Alchemy Modular Account Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Riley Holterhus, Lead Security Researcher

Blockdev, Security Researcher

Christos Pap, Junior Security Researcher

Report prepared by: Lucas Goiriz

January 31, 2024

Contents

1 About Spearbit **4**

2 Introduction **4**

3 Risk classification **4**
3.1 Impact . . . 4
3.2 Likelihood . . . 4
3.3 Action required for severity levels . . . 4

4 Executive Summary **5**

5 Findings **6**
5.1 High Risk . . . 6
5.1.1 SESSION_KEY_DATA_PREFIX causes storage collision in _sessionKeyDataOf() . . . 6
5.2 Medium Risk . . . 7
5.2.1 Many ways to get around _getTokenSpendAmount() . . . 7
5.2.2 State is not cached properly before validation/execution steps . . . 8
5.2.3 Prevent createAccount() user error . . . 9
5.2.4 All self-calls with no runtime validator are valid . . . 9
5.2.5 Uninstalling plugins can still be blocked . . . 10
5.2.6 Using invalid owners will brick the MSCA . . . 11
5.2.7 Valid ERC-1271 signature can be rejected . . . 11
5.2.8 Internal plugin functions can be setup to be externally accessible . . . 12
5.3 Low Risk . . . 13
5.3.1 onUninstall() of SessionKeyPermissionsPlugin doesn't clear or invalidate data . . . 13
5.3.2 hasPostOnlyHooks can be incorrectly true . . . 14
5.3.3 Add explicit checks for self-dependencies . . . 14
5.3.4 The sessionKeys of SessionKeyPermissionsPlugin are not linked to SessionKeyPlugin . . . 15
5.3.5 Usage of .transfer instead of .call may make funds unable to be withdrawn . . . 16
5.3.6 _domainSeparator() can be made more unique . . . 17
5.3.7 allocateAssociatedStorageKey() doesn't clear the upper bits of all parameters . . . 18
5.3.8 Use Ownable2Step to prevent accidental transfers . . . 18
5.3.9 rotateKey() may overwrite the key registered with newSessionKey . . . 18
5.3.10 Possible difference between interfaceId and exposed function selectors of plugins . . . 19
5.3.11 Check on dependencyInterfaceIds is limited . . . 19
5.3.12 Check for plugin in _exec() can be done easier for installed plugins . . . 20
5.4 Gas Optimization . . . 21
5.4.1 _checkAndUpdateGasLimitUsage() does a storage update that is sometimes reverted . . . 21
5.4.2 Already calculated addition can be used in the _checkAndUpdateGasLimitUsage function . . . 21
5.4.3 Function _checkUserOpPermissions() repeatedly retrieves sessionKeyData.contractAccessControlType . . . 22
5.4.4 dependencies.length can be cached earlier to save gas . . . 22
5.4.5 Similar functions _onInstall() and updateSessionKeys() have different checks . . . 22
5.4.6 _isValidERC1271OwnerTypeSignature can cache variables . . . 23
5.4.7 Remove extra argument from _checkCallPermissions() . . . 24
5.4.8 tryDecrement() and tryIncrement() can be optimized . . . 24
5.4.9 tryRemoveKnown() does redundant operations . . . 25
5.4.10 Flip the order of owner addition and removal . . . 25
5.4.11 Use cached callsLength instead of calls.length . . . 26
5.4.12 Flip the condition for potential gas saving . . . 26
5.4.13 hasRequiredPaymaster can be removed . . . 26
5.4.14 AssociatedLinkedListSetLib.getAll() iterates over the loop just to get its size . . . 27
5.4.15 Unnecessary assignments in assembly . . . 27
5.4.16 Redundant SSTOREs in clear() . . . 28
5.4.17 Redundant check in validateUserOp() . . . 28

5.4.18 Function `initialize()` can use `calldata` . . . 29
5.4.19 `_installPlugin()` repeatedly uses `manifest.permittedCallHooks[i]` . . . 29
5.4.20 Check in `_installPlugin()` can be done earlier . . . 30
5.4.21 Clearing upper bits can be done in a more efficient way . . . 31
5.5 Informational . . . 31
5.5.1 Validation reverts and `SIG_VALIDATION_FAILED` are not differentiated . . . 31
5.5.2 Using `||` for bitwise `OR` operations might be confusing . . . 32
5.5.3 Redundant comment in the `_revertOnRuntimePluginFunctionFail` function . . . 32
5.5.4 ERC-6900 specification not clear on duplicate hooks . . . 32
5.5.5 `UUPSUpgradeable` has a different `solidity` version . . . 33
5.5.6 `_updateSpendLimits()` starts a new `interval` . . . 33
5.5.7 Make linked list `clear()` implementations more consistent . . . 33
5.5.8 `SessionKeys` can collect a large `allowance` while staying under the radar . . . 34
5.5.9 No limits on the validity of `sessionKey` . . . 35
5.5.10 Requirements for `_coalescePreValidation()` can be better enforced . . . 36
5.5.11 `_SIG_VALIDATION_PASSED` / `_SIG_VALIDATION_FAILED` not used optimally . . . 37
5.5.12 `_checkUserOpPermissions()` uses a special `nonce` which requires detailed knowledge . . . 38
5.5.13 `_checkUserOpPermissions()` doesn't do checks on ERC20 tokens . . . 38
5.5.14 Plugin `permissions` are not standardized . . . 39
5.5.15 `preUserOpValidationHook()` and `preExecutionHook()` use a different default return value . . . 39
5.5.16 Race conditions with `updateKeyPermissions()` . . . 40
5.5.17 `oldSessionKey` can be reused . . . 40
5.5.18 No function to deregister / invalidate a `sessionKey` . . . 41
5.5.19 Functions of `SessionKeyPermissionsLoupe()` could check `validAfter` / `validUntil` . . . 41
5.5.20 Two ways to retrieve `keyId` . . . 41
5.5.21 Variable name `modifyOwnershipPermission` not logical . . . 42
5.5.22 The `interfaceIds` array of the manifest may contain duplicate elements . . . 43
5.5.23 Linking to other plugins is error-prone . . . 43
5.5.24 Some struct elements can be better documented . . . 44
5.5.25 Session key logic could be prepared for other signature schemas . . . 45
5.5.26 Comment box not used consistently . . . 46
5.5.27 Naming inconcistencies between the `fallback` and the `executeFromPlugin` functions . . . 47
5.5.28 "*Session keys may not be contracts*" is not enforced . . . 47
5.5.29 Some custom errors could be more informative . . . 47
5.5.30 Comments in `BasePlugin` are incomplete . . . 48
5.5.31 `_isValidERC1271OwnerTypeSignature()` could run out of gas . . . 49
5.5.32 Initialization of index variables in `for` loops is not consistent . . . 50
5.5.33 Not clear why view function are not added to `userOpValidationFunctions` . . . 50
5.5.34 Missing or Incomplete NatSpec . . . 51
5.5.35 Comment in `BasePlugin` for `onInstall()` is inaccurate . . . 51
5.5.36 NatSpec comments in the `withdraw` function can be improved . . . 52
5.5.37 Generic function `eip712Domain()` can't be used by other plugins . . . 53
5.5.38 Similar functions `isSentinel()` and `hasNext()` are written differently . . . 53
5.5.39 Similar functions `getAll()` and `tryRemove()` are written differently . . . 54
5.5.40 Collapse `if` / `else` in favor of simplicity . . . 54
5.5.41 The project may fail to deploy or may not work properly on chains that are not compatible with the Shanghai hardfork . . . 55
5.5.42 Keccaks for constants are evaluated at compile time . . . 55
5.5.43 One `assembly` block without `("memory-safe")` . . . 56
5.5.44 Missing `OwnerUpdated` event emission in `onInstall` and `onUninstall` functions . . . 57
5.5.45 Comments for `onHookApply()` could be extended . . . 57
5.5.46 Wrong keccak value for storage prefix . . . 57
5.5.47 `executeFromPluginExternal()` can be made more readable . . . 58
5.5.48 `unchecked` increment pattern is no longer necessary . . . 58
5.5.49 `_resolveManifestFunction()` can be refactored to enhance readability . . . 59
5.5.50 `_resolveManifestFunction()` could revert . . . 60

5.5.51 Other smart contract wallets could be plugins 60
5.5.52 `_addHooks()` can call `_assertNotNullFunction()` 61
5.5.53 Unused code 62
5.5.54 Upgrade `AccountStorageInitializable` to the latest version of the OpenZeppelin library . . 62
5.5.55 Typos 63

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Alchemy provides the leading blockchain development platform powering millions of users in 197 countries worldwide. Their mission is to provide developers with the fundamental building blocks they need to create the future of technology.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of msca according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

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

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Alchemy engaged with Spearbit to review the modular-account protocol. In this period of time a total of **97** issues were found.

Summary

Project Name	Alchemy
Repository	modular-account
Commit	0e3fd1...a1f865
Type of Project	Smart Contract Wallet
Audit Timeline	Nov 20 to Dec 8
Three week fix period	Jan 2 - Jan 23

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	8	6	2
Low Risk	12	9	3
Gas Optimizations	21	20	1
Informational	55	38	17
Total	**97**	**74**	**23**

5 Findings

5.1 High Risk

5.1.1 `SESSION_KEY_DATA_PREFIX` causes storage collision in `_sessionKeyDataOf()`

Severity: High Risk

Context: SessionKeyPermissionsBase.sol#L82-L83, SessionKeyPermissionsBase.sol#L124-L135

Description: In function `_sessionKeyDataOf()`, the `bytes32` variable `sessionKeyDataKey` combines `id` with `SESSION_KEY_DATA_PREFIX`:

```
function _sessionKeyDataOf(address associated, SessionKeyId id) ... {
    uint256 prefixAndBatchIndex = uint256(bytes32(SESSION_KEY_DATA_PREFIX));
    bytes memory associatedStorageKey = PluginStorageLib.allocateAssociatedStorageKey(associated,
↪   prefixAndBatchIndex, 1);

    bytes32 sessionKeyDataKey = bytes32(abi.encodePacked(SESSION_KEY_DATA_PREFIX,
↪   SessionKeyId.unwrap(id)));
    return _toSessionKeyData(PluginStorageLib.associatedStorageLookup(associatedStorageKey,
↪   sessionKeyDataKey));
}
```

This is incorrect, because `SESSION_KEY_DATA_PREFIX` is a `bytes4` value, and the `id` unwraps to a `bytes32`. This means the `bytes32` casting will truncate the data, and the last 4 bytes of the `id` will not be used.

Since the `SessionKeyPermissionsPlugin` increments the `_keyIdCounter` starting at 1, essentially all session keys will start with 28 zero bytes, and will thus share the same storage. In the worst case, a less privileged session key might steal/grief using the privileges meant for a more trusted session key.

Recommendation: Since the `SESSION_KEY_DATA_PREFIX` is already being used in the `prefixAndBatchIndex` variable, simply remove it from the `sessionKeyDataKey`:

```
- bytes32 sessionKeyDataKey = bytes32(abi.encodePacked(SESSION_KEY_DATA_PREFIX,
↪   SessionKeyId.unwrap(id)));
+ bytes32 sessionKeyDataKey = SessionKeyId.unwrap(id);
```

This change also matches with the description of the `SessionKeyData` in the comments:

```
// SessionKeyData (96 bytes)
// 12 padding zeros || associated address || SESSION_KEY_DATA_PREFIX || batch index || sessionKeyId
```

Alchemy: Solved in PR 14 and PR 81.

Spearbit: Verified.

5.2 Medium Risk

5.2.1 Many ways to get around `_getTokenSpendAmount()`

Severity: Medium Risk

Context: SessionKeyPermissionsPlugin.sol#L359-L414, SessionKeyPermissionsPlugin.sol#L603-L673, MSCA-Specs-Manual

Description: Function `_getTokenSpendAmount()` tries to match all cases where tokens are send. In line with the MSCA-Specs-Manual: *ERC-20 Spend Limits only track calls to transfer or approve*.

However there are several other ways to transfer tokens and/or change allowances. At least the following other ways exist:

- `increaseAllowance()`-
- `transferFrom()`, see weird-erc20 transferfrom-with-src--msgsender-
- aliasses to one of the other functions. For example Dai has these functions:

```
function push(address usr, uint wad) external {
    transferFrom(msg.sender, usr, wad);
}
function pull(address usr, uint wad) external {
    transferFrom(usr, msg.sender, wad);
}
function move(address src, address dst, uint wad) external {
    transferFrom(src, dst, wad);
}
```

All the alternative ways have to be explicitly blocked otherwise they can be abused and the `sessionKey` can transfer funds without permission. Explit blocking is difficult and error-prone, especially because there is no limit to the possible aliases.

```
 function _updateLimitsPreExec(address account, bytes calldata callData) internal {
    // ...
    if (contractData.isERC20WithSpendLimit) {
        uint256 spendAmount = _getTokenSpendAmount(account, call.target, call.data);
        // ...
    }
    // ...
}
function _getTokenSpendAmount(address account, address token, bytes memory callData) /*...*/ {
    // ...
    if (selector == IERC20.transfer.selector) {
       // ...
    } else if (selector == IERC20.approve.selector) {
        // ...
    }
    // ...
    return 0;
}
```

Recommendation: For all token addresses with `isERC20WithSpendLimit` set, only allow the selectors `transfer`, `approve` and view functions. If the view functions aren't relevant then a `revert` could be added to the end of `_getTokenSpendAmount()`:

```
function _getTokenSpendAmount(address account, address token, bytes memory callData) ... {
    ...
    if (selector == IERC20.transfer.selector) {
       ...
    } else if (selector == IERC20.approve.selector) {
        ...
    }
    ...
-   return 0;
+   revert(...);
}
```

The check could also added to `_checkCallPermissions()`. Here is an example for implementation. If the view functions aren't relevant they can be removed.

```
  function _checkCallPermissions(...) ... {
      bytes4 selector = bytes4(callData);
      ContractData storage contractData = _contractDataOf(msg.sender, keyId, target);
+     if (contractData.isERC20WithSpendLimit && !isAllowedERC20Function(selector) )
+         return false;
      // ...
  }

+ function isAllowedERC20Function(bytes4 selector) internal pure returns (bool) {
+     return selector == IERC20.transfer.selector
+         || selector == IERC20.approve.selector
+         || selector == IERC20.totalSupply.selector
+         || selector == IERC20.balanceOf.selector
+         || selector == IERC20.allowance.selector
+         || selector == ERC20.name.selector
+         || selector == ERC20.symbol.selector;
+ }
```

Alchemy: Solved in PR 66.

