Rhinestone+Biconomy: Smartsessions Core

Security Review

Cantina Managed review by:

Riley Holterhus, Lead Security Researcher

Blockdev, Security Researcher

Chinmay Farkya, Associate Security Researcher

October 8, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Rhinestone is the leading ERC-7579 modular smart account infrastructure platform. We provide tools and services that help developers build, deploy, and manage the smart account modules that will power the next wave of powerful onchain products with seamless UX.

From Sep 3rd to Sep 10th the Cantina team conducted a review of rhinestone-smartsessions on commit hash a6fc609c. In total, the team identified **44** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 7
- Medium Risk: 7
- Low Risk: 12
- Gas Optimizations: 2
- Informational: 16

This report is one of two from the review, focusing specifically on findings related to the core codebase rather than the external policy contracts.

3 Findings

3.1 High Risk

3.1.1 `ValidationData` not always intersected properly

Severity: High Risk

Context: PolicyLib.sol#L73-L79

Description: Throughout the codebase, there are calls to multiple policy contracts, and the `ValidationData` return values from these calls are intended to be combined using the `intersectValidationData()` function. However, there are currently two locations in the codebase where the `intersectValidationData()` function is not properly called, and previous `ValidationData` results are overwritten instead of combined. These two locations are:

1. In `PolicyLib.sol`, the `check()` function combines each policy's return value with itself instead of accumulating across the entire loop:

```
function check(/* ... */) internal returns (ValidationData vd) {
    // ....
    for (uint256 i; i < length; i++) {
        // ...
        uint256 validationDataFromPolicy = uint256(bytes32(policies[i].safeCall({ callData:
↪  callOnIPolicy })));
        vd = ValidationData.wrap(validationDataFromPolicy);
        // ...
        if (vd.isFailed()) revert ISmartSession.PolicyViolation(permissionId, policies[i]);
        // ...
        vd = vd.intersectValidationData(vd);
    }
}
```

2. In `SmartSession.sol`, the `_enforcePolicies()` function overwrites the userOp policy return value with the action policy return value:

```
function _enforcePolicies(/* ... */) internal returns (ValidationData vd) {
    // ...
    vd = $userOpPolicies.check({/* ... */});
    // ...
    if (selector == IERC7579Account.execute.selector) {
        // ...
        else if (callType == CALLTYPE_BATCH) {
            vd = $actionPolicies.actionPolicies.checkBatch7579Exec({/* ... */});
        }
        // DEFAULT EXEC & SINGLE CALL
        else if (callType == CALLTYPE_SINGLE) {
            // ...
            vd = $actionPolicies.actionPolicies.checkSingle7579Exec({/* ... */});
        }
        // ...
    }
    // ...
    else {
        // ...
        vd = $actionPolicies.actionPolicies[actionId].check({/* ... */});
    }
    // ...
}
```

The consequences of this issue are partially mitigated by the `isFailed()` check, as this helper function reverts if any individual policy fails. However, unexpected behavior could arise if `intersectValidationData()` returns an intermediate error, or if a policy returns a `validAfter` or `validUntil` timestamp, since these values will sometimes be overwritten.

Recommendation: Update the logic in the two locations mentioned above to properly intersect return data instead of overwriting it.

Rhinestone: Fixed in PR 60.

Cantina Managed: Verified.

3.1.2 Enable mode can be frontrun to add policies for a different `permissionId`

Severity: High Risk

Context: SmartSession.sol#L139-L213

Description: `SmartSession` offers an enable mode for when a smart account wants to enable policies within the same call when the related actions get executed. In this case, it first enables the required policies during the validation phase and enforces them as well to allow desired actions during the execution phase of the same call.

`SmartSession::_enablePolicies()` handles this flow. This logic uses a separate enable-signature to validate that the call was authorized from the `userOp.sender`.

The `permissionID` is first parsed from `userOp.signature` in `validateUserOp()`, and then it is further decoded inside `_enablePolicies()` to get `enableData` which has all the info about the session to be enabled.

The problem is that this `permissionID` is not included in the signature digest. Moreover, it is not checked to be equivalent to the `enableData` session information. Even if the `permissionID` encoded into `userOp.signature` is changed, the operation will go on smoothly.

As a result, when a smart account wants to add policies to a session X, they sign a call and send it. A bundler/malicious user sees the call and frontruns it to replace `permissionID` in the packed `userOp.signature`. Since `userOp.signature` is not a part of the `userOphash`, so nothing changes for the signature validation process.

Hence, the operation goes through and the policies that were intended for permission X go through and get installed on `permissionID` Y of the caller's choice.

This exploit has three requirements:

1. The `permissionID` of the malicious caller also needs to be installed in some way for the smart account.
2. The `$signerNonce` for permisisonID Y is the same as the original `permissionID` X.
3. Depends on the `sessionvalidator` implementation. If both are using the same, this exploit can work.

This problem could allow someone who's involved in a less-privileged `permissionID` to frontrun and steal policies that were intended to be given to a more privileged `permissionID`.

Recommendation: Add `permissionID` into the `getAndVerifyDigest()` function so that it is always verified. Furthermore, move the check at Line 183 out of the if clause so that `permisisonID` is always guaranteed to be associated with the `enableData` (which is validated as it is part of the signature).

Rhinestone: Fixed in PR 80.

Cantina Managed: Verified.

3.1.3 `removeSession()` doesn't remove data completely from installed policy modules

Severity: High Risk

Context: SmartSessionBase.sol#L235-L255

Description: When a smart account uninstalls the `SmartSession` module or removes a session, the `removeSession()` function is called which is responsible for deleting all associated data from the policy modules etc. connected with that session.

According to `ERC7579` specs, any uninstallation of a module should not leave remnant storage.

The problem with `removeSession()` is that it forgets to delete two things:

1. `$actionPolicies.enabledActionIds[permissionId]`.
2. `$sessionValidators[permissionId][smartAccount]`.

This breaks compliance with ERC7579 and in some cases could also lead to problems when the same module is re-installed for the smart account.

Recommendation: Add code in `removeSession()` to disable the `enabledActionIDs` and `sessionValidators` associated with the `permissionID` of the session that is being removed.

Rhinestone: Fixed in PR 62.

Cantina Managed: Verified.

3.1.4 ERC-7739 does not use correct `verifyingContract` address

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: ERC-7739 introduces a rehashing scheme designed to prevent signature replay across smart accounts that share signers. To accomplish this, the ERC-7739 logic wraps the hash provided to the ERC-1271 `isValidSignature()` function within a new EIP-712 typed struct. This struct differentiates the hashes verified by each account by using the account's own address as the `verifyingContract`, which ensures that signatures are unique to each account and cannot be replayed.

However, in the current codebase, the `verifyingContract` is set to the address of the `SmartSession` rather than the address of each individual account. This approach defeats the main purpose of using ERC-7739, as it causes all accounts to modify their hashes in the same predictable way. As a result, two smart accounts that share a signer could be vulnerable to signature replay.

Recommendation: Change the `_erc1271IsValidSignatureViaNestedEIP712()` function to use an `eip712Domain()` that uses the smart account's address as the `verifyingContract` instead of the `SmartSession` address. For example:

```
  function _typedDataSignFields() private view returns (bytes32 m) {
      (
          bytes1 fields,
          string memory name,
          string memory version,
          uint256 chainId,
-         address verifyingContract,
+         ,
          bytes32 salt,
          uint256[] memory extensions
      ) = eip712Domain();
+     address verifyingContract = msg.sender;
      /// @solidity memory-safe-assembly
      assembly {
          m := mload(0x40) // Grab the free memory pointer.
          mstore(0x40, add(m, 0x120)) // Allocate the memory.
          // Skip 2 words for the `typedDataSignTypehash` and `contents` struct hash.
          mstore(add(m, 0x40), shl(248, byte(0, fields)))
          mstore(add(m, 0x60), keccak256(add(name, 0x20), mload(name)))
          mstore(add(m, 0x80), keccak256(add(version, 0x20), mload(version)))
          mstore(add(m, 0xa0), chainId)
          mstore(add(m, 0xc0), shr(96, shl(96, verifyingContract)))
          mstore(add(m, 0xe0), salt)
          mstore(add(m, 0x100), keccak256(add(extensions, 0x20), shl(5, mload(extensions))))
      }
  }
```

Rhinestone: Fixed in PR 64 and PR 77.

Cantina Managed: Verified.

3.1.5 Arrays are not properly cleared due to iteration during removal

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In two different parts of the codebase, an array is iterated over while its elements are simultaneously removed. Because each removal will shift the indices of the remaining elements, both locations of this behavior will fail to remove all elements that are intended to be removed.

