest is Test {
    using FlatBytesLib for FlatBytesLib.Bytes;

    FlatBytesLib.Bytes b;
    uint256 u;

    function test_flatBytesLibOverflow() public {
        u = 1337;
        b.store(new bytes(320));
        assertEq(u, 11);
    }
}
```

Recommendation: Calculate the number of `bytes32` needed to store data as such:

```
  // Find 32 bytes segments nb
  totalLength = data.length;
  if (totalLength > 32 * 10) revert();
- uint256 dataNb = totalLength / 32 + 1;
+ uint256 dataNb = (totalLength + 31) / 32;
```

This rounds `totalLength / 32` up to the nearest whole number. Note that the change has to be implemented in both `toArray()` and `toBytes()`.

Rhinestone: Fixed in PR #3.

Renascence: Verified, the recommendation was implemented. The number of slots in the `Data` struct was also changed to 32.

Informational

[I-1] Having too many policies could DOS due to OOG

Context:

- SmartSessionBase.sol
- PolicyLib

Description:

```
    function removeSession(PermissionId permissionId) public {
        if (permissionId == EMPTY_PERMISSIONID) revert InvalidSession(permissionId);
>         $userOpPolicies.policyList[permissionId].removeAll(msg.sender);
>         $erc1271Policies.policyList[permissionId].removeAll(msg.sender);

        // Remove all Action policies for this session
        uint256 actionLength =
$actionPolicies.enabledActionIds[permissionId].length(msg.sender);
        for (uint256 i; i < actionLength; i++) {
            ActionId actionId =
ActionId.wrap($actionPolicies.enabledActionIds[permissionId].at(msg.sender, i));
>             $actionPolicies.actionPolicies[actionId].policyList[permissionId].remove
All(msg.sender);
        }

>         $actionPolicies.enabledActionIds[permissionId].removeAll(msg.sender);
```

```
    function disableActionId(PermissionId permissionId, ActionId actionId) public {
        if ($enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId))
== false) {
            revert InvalidSession(permissionId);
        }

>         $actionPolicies.actionPolicies[actionId].policyList[permissionId].removeAll(
msg.sender);
```

```
    function check(
        Policy storage $self,
        PackedUserOperation calldata userOp,
        PermissionId permissionId,
        bytes memory callOnIPolicy,
        uint256 minPolicies
    )
        internal
        returns (ValidationData vd)
    {
        // @audit-issue-reported unbounded cost
>         address[] memory policies = $self.policyList[permissionId].values({ account:
userOp.sender });
        uint256 length = policies.length;
```

Recommendation: You could set a limit on the maximum of policies. Specifically, 128.

Rhinestone: Fixed in PR #129.

Renascence: Verified, a check was added to `_push()` to ensure that only a maximum of 128 elements can be added to the array.

[I-2] Code improvements

Description:

1. The code below will emit an event regardless of the result. Might be a good idea to only emit on success.

```
for (uint256 i; i < length; i++) {
    address policy = policies[i];
    $policy.policyList[permissionId].remove(smartAccount, policy);
    emit ISmartSession.PolicyDisabled(permissionId, policyType, address(policy),
smartAccount);
```

2. This check exists in multiple places. Might be a good idea to extract it into a modifier.

```
if ($enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId)) ==
false) {
    revert InvalidSession(permissionId);
}
```

3. The `PERSONAL_SIGN_TYPEHASH` variable is unused. It can be removed

```
bytes32 internal constant _PERSONAL_SIGN_TYPEHASH =
    0x983e65e5148e570cd828ead231ee759a8d7958721a768f93bc4483ba005c32de;
```

4. To be consistent with `SmartSession._enablePolicies()`, consider checking if the session validator is set before calling `enable()`:

```
$sessionValidators.enable({
    permissionId: permissionId,
    smartAccount: msg.sender,
    sessionValidator: session.sessionValidator,
    sessionValidatorConfig: session.sessionValidatorInitData,
    useRegistry: true
});
```

5. There's a missing return keyword here:

```
    function getActionPolicies(address account, PermissionId permissionId, ActionId
actionId) external view returns (address[] memory) {
>
$actionPolicies.actionPolicies[actionId].policyList[permissionId].values(account);
    }
```

Rhinestone: Fixed in PR #131.

Renascence: Verified, the recommended fixes were implemented.