Spearbit: Verified.

5.2.2 State is not cached properly before validation/execution steps

Severity: Medium Risk

Context: UpgradeableModularAccount.sol

Description: In ERC-6900, hooks can be installed to run custom logic before and after execution/validation functions. Since execution functions, validation functions, and the hooks themselves can all update the account's storage, there is a consideration of which state should be used during each of these steps. The EIP implies that the relevant hooks/functions should be cached *before* each step begins:

> Notably, for the `uninstallPlugin` native function, the post execution hooks defined for it prior to the uninstall MUST run afterwards.

However, this is currently not the case, and there are inconsistencies between functions. For one example, `postOnlyHooks` are not cached, which means the changes from `installPlugin()`/`uninstallPlugin()` will immediately affect the hooks used after execution. Another example is in `executeFromPlugin()`, where the effects of `_doPrePermittedCallHooks()` can technically change the `plugin` and pre-exec hooks later used. On the other hand, the `fallback()` function *does* cache the `plugin` and the existence of any `postOnlyHooks`, but doesn't cache the `postOnlyHooks` themselves.

Recommendation: After discussing with Alchemy, the recommended behavior is to cache state once before validation (runtime or userOp) and once before execution. The state that is cached should be precisely the state about to be used. Notably, this implies the validation phase *can* affect the control flow of the execution phase,

and can even change the plugin used. Since userOp validation spans multiple sub-calls, this behavior will be the simplest to implement and reason about.

It is also recommended to describe this behavior in the ERC-6900 spec, so that other implementations can do the same.

Alchemy: Solved in PR 58.

Spearbit: Verified.

5.2.3 Prevent `createAccount()` user error

Severity: Medium Risk

Context: MultiOwnerMSCAFactory.sol, MultiOwnerTokenReceiverMSCAFactory.sol

Description: In both factory implementations, the `createAccount()` function will revert if the user makes any of the following mistakes:

- If `owners.length` is zero.
- If `owners.length` is too large (to require more gas than the block limit).
- If `address(0)` exists in `owners`.
- If duplicate addresses exist in `owners`.

Although the probability is low, this would be a problem if the user doesn't realize their mistake until *after* they've received funds at their counterfactual address.

Recommendation: Consider making the `getAddress()` function revert in these conditions, so that users would catch their mistake earlier.

Alchemy: Solved in PR 26, PR 49 and PR 82.

Spearbit: Verified.

5.2.4 All self-calls with no runtime validator are valid

Severity: Medium Risk

Context: UpgradeableModularAccount.sol#L616-L635

Description: The `_doRuntimeValidation()` function is intended to revert if the relevant `runtimeValidationFunction` is empty, except for the special case when it's a self-call to `installPlugin()` or `upgradeToAndCall()`:

```
if (
    runtimeValidationFunction == FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE
        && (
            (
                msg.sig != IPluginManager.installPlugin.selector
                    || msg.sig != UUPSUpgradeable.upgradeToAndCall.selector
            ) && msg.sender != address(this)
        )
) {
    // Runtime calls cannot be made against functions with no
    // validator, except in the special case of self-calls to
    // `installPlugin` and `upgradeToAndCall`, to enable removing the plugin protecting
    // `installPlugin` and installing a different one as part of
    // a single batch execution, and/or to enable upgrading the account implementation.
    revert RuntimeValidationFunctionMissing(msg.sig);
}
```

However, this special case is implemented with the `||` and `&&` flipped, and the `msg.sig` check always evaluates to `true`. This means validation will succeed for *any* self-call with an empty validator, which is not intended.

Recommendation: To achieve the desired behavior, flip the `||` and the `&&` in the `if` statement:

```
if (
    runtimeValidationFunction == FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE
        && (
            (
                msg.sig != IPluginManager.installPlugin.selector
                    && msg.sig != UUPSUpgradeable.upgradeToAndCall.selector
            ) || msg.sender != address(this)
        )
) {
    // Runtime calls cannot be made against functions with no
    // validator, except in the special case of self-calls to
    // `installPlugin` and `upgradeToAndCall`, to enable removing the plugin protecting
    // `installPlugin` and installing a different one as part of
    // a single batch execution, and/or to enable upgrading the account implementation.
    revert RuntimeValidationFunctionMissing(msg.sig);
}
```

Alchemy: Solved in PR 22.

Spearbit: Verified.

5.2.5 Uninstalling plugins can still be blocked

Severity: Medium Risk

Context: UpgradeableModularAccount.sol#L409-L440

Description: The function `uninstallPlugin()` has an option to `forceUninstall`. However `preUserOpValidation` / `preRuntimeValidation` hooks or `userOpValidationFunctions` or `runtimeValidationFunctions` or `preExecution` / `postExection` hooks of `uninstallPlugin` can still prevent `uninstallPlugin` from executing. Due to this, it might be difficult or impossible to remove a malicious plugin. And thus (remaining) funds would remain inaccesible. This could potentially damage the reputation of the project.

```
function uninstallPlugin(/*...*/) /*...*/ {
    // ...
    args.forceUninstall = decodedConfig.forceUninstall;
    // ...
}
```

Recommendation: Doublecheck the effort that could/should be put in removing malicious plugins. A potential solution would be a function in the MSCA that can unconditionally trigger an upgrade can exclusively be claimed by one plugin, for example the `MultiOwnerPlugin`. It must be protected by additional security guaranties, for example by signatures of multiple owners.

Alchemy: We have decided to acknowledge the issue, document the potential risk in all our public facing docs, and no action in code for MVP. We have also created tasks to revisit in the next version.

Spearbit: Acknowledged.

5.2.6 Using invalid owners will brick the MSCA

Severity: Medium Risk

Context: MultiOwnerPlugin.sol#L82-L94, MultiOwnerPlugin.sol#L191-L197

Description: When adding or updating the `owners` of the MSCA via the `MultiOwnerPlugin`, the resulting set of `owners` could potentially contain only invalid `owners`. This is also possible during the creation of the MSCA. If this occurs then the MSCA is bricked.

Additionally any address can be added as an `owner` of the MSCA, also without the permission of the person responsible for that address. This could be abused by adding an innocent address to a MSCA containing stolen funds.

OZ Ownable2Step has a solution for a similar issue where the new owner is verified via `acceptOwnership()`.

```
function _onInstall(bytes calldata data) internal override isNotInitialized(msg.sender) {
    // ...
    _addOwnersOrRevert(_owners, msg.sender, initialOwners);
}
function updateOwners(address[] memory ownersToAdd, address[] memory ownersToRemove) /*...*/ {
    // ...
    _addOwnersOrRevert(_owners, msg.sender, ownersToAdd);
    _removeOwnersOrRevert(_owners, msg.sender, ownersToRemove);
    // ...
}
```

Recommendation: A potential solution is to add `signatures` from (one or more of) the owner(s) in the relevant functions calls and check that the signatures can be verified. Alternatively, front end checks could be added.

Alchemy: When updating owners, if the user supplies invalid addresses (e.g., addresses that the user does not have access to) as owners, the modular account can be rendered unusable. This can happen during MultiOwnerPlugin installation, or when the user updates the owner set later.

We have checks in place for common mistakes, but there is no easy way to prevent users from adding wrong addresses that they don't have access to without sacrificing user experience. Even if we decide to fix this, the issue still exists for counterfactual addresses. When getting counterfactual address, the factory asks for an array of owners to be passed in. That array is eventually used to create the account. Here, we also have checks in place for common mistakes, but there is no way to prevent users adding wrong addresses that they don't have access to.

This is an user error that is very tricky to prevent from the contract side. All things considered, we choose to acknowledge the risk, and encourage the client to educate users to avoid making such a mistake.

Spearbit: Acknowledged.

5.2.7 Valid ERC-1271 signature can be rejected

Severity: Medium Risk

Context: MultiOwnerPlugin.sol#L140-L151, MultiOwnerPlugin.sol#L213-L224

Description: `isValidSignature()` may reject a valid ERC-1271 mistaking it for an EOA signature:

```
(address signer, ECDSA.RecoverError error) = ECDSA.tryRecover(messageHash, signature);
if (error == ECDSA.RecoverError.NoError) {
    if (_owners.contains(msg.sender, CastLib.toSetValue(signer))) {
        return _1271_MAGIC_VALUE;
    } else {
        return _1271_MAGIC_VALUE_FAILURE;
    }
} else {
    if (_isValidERC1271OwnerTypeSignature(msg.sender, messageHash, signature)) {
        return _1271_MAGIC_VALUE;
    }
}
```

Take this instance as an example: if a Gnosis safe contract `C` is one of the owners of an MSCA, and `C`'s own owner is `E` (an EOA).

`E` signs a message and uses ERC-1271 so that `C` approves `E`'s ECDSA signature. Now when this signature is passed to the MSCA, it first recovers the signer which will be `E`. Since `E` is not in the owner list of MSCA, failure will be returned.

However, this signature should be treated as a valid signature due to ERC-1271. `userOpValidationFunction()` has the same issue.

Recommendation: Update `isValidSignature()` and `userOpValidationFunction()` to check for ERC-1271 signature if for any reason ECSDA signature validity fails. Here's the suggested update for `isValidSignature()`:

```
  if (error == ECDSA.RecoverError.NoError) {
      if (_owners.contains(msg.sender, CastLib.toSetValue(signer))) {
          return _1271_MAGIC_VALUE;
-     } else {
-         return _1271_MAGIC_VALUE_FAILURE;
      }
- } else {
+ }
  if (_isValidERC1271OwnerTypeSignature(msg.sender, messageHash, signature)) {
      return _1271_MAGIC_VALUE;
  }
- }
```

Alchemy: Solved in PR 15.

Spearbit: Verified.

5.2.8 Internal plugin functions can be setup to be externally accessible

Severity: Medium Risk

Context: PluginManagerInternals.sol#L71-L93, IPlugin.sol

Description: The function `_setExecutionFunction()` prevents several classes of function selectors to be used. The functions from `IPlugin` are not checked so they could be configured to be accessed externally. As these function have no additional access control, this way the inner working of the plugins could be made accessible and thus disturbed.

In practice this is unlikely to happen, but if it would happen the impact could be high.

```
function _setExecutionFunction(bytes4 selector, address plugin) internal {
   // ...
    if (KnownSelectors.isNativeFunction(selector)) {
        revert NativeFunctionNotAllowed(selector);
    }
    // ...
    if (KnownSelectors.isErc4337Function(selector)) {
        revert Erc4337FunctionNotAllowed(selector);
    }
    // ...
}
```

Recommendation: In `_setExecutionFunction()` check that `selector` isn't one of the `selectors` of `IPlugin`.

Alchemy: Solved in PR 17.

Spearbit: Verified.

5.3 Low Risk

5.3.1 `onUninstall()` of `SessionKeyPermissionsPlugin` doesn't clear or invalidate data

Severity: Low Risk

Context: SessionKeyPermissionsPlugin.sol#L122

Description: The `onUninstall()` of `SessionKeyPermissionsPlugin` doesn't clear or invalidate any data. If the plugin would be installed again in the future, all the old `sessionKey` related data is valid again.

```
function onUninstall(bytes calldata) external override {}
```

Recommendation: Consider to invalidate all data by adding a field to all `PluginStorageLib` fields, that is incremented for each `onUninstall()`. For example in the following way:

```
  function onUninstall(bytes calldata) external override {
+     _keyIdCounter[msg.sender] +=  0x1_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000;
+     // invalidate all SessionKeyData ContractData FunctionData
  }
```

Incorporate the value `_keyIdCounter[msg.sender] >> 24*8` somewhere in the `_sessionKeyIdOf()` to invalidate the `SessionKeyId` data. Depending on preference, this could also be achieved by making the `_keyIdCounter` store a struct that tracks these two types of values in a single packed storage slot.

Example of a struct that has the some data layout:

```
struct KeyIdCounter {
    uint64 currentValidBatch;
    uint192 counter;
}
```

Alchemy: Solved in PR 62 by merging `SessionKeyPlugin` and `SessionKeyPermissionsPlugin`.

Spearbit: Verified.

5.3.2 `hasPostOnlyHooks` can be incorrectly `true`

Severity: Low Risk

Context: PluginManagerInternals.sol#L126-L132, PluginManagerInternals.sol#L172-L178

Description: In the `_addExecHooks()` and `_addPermittedCallHooks()` functions, the `hasPostOnlyHooks` storage boolean is set to `true` as long as `postExecHook != FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE`. This is incorrect in the case where the `preExecHook` is also non-empty, since this would mean it's an associated post hook.

Fortunately, this is not a huge concern, because it will most likely result in entering an empty loop that does nothing. However, this bug can technically result in a different control flow due to how `postOnlyHooks` are cached. This is being addressed in the issue "State is not cached properly before validation/execution steps".

Recommendation: Only set `hasPostOnlyHooks` to `true` if the `preExecHook` is empty and the `postExecHook` is non-empty. Using `_addExecHooks()` as an example:

```
  function _addExecHooks(bytes4 selector, FunctionReference preExecHook, FunctionReference postExecHook)
      internal
  {
      SelectorData storage selectorData = _getAccountStorage().selectorData[selector];

      _addHooks(selectorData.executionHooks, selector, preExecHook, postExecHook);

      if (preExecHook != FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE) {
          selectorData.hasPreExecHooks = true;
-     }
-     if (postExecHook != FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE) {
+     } else if (postExecHook != FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE) {
          selectorData.hasPostOnlyExecHooks = true;
      }
  }
```

Alchemy: Solved in PR 20.

Spearbit: Verified.

5.3.3 Add explicit checks for self-dependencies

Severity: Low Risk

Context: PluginManagerInternals.sol#L352-L373, PluginManagerInternals.sol#L441-L477

Description: In the `_installPlugin()` function, users are meant to specify dependencies in the `dependencies` and `injectedHooks` arguments. For both arguments, it would be incorrect to allow the `plugin` to be a dependency of itself, since the `dependentCount` tracking would break.