For example, consider the `_removeAll()` function:

```
function _removeAll(Set storage set, address account) internal {
    uint256 len = _length(set, account);
    for (uint256 i; i < len; i++) {
        _remove(set, account, _at(set, account, i));
    }
}
```

Since each call to `_remove()` will decrease the length of the set and will shuffle the elements inside of it, the second half of this loop will attempt to remove elements at indices that no longer exist.

The other location of this problem is in the `onUninstall()` function, which has the below implementation:

```
function onUninstall(bytes calldata /*data*/ ) external override {
    uint256 configIdsCnt = $enabledSessions.length({ account: msg.sender });

    for (uint256 i; i < configIdsCnt; i++) {
        PermissionId configId = PermissionId.wrap($enabledSessions.at({ account: msg.sender, index: i }));
        removeSession(configId);
    }
}
```

In both cases, arrays are not fully cleared as expected, leaving permissions and other associated values active in storage despite the intent to remove them.

Recommendation: To ensure that all elements are removed, consider changing the implementation of both functions to remove indices in reverse order. This approach would fix the issue, as each loop iteration would pop the last element, and no index shuffling would occur.

Rhinestone: Fixed in PR 65.

Cantina Managed: Verified. The `_removeAll()` function is fixed since it now removes elements in reverse order. The `onUninstall()` function is fixed since it now continuously removes the element at index 0.

3.1.6 Native account functions don't correctly handle `RETRY_WITH_FALLBACK`

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In the current implementation of the `tryCheck()` function, if the minimum policy threshold (`minPolicies`) is not met, the function returns `RETRY_WITH_FALLBACK` to indicate that the system should re-attempt the action policy check using the `FALLBACK_ACTIONID`. This can be seen in the following code snippets:

```
function tryCheck(/* ... */) /* ... */ {
    // ...
    if (minPolicies > length) {
        return RETRY_WITH_FALLBACK;
    }
    // ...
}

function checkSingle7579Exec(/* ... */) /* ... */ {
    // ...
    vd = $policies[actionId].tryCheck({/* ... */});
    // ...
    if (vd == RETRY_WITH_FALLBACK) {
        // If no policies were configured for FALLBACK_ACTIONID for this PermissionId, this will revert
        vd = $policies[FALLBACK_ACTIONID].check({/* ... */});
    }
    return vd;
}
```

This fallback behavior is not properly implemented in the `_enforcePolicies()` function. More specifically, when native account function calls are being verified, the `tryCheck()` function is used but the return value is passed directly into the `intersect()` function:

```
vd = vd.intersect(
    $actionPolicies.actionPolicies[actionId].tryCheck({/* ... */})
);
```

As a result, any native account function that does not have an action policy will directly pass `RETRY_WITH_FALLBACK` into `intersect()`. Since `RETRY_WITH_FALLBACK` is defined as `ValidationData.wrap(uint256(0x50FFBAAD))`, this can lead to unexpected results instead of a failure.

Recommendation: Evaluate whether native account functions should support the `RETRY_WITH_FALLBACK` behavior. If not, consider replacing `tryCheck()` with `check()` in `_enforcePolicies()`. If so, consider adding logic to properly handle the `RETRY_WITH_FALLBACK` return value, similar to `checkSingle7579Exec()`.

Rhinestone: Fixed in PR 112 by replacing `tryCheck()` with `check()`.

Cantina Managed: Verified.

3.1.7 Installation reentrancy concerns

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When a user installs a session, several different storage locations are initialized sequentially. For example, consider the `enableSession()` implementation below:

```
function enableSessions(Session[] calldata sessions) public returns (PermissionId[] memory permissionIds) {
    uint256 length = sessions.length;
    if (length == 0) revert InvalidData();

    permissionIds = new PermissionId[](length);

    for (uint256 i; i < length; i++) {
        Session calldata session = sessions[i];
        PermissionId permissionId = session.toPermissionId();
        // ...
        $enabledSessions.add({ /* ... */});
        // ...
        $sessionValidators.enable({/* ... */});
        // ...
        $userOpPolicies.enable({/* ... */});
        // ...
        $erc1271Policies.enable({/* ... */});
        $enabledERC7739Content.enable(/* ... */);
        // ...
        $actionPolicies.enable({/* ... */});
        // ...
    }
}
```

Note that any of these storage initialization steps may result in an external call, which would grant control flow to another contract. One low-risk example is with the external calls to `registry.checkForAccount()`. A higher-risk example is the `initializeWithMultiplexer()` function, which is invoked on each installed policy. While these calls are generally trusted (as the user chooses the policies they install), even a non-malicious policy could inadvertently give up control flow to an untrusted contract.

If an attacker does gain control flow during an installation, they could potentially exploit the unfinished installation state. For instance, in the above code, notice that `$sessionValidators.enable()` happens before `$erc1271Policies.enable()`. So, if an attacker gains control flow after `$sessionValidators.enable()` but before `$erc1271Policies.enable()`, they may be able to successfully call `isValidSignature()`, bypassing all of the `$erc1271Policies` that are yet to be installed.

Recommendation: There are two recommendations to help mitigate this issue:

1. Reorder the calls. Moving the `$enabledSessions.add()` and `$sessionValidators.enable()` calls to be at the end of the installation process would be safer. These are the most important storage values, so if an attacker gains control flow while these are still unset, there is much less risk.

2. Document this behavior, especially in any documentation relating to `initializeWithMultiplexer()`. It would be ideal if policy developers minimized external calls during session installation.

Rhinestone: Fixed in PR 102.

Cantina Managed: Verified.

3.2 Medium Risk

3.2.1 Uninstalling the fallback submodule of SmartSession will break the whole validation process

Severity: Medium Risk

Context: SmartSessionBase.sol#L282-L287, SmartSessionBase.sol#L300-L307

Description: The `SmartSession` can behave as a multi-type module where it is serving as the "validator" for a smart account and also serve as a fallback handler for `supportsNestedTypedDataSign()` callbacks.

Since ERC7739 usage would require the smart account to have a fallback to expose a "*supportsNestedTypedDataSign*" view function, the same `SmartSession` module can be used as a fallback handler for this method by installing it on the smart account.

The problem here is that the way it is implemented right now breaks compatibility with some smart wallet providers like Biconomy.

For Biconomy's nexus smart account, it can be seen in ModuleManager.sol#L346 that whenever any component(validator/fallback handler etc...) is removed from the account, `onUninstall` is called on the module address.

It is expected to be the job of the module to distinguish what part of the multi-type module is being uninstalled.

But `SmartSession` lacks logic to specifically uninstall the fallback submodule when the smart account would want to remove it as a fallback handler and continue using it as a validator. When `SmartSessionBase::onUninstall()` is called, it completely wipes out the session storage.

A smart account trying to uninstall only the fallback handler part of the `SmartSession` module breaks the entire session and validation operations thereafter.

Recommendation: Move the fallback handler part to an independent submodule and make it an optional feature, in order to ensure compatibility with all wallet providers.

Rhinestone:

Fixed in PR 64.

Cantina Managed: Verified.

3.2.2 `_enablePolicies()` does not install `ERC7739Content`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `validateUserOp()` function, the `_enablePolicies()` logic allows an account to atomically install policies during validation. Part of the data that's provided to the installation logic is `enableData.sessionToEnable.erc7739Policies.allowedERC7739Content`, which is an array of type names that are allowed to be used during an ERC-7739 compliant `isValidSignature()` call.

Despite this data being provided to `_enablePolicies()`, there is currently no logic to use it. As a result, accounts that install policies through `_enablePolicies()` will have incorrectly installed permissions, and future calls to `isValidSignature()` will fail.

Recommendation: Add logic to `_enablePolicies()` to install `enableData.sessionToEnable.erc7739Policies.allowed`

```
  // Enable ERC1271 policies
  $erc1271Policies.enable({
    policyType: PolicyType.ERC1271,
    permissionId: permissionId,
    configId: permissionId.toErc1271PolicyId().toConfigId(),
    policyDatas: enableData.sessionToEnable.erc7739Policies.erc1271Policies,
    smartAccount: account,
    useRegistry: useRegistry
  });
+ $enabledERC7739Content.enable(enableData.sessionToEnable.erc7739Policies.allowedERC7739Content,
↪  permissionId, account);
```

Rhinestone: Fixed in PR 71.

Cantina Managed: Verified.