For the `injectedHooks` logic, technically there is nothing stopping this. Indeed, the following `plugins.contains()` check would pass, because at this point the `plugin` will have been added to the account's storage:

```
for (uint256 i = 0; i < length;) {
    InjectedHook memory hook = injectedHooks[i];

    storage_.pluginData[plugin].injectedHooks[i] = StoredInjectedHook({
        providingPlugin: hook.providingPlugin,
        selector: hook.selector,
        preExecHookFunctionId: hook.injectedHooksInfo.preExecHookFunctionId,
        isPostHookUsed: hook.injectedHooksInfo.isPostHookUsed,
        postExecHookFunctionId: hook.injectedHooksInfo.postExecHookFunctionId
    });

    // Increment the dependent count for the plugin providing the hook.
    storage_.pluginData[hook.providingPlugin].dependentCount += 1;

    if (!storage_.plugins.contains(CastLib.toSetValue(hook.providingPlugin))) {
        revert MissingPluginDependency(hook.providingPlugin);
    }

    // ...
}
```

For the `dependencies` logic it *is* impossible to include the plugin as a dependent of itself, since it checks that `storage_.pluginData[dependencyAddr].manifestHash == bytes32(0)`, and this check happens before the `storage_.pluginData[plugin].manifestHash = manifestHash` assignment. However, a more explicit check could be useful.

Recommendation: Consider explicitly disallowing a plugin from installing itself as a dependency. This can be achieved with explicit checks (i.e. `require(dependencyAddr != plugin)` and `require(hook.providingPlugin != plugin)`), or by rearranging the current checks to occur before the plugin is included in the account's storage.

Alchemy: Solved in PR 33.

Spearbit: Verified.

5.3.4 The `sessionKeys` of `SessionKeyPermissionsPlugin` are not linked to `SessionKeyPlugin`

Severity: Low Risk

Context: SessionKeyPermissionsPlugin.sol#L44-L116, SessionKeyPermissionsPlugin.sol#L180-L202

Description: The functions `registerKey()` and `rotateKey()` don't check if the `sessionKey` corresponds to a `sessionKey` in `SessionKeyPlugin`. This allows registring keys that are not present in `SessionKeyPlugin`, so they can't be used (because the transaction is run via `executeWithSessionKey` of `SessionKeyPlugin`).

Linking the `sessionKeys` of `SessionKeyPermissionsPlugin` to the sessionKeys of `SessionKeyPlugin` also gives the benefit that they can be enumerated via `getSessionKeys()`.

Additionally `SessionKeyPermissionsPlugin` allows `sessionKey==0`, while `SessionKeyPlugin` doesn't allow it.

```
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {

    function registerKey(address sessionKey, bytes32 tag) external override {
        // no check on sessionKey
    }
    function rotateKey(address oldSessionKey, address newSessionKey) external override {
        // no check on newSessionKey
    }
    function pluginManifest() external pure override returns (PluginManifest memory) {
        // ...
        manifest.preUserOpValidationHooks[0] = ManifestAssociatedFunction({
            executionSelector: ISessionKeyPlugin.executeWithSessionKey.selector,
```

```
            associatedFunction: ManifestFunction({
                functionType: ManifestAssociatedFunctionType.SELF,
                functionId: uint8(FunctionId.PRE_USER_OP_VALIDATION_HOOK_CHECK_PERMISSIONS),
                dependencyIndex: 0 // Unused.
            })
        });
    function pluginManifest() external pure override returns (PluginManifest memory) {
        // ...
        manifest.executionHooks = new ManifestExecutionHook[](1);
        manifest.executionHooks[0] = ManifestExecutionHook({
            executionSelector: ISessionKeyPlugin.executeWithSessionKey.selector,
            preExecHook: ManifestFunction({
                functionType: ManifestAssociatedFunctionType.SELF,
                functionId: uint8(FunctionId.PRE_EXECUTION_HOOK_UPDATE_LIMITS),
                dependencyIndex: 0 // Unused.
            }),
           // ...
        });
    }
    function preUserOpValidationHook(uint8 functionId, UserOperation calldata userOp, bytes32) /*...*/ {
        if (functionId == uint8(FunctionId.PRE_USER_OP_VALIDATION_HOOK_CHECK_PERMISSIONS)) {
            return _checkUserOpPermissions(userOp);
        }
        revert NotImplemented();
    }
    function preExecutionHook(uint8 functionId, address, uint256, bytes calldata data) /*...*/ {
        if (functionId == uint8(FunctionId.PRE_EXECUTION_HOOK_UPDATE_LIMITS)) {
            _updateLimitsPreExec(msg.sender, data);
        }
        return "";
    }
// ...
```

Recommendation: Consider checking the `sessionKey` is valid via `isSessionKey()`.

Alchemy: Solved in PR 62 by merging `SessionKeyPlugin` and `SessionKeyPermissionsPlugin`.

Spearbit: Verified.

5.3.5 Usage of `.transfer` instead of `.call` may make funds unable to be withdrawn

Severity: Low Risk

Context: MultiOwnerMSCAFactory.sol#L98, MultiOwnerTokenReceiverMSCAFactory.sol#L106

Description: The `MultiOwnerTokenReceiverMSCAFactory` and the `MultiOwnerTokenReceiverMSCAFactory` factories use Solidity's `transfer()` function to withdraw funds from the contract. The `.transfer()` function forwards a fixed gas amount of `2300`. However, the gas cost for some opcodes might change in the future, as mentioned in CONSENSYS Diligence's article.

This method could lead to problems if the recipient is a smart contract with a payable fallback function that requires more than `2300` gas units, or if the function is invoked through another proxy that increases gas consumption beyond `2300` units. In addition, on certain chains the gas cost cost can be higher than in Mainnet, and can result in issues, like in zkSync Era.

Recommendation: It is advisable to use the `.call()` function instead and verify the success of the `.call` function by checking the returned boolean value.

Alchemy: Solved in PR 24.

Spearbit: Verified.

5.3.6 `_domainSeparator()` can be made more unique

Severity: Low Risk

Context: MultiOwnerPlugin.sol#L56-L57, MultiOwnerPlugin.sol#L111-L127, MultiOwnerPlugin.sol#L161-L170, MultiOwnerPlugin.sol#L378-L380

Description: The `_domainSeparator()` is based on the MSCA contract address and is not linked to the address of the plugin, while the plugin is doing the verification so it is closer to `verifyingContract`. With multiple versions of an Owner plugin, signatures could potentially be re-used. The `_domainSeparator()` could be made more unique by adding a salt, based on the address of the plugin.

```
function eip712Domain() /*...*/ {
    (fields, name, version, chainId,, salt, extensions) = super.eip712Domain();
    verifyingContract = msg.sender;
}
function _domainSeparator(address account) internal view returns (bytes32) {
    return keccak256(abi.encode(_TYPE_HASH, _HASHED_NAME, _HASHED_VERSION, block.chainid, account));
}
function encodeMessageData(address account, bytes memory message) /*...*/ {
    bytes32 messageHash = keccak256(abi.encode(ERC6900_TYPEHASH, keccak256(message)));
    return abi.encodePacked("\x19\x01", _domainSeparator(account), messageHash);
}
```

Recommendation: Consider adding a salt. Here is an example of an implementation:

```
  bytes32 private constant _TYPE_HASH =
-     keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");
+     keccak256("EIP712Domain(string name,string version,uint256 chainId,address
↪  verifyingContract,bytes32 salt)");

+ bytes32 private constant _SALT = bytes32(bytes20(address(this)));

  function eip712Domain() /*...*/ {
      (fields, name, version, chainId,, salt, extensions) = super.eip712Domain();
+     fields |= hex"10"; // also use salt
      verifyingContract = msg.sender;
+     salt = _SALT;
  }

  function _domainSeparator(address account) internal view returns (bytes32) {
-     return keccak256(abi.encode(_TYPE_HASH, _HASHED_NAME, _HASHED_VERSION, block.chainid, account));
+     return keccak256(abi.encode(_TYPE_HASH, _HASHED_NAME, _HASHED_VERSION, block.chainid,
↪  account,_SALT));
  }
```

Note: See this OpenZeppelin issue for implementation details.

Alchemy: Fixed in PR 38.

Spearbit: Verified.

5.3.7 `allocateAssociatedStorageKey()` doesn't clear the upper bits of all parameters

Severity: Low Risk

Context: PluginStorageLib.sol#L17-L38

Description: The function `allocateAssociatedStorageKey()` clears the upper bits of `keySize` but is doesn't clear the upper bits of `addr`. To be consistent and safe its good to clear the upper bits of `addr` too.

```
function allocateAssociatedStorageKey(address addr, uint256 batchIndex, uint8 keySize) /*...*/ {
    assembly ("memory-safe") {
        // Clear any dirty upper bits of keySize to prevent overflow
        keySize := and(keySize, 0xff)
        // ...
        mstore(add(key, 32), addr)
        // ...
    }
}
```

Recommendation: Consider clearing the upper bits of `addr`.

Alchemy: Solved in PR 35.

Spearbit: Verified.

5.3.8 Use `Ownable2Step` to prevent accidental transfers

Severity: Low Risk

Context: MultiOwnerTokenReceiverMSCAFactory.sol#L18, MultiOwnerMSCAFactory.sol#L18

Description: If a wrong address is passed to the function `transferOwnership()` exposed by `MultiOwnerTokenReceiverMSCAFactory` or `MultiOwnerMSCAFactory`, privileged functionality is permanently lost.

Recommendation: Consider using `Ownable2Step` instead of `Ownable`. Now, the proposed owner has to explicitly accept ownership which at least ensures that the address is valid.

Alchemy: Solved in PR 23 and PR 123.

Spearbit: Verified.

5.3.9 `rotateKey()` may overwrite the key registered with `newSessionKey`

Severity: Low Risk

Context: SessionKeyPermissionsPlugin.sol#L61

Description: `rotateKey()` transfers `keyId` from `oldSessionKey` to `newSessionKey` but doesn't check if `newSessionKey` is already in use. This may lead to an overwrite for `newSessionKey`:

```
SessionKeyId keyId = _sessionKeyIdOf(msg.sender, oldSessionKey);
_assertRegistered(keyId, oldSessionKey);
_updateSessionKeyId(msg.sender, oldSessionKey, SessionKeyId.wrap(bytes32(0)));
_updateSessionKeyId(msg.sender, newSessionKey, keyId);
```

Recommendation: Before update `newSessionKey`, check if it has a key registered with it. Note the change in behavior with this recommendation: `rotateKey()` for same `oldSessionKey` and `newSessionKey` will work, but it no longer works once `newSessionKey` is checked to be empty.

Alchemy: Solved in PR 62.

Spearbit: Verified.

5.3.10 Possible difference between `interfaceId` and exposed function `selector`s of plugins

Severity: Low Risk

Context: TokenReceiverPlugin.sol#L105-L108

Description: Plugins can expose an `interfaceId` while not completely implementing and exposing all functions. In theory, this could be checked by combining the exposed function `selector` and comparing that to the `interfaceId`. However, with the current Manifest setup this doesn't work if multiple `interfaceId`s are combined like `TokenReceiverPlugin` does.

A usecase for having an incomplete implementation would be a plugin that extends another plugin to create a larger feature set. Assuming the dependencies are setup correctly this could function without issues. However in most cases an incomplete implementation is unwanted.

An incomplete implementation should normally be detected via a security review of the plugin.

```
function pluginManifest() external pure override returns (PluginManifest memory) {
    // ...
    manifest.interfaceIds = new bytes4[](3);
    manifest.interfaceIds[0] = type(IERC721Receiver).interfaceId;
    manifest.interfaceIds[1] = type(IERC777Recipient).interfaceId;
    manifest.interfaceIds[2] = type(IERC1155Receiver).interfaceId;
    // ...
}
```

Recommendation: Check if you want to allow extensions of plugin. If so that then no changes are needed. Otherwise consider having an automated way to check for the consistency between `interfaceId` and the exposed function `selector`s. This would require a change in the Manifest because the `interfaceId` and exposed function `selector` have to be linked.

Note: the `interfaceId` is the `XOR` of all function `selector`s in the interface.

Alchemy: Acknowledged.

Spearbit: Acknowledged.

5.3.11 Check on `dependencyInterfaceIds` is limited

Severity: Low Risk

Context: PluginManagerInternals.sol#L362-L365

Description: The function `_installPlugin()` checks the dependency supports the expected interface. When specifying this value in the `Manifest`, the `largest` relevant `interfaceId` should be used.

However the `smallest` relevant `interfaceId` : `type(IPlugin).interfaceId)` can also be used by mistake. This isn't easily detected.

For example the plugins `SessionKeyPlugin` and `SessionKeyPermissionsPlugin` use `type(IPlugin).interfaceId` to link to `owner` plugins. These `owner` plugins don't need to have any other functions defined. So its dificult to specify that an `owner` plugins is expected here.

Note: Also see the issue "Possible difference between `interfaceId` and exposed function `selector`s of plugins".

```
function _installPlugin(/*...*/) /*...*/ {
    // ...
    for (uint256 i = 0; i < length;) {
        // ...
        // Check that the dependency supports the expected interface.
        if (!ERC165Checker.supportsInterface(dependencyAddr, manifest.dependencyInterfaceIds[i])) {
            revert InvalidDependenciesProvided();
        }
        // ...
    }
    // ...
}
```

Recommendation: Consider having a way to identify specific functionality of plugins and use that to make sure the right plugins are referred to.

Alchemy: Acknowledged, but requires a spec update to fix so no changes for now. Will be addressed in a future ERC update.

Spearbit: Acknowledged.

5.3.12 Check for plugin in `_exec()` can be done easier for installed plugins

Severity: Low Risk

Context: AccountExecutor.sol#L23-L37

Description: The function `_exec()` uses a check on `supportsInterface()` to check whether a target is an installed or uninstalled plugin. This checks fails if the plugin doesn't implement `supportsInterface()` in a correct way, either by accident (which is unlikely) or on purpose. This risk of abuse is low because it also involves other actors calling the plugin in the wrong way.

However for installed plugins there is a more straightforward way to check the `target` is a `plugin`, which costs about the same amount of gas.

```
function _exec(address target, uint256 value, bytes memory data) internal returns (bytes memory result)
↪ {
    if (ERC165Checker.supportsInterface(target, type(IPlugin).interfaceId)) {
        revert PluginCallDenied(target);
    }
    // ... // execute
}
```

Recommendation: Consider checking the `target` is an installed plugin and then revert, in function `_exec()`. This could be done with the following code:

```
  function _exec(/*...*/ , AccountStorage storage storage_ ) internal returns (bytes memory result) {
      // ...
+     if (storage_.pluginData[target].manifestHash != bytes32(0)) {  // its installed so certainly not
↪  allowed
+         revert PluginCallDenied(target);
+     }
      if (ERC165Checker.supportsInterface(target, type(IPlugin).interfaceId)) {
          revert PluginCallDenied(target);
      }
      // ... // execute}
```

Alchemy: If a plugin fails the interface check, this would throw during installation so it cannot be installed. However, it could be useful for plugins that have a non-pure implementation of supportsInterface. The storage check instead of the ERC165 check costs ~1500 more gas for both against an EOA and to a non plugin contract. For this reason we'll accept this issue as informational, but opt to not do this fix.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 `_checkAndUpdateGasLimitUsage()` does a storage update that is sometimes reverted

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L472-L542

Description: Function `_checkAndUpdateGasLimitUsage()` updates `keyData.gasLimit.limitUsed` even if `validationSuccess == false`. As this is a storage variable its relatively expensive. Because `validationSuccess == false` the transaction will be reverted and the original value is restored. This might not be obvious to developers/reviewers.

```
function _checkAndUpdateGasLimitUsage(uint256 newUsage, SessionKeyData storage keyData) /*...*/ {
    // ...
    validationSuccess = newTotalUsage <= gasLimit;
    keyData.gasLimit.limitUsed += newUsage;
    // ...
    return (validationSuccess, validAfter);
}
```

Recommendation: Consider changing the code as below. Alternatively add a comment about the `revert` when `validationSuccess==false`.

```
  function _checkAndUpdateGasLimitUsage(uint256 newUsage, SessionKeyData storage keyData) ... {
      // ...
      validationSuccess = newTotalUsage <= gasLimit;
+     if (validationSuccess)
          keyData.gasLimit.limitUsed += newUsage;
      // ...
  }
```

Alchemy: Solved in PR 30.

Spearbit: Verified.

5.4.2 Already calculated addition can be used in the `_checkAndUpdateGasLimitUsage` function

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L501

Description: In the `_checkAndUpdateGasLimitUsage` function, when `refreshInterval` is `0`, the `keyData.gasLimit.limitUsed` is updated accordingly. However, instead of assigning the already calculated addition, it's recomputed. This will simplify the code and it will save an `SLOAD` and an `ADD` operation.

Recommendation: It is recommended to replace the `keyData.gasLimit.limitUsed += newUsage` with `keyData.gasLimit.limitUsed = newTotalUsage;`

```
  if (refreshInterval == 0) {
      // We don't have a refresh interval reset, so just check that the gas limits are not exceeded and
      // update their amounts.
      validationSuccess = newTotalUsage <= gasLimit;
-     keyData.gasLimit.limitUsed += newUsage;
+     keyData.gasLimit.limitUsed = newTotalUsage;
      // ...
```

Alchemy: Solved in PR 31.

Spearbit: Verified.

5.4.3 Function `_checkUserOpPermissions()` repeatedly retrieves `sessionKeyData.contractAccessControlType`

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L242-L323

Description: Function `_checkUserOpPermissions()` repeatedly retrieves `sessionKeyData.contractAccessControlType` in a `for` loop. Some gas can be saved by caching this.

```
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {
    function _checkUserOpPermissions(UserOperation calldata userOp) internal returns (uint256) {
       // ...
        (SessionKeyData storage sessionKeyData, SessionKeyId keyId) = _loadSessionKey(msg.sender,
↪  sessionKey);
        // ...
        for (uint256 i = 0; i < callsLength;) {
            /*...*/   && _checkCallPermissions(sessionKeyData.contractAccessControlType, /*...*/  );
            // ...
        }
        // ...
    }
}
```

Recommendation: Consider storing `sessionKeyData.contractAccessControlType` in a local variable.

Alchemy: Solved in PR 39.

Spearbit: Verified

5.4.4 `dependencies.length` can be cached earlier to save gas

Severity: Gas Optimization

Context: PluginManagerInternals.sol#L352

Description: In the `_installPlugin` function, the `dependencies.length` is cached before looping to each element. However, a few lines above, at `PluginManagerInternals#L348`, there is a check to ensure that the dependencies match the manifest, which uses the `dependencies.length`.

Recommendation: It is recommended to cache the `dependencies.length` earlier to save some gas.

Alchemy: Solved in PR 40.

Spearbit: Verified.

5.4.5 Similar functions `_onInstall()` and `updateSessionKeys()` have different checks

Severity: Gas Optimization

Context: SessionKeyPlugin.sol#L85-L116, SessionKeyPlugin.sol#L182-L201, AssociatedLinkedListSetLib.sol#L46-L82

Description: Function `_onInstall()` checks for `sessionKey == address(0)`, while the similar function `updateSessionKeys()` doesn't. This adds no risk because the function `tryAdd()` also checks for `0`, however it is unconsistent.

```
function _onInstall(bytes calldata data) internal override isNotInitialized(msg.sender) {
    // ...
    for (uint256 i = 0; i < length;) {
        address sessionKey = sessionKeysToAdd[i];
        if (sessionKey == address(0)) {
            revert InvalidSessionKey(sessionKey);
        }
        if (!_sessionKeys.tryAdd(msg.sender, CastLib.toSetValue(sessionKey))) {
            revert // ...
        }
        // ...
    }
    // ...
}
function updateSessionKeys(/*...*/) /*...*/ {
    // ...
    for (uint256 i = 0; i < length;) {
        if (!_sessionKeys.tryAdd(msg.sender, CastLib.toSetValue(sessionKeysToAdd[i]))) {
            revert // ...
        }
        // ...
    }
    // ...
}
function tryAdd(AssociatedLinkedListSet storage set, address associated, SetValue value) /*...*/ {
    bytes32 unwrappedKey = bytes32(SetValue.unwrap(value));
    if (unwrappedKey == bytes32(0)) {
        // Cannot add the zero value
        return false;
    }
    // ...
}
```

Recommendation: Consider making the implementations consistent. The `0` check in `_onInstall()` can be removed because its also done in `tryAdd`.

Alchemy: Solved in PR 41.

Spearbit: Verified.

5.4.6 `_isValidERC1271OwnerTypeSignature` can cache variables

Severity: Gas Optimization

Context: MultiOwnerPlugin.sol#L416-L432

Description: Most `for` loops cache the array length. The exception is function `_isValidERC1271OwnerTypeSignature()`. Caching saves some gas and is more consistent:

```
function _isValidERC1271OwnerTypeSignature(address associated, bytes32 digest, bytes memory signature)
    // ...
    for (uint256 i; i < owners_.length;) {
        // ...
    }
    // ...
}
```

Recommendation: Consider storing `owners_.length` in a local variable to cache it.

Alchemy: Solved in PR 42.

Spearbit: Verified.

5.4.7 Remove extra argument from `_checkCallPermissions()`

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L331

Description: `_checkCallPermissions()` takes a unnecessary argument for `value`:

```
function _checkCallPermissions(
    ContractAccessControlType accessControlType,
    SessionKeyId keyId,
    address target,
    uint256, /*value*/
    bytes memory callData
)
```

Native token spending is checked through `_checkSpendLimitUsage()`.

Recommendation: Remove the extra `uint256` argument from `_checkCallPermissions()`.

Alchemy: Solved in PR 43.

Spearbit: Verified.

5.4.8 `tryDecrement()` and `tryIncrement()` can be optimized

Severity: Gas Optimization

Context: CountableLinkedListSetLib.sol#L48-L62

Description: Functions `tryDecrement()` and `tryIncrement()` do operations on the highest byte of `flags` by splitting `flags` in two bytes first. The operations can also be done without splitting, which saves some gas.

```
function tryDecrement(LinkedListSet storage set, SetValue value) internal returns (bool) {
    // ...
    uint16 flags = set.getFlags(value);
    // Use the upper 8 bits of the (16-bit) flag for the counter.
    uint16 counter = flags >> 8;
    if (counter == 0) {
        return set.tryRemove(value);
    }
    unchecked {
        --counter;
    }
    return set.trySetFlags(value, (counter << 8) | (flags & 0xFF));
}
```

Recommendation: Consider changing the code to:

```
function tryDecrement(LinkedListSet storage set, SetValue value) internal returns (bool) {
    // ...
    uint16 flags = set.getFlags(value);
    if (flags < 0x100) {
        return set.tryRemove(value);
    }
    unchecked {
         flags -= 0x100;
    }
    return set.trySetFlags(value, flags);
}
```

Change `tryIncrement()` in a similar way.

Alchemy: Solved in PR 56.

Spearbit: Verified.

5.4.9 `tryRemoveKnown()` does redundant operations

Severity: Gas Optimization

Context: AssociatedLinkedListSetLib.sol#L139-L173, AssociatedLinkedListSetLib.sol#L412-L419, LinkedListSetLib.sol#L54-L79

Description: The function `tryRemoveKnown()` clears the flags of `next` and removes `HAS_NEXT_FLAG` if it was set. Then it adds (`OR`s) the `HAS_NEXT_FLAG` again if it was set. This is the same as using `clearUserFlags()`, because that keeps the `HAS_NEXT_FLAG`. Using `clearUserFlags()` saves some gas.

```
bytes32 constant HAS_NEXT_FLAG = bytes32(uint256(2));
function tryRemoveKnown(/*...*/) /*...*/ {
    // ...
    bytes32 next = _load(valueSlot);
    // ...
    _store(prevSlot, clearFlags(next) | /*...*/ | (next & HAS_NEXT_FLAG));
    // ...
}
function clearFlags(bytes32 val) internal pure returns (bytes32) {
    return val & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0001;
}
function clearUserFlags(bytes32 val) internal pure returns (bytes32) {
    return val & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF0003;
}
```

Recommendation: Consider changing the code of `tryRemoveKnown()` to:

```
  function tryRemoveKnown(/*...*/) /*...*/ {
      // ...
-     _store(prevSlot, clearFlags(next) | /*...*/ | (next & HAS_NEXT_FLAG));
+     _store(prevSlot, clearUserFlags(next) | /*...*/ ;
      // ...
  }
```

Function `tryRemoveKnown()` of `LinkedListSetLib` can be changed in a similar way.

Alchemy: Solved in PR 52.

Spearbit: Verified.

5.4.10 Flip the order of owner addition and removal

Severity: Gas Optimization

Context: MultiOwnerPlugin.sol#L86-L87

Description: `updateOwners(address[] memory ownersToAdd, address[] memory ownersToRemove)` first adds the new owners and then removes the specified owners:

```
_addOwnersOrRevert(_owners, msg.sender, ownersToAdd);
_removeOwnersOrRevert(_owners, msg.sender, ownersToRemove);
```

This wastes gas as `_removeOwnersOrRevert()` has to iterate over all the newly added owners.

Recommendation: Removing the owners and then adding them will save gas as `_removeOwnersOrRevert()` will not have to iterate over the newly added owners. Note that in current code, if an owner is present in both the lists, `updateOwners()` will not include it in the final owner list. However with this change, that owner will now be added to the list as addition happens after removal. This behavior should be documented in its Natspec.

Another version can be added where the `prev` is specified for cheaper removal using `tryRemoveKnown()`.

Alchemy: Solved in PR 50.

Spearbit: Verified.

5.4.11 Use cached `callsLength` instead of `calls.length`

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L255

Description: In the `_checkUserOpPermissions` function, the `calls.length` is cached in the `callsLength` variable. However, in the next line the uncached `calls.length` is used again to calculate whether the `validationSuccess` should be true or false.

Recommendation: It's recommended to use the cached `callsLength` variable instead of recalculating the length of the `calls` array.

Alchemy: Solved in PR 39.

Spearbit: Verified.

5.4.12 Flip the condition for potential gas saving

Severity: Gas Optimization

Context: SessionKeyPermissionsPlugin.sol#L321

Description: The following condition `!validationSuccess ? 1 : 0` in below snippet can be flipped to potentially save gas:

```
return uint160(!validationSuccess ? 1 : 0) | (uint256(sessionKeyData.validUntil) << 160)
    | (uint256(currentValidAfter) << (208));
```

Recommendation: Update the code as:

```
```solidity
- return uint160(!validationSuccess ? 1 : 0) | (uint256(sessionKeyData.validUntil) << 160)
+ return  uint160(validationSuccess ? 0 : 1) | (uint256(sessionKeyData.validUntil) << 160)
      | (uint256(currentValidAfter) << (208));
```

We also suggest benchmarking the changes to check if the changes aimed at gas optimizations are actually saving gas.

Alchemy: Solved in PR 63.

Spearbit: Verified.

5.4.13 `hasRequiredPaymaster` can be removed

Severity: Gas Optimization

Context: SessionKeyPermissionsBase.sol#L17

Description: `SessionKeyData` struct has two fields:

```
struct SessionKeyData {
   // ...
   bool hasRequiredPaymaster;
   // ...
   address requiredPaymaster;
}
```
```

`hasRequiredPaymaster` is equivalent to `requiredPaymaster != address(0)`. Hence, this fields can be removed and its usage can be replaced with the equivalent check. This removes the need to have two parameters to maintain for paymaster.

Recommendation: Remove `hasRequiredPaymaster` from `SessionKeyData` struct. Remove all its assignments and replace its usage with `requiredPaymaster != address(0)`. Update `_setRequiredPaymaster()` as:

```
function _setRequiredPaymaster(SessionKeyData storage sessionKeyData, address requiredPaymaster)
↪  internal {
    sessionKeyData.requiredPaymaster = requiredPaymaster;
}
```

Alchemy: Using `hasRequiredPaymaster` allows us to potentially save an extra sload when native token limits are bypassed, and `hasRequiredPaymaster` is false. There aren't any great ways to pack/arrange this struct more efficiently without guessing at future usage patterns. We'll keep this as is.

Spearbit: Acknowledged.

5.4.14 `AssociatedLinkedListSetLib.getAll()` iterates over the loop just to get its size

Severity: Gas Optimization

Context: AssociatedLinkedListSetLib.sol#L354

Description: `AssociatedLinkedListSetLib.getAll()` returns the entire list in memory. To do that it first gets the total number of elements in the list by iterating of the list and incrementing `count` by 1. Then it does a for-loop from `0` to `count` and loads all elements from storage to memory. This makes it a gas intensive operation.

Recommendation: You avoid this extra loop to get `count` by either maintaining `count` for each list or using assembly as done for `LinkedListSetLib.getAll()`.

Alchemy: Fixed in PR 115.

Spearbit: Verified.

5.4.15 Unnecessary assignments in assembly

Severity: Gas Optimization

Context: AssociatedLinkedListSetLib.sol#L431-L432

Description: There is no need to introduce `keyWord2` in the following assembly code:

```
assembly ("memory-safe") {
    // Store the value in the last word.
    let keyWord2 := value
    mstore(add(keyBuffer, 0x60), keyWord2)
    slot := keccak256(keyBuffer, 0x80)
}
```

Recommendation: Update the code as follows:

```
  assembly ("memory-safe") {
      // Store the value in the last word.
-     let keyWord2 := value
-     mstore(add(keyBuffer, 0x60), keyWord2)
+     mstore(add(keyBuffer, 0x60), value)
      slot := keccak256(keyBuffer, 0x80)
  }
```

Alchemy: Solved in PR 54.