3.2.3 `onUninstall()` has unbounded gas costs

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Since the `SmartSession` contract is an ERC-7579 compliant module, it implements an `onUninstall()` function, which is triggered if `SmartSession` is ever removed from an ERC-7579 account. Currently, this function iterates through all enabled sessions and removes all of the associated storage:

```
function onUninstall(bytes calldata /*data*/ ) external override {
    uint256 configIdsCnt = $enabledSessions.length({ account: msg.sender });

    for (uint256 i; i < configIdsCnt; i++) {
        PermissionId configId = PermissionId.wrap($enabledSessions.at({ account: msg.sender, index: i }));
        removeSession(configId);
    }
}

function removeSession(PermissionId permissionId) public {
    if (permissionId == EMPTY_PERMISSIONID) revert InvalidSession(permissionId);

    // Remove all UserOp policies for this session
    $userOpPolicies.policyList[permissionId].removeAll(msg.sender);

    // Remove all ERC1271 policies for this session
    $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].get(msg.sender, i));
        $actionPolicies.actionPolicies[actionId].policyList[permissionId].removeAll(msg.sender);
    }

    // Remove all ERC1271 policies for this session
    $enabledSessions.remove({ account: msg.sender, value: PermissionId.unwrap(permissionId) });
    $enabledERC7739Content[permissionId].removeAll(msg.sender);
    emit SessionRemoved(permissionId, msg.sender);
}
```

Note that with this logic, there is no limit to the amount of gas that `onUninstall()` will require. In the worst case, the `onUninstall()` function can require more gas than the entire block gas limit, making it impossible for the function to ever be successfully executed.

If the `onUninstall()` call is required by the account to succeed, it could prevent the account from ever uninstalling the module. If `onUninstall()` is wrapped in a try-catch block by the account, any non-cleared state might lead to unexpected behavior if the module is re-installed later.

Recommendation: To ensure `onUninstall()` can always succeed within a reasonable gas limit, consider implementing an upper bound on the number of sessions, associated policies, and other storage values that can be installed on an account.

Since enforcing an upper bound across all associated storage would be a significant change, another approach is to assess the impact of `onUninstall()` potentially reverting due to an out-of-gas error. If the likelihood of this error is low, simply preventing users from re-installing a module that fails to uninstall properly might be acceptable.

Rhinestone: Fixed in PR 69.

Cantina Managed: Verified. A check has been added to the `onInstall()` function to revert if the account has lingering `$enabledSessions`. As a result, if an account does not succeed in uninstalling the module, it will not be permitted to re-install it until the storage is properly cleared.

3.2.4 `requirePolicyType()` missing revert logic

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `requirePolicyType()` function is used during policy installation to ensure the policy implements the expected interface. This function has the following implementation:

```
function requirePolicyType(address policy, PolicyType policyType) internal view {
    bool supportsInterface;
    if (policyType == PolicyType.USER_OP) {
        supportsInterface = IPolicy(policy).supportsInterface(IUserOpPolicy.checkUserOpPolicy.selector);
    } else if (policyType == PolicyType.ACTION) {
        supportsInterface = IPolicy(policy).supportsInterface(IActionPolicy.checkAction.selector);
    } else if (policyType == PolicyType.ERC1271) {
        supportsInterface = IPolicy(policy).supportsInterface(I1271Policy.check1271SignedAction.selector);
    } else {
        revert UnsupportedPolicy(policy);
    }
}
```

Currently, this function does not make use of its `supportsInterface` boolean, so if `supportsInterface()` returns `false`, the installation will still proceed without a revert.

Recommendation: Change the `requirePolicyType()` function to revert if `supportsInterface` is `false`. For example, consider changing to the following implementation:

```
function requirePolicyType(address policy, PolicyType policyType) internal view {
    bool supportsInterface;
    if (policyType == PolicyType.USER_OP) {
        supportsInterface = IPolicy(policy).supportsInterface(IUserOpPolicy.checkUserOpPolicy.selector);
    } else if (policyType == PolicyType.ACTION) {
        supportsInterface = IPolicy(policy).supportsInterface(IActionPolicy.checkAction.selector);
    } else if (policyType == PolicyType.ERC1271) {
        supportsInterface = IPolicy(policy).supportsInterface(I1271Policy.check1271SignedAction.selector);
    }
    if (!supportsInterface) revert UnsupportedPolicy(policy);
}
```

Rhinestone: Fixed in PR 107.

Cantina Managed: Verified.

3.2.5 Aggregators in `ValidationData` break core logic

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In ERC-4337, the `ValidationData` return type from `validateUserOp()` is a packed `uint256` with three components:

```
/**
 ...
 * @param aggregator  - address(0) - The account validated the signature by itself.
 *                      address(1) - The account failed to validate the signature.
 *                      otherwise - This is an address of a signature aggregator that must
 *                                  be used to validate the signature.
 * @param validAfter  - This UserOp is valid only after this timestamp.
 * @param validaUntil - This UserOp is valid only up to this timestamp.
 */
struct ValidationData {
    address aggregator;
    uint48 validAfter;
    uint48 validUntil;
}
```

Notice that this definition reserves `address(0)` and `address(1)` as special cases for the aggregator component, meaning the signature was self-validated or failed, respectively. Any other value represents a signature aggregator address.

In the `SmartSession` codebase, there are multiple locations where logic will break if a full aggregator address (anything other than `address(0)` or `address(1)`) is used. For example, consider the `isFailed()` function:

```
function isFailed(ValidationData packedData) internal pure returns (bool sigFailed) {
    sigFailed = (ValidationData.unwrap(packedData) & 1) == 1;
}
```

Since this function only inspects the least significant bit of the `ValidationData`, any aggregator address with a least significant bit of 1 will be considered a failure.

On the other hand, consider the `intersect()` function in the `ValidationDataLib`:

```
function intersect(ValidationData a, ValidationData b) internal pure returns (ValidationData validationData) {
    assembly {
        // xor(a,b) == shows only matching bits
        // and(xor(a,b), 0x000000000000000000000000ffffffffffffffffffffffffffffffffffffffff) ==
        // filters out the validAfter and validUntil bits
        // if the result is not zero, then aggregator part is not matching
        // validCase :
        // a == 0 || b == 0 || xor(a,b) == 0
        // invalidCase :
        // a mul b != 0 && xor(a,b) != 0
        let sum := shl(96, add(a, b))
        switch or(
            iszero(and(xor(a, b), 0x000000000000000000000000ffffffffffffffffffffffffffffffffffffffff)),
            or(eq(sum, shl(96, a)), eq(sum, shl(96, b)))
        )
        case 1 {
            validationData := and(or(a, b), 0x000000000000000000000000ffffffffffffffffffffffffffffffffffffffff)
            // validAfter
            let a_vd := and(0xffffffffffff0000000000000000000000000000000000000000000000000000, a)
            let b_vd := and(0xffffffffffff0000000000000000000000000000000000000000000000000000, b)
            validationData := or(validationData, xor(a_vd, mul(xor(a_vd, b_vd), gt(b_vd, a_vd))))
            // validUntil
            a_vd := and(0x000000000000ffffffffffff0000000000000000000000000000000000000000, a)
            if iszero(a_vd) { a_vd := 0x000000000000ffffffffffff0000000000000000000000000000000000000000 }
            b_vd := and(0x000000000000ffffffffffff0000000000000000000000000000000000000000, b)
            if iszero(b_vd) { b_vd := 0x000000000000ffffffffffff0000000000000000000000000000000000000000 }
            let until := xor(a_vd, mul(xor(a_vd, b_vd), lt(b_vd, a_vd)))
            if iszero(until) { until := 0x000000000000ffffffffffff0000000000000000000000000000000000000000 }
            validationData := or(validationData, until)
        }
        default { validationData := 1 }
    }
}
```

Notice that this function specially handles all aggregator bits and returns `address(1)` in the case of a mismatch.

This could be problematic, since later in the `setSig()` function, `xor()` is used to signal an error in the least significant bit:

```
function setSig(
    ValidationData validationData,
    bool sigFailed
)
    internal
    pure
    returns (ValidationData _validationData)
{
    assembly {
        _validationData := xor(validationData, sigFailed)
    }
}
```

Depending on how far a full aggregator address can intentionally or unintentionally enter the `SmartSession` codebase, this can result in two errors "canceling out" to make a success due to the `xor()`.

Recommendation: Since aggregators are likely not intended to be supported, consider simplifying the logic referenced above, and add explicit checks to ensure the aggregator component is either `address(0)` or `address(1)`. Additionally, consider changing the `setSig()` function to avoid `xor()` logic, since it can result in errors "canceling out", depending on how errors are propagated in the overall control flow.

Rhinestone: Got rid of the setSig function entirely in PR 113.

Cantina Managed: Verified. The `setSig()` function has been removed and replaced with an explicit return of `ERC4337_VALIDATION_FAILED` when the session validator fails. Additionally, the `isFailed()` function has been updated to treat any non-zero bit in the aggregator component of `ValidationData` as an error, which will help prevent problems created by malicious or incorrectly designed policies that return unexpected data.