Spearbit: Verified.

5.4.16 Redundant `SSTORE`s in `clear()`

Severity: Gas Optimization

Context: LinkedListSetLib.sol#L128, AssociatedLinkedListSetLib.sol#L192-L193

Description: `LinkedListSetLib.clear()` and `AssociatedLinkedListSetLib.clear()` do redundant operations to map `SENTINEL_VALUE` to 0. This is already done in the first iteration of the preceding do-while loop. For example, consider `LinkedListSetLib.clear()`:

```
bytes32 cursor = SENTINEL_VALUE;

do {
    bytes32 next = clearFlags(map[cursor]);
    map[cursor] = bytes32(0);
    cursor = next;
} while (!isSentinel(cursor) && cursor != bytes32(0));

map[SENTINEL_VALUE] = bytes32(0);
```

For the first iteration, `cursor = SENTINEL_VALUE`, hence `map[cursor] = bytes32(0)` maps `SENTINEL_VALUE` to 0. The loop never reassigns it as it exits as soon as `cursor` reaches `SENTINEL_VALUE` again. Hence the same assignment after the loop is not necessary.

Recommendation: Delete the redundant assignments:

- LinkedListSetLib.sol#L128:

```
- map[SENTINEL_VALUE] = bytes32(0);
```

- AssociatedLinkedListSetLib.sol#L192-L193:

```
- StoragePointer sentinelSlot = _mapLookup(keyBuffer, SENTINEL_VALUE);
- _store(sentinelSlot, bytes32(0));
```

Alchemy: Solved in PR 51.

Spearbit: Verified.

5.4.17 Redundant check in `validateUserOp()`

Severity: Gas Optimization

Context: UpgradeableModularAccount.sol#L178-L193, UpgradeableModularAccount.sol#L776-L781

Description: The function `validateUserOp()` checks `userOp.callData.length < 4` and function `_selectorFromCallData()` checks it again.

```
function validateUserOp(/*...*/) /*...*/ {
    // ...
    if (userOp.callData.length < 4) {
        revert UnrecognizedFunction(bytes4(userOp.callData));
    }
    bytes4 selector = _selectorFromCallData(userOp.callData);
    // ...
}
function _selectorFromCallData(bytes calldata data) internal pure returns (bytes4) {
    if (data.length < 4) {
        revert UnrecognizedFunction(bytes4(data));
    }
    // ...
}
```

Recommendation: Consider removing the check in `validateUserOp()`:

```
  function validateUserOp(/*...*/) /*...*/ {
      // ...
-     if (userOp.callData.length < 4) {
-         revert UnrecognizedFunction(bytes4(userOp.callData));
      }
      // ...
  }
```

Alchemy: Solved in PR 53.

Spearbit: Verified.

5.4.18 Function `initialize()` can use `calldata`

Severity: Gas Optimization

Context: UpgradeableModularAccount.sol#L99

Description: The function `initialize()` is `external` and has a `memory` parameter. This could also be `calldata` to save some gas.

```
function initialize(address[] memory plugins, bytes calldata pluginInitData) /*...*/ {
    // ...
}
```

Recommendation: Consider changing the code to:

```
- function initialize(address[] memory plugins, bytes calldata pluginInitData) /*...*/ {
+ function initialize(address[] calldata plugins, bytes calldata pluginInitData) /*...*/ {
      // ...
  }
```

Alchemy: Solved in PR 18.

Spearbit: Verified.

5.4.19 `_installPlugin()` repeatedly uses `manifest.permittedCallHooks[i]`

Severity: Gas Optimization

Context: PluginManagerInternals.sol#L570-L588

Description: The function `_installPlugin()` repeatedly uses `manifest.permittedCallHooks[i]`. This can be optimized, as is done in other locations of the same function:

```
function _installPlugin(
    // ...
    _addPermittedCallHooks(
        manifest.permittedCallHooks[i].executionSelector,
        /*...*/ ,
        _resolveManifestFunction(
            manifest.permittedCallHooks[i].preExecHook,
            // ...
        ),
        _resolveManifestFunction(
            manifest.permittedCallHooks[i].postExecHook,
            // ...
        )
    );
    // ...
}
```

Recommendation: Consider setting and using:

```
ManifestExecutionHook memory mh = manifest.permittedCallHooks[i];
```

Alchemy: The `permittedCallHooks` functionality was removed in PR 76, which resolves this suggestion.

Spearbit: Verified.

5.4.20 Check in `_installPlugin()` can be done earlier

Severity: Gas Optimization

Context: PluginManagerInternals.sol#L461-L463

Description: Function `_installPlugin()` does a check that could be done earlier. This is more logical and saves some gas in case of a revert.

```
function _installPlugin(/*...*) /*...*/ {
    // ...
    for (uint256 i = 0; i < length;) {
        // ...
        storage_.pluginData[plugin].injectedHooks[i] = StoredInjectedHook({providingPlugin:
↪  hook.providingPlugin, /*...*/ } );
        storage_.pluginData[hook.providingPlugin].dependentCount += 1;
        if (!storage_.plugins.contains(CastLib.toSetValue(hook.providingPlugin))) {
            revert MissingPluginDependency(hook.providingPlugin);
        }
        // ...
    }
    // ...
}
```

Recommendation: Consider changing the code to:

```
  function _installPlugin(/*...*/) /*...*/ {
      // ...
      for (uint256 i = 0; i < length;) {
          // ...
+         if (!storage_.plugins.contains(CastLib.toSetValue(hook.providingPlugin))) {
+             revert MissingPluginDependency(hook.providingPlugin);
+         }
          storage_.pluginData[plugin].injectedHooks[i] = StoredInjectedHook({providingPlugin:
↪  hook.providingPlugin, ... } );
          storage_.pluginData[hook.providingPlugin].dependentCount += 1;
-         if (!storage_.plugins.contains(CastLib.toSetValue(hook.providingPlugin))) {
-             revert MissingPluginDependency(hook.providingPlugin);
-         }
          // ...
      }
      // ...
  }
```

Alchemy: Solved in PR 33.

Spearbit: Verified.

5.4.21 Clearing upper bits can be done in a more efficient way

Severity: Gas Optimization

Context: AccountExecutor.sol#L118-L131, UUPSUpgradeable.sol#L86

Description: The function `_updatePluginCallBufferSelector()` uses `and` to clear upper and lower bits. Contract `UUPSUpgradeable` uses a cheaper way to clear upper bits as well as lower bits.

```
function _updatePluginCallBufferSelector(bytes memory buffer, bytes4 pluginSelector) internal pure {
    assembly ("memory-safe") {
        // ...
        // Clear the upper 4 bytes of the existing word
        existingWord := and(existingWord, shr(32, not(0)))
        // ...
        // Clear the lower 28 bytes of the selector
        pluginSelector := and(pluginSelector, shl(224, 0xFFFFFFFF))
    }
}
```

Recommendation: Consider changing the code to:

```
- existingWord := and(existingWord, shr(32, not(0)))
+ existingWord := shr(32, shl(32, existingWord))
  // ...
- pluginSelector := and(pluginSelector, shl(224, 0xFFFFFFFF))
+ pluginSelector := shl(224, shr(224, pluginSelector)) // note shr and shl are reversed here
```

Alchemy: Solved in PR 55.

Spearbit: Verified.

5.5 Informational

5.5.1 Validation reverts and `SIG_VALIDATION_FAILED` are not differentiated

Severity: Informational

Context: AccountExecutor.sol#L94-L97, SessionKeyPlugin.sol#L160-L179, MultiOwnerPlugin.sol#L205-L226

Description: To fully comply with the ERC-4337 specification, `validateUserOp()` should return `SIG_VALIDATION_FAILED` when an invalid signature is still syntactically correct, and all other signature errors should revert. This is not implemented perfectly in the current plugins, since only an incorrect `uint8 functionId` will cause a revert. Also, this behavior is incorrectly handled in the account itself, since the `_executeUserOpPluginFunction()` function catches reverts and translates them to `SIG_VALIDATION_FAILED`.

Recommendation: Update each `userOpValidationFunction()` to revert in scenarios where the `userOp.signature` is not syntactically correct. Also, change `_executeUserOpPluginFunction()` to bubble up reverts instead of translating them to `SIG_VALIDATION_FAILED`.

Alchemy: Solved in PR 46 and PR 57.

Spearbit: Verified.

5.5.2 Using || for bitwise OR operations might be confusing

Severity: Informational

Context: AccountStorageV1.sol#L23, AccountStorageV1.sol#L25

Description: In `AccountStorageV1`, the structure of the keys is explained in inline comments. However, to demonstrate the structure, || is used instead of | for bitwise OR operations.

```
mapping(bytes4 => SelectorData) selectorData;
// bytes24 key = address(calling plugin) || bytes4(selector of execution function)
mapping(bytes24 => PermittedCallData) permittedCalls;
// key = address(calling plugin) || target address
mapping(IPlugin => mapping(address => PermittedExternalCallData)) permittedExternalCalls;
```

Recommendation: To enhance the readability of the inline comments, it is advisable to replace || with |.

Alchemy: We use || as a concatenate operator. It's more common for these to appear in technical documents - some EIPs use || as concat, and especially so in academic cryptography, it seems to originate from set theory notation. It's definitely confusing since in most programming languages, || means logical OR. Solved by adding the comment "|| for variables in comments refers to the concat operator" in PR 64.

Spearbit: Verified.

5.5.3 Redundant comment in the _revertOnRuntimePluginFunctionFail function

Severity: Informational

Context: AccountExecutor.sol#L211

Description: The comment in the #L211 appears to be left as a duplicate line from #L210. This comment seems to be redundant and it can be removed.

Recommendation: It's recommended to remove the comment at AccountExecutor.sol#L211.

Alchemy: Fixed in PR 74.

Spearbit: Verified.

5.5.4 ERC-6900 specification not clear on duplicate hooks

Severity: Informational

Context: MSCA-Specs-Manual

Description: The ERC-6900 specification does not currently define explicitly the way an account should behave for duplicate hooks.

The MSCA-Specs-Manual documents choices, which are implemented in the code. However other smart contract accounts could make different choices. It would be helpful to make a recommendation here in the future.

Recommendation: Consider making the ERC-6900 specification more explicit on duplicate hooks.

Alchemy: With the fix for `MSCA-16 Lack of Function Call Context for Post-Execution Hooks` we're removing the ability for hooks to be provided as dependencies as described in Hook Simplification Proposal. This should eliminate the cases where duplicate hooks can occur, except for when a plugin applies the same hook twice within its manifest (most likely a bad config), or when one of the magic values are used. Still, agreed it'd be good to provide explicit guidance on this in the standard.

Spearbit: Acknowledged.

5.5.5 `UUPSUpgradeable` has a different `solidity` version

Severity: Informational

Context: UUPSUpgradeable.sol#L2

Description: The (forked) contract `UUPSUpgradeable` is the only contract with a different `solidity` version. This is the same as in the original version of the contract. Just added here because there is a difference, there is no real risk in using this.

```
pragma solidity ^0.8.4;
abstract contract UUPSUpgradeable {
}
```

Recommendation: To be consistent the `solidity` version could be changed. Alternatively its also a good approach to keep the forked contract as close to the original as possible.

Alchemy: Acknowledged, we prefer to let Solady use this audit report without needing any changes on their end, so will keep it as ^0.8.4.

Spearbit: Acknowledged.

5.5.6 `_updateSpendLimits()` starts a new `interval`

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L811-L833

Description: Function `_updateSpendLimits()` starts a new `interval`, so its important to be careful with these updates. For example if an update is done when 20% of the time has passed and 100% of the funds for the `interval` have been spend, then a new `interval` starts, which allows spending 100% again.

```
function _updateSpendLimits( /*...*/ ) /*...*/ {
    // ...
    timeInfo.refreshInterval = newRefreshInterval;
    // ...
    timeInfo.lastUsed = uint48(block.timestamp);
}
```

Recommendation: Consider adding a comment about this risk.

Alchemy: Fixed in PR 75.

Spearbit: Verified.

5.5.7 Make linked list `clear()` implementations more consistent

Severity: Informational

Context: AssociatedLinkedListSetLib.sol#L179-L194

Description: The AssociatedLinkedListSetLib.sol `clear()` function is as follows:

```
function clear(AssociatedLinkedListSet storage set, address associated) internal {
    TempBytesMemory keyBuffer = _allocateTempKeyBuffer(set, associated);

    bytes32 cursor = SENTINEL_VALUE;

    do {
        bytes32 cleared = clearFlags(cursor);
        StoragePointer cursorSlot = _mapLookup(keyBuffer, cleared);
        bytes32 next = _load(cursorSlot);
        _store(cursorSlot, bytes32(0));
        cursor = next;
    } while (!isSentinel(cursor) && cursor != bytes32(0));

    StoragePointer sentinelSlot = _mapLookup(keyBuffer, SENTINEL_VALUE);
    _store(sentinelSlot, bytes32(0));
}
```

Technically, this differs from the `clear()` function in `LinkedListSetLib`, since the cursor's flags are not cleared before the `while` condition. Fortunately, this doesn't cause any problems unless the cursor is only flag bits, which is currently impossible.

Recommendation: Consider making the implementations more consistent by clearing the flags before the `while` condition in the following way:

```
function clear(AssociatedLinkedListSet storage set, address associated) internal {
    TempBytesMemory keyBuffer = _allocateTempKeyBuffer(set, associated);

    bytes32 cursor = SENTINEL_VALUE;

    do {
        StoragePointer cursorSlot = _mapLookup(keyBuffer, cursor);
        bytes32 next = clearFlags(_load(cursorSlot));
        _store(cursorSlot, bytes32(0));
        cursor = next;
    } while (!isSentinel(cursor) && cursor != bytes32(0));

}
```

Also see issue "Redundant `SSTORE`s in `clear()`".

Alchemy: Fixed in PR 69.

Spearbit: Verified.