3.2.6 `FALLBACK_ACTIONID` policies can be used directly

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When an action policy check fails to meet the `minPolicies` threshold, the code is designed to re-attempt the check using the policies installed on the `FALLBACK_ACTIONID`. This value is computed with the following logic:

```
// ActionId for a fallback action policy. This id will be used if both action
// target and selector are set to 1. During validation if the current target and
// selector does not have a set action policy, then the fallback will be used if
// enabled.
address constant FALLBACK_TARGET_FLAG = address(1);
bytes4 constant FALLBACK_TARGET_SELECTOR_FLAG = 0x00000001;
// 0xd884b6afa19f8ace90a388daca691e4e28f20cdac5aeefd46ad8bd1c074d28cf
ActionId constant FALLBACK_ACTIONID =
    ActionId.wrap(keccak256(abi.encodePacked(FALLBACK_TARGET_FLAG, FALLBACK_TARGET_SELECTOR_FLAG)));
```

Note that in the `checkSingle7579Exec()` function, the user can freely specify the `target` and `targetSig`, with the exception of self-calls to `execute()`:

```
function checkSingle7579Exec(/* ... */) /* ... */ {
    // ...
    if (targetSig == IERC7579Account.execute.selector && target == userOp.sender) {
        revert ISmartSession.InvalidSelfCall();
    }

    // Generate the action ID based on the target and function selector
    ActionId actionId = target.toActionId(targetSig);

    // Check the relevant action policy
    vd = $policies[actionId].tryCheck({/* ... */});
    // ...
}
```

As a result, it is possible for a user to set `target == address(1)` and `targetSig == bytes4(0x00000001)` in order to match the `FALLBACK_ACTIONID` and directly use its policies. Depending on the the session validator, this may allow an unexpected call to the `ecRecover` precompile at `address(1)`. This is likely not intended and it would make sense to explicitly prevent this behavior.

Recommendation: Consider explicitly preventing using the `FALLBACK_ACTIONID` outside of the fallback logic. For example, consider adding the following check:

```
  function checkSingle7579Exec(/* ... */) /* ... */ {
      // ...
      if (targetSig == IERC7579Account.execute.selector && target == userOp.sender) {
          revert ISmartSession.InvalidSelfCall();
      }
+     if (target == address(1)) revert();

      // Generate the action ID based on the target and function selector
      ActionId actionId = target.toActionId(targetSig);

      // Check the relevant action policy
      vd = $policies[actionId].tryCheck({/* ... */});
      // ...
  }
```

Rhinestone: Fixed in PR 105.

Cantina Managed: Verified.

3.2.7 Batch executions array can be empty

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SmartSession` codebase, at least one action policy is generally required to be set and verified for `validateUserOp()` to succeed. This is enforced with a `minPolicies` argument of 1, and if this threshold isn't met initially, the system re-attempts validation using the `FALLBACK_ACTIONID`.

However, there is currently one scenario where `validateUserOp()` can succeed without validating any action policies. Specifically, in the `checkBatch7579Exec()` function, it is technically possible for the `executions` array to be empty in the following code:

```
function checkBatch7579Exec(
    mapping(ActionId => Policy) storage $policies,
    PackedUserOperation calldata userOp,
    PermissionId permissionId,
    uint256 minPolicies
)
    internal
    returns (ValidationData vd)
{
    // Decode the batch of 7579 executions from the user operation's call data
    Execution[] calldata executions = userOp.callData.decodeUserOpCallData().decodeBatch();
    uint256 length = executions.length;

    // Iterate through each execution in the batch
    for (uint256 i; i < length; i++) {
        Execution calldata execution = executions[i];

        // Check policies for the current execution and intersect the result with previous checks
        ValidationData _vd = checkSingle7579Exec({
            $policies: $policies,
            userOp: userOp,
            permissionId: permissionId,
            target: execution.target,
            value: execution.value,
            callData: execution.callData,
            minPolicies: minPolicies
        });

        vd = vd.intersect(_vd);
    }
}
```

If the `executions` array is empty, no action policies are checked, since no calls are ever made.

While this may not be directly exploitable, it introduces a potential deviation from the codebase's intended design. Also, even though this would result in a no-op call to the account's `execute()` function, gas fees may still be incurred by the account, so it's important to ensure the intended validation is always enforced.

Recommendation: To prevent this behavior, consider enforcing that the `executions` array is non-empty in `checkBatch7579Exec()`.

Rhinestone: Fixed in PR 106.

Cantina Managed: Verified.

3.3 Low Risk

3.3.1 `_get()` doesn't do length check

Severity: Low Risk

Context: AssociatedArrayLib.sol#L21

Description: `AssociatedArrayLib._get()` doesn't do a length check, so if `index` is out of bounds, it returns a garbage value.

Recommendation: Revert if `index` is out of bounds.

Rhinestone: Fixed in PR 65.

Cantina Managed: Verified.

3.3.2 `areEnabled()` functions return different values for empty policy list

Severity: Low Risk

Context: PolicyLib.sol#L265, PolicyLib.sol#L306

Description: `areEnabled(EnumerableActionPolicy storage $self, ...)` doesn't explicitly handle `length == 0` case (returns `false`) like the function `areEnabled(Policy storage $policies, ...)` above (returns `true`).

Recommendation: Update the functions to ensure consistent behavior.

Rhinestone: Fixed in PR 71.

Cantina Managed: Verified.

3.3.3 `_DOMAIN_SEPARATOR` can omit unused values

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `HashLib.sol` file defines two constants:

```
/// @dev `keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)")`.
bytes32 constant _DOMAIN_TYPEHASH = 0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f;

// keccak256(abi.encode(_DOMAIN_TYPEHASH, keccak256("SmartSession"), keccak256(""), 0, address(0)));
// One should use the same domain separator where possible
// or provide the following EIP712Domain struct to the signTypedData() function
// Name: "SmartSession" (string)
// Version: "" (string)
// ChainId: 0 (uint256)
// VerifyingContract: address(0) (address)
// it is introduced for compatibility with signTypedData()
// all the critical data such as chainId and verifyingContract are included
// in session hashes
// https://docs.metamask.io/wallet/reference/eth_signtypeddata_v4
bytes32 constant _DOMAIN_SEPARATOR = 0xa82dd76056d04dc31e30c73f86aa4966336112e8b5e9924bb194526b08c250c1;
```

Notice that the `_DOMAIN_SEPARATOR` uses zero for the values of the `version`, `chainId`, and `verifyingContract` fields. This is because the `_DOMAIN_SEPARATOR` is used in the `multichainDigest()` function, which verifies signatures that aren't tied to a specific chain.

According to the EIP-712 spec, unused fields in the domain separator can be omitted:

> Protocol designers only need to include the fields that make sense for their signing domain. Unused fields are left out of the struct type.
>
> - `string name` the user readable name of signing domain, i.e. the name of the DApp or the protocol.
> - `string version` the current major version of the signing domain. Signatures from different versions are not compatible.

- `uint256 chainId` the EIP-155 chain id. The user-agent should refuse signing if it does not match the currently active chain.
- `address verifyingContract` the address of the contract that will verify the signature. The user-agent may do contract specific phishing prevention.
- `bytes32 salt` an disambiguating salt for the protocol. This can be used as a domain separator of last resort.

Therefore, instead of using zero values, it may be more appropriate to simply omit the unused fields.

Recommendation: Consider removing the zeroed-out values from the `_DOMAIN_SEPARATOR` calculation and the `_DOMAIN_TYPEHASH` definition. Then, recalculate the hashes accordingly.

Rhinestone: Fixed in PR 68.

Cantina Managed: Verified.

3.3.4 `SESSION_NOTATION` missing `nonce`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `Session` struct has the following EIP-712 type definition:

```
string constant SESSION_NOTATION =
    "Session(address account,address smartSession,uint8 mode,address sessionValidator,bytes32 salt,bytes
↪   sessionValidatorInitData,PolicyData[] userOpPolicies,ERC7739Data erc7739Policies,ActionData[] actions)";
```

This struct is hashed in the following function:

```
function _sessionDigest(
    Session memory session,
    address account,
    address smartSession, // for testing purposes
    SmartSessionMode mode,
    uint256 nonce
)
    internal
    pure
    returns (bytes32 _hash)
{
    // chainId is not needed as it is in the ChainSession
    _hash = keccak256(
        abi.encode(
            SESSION_TYPEHASH,
            account,
            smartSession,
            uint8(mode), // Include mode as uint8
            address(session.sessionValidator),
            session.salt,
            keccak256(session.sessionValidatorInitData),
            session.userOpPolicies.hashPolicyDataArray(),
            session.erc7739Policies.hashERC7739Data(),
            session.actions.hashActionDataArray(),
            nonce
        )
    );
}
```

Notice that in the `_sessionDigest()` function, an additional `nonce` value is included in the hash, which is not present in the `SESSION_NOTATION` definition. This discrepancy will cause issues off-chain in how the data is presented to the user for signing.

Recommendation: Add the `uint256 nonce` value to the `SESSION_NOTATION` definition:

```
string constant SESSION_NOTATION =
    "Session(address account,address smartSession,uint8 mode,address sessionValidator,bytes32 salt,bytes
↪   sessionValidatorInitData,PolicyData[] userOpPolicies,ERC7739Data erc7739Policies,ActionData[]
↪   actions,uint256 nonce)";
```

Rhinestone: Fixed in PR 70.

Cantina Managed: Verified.

3.3.5 `ActionData` incorrect member ordering

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `ActionData` EIP-712 struct is defined as follows:

```
string constant ACTION_DATA_NOTATION = "ActionData(address actionTarget, bytes4
↪  actionTargetSelector,PolicyData[] actionPolicies)";
bytes32 constant ACTION_DATA_TYPEHASH = keccak256(bytes(ACTION_DATA_NOTATION));
```

This struct is used in the following function:

```
function hashActionData(ActionData memory actionData) internal pure returns (bytes32) {
    return keccak256(
        abi.encode(
            ACTION_DATA_TYPEHASH,
            actionData.actionTargetSelector,
            actionData.actionTarget,
            hashPolicyDataArray(actionData.actionPolicies)
        )
    );
}
```

Notice that in the `hashActionData()` function, the `actionTargetSelector` and `actionTarget` are encoded in a different order than how they are defined in `ACTION_DATA_NOTATION`. This discrepancy will cause issues with how the data is presented to the user for signing off-chain.

Recommendation: Adjust the order of encoding in either `hashActionData()` or `ACTION_DATA_NOTATION` to ensure both locations match.

Rhinestone: Fixed in PR 70.

Cantina Managed: Verified.

3.3.6 EIP-712 types are not sorted correctly

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When an EIP-712 type references other struct types, the set of all referenced structs should be sorted alphabetically by name and then appended to the main type. This is outlined in the EIP-712 spec:

> If the struct type references other struct types (and these in turn reference even more struct types), then the set of referenced struct types is collected, sorted by name and appended to the encoding. An example encoding is `Transaction(Person from,Person to,Asset tx)Asset(address token,uint256 amount)Person(address wallet,string name)`.

In the `HashLib.sol` file, some EIP-712 type definitions do not adhere to this sorting requirement. Specifically:

1. The `SESSION_TYPEHASH` should append the referenced types in the order of `"ActionData"`, `"ERC7739Data"`, and `"PolicyData"`. However, it currently appends them in the following order:

```
bytes32 constant SESSION_TYPEHASH = keccak256(
    abi.encodePacked(
        bytes(SESSION_NOTATION), bytes(POLICY_DATA_NOTATION), bytes(ACTION_DATA_NOTATION),
↪  bytes(ERC7739_DATA_NOTATION)
    )
);
```

2. The `CHAIN_SESSION_TYPEHASH` should append the referenced types in the order of `"ActionData"`, `"ERC7739Data"`, `"PolicyData"`, and `"Session"`. However, it currently appends them in the following order:

```
bytes32 constant CHAIN_SESSION_TYPEHASH = keccak256(
    abi.encodePacked(
        bytes(CHAIN_SESSION_NOTATION),
        bytes(SESSION_NOTATION),
        bytes(POLICY_DATA_NOTATION),
        bytes(ACTION_DATA_NOTATION),
        bytes(ERC7739_DATA_NOTATION)
    )
);
```

3. The `MULTICHAIN_SESSION_TYPEHASH` should append the referenced types in the order of `"ActionData"`, `"ChainSession"`, `"ERC7739Data"`, `"PolicyData"`, and `"Session"`. However, it currently appends them in the following order:

```
bytes32 constant MULTICHAIN_SESSION_TYPEHASH = keccak256(
    abi.encodePacked(
        bytes(MULTI_CHAIN_SESSION_NOTATION),
        bytes(CHAIN_SESSION_NOTATION),
        bytes(SESSION_NOTATION),
        bytes(POLICY_DATA_NOTATION),
        bytes(ACTION_DATA_NOTATION),
        bytes(ERC7739_DATA_NOTATION)
    )
);
```

As a result, the current implementation does not fully comply with the EIP-712 spec, which could lead to issues when presenting data for the end user to sign.

Recommendation: Update the typehash definitions in `HashLib.sol` to append the referenced types in the correct alphabetical order.

Rhinestone: Fixed in PR 70.

Cantina Managed: Verified.

3.3.7 Extra space in struct encoding for EIP-712

Severity: Low Risk

Context: HashLib.sol#L10

Description: `ACTION_DATA_NOTATION` is defined as:

```
string constant ACTION_DATA_NOTATION = "ActionData(address actionTarget, bytes4
↪  actionTargetSelector,PolicyData[] actionPolicies)";
```

Notice the space between `,` and `bytes4` for the second argument. According to EIP-712, a struct encoding shouldn't have this space and will not result in proper off-chain integration for signature and verification.

Recommendation: Add the missing space:

```
- string constant ACTION_DATA_NOTATION = "ActionData(address actionTarget, bytes4
↪  actionTargetSelector,PolicyData[] actionPolicies)"
+ string constant ACTION_DATA_NOTATION = "ActionData(address actionTarget,bytes4
↪  actionTargetSelector,PolicyData[] actionPolicies)"
```

Rhinestone: Fixed in PR 59.

Cantina Managed: Verified.

3.3.8 Limit the return data copied to memory

Severity: Low Risk

Context: PolicyLib.sol#L72

Description: What `safeCall()` does:

- Revert if the call reverts.
- Return the returnData as bytes memory.

Eventually only 32 bytes are used from the return data.

Best practice is to limit the size of the return data for untrusted or semi-trusted external calls. External calls can cause "return data bombing" where it returns so much data that all gas is exhausted in copying it to memory.

Recommendation: You can use ExcessivelySafeCall where you can limit the size of data being copied to memory (in this case 32 bytes).

Rhinestone: Fixed in PR 67.

Cantina Managed: Verified.

3.3.9 `$enabledERC7739Content.enable()` is called twice

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `_enablePolicies()` function, the following code enables ERC-1271 related storage:

```
// Enable ERC1271 policies
$enabledERC7739Content.enable({
    contents: enableData.sessionToEnable.erc7739Policies.allowedERC7739Content,
    permissionId: permissionId,
    smartAccount: account
});
$erc1271Policies.enable({
    policyType: PolicyType.ERC1271,
    permissionId: permissionId,
    configId: permissionId.toErc1271PolicyId().toConfigId(),
    policyDatas: enableData.sessionToEnable.erc7739Policies.erc1271Policies,
    smartAccount: account,
    useRegistry: useRegistry
});
$enabledERC7739Content.enable(
    enableData.sessionToEnable.erc7739Policies.allowedERC7739Content, permissionId, account
);
```

Notice that `$enabledERC7739Content.enable()` is called twice. As a result, the second call will be a no-op and can be removed.

Recommendation: Remove one of the two calls to `$enabledERC7739Content.enable()`.

Rhinestone: Fixed in PR 102.

Cantina Managed: Verified.

3.3.10 `enabledActionIds` are only cleared on full uninstalls

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `SmartSession` codebase, action policies are stored using the `EnumerableActionPolicy` data structure:

```
struct EnumerableActionPolicy {
    mapping(ActionId => Policy) actionPolicies;
    mapping(PermissionId => EnumerableSet.Bytes32Set) enabledActionIds;
}
```

In this data structure, the `enabledActionIds` field tracks the set of all enabled action ids, while the `action-Policies` field stores the details of each of these policies.

If a user wishes to remove action policies, they can either call `removeSession()`, which removes the entire session, or `disableActionPolicies()`, which has the following implementation:

```
function disableActionPolicies(PermissionId permissionId, ActionId actionId, address[] calldata policies)
↪   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);
    }

    // Disable the specified action policies for the given action ID
    $actionPolicies.actionPolicies[actionId].disable({
        policyType: PolicyType.ACTION,
        smartAccount: msg.sender,
        permissionId: permissionId,
        policies: policies
    });
}
```

Notice that this function does not affect the `enabledActionIds` storage, even if all policies are removed as a result of the `disable()` call. This means the `enabledActionIds` storage can contain action ids that no longer have associated policies, and these action ids can only be removed by uninstalling the entire session.