5.5.8 `SessionKeys` can collect a large `allowance` while staying under the radar

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L359-L414, SessionKeyPermissionsPlugin.sol#L549-L589, SessionKeyPermissionsPlugin.sol#L603-L673

Description: A `sessionKey` can stay *under the radar* and maximize its `allowance` by increasing its `allowance` every `interval`. No `tokens` are moved yet, but the `allowance` steadily increases. Then suddenly an unexpected large amount of tokens can be used by the `sessionKey`.

```
function _updateLimitsPreExec(address account, bytes calldata callData) internal {
    // ...
    if (contractData.isERC20WithSpendLimit) {
        // ...
        uint256 spendAmount = _getTokenSpendAmount(account, call.target, call.data);
        if (  !_runtimeUpdateSpendLimitUsage(
```

```
                spendAmount, contractData.erc20SpendLimitTimeInfo, contractData.erc20SpendLimit
            )  ) {
            revert ERC20SpendLimitExceeded(msg.sender, sessionKey, call.target);
        }
    }
    // ...
}
function _getTokenSpendAmount(address account, address token, bytes memory callData) /*...*/ {
   // ...
    } else if (selector == IERC20.approve.selector) {
        // ...
        uint256 existingAllowance = IERC20(token).allowance(account, spender);
        // ...
        approveAmount := mload(add(callData, 68))
        // ...
        return approveAmount - existingAllowance;  // so the increase in approval
        // ...
    }
}
function _runtimeUpdateSpendLimitUsage(uint256 newUsage, /*...*/) /*...*/ {
    uint256 currentUsage = limit.limitUsed;
    if (/*...*/ lastUsed + refreshInterval > block.timestamp) {
        // ...
        newTotalUsage = newUsage + currentUsage;
        // ...
        if (/*...*/ newTotalUsage > spendLimit) {
            // If we overflow, or if the limit is exceeded, fail here and revert in the parent context.
            return false;
        }
        limit.limitUsed = newTotalUsage;
    } else {
        // reset limit.limitUsed;
    }
    // ...
}
```

Recommendation: Consider monitoring for this behaviour, for example showing the total allowance in a dashboard of a wallet user interface.

Alchemy: Solved in PR 28. We now decrease the session key's limits based on the `uint256` in the `approve`, disregarding the actual increase in allowance. Thus the session key can only have a maximum allowance of the given limit.

Spearbit: Verified.

5.5.9 No limits on the validity of `sessionKey`

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L44-L54

Description: There are no limits on the validity of `sessionKey`, that means that in practice there is also no reason to use `rotateKey()`.

```
function registerKey(address sessionKey, bytes32 tag) external override {
    // ...
}
```

Recommendation: Consider having a `validAfter`/`validUntil` on the `sessionKey` level.

Alchemy: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Requirements for `_coalescePreValidation()` can be better enforced

Severity: Informational

Context: ValidationDataHelpers.sol#L5-L26, UpgradeableModularAccount.sol#L504-L574

Description: The function `_coalescePreValidation()` only works correctly if `uint160(validationData1)` and `uint160(validationData2)` are `0` or `1`. This is enforced in `_doUserOpValidation()` which is currently also the only functions that calls `_coalescePreValidation()`.

If the function `_coalescePreValidation()` is used elsewhere, this might not be enforced.

```
function _doUserOpValidation(
    // ...
    for (uint256 i = 0; i < preUserOpValidationHooksLength;) {
        // ...
        if (uint160(currentValidationData) > 1) {
            // If the aggregator is not 0 or 1, it is an unexpected value
            revert UnexpectedAggregator(plugin, functionId, address(uint160(currentValidationData)));
        }
        validationData = _coalescePreValidation(validationData, currentValidationData);
        // ...
    }
    // ...
}
function _coalescePreValidation(uint256 validationData1, uint256 validationData2) ... {
    // ...
    // Once we know that the authorizer field is 0 or 1, we can safely bubble up SIG_FAIL with bitwise
↪   OR
    resValidationData |= uint160(validationData1) | uint160(validationData2);
}
```

Recommendation: Consider moving the check to `_coalescePreValidation()`. Alternative document the requirements at function `_coalescePreValidation()`.

```
  function _doUserOpValidation(
      // ...
      for (uint256 i = 0; i < preUserOpValidationHooksLength;) {
          // ...
-         if (uint160(currentValidationData) > 1) {
-             // If the aggregator is not 0 or 1, it is an unexpected value
-             revert UnexpectedAggregator(plugin, functionId, address(uint160(currentValidationData)));
-         }
          validationData = _coalescePreValidation(validationData, currentValidationData);
          // ...
      }
      // ...
  }
  function _coalescePreValidation(uint256 validationData1, uint256 validationData2) /*...*/ {
      // ...
-     // Once we know that the authorizer field is 0 or 1, we can safely bubble up SIG_FAIL with
↪   bitwise OR
      resValidationData |= uint160(validationData1) | uint160(validationData2);
+     if (uint160(resValidationData ) > 1) {
+         // If the aggregator is not 0 or 1, it is an unexpected value
+         revert UnexpectedAggregator(address(uint160(currentValidationData)));
+     }
+     // now we know both `uint160(validationData1)` and `uint160(validationData2)` were either `0` or
↪   `1`
  }
```

Alchemy: Fixed in PR 78 by documenting this requirement in `_coalescePreValidation()`.

Spearbit: Verified.

5.5.11 `_SIG_VALIDATION_PASSED` / `_SIG_VALIDATION_FAILED` not used optimally

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L242-L323, SessionKeyPlugin.sol#L41-L42, MultiOwnerPlugin.sol#L63-L66, UpgradeableModularAccount.sol#L68, ValidationDataHelpers.sol#L29-L49

Description: On several place the values of `0` and `1` are used to indicate signature validation/failure. This could be replaced by the constants `_SIG_VALIDATION_PASSED` / `_SIG_VALIDATION_FAILED` to improve readability.

These constants are defined in several contracts. For maintenance its easier to put the in one file. The constant `_SIG_VALIDATION_FAILED` is also defined in `UpgradeableModularAccount` but its is not used there.

```
function _checkUserOpPermissions(UserOperation calldata userOp) internal returns (uint256) {
    // ...
    // Validation return data is 1 in the case of an invalid signature,
    // otherwise a packed struct of the aggregator address (0 here), ...
    // ...
    return uint160(!validationSuccess ? 1 : 0) | (uint256(sessionKeyData.validUntil) << 160)
        | (uint256(currentValidAfter) << (208));
}
contract MultiOwnerPlugin is BasePlugin, IMultiOwnerPlugin, IERC1271, EIP712 {
    // ERC-4337 specific value: signature validation passed
    uint256 internal constant _SIG_VALIDATION_PASSED = 0;
    // ERC-4337 specific value: signature validation failed
    uint256 internal constant _SIG_VALIDATION_FAILED = 1;
}
contract SessionKeyPlugin is BasePlugin, ISessionKeyPlugin {
    uint256 internal constant _SIG_VALIDATION_PASSED = 0;
    uint256 internal constant _SIG_VALIDATION_FAILED = 1;
}
contract UpgradeableModularAccount is ... {
    uint256 internal constant _SIG_VALIDATION_FAILED = 1;
}
function _coalesceValidation(uint256 preValidationData, uint256 validationData) ... {
     // ...
     resValidationData |= uint160(preValidationData) == 1 ? 1 : uint160(validationData);
}
```

Recommendation: Consider the following:

- Moving the definitions of `_SIG_VALIDATION_PASSED` / `_SIG_VALIDATION_FAILED` to a central place.
- Replacing the relevant values of `0` and `1` with `_SIG_VALIDATION_PASSED` / `_SIG_VALIDATION_FAILED`.
- Removing `_SIG_VALIDATION_FAILED` from `UpgradeableModularAccount`.

Alchemy: Solved in PR 73.

Spearbit: Verified.

5.5.12 `_checkUserOpPermissions()` uses a special `nonce` which requires detailed knowledge

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L242-L323

Description: Function `_checkUserOpPermissions()` requires a special `nonce` in some situations. There is no straightforward way to create this `nonce` without knowing the implementation details.

```
function _checkUserOpPermissions(UserOperation calldata userOp) internal returns (uint256) {
    // ...
    if (sessionKeyData.hasGasLimit) {
        // ...
        if (uint192(userOp.nonce >> 64) != uint192(uint160(sessionKey))) {
            validationSuccess = false;
        }
    }
    // ...
}
```

Recommendation: Consider adding a `loupe` function that generates a `nonce` in combination with the `sessionKey`.

Alchemy: Plugins in `ERC-6900` are not tied to a specific `EntryPoint`, and accounts do not have a mandatory interface function for exposing their chosen `EntryPoint` contract. So, the only way to handle this would be to install a new execution function on the account, which raises the gas cost of installation, and thus we've chosen not to add a loupe function for it. As such, we will acknowledge that the special nonce is required to be computed by the client manually.

Spearbit: Acknowledged.

5.5.13 `_checkUserOpPermissions()` doesn't do checks on ERC20 tokens

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L242-L323

Description: The function `_checkUserOpPermissions()` of `SessionKeyPermissionsPlugin` doesn't do checks on ERC20 tokens. This also means `spendLimitValidAfter` of ERC20 tokens isn't used. Developers wanting to use this functionality might find themselves perplexed by its nonfunctional behavior..

```
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {
    function _checkUserOpPermissions(UserOperation calldata userOp) internal returns (uint256) {
        // no checks on ERC20 tokens

        return uint160(!validationSuccess ? 1 : 0) | (uint256(sessionKeyData.validUntil) << 160)
            | (uint256(currentValidAfter) << (208));
    }
    // ...
```

Recommendation: Consider adding a comment to function `_checkUserOpPermissions()` about this.

Alchemy: Fixed in PR 79.

Spearbit: Verified.

5.5.14 Plugin `permissions` are not standardized

Severity: Informational

Context: MultiOwnerPlugin.sol#L353

Description: The function `pluginMetadata()` exports strings that show the required `permissions` to the user. They are currently not standardized which makes it more difficult to reason about the permissions, both for users and for check scripts.

```
function pluginMetadata() external pure virtual override returns (PluginMetadata memory) {
    //...
    string memory modifyOwnershipPermission = "Modify Ownership";
    // ...
}
```

Recommendation: Consider standardizing the permissions. For an example see: Android Manifest.permission.

Alchemy: Acknowledge that this should have some format, but will keep it as is and revisit in a future standard improvement.

Spearbit: Acknowledged.

5.5.15 `preUserOpValidationHook()` and `preExecutionHook()` use a different default return value

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L96-L116

Description: Functions `preUserOpValidationHook()` and `preExecutionHook()` use a different default return value in case the `functionId` isn't found. Function `preUserOpValidationHook()` uses `revert` and `preExecutionHook()` uses `return`. This is inconsistent.

```
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {
    function preUserOpValidationHook(uint8 functionId, UserOperation calldata userOp, bytes32) /*...*/ {
        if (functionId == /*...*/ ) {
            return // ...
        }
        revert NotImplemented();
    }
    function preExecutionHook(uint8 functionId, address, uint256, bytes calldata data) /*...*/ {
        if (functionId == /*...*/ ) {
            // ...
        }
        return "";
    }
}
```

Recommendation: Consider using the same default for these functions.

Alchemy: Solved in PR 47.

Spearbit: Verified.

5.5.16 Race conditions with `updateKeyPermissions()`

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L67-L77

Description: There are some race conditions with the use of `registerKey()` and `updateKeyPermissions()`:

- The default value for `gasLimit` is `false`. The default value for `validAfter` / `validUntil` == 0, which is interpreted as always valid. So as soon as a `sessionKey` is registered via `registerKey()` it can start using gas, before any other `updateKeyPermissions()` have been done. Although by default no external calls are allowed, `executeWithSessionKey()` could use an empty array of of calls and still use gas.
- If `updateKeyPermissions()` uses seperate function calls, the permissions could be temporarily in an inconsistent state. This is especially relevant if the sessionKey is already in use.

```
function updateKeyPermissions(address sessionKey, bytes[] calldata updates) external override {
    // ...
}
```

Recommendation: Be careful of the order of updates. Preferably use `executeBatch()` for the combination of `registerKey()` and `updateKeyPermissions()` if its important to limit gas. It might be helpful have a default gas limit to prevent the race condition directly after `registerKey()`.

Alchemy: Solved by adding atomic permissions setup to `addSessionKey`, removing the need for `executeBatch`, in PR 80.

Spearbit: Verified.

5.5.17 `oldSessionKey` can be reused

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L57-L64

Description: After using `rotateKey()`, an `oldSessionKey` can immediately be reused again, via `registerKey()`. This will generate a new `sessionKeyId`. Could be confusing because any users of the `oldSessionKey` then have different permissions.

```
function rotateKey(address oldSessionKey, address newSessionKey) external override {
   SessionKeyId keyId = _sessionKeyIdOf(msg.sender, oldSessionKey);
   _assertRegistered(keyId, oldSessionKey);
   _updateSessionKeyId(msg.sender, oldSessionKey, SessionKeyId.wrap(bytes32(0)));
   _updateSessionKeyId(msg.sender, newSessionKey, keyId);
   // ...
}
```

Recommendation: If this is undesirable to reuse `sessionKeys` the following can be done: Set `_updateSessionKeyId(msg.sender, oldSessionKey, ...)` to a special value, which invalidates the `sessionKey`.

Alchemy: For flexibility and simplicity, we won't be permanently invalidating old session key addresses. We have explained the behavior of reused keys after removal or rotation in our documentation.

Spearbit: Acknowledged.

5.5.18 No function to deregister / invalidate a `sessionKey`

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L44-L54

Description: `SessionKeyPermissionsPlugin` has a way to register `sessionKeys`, via `registerKey()`, but there is no way to undo this.

```
function registerKey(address sessionKey, bytes32 tag) external override {
    // ...
}
```

Recommendation: Consider having a function to deregister / invalidate a `sessionKey`.

Alchemy: The function `removeSessionKey` was added in PR 62, along with key permissions being deleted during uninstallation, providing two ways to de-register a key.

Spearbit: Verified.

5.5.19 Functions of `SessionKeyPermissionsLoupe()` could check `validAfter` / `validUntil`

Severity: Informational

Context: SessionKeyPermissionsLoupe.sol#L44-L51

Description: The functions in `SessionKeyPermissionsLoupe()` don't take into account the `validAfter` / `validUntil`. The result could be given only if `block.timestamp` is within the range of `validAfter` / `validUntil`. However this will take more gas and might be limiting.

```
function getKeyTimeRange(address account, address sessionKey) /*...*/ {
    (SessionKeyData storage sessionKeyData,) = _loadSessionKey(account, sessionKey);
    return (sessionKeyData.validAfter, sessionKeyData.validUntil);
}
```

Recommendation: Consider checking `validAfter` / `validUntil` in functions of `SessionKeyPermissionsLoupe()`.

Alchemy: We won't be adding this check to the `loupe` functions, because we think it's useful to be able to inspect a key's permissions before it is active or after it is inactive, and don't want to create multiple loupe function definitions.

Spearbit: Acknowledged.

5.5.20 Two ways to retrieve `keyId`

Severity: Informational

Context: SessionKeyPermissionsLoupe.sol#L19-L51, SessionKeyPermissionsLoupe.sol#L78-L100, SessionKeyPermissionsBase.sol#L139-L147

Description: There are two ways to retrieve the `keyId`:

- Via `_loadSessionKey`, which has the extra gas overhead of also retrieving and then ignoring `_sessionKeyDataOf()`.
- Via `_sessionKeyIdOf`, which then also requires calling `_assertRegistered()`.

This is not consistent and not always optimal.

```
function getERC20SpendLimitInfo(address account, address sessionKey, address token) /*...*/ {
    (, SessionKeyId keyId) = (account, sessionKey);
    // ...
}
function getAccessControlEntry(address account, address sessionKey, address contractAddress) /*...*/ {
    SessionKeyId keyId = _sessionKeyIdOf(account, sessionKey);
    _assertRegistered(keyId, sessionKey);
   // ...
}
function getRequiredPaymaster(address account, address sessionKey) external view returns (address) {
    SessionKeyId id = _sessionKeyIdOf(account, sessionKey);
    _assertRegistered(id, sessionKey);
    SessionKeyData storage sessionKeyData = _sessionKeyDataOf(account, id);
    // ...
}
function _loadSessionKey(address associated, address sessionKey) /*...*/ {
    SessionKeyId id = _sessionKeyIdOf(associated, sessionKey);
    _assertRegistered(id, sessionKey);
    return (_sessionKeyDataOf(associated, id), id);
}
```

Recommendation: Consider making retrieving the `keyId`. Possibly use an extra function like:

```
function assertedSessionKeyId(address associated, address sessionKey) internal view returns
↪  SessionKeyId keyId {
    SessionKeyId id = _sessionKeyIdOf(associated, sessionKey);
    _assertRegistered(id, sessionKey);
    return id;
}
```

Function `getRequiredPaymaster()` can be changed in the following way:

```
  function getRequiredPaymaster(address account, address sessionKey) external view returns (address) {
-     SessionKeyId id = _sessionKeyIdOf(account, sessionKey);
-     _assertRegistered(id, sessionKey);
-     SessionKeyData storage sessionKeyData = _sessionKeyDataOf(account, id);
+     (SessionKeyData storage sessionKeyData,) = _loadSessionKey(account, sessionKey);
     // ...
  }
```

Alchemy: Solved in PR 93.

Spearbit: Verified.

5.5.21 Variable name `modifyOwnershipPermission` not logical

Severity: Informational

Context: SessionKeyPlugin.sol#L281-L298, SessionKeyPermissionsPlugin.sol#L208-L225

Description: The functions `pluginMetadata()` in `SessionKeyPlugin` and `SessionKeyPermissionsPlugin` have a variable named `modifyOwnershipPermission`, which doesn't seem logical.

```
contract SessionKeyPlugin is BasePlugin, ISessionKeyPlugin {
    function pluginMetadata() external pure virtual override returns (PluginMetadata memory) {
        // ...
        string memory modifyOwnershipPermission = "Modify Session Keys";
        // ...
    }
}
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {
    function pluginMetadata() external pure virtual override returns (PluginMetadata memory) {
        // ...
        string memory modifyOwnershipPermission = "Modify Session Key Permissions";
        // ...
    }
}
```

Recommendation: Consider changing the variable name to a more generic name:

```
- string memory modifyOwnershipPermission = // ...
+ string memory permission = // ...
```

Alchemy: Solved in PR 62.

Spearbit: Verified.

5.5.22 The `interfaceIds` array of the manifest may contain duplicate elements

Severity: Informational

Context: PluginManagerInternals.sol#L596-L602

Description: When a plugin is installed, the interface IDs associated with the plugin are activated for the account. This is achieved by iterating through the `interfaceIds` array of the manifest and updating the interfaceId in the `supportedInterfaces` mapping. However, the current implementation lacks a mechanism to check for duplicate entries in the `interfaceIds` array.

Recommendation: Although duplicate elements in the `interfaceIds` array do not currently cause issues for the account, this could be mitigated by ordering the `interfaceIds` within the manifest and checking that they are ordered and not equal to the previous one.

Alchemy: We anticipate some larger future changes around dependencies and the plugin manifest in ERC-6900. We will acknowledge this issue and revisit after standard improvements.

Spearbit: Acknowledged.

5.5.23 Linking to other plugins is error-prone

Severity: Informational

Context: SessionKeyPlugin.sol#L212-L216, IMultiOwnerPlugin.sol#L7-L10, SessionKeyPluginWithMultiOwner.t.sol

Description: Linking to other plugins seems error-prone: this could lead to confusion/mistakes/compatibility errors/ requirements for expensive reviews. A perfectly safe and audited plugin can still introduce security risks by using the wrong deployment parameters. The main goal of the manifests is to prevent that from happening.

This shows the configuration that is required, in a test in SessionKeyPluginWithMultiOwner.t.sol:

```
contract SessionKeyPluginWithMultiOwnerTest is Test {
    dependencies[0] = FunctionReferenceLib.pack(
         address(multiOwnerPlugin),
         uint8(IMultiOwnerPlugin.FunctionId.USER_OP_VALIDATION_OWNER)
    );
    dependencies[1] = FunctionReferenceLib.pack(
        address(multiOwnerPlugin),
        uint8(IMultiOwnerPlugin.FunctionId.RUNTIME_VALIDATION_OWNER_OR_SELF));
    // ...
    /*...*/.installPlugin({/*...*/, dependencies: dependencies, /*...*/});
    // ...
}
```

```
interface IMultiOwnerPlugin {
    enum FunctionId {
        RUNTIME_VALIDATION_OWNER_OR_SELF, // require owner or self access
        USER_OP_VALIDATION_OWNER // require owner access
    }
}
contract SessionKeyPlugin is BasePlugin, ISessionKeyPlugin {
    uint256 internal constant _MANIFEST_DEPENDENCY_INDEX_OWNER_USER_OP_VALIDATION = 0;
    uint256 internal constant _MANIFEST_DEPENDENCY_INDEX_OWNER_RUNTIME_VALIDATION = 1;

    function pluginManifest() external pure override returns (PluginManifest memory) {
        // ...
        manifest.dependencyInterfaceIds[_MANIFEST_DEPENDENCY_INDEX_OWNER_USER_OP_VALIDATION] =
            type(IPlugin).interfaceId;
        manifest.dependencyInterfaceIds[_MANIFEST_DEPENDENCY_INDEX_OWNER_RUNTIME_VALIDATION] =
            type(IPlugin).interfaceId;
        // ...
    }
}
```

Recommendation: Consider standardizing the `functionIds`. This probably requires expanding the size to at least `bytes4`. Plugins should be able to indicate which `functionIds` they support, possible via a dedicated / standardized `interfaceId`.

Possibly combine this with standardizing a modules registry. See this example.

Also see the issue "Plugin `permissions` are not standardized".

Alchemy: We acknowledge this issue, but will not make a fix for now

Spearbit: Acknowledged.

5.5.24 Some struct elements can be better documented

Severity: Informational

Context: SessionKeyPermissionsBase.sol#L11-L28

Description: The `SessionKeyPermissionsBase` abstract contract includes several structs that are employed in the `SessionKeyPermissionsPlugin` contract. However, the documentation for these struct elements is incomplete. Currently, only a few elements are documented, leaving the rest without detailed explanations:

```
struct SessionKeyData {
     // Contract access control type
    ContractAccessControlType contractAccessControlType;
    // Key time range: limits when a key may be used.
    uint48 validAfter;
    uint48 validUntil;
    bool hasRequiredPaymaster;
    bool hasGasLimit;
    bool gasLimitResetThisBundle;
    // Native token spend limits
    bool nativeTokenSpendLimitBypassed; // By default, spend limits ARE enforced and the limit is zero.
    SpendLimitTimeInfo gasLimitTimeInfo;
    SpendLimitTimeInfo nativeTokenSpendLimitTimeInfo;
    // Required paymaster rule
    address requiredPaymaster;
    SpendLimit gasLimit;
    SpendLimit nativeTokenSpendLimit;
}

/// @dev These structs are not held in an Associated Enumerable set, so the elements must be emitted
↪  from
/// events to use offchain.
struct ContractData {
    bool isOnList;
    bool checkSelectors;
    bool isERC20WithSpendLimit;
    SpendLimitTimeInfo erc20SpendLimitTimeInfo;
    SpendLimit erc20SpendLimit;
}

struct FunctionData {
    bool isOnList;
}
```

Recommendation: To ensure clarity and comprehensive understanding, it is recommended to thoroughly document all struct elements within the `SessionKeyPermissionsBase` abstract contract.

Alchemy: Solved in PR 86.

Spearbit: Verified.

5.5.25 Session key logic could be prepared for other signature schemas

Severity: Informational

Context: SessionKeyPlugin.sol#L160-L179

Description: Function `userOpValidationFunction()` of `SessionKeyPlugin` currently only allows to check the session key via `ecrecover`. It might be useful to also prepare for `secp256r1` pairs, so the session key can used by an iOS or Android device. There are a few solidity implementations and there could be a precompile (see EIP-7212).

```
function userOpValidationFunction(uint8 functionId, UserOperation calldata userOp, bytes32 userOpHash)
↪  /*...*/ {
    // ...
    (address recoveredSig, ECDSA.RecoverError err) = hash.tryRecover(userOp.signature);
    if (err == ECDSA.RecoverError.NoError && sessionKey == recoveredSig) {
        return _SIG_VALIDATION_PASSED;
    }
    // ...
}
```

Recommendation: Consider preparing the code for `secp256r1` pairs.

Alchemy: Good suggestion. Out of scope for v1 but will definitely consider for v2.

Spearbit: Acknowledged.

5.5.26 Comment box not used consistently

Severity: Informational

Context: SessionKeyPlugin.sol

Description: Plugins have comment boxes to seperate the different functions. This is useful but not applied consistently.

```
MultiOwnerPlugin.sol
78:     //  |  Execution functions     |
97:     //  |  Execution view functions    |
157:    //  |  Plugin view functions     |
187:    //  |  Plugin interface functions     |
310:    //  |  EIP-165     |
319:    //  |  Internal     |

TokenReceiverPlugin.sol
27:     //  |  Execution functions     |
60:     //  |  Plugin interface functions     |

SessionKeyPlugin.sol
51:     //  |  Execution functions     |
119:    //  |  Plugin view functions     |
156:    //  |  Plugin interface functions     |
286:    //  |  EIP-165     |

SessionKeyPermissionsPlugin.sol
40:     //  |  Execution functions     |
94:     //  |  Plugin interface functions     |
210:    //  |  EIP-165     |
220:    //  |  Internal / Private functions     |
```

Recommendation: Consider applying the comment boxes consistently. In particular, consider seperating out three different classes of view functions:

- Accessible via UserOp and Runtime Validation.
- Only accessible via UserOp.
- Only accessible directly on the plugin.

Alchemy: Solved in PR 92.

Spearbit: Verified.

5.5.27 Naming inconcistencies between the `fallback` and the `executeFromPlugin` functions

Severity: Informational

Context: UpgradeableModularAccount.sol#L240, UpgradeableModularAccount.sol#L131

Description: There are a couple of naming differences between the fallback and the executeFromPlugin function:

- The variable named `success` in the `executeFromPlugin` function is referred to as `execSuccess` in the `fallback` function.
- The variable `returnData` in the `executeFromPlugin` function is named `execReturnData` in the corresponding section of the `fallback` function.

Recommendation: To ensure consistency and clarity in the codebase, it is advisable to standardize the naming conventions between these two functions.

Alchemy: Solved in PR 91.

Spearbit: Verified.

5.5.28 "*Session keys may not be contracts*" is not enforced

Severity: Informational

Context: MSCA-Specs-Manual

Description: The MSCA-Specs-Manual contains: *Session keys may not be contracts*, however this is not enforced in the code. Checking for contracts could be helpful to prevent mistakes, although mistakes are easy to correct.

Recommendation: Consider checking that Session keys are not contracts in `SessionKeyPlugin` and `SessionKeyPermissionsPlugin`.

Alchemy: We will not be adding an `extcodesize` check here to prevent warming the `session key`'s address and incurring the extra gas cost. If a contract is accidentally added, it will be unable to provide a valid `ECDSA` signature for `user op validation`, and will be unusable. As you stated, the mistake of adding a `contract` as a `session key` is also easy to correct.

Spearbit: Acknowledged.

5.5.29 Some custom errors could be more informative

Severity: Informational

Context: BasePlugin.sol#L21, MultiOwnerPlugin.sol#L205-L226, SessionKeyPermissionsPlugin.sol#L775, SessionKeyPermissionsPlugin.sol#L718

Description: It has been observed that certain errors could convey more information if additional arguments are included. There are listed in the recommendation section bellow, along with the proposed suggestion.

Recommendation:

- The error `NotImplemented()` is used to indicate a function or a `FunctionId` is not implemented. As its used in several functions, troubleshooting this error might be time consuming. Consider adding the `function selector` (`msg.sig`) and the `functionId` to the error `NotImplemented()`:

```
error NotImplemented();

contract MultiOwnerPlugin is BasePlugin, IMultiOwnerPlugin, IERC1271, EIP712 {
    function userOpValidationFunction(uint8 functionId, UserOperation calldata userOp, bytes32
↪   userOpHash) /*...*/ {
        if (functionId == uint8(FunctionId.USER_OP_VALIDATION_OWNER)) {
            // ...
        } // else
        revert NotImplemented();
    }
}
```

- The `InvalidToken` can be more informative by adding the `token` in the revert message. Currently, this error is only used when `token` is `address(0)`:

```
function _setERC20SpendLimit(SessionKeyId keyId, address token, uint256 spendLimit, uint48
↪   refreshInterval)
    internal
{
    if (token == address(0)) {
        revert InvalidToken();
    }
// ...
```

- The `InvalidPermissionsUpdate` error could be made more descriptive by adding the `updateSelector` as a parameter:

```
// ...
    abi.decode(update[4:], (address, uint256, uint48));
    _setERC20SpendLimit(keyId, token, spendLimit, refreshInterval);
} else if (updateSelector == ISessionKeyPermissionsUpdates.setGasSpendLimit.selector) {
    (uint256 spendLimit, uint48 refreshInterval) = abi.decode(update[4:], (uint256, uint48));
    _setGasSpendLimit(sessionKeyData, spendLimit, refreshInterval);
} else if (updateSelector == ISessionKeyPermissionsUpdates.setRequiredPaymaster.selector) {
    address requiredPaymaster = abi.decode(update[4:], (address));
    _setRequiredPaymaster(sessionKeyData, requiredPaymaster);
} else {
    revert InvalidPermissionsUpdate();
```

Alchemy: Solved in PR 94.