Recommendation: Consider modifying the `disableActionPolicies()` function to remove the action id from the `enabledActionIds` mapping if no policies remain after the `disable()` call:

```
  function disableActionPolicies(PermissionId permissionId, ActionId actionId, address[] calldata policies)
↪   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);
      }

      // Disable the specified action policies for the given action ID
      $actionPolicies.actionPolicies[actionId].disable({
          policyType: PolicyType.ACTION,
          smartAccount: msg.sender,
          permissionId: permissionId,
          policies: policies
      });

+     if ($actionPolicies.actionPolicies[actionId].policyList[permissionId].length(msg.sender) == 0) {
+         $actionPolicies.enabledActionIds[permissionId].remove(msg.sender, ActionId.unwrap(actionId));
+     }
  }
```

Rhinestone: Fixed in PR 103.

Cantina Managed: Verified.

3.3.11 ERC-1271 logic doesn't verify against `$enabledSessions`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In general, when a permission id is used in an important function of the `SmartSession` codebase, there is a check that the user's `$enabledSessions` storage contains the permission id. However, this check is missing in the ERC-1271 logic.

Fortunately, this is unlikely to be exploitable due to the other checks in the ERC-1271 logic, for example the checks against the `$enabledERC7739Content`.

Recommendation: For consistency and for extra safety, consider adding a check against `$enabledSessions` in the ERC-1271 control flow. For example:

```
  function _erc1271IsValidSignatureNowCalldata(
      address sender,
      bytes32 hash,
      bytes calldata signature,
      bytes calldata contents
  )
      internal
      view
      virtual
      override
      returns (bool valid)
  {
      bytes32 contentHash = string(contents).hashERC7739Content();
      PermissionId permissionId = PermissionId.wrap(bytes32(signature[0:32]));
      signature = signature[32:];
+     if (!$enabledSessions.contains(msg.sender, PermissionId.unwrap(permissionId))) return false;
      if (!$enabledERC7739Content[permissionId].contains(msg.sender, contentHash)) return false;
      valid = $erc1271Policies.checkERC1271({
          account: msg.sender,
          requestSender: sender,
          hash: hash,
          signature: signature,
          permissionId: permissionId,
          configId: permissionId.toErc1271PolicyId().toConfigId(),
          minPoliciesToEnforce: 0
      });

      if (!valid) return false;
      // this call reverts if the ISessionValidator is not set or signature is invalid
      return $sessionValidators.isValidISessionValidator({
          hash: hash,
          account: msg.sender,
          permissionId: permissionId,
          signature: signature
      });
  }
```

Rhinestone: Fixed in PR 104.

Cantina Managed: Verified.

3.3.12 `gasleft()` should be avoided during ERC-4337 validation stage

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The `validateUserOp()` function will be called during the validation stage of an ERC-4337 transaction, which means it's important that the validation rules are followed in this function. Otherwise, transactions may be dropped by ERC-4337 bundlers. These validation rules are outlined in ERC-4337 and ERC-7562.

One of these validation rules is the following:

- **[OP-012]** `GAS` (`0x5A`) opcode is allowed, but only if followed immediately by `*CALL` instructions, else it is blocked. This is a common way to pass all remaining gas to an external call, and it means that the actual value is consumed from the stack immediately and cannot be accessed by any other opcode.

Currently, this rule is not fully adhered to in the codebase. Specifically, this is not followed in the `PolicyLib` contract, where the `callPolicy()` function passes `gasleft()` as an argument to the `excessivelySafeCall()` function:

```
function callPolicy(/* ... */) /* ... */ {
    //  ...
    (bool success, bytes memory returnDataFromPolicy) =
        policy.excessivelySafeCall({ _gas: gasleft(), _value: 0, _maxCopy: 32, _calldata: callOnIPolicy });
    // ...
}
```

Recommendation: to better align with the ERC-4337 validation rules, consider replacing `gasleft()` with `type(uint256).max`. This will accomplish the same result of forwarding all remaining gas.

Note that there is no error for attempting to use more gas than is currently available, as this has been allowed since EIP-150.

Rhinestone: Fixed in PR 109.

Cantina Managed: Verified.

3.4 Gas Optimization

3.4.1 Return boolean expression directly

Severity: Gas Optimization

Context: SmartSessionModeLib.sol#L8-L25, SmartSessionBase.sol#L315

Description: These function bodies can be simplified to just return the boolean expression.

Recommendation: Refactor these functions as:

- SmartSessionModeLib.sol#L8-L25:

```
  - if (A) {
  -   return true;
  - }
  - return false;
  + return A;
```

- SmartSessionBase.sol#L315:

```
  - if (typeID == ERC7579_MODULE_TYPE_VALIDATOR) return true;
  + return typeID == ERC7579_MODULE_TYPE_VALIDATOR;
```

Rhinestone: Fixed in PR 61 and PR 119.

Cantina Managed: Verified.

3.4.2 Repeated operations to compute same value

Severity: Gas Optimization

Context: AssociatedArrayLib.sol#L30-L33

Description: `AssociatedArrayLib._getAll()` does a lot of repeated operation, mainly `keccak256` to find the length slot in each iteration.

Here's `_get()`:

```
mstore(0x00, account)
mstore(0x20, s.slot)
value := sload(add(keccak256(0x00, 0x40), mul(0x20, add(index, 1))))
```

Here's `_length()`:

```
mstore(0x00, account)
mstore(0x20, s.slot)
__length := sload(keccak256(0x00, 0x40))
```

Each iteration stores `account` and `s.slot` in the scratch space and computes its hash to get the length storage slot.

Recommendation: Refactor code to compute the length slot just once.

Rhinestone: Fixed in PR 66.

Cantina Managed: Verified.

3.5 Informational

3.5.1 Typos and documentation errors

Severity: Informational

Context: IPolicy.sol#L19, ISmartSession.sol#L61, ISmartSession.sol#L62, ISmartSession.sol#L78, SmartSession.sol#L82, SmartSessionBase.sol#L53

Description: The code comments have typos at some places:

1. Redundant comment in `SmartSessionBase`: `enableUserPolicies()`.
2. Wrong comment in `SmartSession`: `validateUserOp()` ⇒ it should be "`userOp.sender = msg.sender`".
3. Spelling mistakes at other places linked to this finding.

Recommendation: Correct the documentation as suggested

Rhinestone: Fixed in PR 59.

Cantina Managed: Verified.

3.5.2 `_remove()` doesn't clear last slot

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `AssociatedArrayLib`, the `_remove()` function has the following implementation:

```
function _remove(Array storage s, address account, uint256 index) private {
    uint256 __length = _length(s, account);
    if (index >= __length) revert AssociatedArray_OutOfBounds(index);
    _set(s, account, index, _get(s, account, __length - 1));
    assembly {
        mstore(0x00, account)
        mstore(0x20, s.slot)
        sstore(keccak256(0x00, 0x40), sub(__length, 1))
    }
}
```

This function swaps the last element of the array into the specified `index` and then reduces the array length by one, which is a method known as "swap and pop". However, note that this implementation does not actually delete the element in the last storage slot, which is a deviation from the similar `_pop()` function:

```
function _pop(Array storage s, address account) private {
    uint256 __length = _length(s, account);
    if (__length == 0) return;
    _set(s, account, __length - 1, 0);
    assembly {
        mstore(0x00, account)
        mstore(0x20, s.slot)
        sstore(keccak256(0x00, 0x40), sub(__length, 1))
    }
}
```

Recommendation: To ensure consistency and to gain a gas refund, consider zeroing out the last slot before reducing the array length in `_remove()`:

```
  function _remove(Array storage s, address account, uint256 index) private {
      uint256 __length = _length(s, account);
      if (index >= __length) revert AssociatedArray_OutOfBounds(index);
      _set(s, account, index, _get(s, account, __length - 1));
+     _set(s, account, __length - 1, 0);
      assembly {
          mstore(0x00, account)
          mstore(0x20, s.slot)
          sstore(keccak256(0x00, 0x40), sub(__length, 1))
      }
  }
```

Rhinestone: Fixed in PR 66.

Cantina Managed: Verified.

3.5.3 ERC-7739 logic doesn't support `personal_sign`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The ERC-7739 spec allows wrapping hashes that originate from either EIP-712 or EIP-191 `personal_sign`. However, in the Smart Session implementation of ERC-7739, hashes originating from `personal_sign` are not supported. This is because the `contents` value passed to `_erc1271IsValidSignatureNowCalldata()` remains set to its default value of `signature` and never reaches the code that would overwrite it:

```
function _erc1271IsValidSignatureViaNestedEIP712(
    address sender,
    bytes32 hash,
    bytes calldata signature
)
    internal
    view
    virtual
    returns (bool result)
{
    bytes32 t = _typedDataSignFields();
    bytes calldata contents = signature;
    /// @solidity memory-safe-assembly
    assembly {
        // ...
        for { } 1 { } {
            // ...
            if or(xor(keccak256(0x1e, 0x42), hash), or(lt(signature.length, 1), iszero(c))) {
                t := 0 // Set `t` to 0, denoting that we need to `hash = _hashTypedData(hash)`.
                mstore(t, _PERSONAL_SIGN_TYPEHASH)
                mstore(0x20, hash) // Store the `prefixed`.
                hash := keccak256(t, 0x40) // Compute the `PersonalSign` struct hash.
                break
            }
            // ...
            contents.offset := add(o, 0x40)
            contents.length := c
            // ...
        }
        mstore(0x40, m) // Restore the free memory pointer.
    }
    if (t == bytes32(0)) hash = _hashTypedData(hash); // `PersonalSign` workflow.
    result = _erc1271IsValidSignatureNowCalldata(sender, hash, signature, contents);
}
```

Since a full signature value wouldn't match any string stored in `$enabledERC7739Content`, this will eventually cause a revert in `_erc1271IsValidSignatureNowCalldata()`.

After discussing with the team, it was confirmed that the lack of support for hashes originating from `personal_sign` is intentional. Therefore, this logic can be simplified to make this limitation more explicit.

Recommendation: Consider explictly removing the logic that is related to supporting `personal_sign` hashes. For example, consider removing the `_PERSONAL_SIGN_TYPEHASH` constant, and explicitly return `false` when hashes that don't originate from EIP-712 are used:

```diff
 function _erc1271IsValidSignatureViaNestedEIP712(
     address sender,
     bytes32 hash,
     bytes calldata signature
 )
     internal
     view
     virtual
     returns (bool result)
 {
     bytes32 t = _typedDataSignFields();
     bytes calldata contents = signature;
     /// @solidity memory-safe-assembly
     assembly {
         let m := mload(0x40) // Cache the free memory pointer.
         // `c` is `contentsType.length`, which is stored in the last 2 bytes of the signature.
         let c := shr(240, calldataload(add(signature.offset, sub(signature.length, 2))))
         for { } 1 { } {
             let l := add(0x42, c) // Total length of appended data (32 + 32 + c + 2).
             let o := add(signature.offset, sub(signature.length, l)) // Offset of appended data.
             mstore(0x00, 0x1901) // Store the "\x19\x01" prefix.
             calldatacopy(0x20, o, 0x40) // Copy the `APP_DOMAIN_SEPARATOR` and `contents` struct hash.
             // Use the `PersonalSign` workflow if the reconstructed hash doesn't match,
             // or if the appended data is invalid, i.e.
             // `appendedData.length > signature.length || contentsType.length == 0`.
             if or(xor(keccak256(0x1e, 0x42), hash), or(lt(signature.length, l), iszero(c))) {
                 t := 0 // Set `t` to 0, denoting that we need to `hash = _hashTypedData(hash)`.
-                mstore(t, _PERSONAL_SIGN_TYPEHASH)
-                mstore(0x20, hash) // Store the `prefixed`.
-                hash := keccak256(t, 0x40) // Compute the `PersonalSign` struct hash.
                 break
             }
             // Else, use the `TypedDataSign` workflow.
             // `TypedDataSign({ContentsName} contents,bytes1 fields,...){ContentsType}`.
             mstore(m, "TypedDataSign(") // Store the start of `TypedDataSign`'s type encoding.
             let p := add(m, 0x0e) // Advance 14 bytes to skip "TypedDataSign(".
             calldatacopy(p, add(o, 0x40), c) // Copy `contentsType` to extract `contentsName`.
             contents.offset := add(o, 0x40) // Set the offset of `contents`.
             contents.length := c // Set the length of `contents`.
             // `d & 1 == 1` means that `contentsName` is invalid.
             let d := shr(byte(0, mload(p)), 0x7fffffe000000000000010000000000) // Starts with `[a-z(]`.
             // Store the end sentinel '(', and advance `p` until we encounter a '(' byte.
             for { mstore(add(p, c), 40) } iszero(eq(byte(0, mload(p)), 40)) { p := add(p, 1) } {
                 d := or(shr(byte(0, mload(p)), 0x120100000001), d) // Has a byte in ", )\x00".
             }
             mstore(p, " contents,bytes1 fields,string n") // Store the rest of the encoding.
             mstore(add(p, 0x20), "ame,string version,uint256 chain")
             mstore(add(p, 0x40), "Id,address verifyingContract,byt")
             mstore(add(p, 0x60), "es32 salt,uint256[] extensions)")
             p := add(p, 0x7f)
             calldatacopy(p, add(o, 0x40), c) // Copy `contentsType`.
             // Fill in the missing fields of the `TypedDataSign`.
             calldatacopy(t, o, 0x40) // Copy the `contents` struct hash to `add(t, 0x20)`.
             mstore(t, keccak256(m, sub(add(p, c), m))) // Store `typedDataSignTypehash`.
             // The "\x19\x01" prefix is already at 0x00.
             // `APP_DOMAIN_SEPARATOR` is already at 0x20.
             mstore(0x40, keccak256(t, 0x120)) // `hashStruct(typedDataSign)`.
             // Compute the final hash, corrupted if `contentsName` is invalid.
             hash := keccak256(0x1e, add(0x42, and(1, d)))
             signature.length := sub(signature.length, l) // Truncate the signature.

             break
         }
         mstore(0x40, m) // Restore the free memory pointer.
     }
-    if (t == bytes32(0)) hash = _hashTypedData(hash); // `PersonalSign` workflow.
+    if (t == bytes32(0)) return false;
     result = _erc1271IsValidSignatureNowCalldata(sender, hash, signature, contents);
 }
```

Rhinestone: Fixed in PR 63.

Cantina Managed: Verified.

3.5.4 `_erc1271CallerIsSafe()` is unused

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `SmartSessionERC7739` contract, the `_erc1271CallerIsSafe()` function is internal and currently unused. The function relates to ERC-7739, and allows certain callers to "skip" the rehashing scheme if they are already known to be safe from signature replay, likely because they include the smart account's address in their hash calculation.

Since `_erc1271CallerIsSafe()` is currently unused, and since it would be a large undertaking to compile a list of all major protocols that can safely skip the rehashing scheme, it would be easiest to remove the function altogether.

Recommendation: Remove the `_erc1271CallerIsSafe()` function from the `SmartSessionERC7739` contract.

Rhinestone: Fixed in PR 63.

Cantina Managed: Verified.

3.5.5 Make `multichainDigest()` a `pure` function

Severity: Informational

Context: HashLib.sol#L82

Description: `multichainDigest()` can be made `pure` instead of `view`.

Recommendation: Make `multichainDigest()` a `pure` function.

Rhinestone: Fixed in PR 59.

Cantina Managed: Verified.

3.5.6 Constant keccak expressions are evaluated at compile time

Severity: Informational

Context: HashLib.sol#L49-L63

Description: Solidity evaluates constant keccak expressions at compile time. The assignments to constant variables can be changed to the keccak expressions instead of the evaluated hash values.

Recommendation: Consider updating the assignments to keccak expressions:

```
bytes32 constant _DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,string version,uint256 chainId,address
↪  verifyingContract)");

bytes32 constant _DOMAIN_SEPARATOR = keccak256(abi.encode(_DOMAIN_TYPEHASH, keccak256("SmartSession"),
↪  keccak256(""), 0, address(0)));
```

Rhinestone: Fixed in PR 68.

Cantina Managed: Verified.

3.5.7 Return `false` if any policy is not enabled

Severity: Informational

Context: PolicyLib.sol#L356, PolicyLib.sol#L398, SmartSessionBase.sol#L382

Description: `PolicyLib.areEnabled(...)` functions and `SmartSessionBase.isPermissionEnabled()` revert when policies are partially enabled. Due to UX reasons, it's consistent to return `false` which matches the behavior of returning `false` when no policy is enabled.

Recommendation: Update the functions to return `false` when any policy isn't enabled. You can consider refactoring the code to return early in that case:

```
- uint256 enabledPolicies;
  for (uint256 i; i < length; i++) {
      PolicyData memory policyData = policyDatas[i];
      IPolicy policy = IPolicy(policyData.policy);

      // check if policy is enabled
-     if ($policies.policyList[permissionId].contains(smartAccount, address(policy))) enabledPolicies++;
+     if (!$policies.policyList[permissionId].contains(smartAccount, address(policy))) return false;
}
+ return true;
- if (enabledPolicies == 0) return false;
- else if (enabledPolicies == length) return true;
- else revert ISmartSession.PartlyEnabledPolicies();
```

Rhinestone:

- Permission A and permission B have the same `permissionId`.
- `permissionA.policies` and `permissionB.policies` only partly overlap.

Then if `permissionA` is enabled, and someone tries to check if `permissionB` is enabled, it would return false. so one tries to enable permission B, and it results that:

- Overlapping policies are reinitiated.
- Policies from permissionA but not permission B are left with initial config.
- Policies from `permissionB` are added.

So the resulting permission is not permission A, neither permission B anymore.

Maybe the method should be renamed:`SmartSessionBase.isPermissionEnabled()` ⇒ `SmartSessionBase.isPermissionFullyEnabled()`.

And also expose `external` methods to check particular policy is enabled under permissionId. So the external parties (such as user op builder sc's and backends) may implement whatever algorithms they want based on granular information on what policy is enabled.

3.5.8 Update `minPoliciesToEnforce`'s Natspec's description

Severity: Informational

Context: PolicyLib.sol#L281, PolicyLib.sol#L300-L301

Description: `minPoliciesToEnforce`, as noted in Natspec, indicates the minimum number of policies that must be checked. Since all policies are enforced if `policies.length >= minPoliciesToEnforce`, its description can be updated to indicate it's the least number of policies to enforce.

Recommendation: Update `minPoliciesToEnforce`'s description to match the code.

Rhinestone: Fixed in PR 19.

Cantina Managed: Verified.

3.5.9 Named return variable never returned

Severity: Informational

Context: SmartSession.sol#L381-L404

Description: `valid` is declared as the named return variable, but is never returned and is instead an intermediate variable to compute return value. This may hurt readability a bit.

Recommendation: Consider using an unnamed return from the function.

Rhinestone: Fixed in PR 118.

Cantina Managed: Verified.

3.5.10 Update comment

Severity: Informational

Context: SmartSession.sol#L84

Description: Update the comment as the relevant code ensures that `userOp.sender == msg.sender`. `account` is used as an alias for `userOp.sender` in the code.

Recommendation: Apply this diff:

```
- // ensure that userOp.sender == account
+ // ensure that userOp.sender == msg.sender
```

COMMENTS:

Rhinestone: We suggest instead:

```
ensure that userOp.sender == msg.sender == msg.sender
```

3.5.11 Delete comment

Severity: Informational

Context: ConfigLib.sol#L126

Description: This comment seems to be a relic of some old code and is not relevant anymore.

Recommendation: Delete the comment.

Rhinestone: Fixed in PR 115.

3.5.12 Missing `toUserOpPolicyId()` conversion

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `_enforcePolicies()` function, the following code is used to check the userOp policies for the permission being used:

```
vd = $userOpPolicies.check({
    userOp: userOp,
    permissionId: permissionId,
    callOnIPolicy: abi.encodeCall(IUserOpPolicy.checkUserOpPolicy, (permissionId.toConfigId(), userOp)),
    minPolicies: 0 // for userOp policies, a min of 0 is ok. since these are not security critical
 });
```

Notice that this code is using `permissionId.toConfigId()` to calculate the `ConfigId` for the call. This differs from other parts of the codebase, where `toUserOpPolicyId()` would be calculated in an intermediate step.

While both calculations produce the same result, it may be preferable to update this logic for consistency.

Recommendation: Consider changing the userOp policy check to use the `toUserOpPolicyId()` function in an intermediate step:

```
  vd = $userOpPolicies.check({
      userOp: userOp,
      permissionId: permissionId,
-     callOnIPolicy: abi.encodeCall(IUserOpPolicy.checkUserOpPolicy, (permissionId.toConfigId(), userOp)),
+     callOnIPolicy: abi.encodeCall(IUserOpPolicy.checkUserOpPolicy,
↪   (permissionId.toUserOpPolicyId().toConfigId(), userOp)),
      minPolicies: 0 // for userOp policies, a min of 0 is ok. since these are not security critical
  });
```

Also, consider removing the following two functions from the `IdLib`, as this is currently their only usage:

```
function toConfigId(PermissionId permissionId, address account) internal pure returns (ConfigId _id) {
    _id = ConfigId.wrap(keccak256(abi.encodePacked(account, permissionId)));
}

function toConfigId(PermissionId permissionId) internal view returns (ConfigId _id) {
    _id = toConfigId(permissionId, msg.sender);
}
```

Rhinestone: Fixed in PR 111.

Cantina Managed: Verified.

3.5.13 No method to clear a subset of `$enabledERC7739Content`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `SmartSessions` codebase has two storage areas related to ERC1271 policies: `$enabledERC7739Content` and `$erc1271Policies`. Currently, users who wish to remove specific entries from their `$enabledERC7739Content` storage are limited to either using the `removeSession()` function, which removes the entire session, or the `disableActionPolicies()` function, which is implemented as follows:

```
function disableERC1271Policies(PermissionId permissionId, address[] calldata policies) 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);
    }

    $enabledERC7739Content[permissionId].removeAll(msg.sender);

    // Disable the specified ERC1271 policies
    $erc1271Policies.disable({
        policyType: PolicyType.ERC1271,
        smartAccount: msg.sender,
        permissionId: permissionId,
        policies: policies
    });
}
```

Notice that this implementation removes all `$enabledERC7739Content` values that the user has installed for the `permissionId`. As a result, there is no way to remove only a subset of values from `$enabledERC7739Content`, which can make it difficult for users to make minor adjustments.

Recommendation: Consider changing the `disableERC1271Policies()` function to allow users to specify which `$enabledERC7739Content` entries they wish to remove. For example:

```
- function disableERC1271Policies(PermissionId permissionId, address[] calldata policies) public {
+ function disableERC1271Policies(PermissionId permissionId, address[] calldata policies, string[] calldata
↪  contents) 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);
      }

+     for (uint256 i; i < contents.length; ++i) {
+         bytes32 contentHash = HashLib.hashERC7739Content(contents[i]);
+         $enabledERC7739Content[permissionId].remove(msg.sender, contentHash);
+     }

-     $enabledERC7739Content[permissionId].removeAll(msg.sender);

      // Disable the specified ERC1271 policies
      $erc1271Policies.disable({
          policyType: PolicyType.ERC1271,
          smartAccount: msg.sender,
          permissionId: permissionId,
          policies: policies
      });
  }
```

Rhinestone: Fixed in PR 116.

Cantina Managed: Verified. The `disableERC1271Policies()` function was changed to accept a `string` array as described above. Also, a new `disableActionId()` function was added that allows all storage associated with a user's `actionId` for a given `permissionId` to be cleared.

3.5.14 `_deinitPolicy()` is unused

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `UniActionPolicy` contract, the `_deinitPolicy()` function is currently unused.

Recommendation: Consider removing the `_deinitPolicy()` function and any related logic, such as the `usedIds` mapping, to simplify the codebase.

Rhinestone: Fixed in PR 110.

Cantina Managed: Verified.

3.5.15 `SMART_SESSION_IMPL` is unused

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `SmartSessionCompatibilityFallback` contains the following `immutable` variable:

```
address internal immutable SMART_SESSION_IMPL;

constructor(address smartSessionImpl) {
    SMART_SESSION_IMPL = smartSessionImpl;
}
```

This variable is internal and currently unused.

Recommendation: Consider removing the `SMART_SESSION_IMPL` variable from the contract.

Rhinestone: Fixed in PR 101.

Cantina Managed: Verified.

3.5.16 ERC-7484 registry checks missing on calls to modules

Severity: Informational

Context: PolicyLib.sol#L50-L72

Description: The ERC-7484 registry is not checked when calls to policy submodules/ sessionValidators are made. The check is performed when policy submodules/ sessionValidators are enabled for a session.

The ERC-7579 standard states:

> the Registry Adapter SHOULD implement the following functionality:
>
> - Revert the transaction flow when the Registry reverts.
> - Query the Registry about module A on installation of A.
> - Query the Registry about module A on execution of A.

While the current logic does perform the query on installation, it fails to do so during enforcing policies to validate actions.

If a user opts-in to using the registry, installs policy submodules that have sufficient support from the ERC7484 attesters, and then subsequently that module is determined to be unsafe and the attesters remove their attestation, then this unsafe module will continue to be used by the smart account since there is no check upon usage.

The impact from using such an unsafe policy module could be very high. It is reasonable to imagine a scenario where a module was initially deemed safe and later found to be unsafe, causing attesters to change their position. As such the likelihood is deemed to be medium with an overall severity of rating high.

Recommendation: Call checkForAccount on the registry for all functions that will invoke calls to the policy modules, basically the `_enforcePolicies` ⇒ `PolicyLib::check()` flow.

Rhinestone: Acknowledged.

Cantina Managed: Acknowledged.