Spearbit: Verified.

5.5.30 Comments in `BasePlugin` are incomplete

Severity: Informational

Context: BasePlugin.sol#L11-L13, BasePlugin.sol#L202-L204

Description: The contract `BasePlugin` contains a few comments mentioning `execute()` and `executeBatch()`, which are not allowed to call plugins. The function `executeFromPluginExternal()` should be added to that, because its also not allowed to call plugins.

```
/// ... This also ensures that plugin interactions cannot
/// happen via the standard execution funtions `execute` and `executeBatch`.

abstract contract BasePlugin is ERC165, IPlugin {
    // ...

    /// ...  This is also used
    /// by the modular account to prevent standard execution functions `execute` and `executeBatch` from
    /// making calls to plugins.

    // ...
}
```

Recommendation: Add `executeFromPluginExternal()` to the comments.

Alchemy: Solved in PR 70.

Spearbit: Verified.

5.5.31 `_isValidERC1271OwnerTypeSignature()` could run out of gas

Severity: Informational

Context: MultiOwnerPlugin.sol#L205-L242, MultiOwnerPlugin.sol#L416-L432

Description: The function `_isValidERC1271OwnerTypeSignature()` does a `for` loop which means it could run out of gas if there are too many owners, possibly in combination with one of the `isValidSignature()` function taking too much gas. For example if the `first` owner account is rogue/buggy it could uses up all the gas and prevent `userOpValidationFunction()` from executing, which could lead to a denial of service. A workaround for this situation is to use `runtimeValidationFunction()`:

```
function userOpValidationFunction(uint8 functionId, UserOperation calldata userOp, bytes32 userOpHash)
↪   /*...*/ {
    // ...
    if (_isValidERC1271OwnerTypeSignature(msg.sender, userOpHash, userOp.signature)) {
        return _SIG_VALIDATION_PASSED;
    // ...
}
function _isValidERC1271OwnerTypeSignature(address associated, bytes32 digest, bytes memory signature)
↪   /*...*/ {
    // ...
    for (uint256 i; i < owners_.length;) {
        if (SignatureChecker.isValidERC1271SignatureNow(owners_[i], digest, signature)) {
            return true;
        }
        // ...
    }
    // ...
}
function runtimeValidationFunction(uint8 functionId, address sender, uint256, bytes calldata) /*...*/ {
    // ...
    // Validate that the sender is an owner of the account, or self.
    if (sender != msg.sender && !isOwnerOf(msg.sender, sender)) {
        revert NotAuthorized();
    }
    // ...
}
```

Recommendation: A potential solution for the `userOpValidation` path is to first check if the `sender` of the `UserOperation` is a valid `owner`.

Alchemy: I think the `userOp.sender` refers to the SCA making the call (which is equal to `msg.sender` in the `userOpValidationFunction` context). `tx.origin` would be the bundler too. If that's the case, I don't think there are good options to short circuit the checks.

Spearbit: Acknowledged.

5.5.32 Initialization of index variables in `for` loops is not consistent

Severity: Informational

Context: MultiOwnerPlugin.sol#L399-L432

Description: Most `for` loops initialize the index variable. The exceptions are functions `_removeOwnersOrRevert()` and `_isValidERC1271OwnerTypeSignature()`. Although its not necessary to initialize the variables, because they are 0 by default, its better to be consistent.

```
function _removeOwnersOrRevert(/*...*/) /*...*/ {
    // ...
    for (uint256 j; j < ownersToRemoveLength;) {
        // ...
    }
    // ...
}
function _isValidERC1271OwnerTypeSignature(address associated, bytes32 digest, bytes memory signature)
    // ...
    for (uint256 i; i < owners_.length;) {
        // ...
    }
    // ...
}
```

Recommendation: Make the initialization of index variables in `for` loops consistent.

Alchemy: Fixed in PR 71.

Spearbit: Verified.

5.5.33 Not clear why view function are not added to `userOpValidationFunctions`

Severity: Informational

Context: MultiOwnerPlugin.sol#L245-L343

Description: The function `pluginManifest()` of `MultiOwnerPlugin` doesn't add the `view` function to `userOpValidationFunctions`, although they are present in `runtimeValidationFunctions`. As we understood from the project:

> We don't expect view functions to be directly called via a user operation, so they were intentionally left out.

```
function pluginManifest() external pure override returns (PluginManifest memory) {
    // ...
    manifest.userOpValidationFunctions[5] = /*...*/ upgradeToAndCall.selector // ...
    // ...
    manifest.runtimeValidationFunctions[5] = /*...*/ upgradeToAndCall.selector // ...
    // ...
    manifest.runtimeValidationFunctions[6] = /*...*/ isValidSignature.selector // ...
    manifest.runtimeValidationFunctions[7] = /*...*/ isOwner.selector // ...
    manifest.runtimeValidationFunctions[8] = /*...*/ owners.selector // ...
    manifest.runtimeValidationFunctions[9] = /*...*/ eip712Domain.selector // ...
    // ...
}
```

Recommendation: Consider adding a comment to explain that `view` functions are not added to `userOpValidationFunctions`.

Alchemy: Solved in PR 95. It's a security risk for SCAs to allow view functions to be called: If an account installs a view function for the `userop` path, a malicious bundler would be able to empty an account by filling blocks full of `userOps` calling that view function. Adding this to our internal list of systemic risks

Spearbit: Verified.

5.5.34 Missing or Incomplete NatSpec

Severity: Informational

Context: MultiOwnerMSCAFactory.sol#L107, MultiOwnerTokenReceiverMSCAFactory.sol#L115, SessionKeyPermissionsPlugin.sol#L603

Description: Some instances of incomplete or missing NatSpec documentation have been found in the codebase. There are listed in the recommendation section with the suggested fix.

Recommendation:

- In the MultiOwnerMSCAFactory.sol#L107 and the MultiOwnerTokenReceiverMSCAFactory.sol#L115, the `getAddress()` function is missing the NatSpec `@return`.
- In SessionKeyPermissionsPlugin.sol#L603, the `@param` for the `token` parameter is missing.

Alchemy: Natspec for `getAddress()` fixed in PR 72. Natspec for `SessionKeyPermissionsPlugin` fixed as part of a separate mitigation in PR 28 which removed the parameter in question.

Spearbit: Verified.

5.5.35 Comment in `BasePlugin` for `onInstall()` is inaccurate

Severity: Informational

Context: BasePlugin.sol#L14-L17, MultiOwnerPlugin.sol#L191-L197, TokenReceiverPlugin.sol#L65, SessionKeyPermissionsPlugin.sol#L119

Description: A comment in `BasePlugin` indicate that `onInstall()` needs to be overriden when installing a plugin when creating an account. `MultiOwnerPlugin` is installed when creating an account but doesn't override `onInstall()`.

The installation of plugins is done via `initialize()` and not in the `constructor`. So the comment seems inaccurate.

Note: The `TokenReceiverPlugin` and `SessionKeyPermissionsPlugin` plugins do override `onInstall()`, but both of these implementations are an empty function, so this is most likely done to save gas.

```
/// Note that the plugins implement BasePlugins cannot be installed when creating an account (aka
↪  installed in the
/// account constructor) unless onInstall is overriden without checking codesize of caller (account).
↪  Checking
/// codesize of account is to prevent EOA from accidentally calling plugin and initiate states which
↪  will make it
/// unusable in the future when EOA can be upgraded into an smart contract account.
abstract contract BasePlugin is ERC165, IPlugin {
  function onInstall(bytes calldata data) external virtual {
        if (msg.sender.code.length == 0) {
            revert NotContractCaller();
        }
        _onInstall(data);
    }
}
contract MultiOwnerPlugin is BasePlugin, IMultiOwnerPlugin, IERC1271, EIP712 {
    function _onInstall(bytes calldata data) internal override isNotInitialized(msg.sender) {
        (address[] memory initialOwners) = abi.decode(data, (address[]));
        if (initialOwners.length == 0) {
            revert EmptyOwnersNotAllowed();
        }
        _addOwnersOrRevert(_owners, msg.sender, initialOwners);
    }
}
contract TokenReceiverPlugin is BasePlugin, IERC721Receiver, IERC777Recipient, IERC1155Receiver {
    function onInstall(bytes calldata) external pure override {}
}
contract SessionKeyPermissionsPlugin is ISessionKeyPermissionsPlugin, SessionKeyPermissionsLoupe,
↪  BasePlugin {
    function onInstall(bytes calldata) external override {}
}
```

Recommendation: Doublecheck and update the comment of `BasePlugin`.

Alchemy: Acknowledging the issue. This is a warning for plugin developers, in the case theyre developing plugins for generic `ERC6900` accounts. In the off chance that there is an account implementation out there that attempt to install any of these plugins in it's `constructor`, it would revert as `codesize` would still be 0 at that point in time. Unless they overwrite the `onInstall` method. I have added this to our internal technical documentation as it's not just a `BasePlugin` issue.

Spearbit: Acknowledged.

5.5.36 NatSpec comments in the `withdraw` function can be improved

Severity: Informational

Context: MultiOwnerMSCAFactory.sol#L92-L93, MultiOwnerTokenReceiverMSCAFactory.sol#L100-L101

Description: The `withdraw` function is used to withdraw both erc20s and native tokens. Currently, he comment with the `@dev` tag states that: `can withdraw stuck erc20s` and the comment with the `@param` tag states that: `address to send native currency to`.

Recommendation: To better reflect the functionality of the `withdraw` function, they can be refactored to:

```
- /// @dev can withdraw stuck erc20s
+ /// @dev can withdraw stuck erc20s or native currency
- /// @param to address to send native currency to
+ /// @param to address to send native currency or erc20s to
```

Alchemy: Solved in PR 96.

Spearbit: Verified.

5.5.37 Generic function `eip712Domain()` can't be used by other plugins

Severity: Informational

Context: MultiOwnerPlugin.sol#L111-L127

Description: The generic function `eip712Domain()` is exported by the `MultiOwnerPlugin`. This does prevent installation of other plugins that may also implement this function. Furthermore, renaming doesn't work because then its not compatible with EIP-5267:

```
function eip712Domain() /*...*/ {
    // ...
}
```

Recommendation: Consider making a generic version of this function.

Alchemy: We think most accounts would have a "default signer" which exposes validators for signatures in the short term. With this assumption, structurally it makes sense to pair the 712domain implementation with these default signer plugins to reduce the number of plugin installations. Acknowledging this as a potential issue that we will revisit in the future, but no plans to split this out for now.

Spearbit: Acknowledged.

5.5.38 Similar functions `isSentinel()` and `hasNext()` are written differently

Severity: Informational

Context: AssociatedLinkedListSetLib.sol#L402-L410, LinkedListSetLib.sol#L299-L307

Description: The functions `isSentinel()` and `hasNext()` of `AssociatedLinkedListSetLib` and `LinkedListSetLib` are similar but are implemented differently. Function `isSentinel()` uses `assembly` while `hasNext` uses `Solidity`.

```
function isSentinel(bytes32 value) internal pure returns (bool ret) {
    assembly ("memory-safe") {
        ret := and(value, 1)
    }
}
function hasNext(bytes32 value) internal pure returns (bool) {
    return value & HAS_NEXT_FLAG != 0;
}
```

Recommendation: Consider changing `isSentinel()` also to Solidity:

```
  function isSentinel(bytes32 value) internal pure returns (bool ret) {
-     assembly ("memory-safe") { ret := and(value, 1) }
+     return value & SENTINEL_VALUE != 0;
  }
```

Alchemy: We're going to acknowledge this but leave the implementation as-is. Using the assembly and operation lets us reinterpret cast the result value for `isSentinel` directly into a `boolean`, which also uses the lowest bit for its representation, while the `HAS_NEXT_FLAG` occupies the second bit and therefore is not safely convertible into a `boolean` without the `!= 0` expression.

Spearbit: Acknowledged.

5.5.39 Similar functions `getAll()` and `tryRemove()` are written differently

Severity: Informational

Context: LinkedListSetLib.sol#L251-L297, AssociatedLinkedListSetLib.sol#L354-L400

Description: The function `getAll()` of `LinkedListSetLib` and `AssociatedLinkedListSetLib` are very similar to `tryRemove()`. `getAll()` uses a `while` loop and uses `hasNext()`. Function `tryRemove()` uses a `do - while` loop and doesn't use `hasNext()`.

Recommendation: Consider rewriting `getAll()` in the same way to make the code more consistent.

Alchemy: Acknowledging the difference, but for now we're not going to refactor this to make the code consistent across `tryRemove` and `getAll`. There is a slight difference in goals for the iteration that supports keeping somewhat different implementations. For `getAll`, using `hasNext` may provide a gas optimization to avoid an extra `sload` for the last element of the set. However, this is not useful to implement for `tryRemove`, because `tryRemove` first checks to see if the set contains the element and short-circuits if it does not. Using `hasNext` would only provide a gas optimization to early-terminate the loop in case that the element is not actually found. However, given that the "set contains" check was just performed, the only way this could happen is if there was storage corruption (i.e. via `delegatecall` or an unsafe upgrade), which is a case we won't optimize for.

Spearbit: Acknowledged.

5.5.40 Collapse `if` / `else` in favor of simplicity

Severity: Informational

Context: SessionKeyPermissionsLoupe.sol#L63-L73, SessionKeyPermissionsLoupe.sol#L114-L130

Description: `getNativeTokenSpendLimitInfo()` and `getGasSpendLimit()` have an `if` / `else` condition based on `hasLimit`. For example, here is `getNativeTokenSpendLimitInfo()`:

```
if (hasLimit) {
    return SpendLimitInfo({
        hasLimit: true,
        limit: sessionKeyData.nativeTokenSpendLimit.limitAmount,
        limitUsed: sessionKeyData.nativeTokenSpendLimit.limitUsed,
        refreshInterval: sessionKeyData.nativeTokenSpendLimitTimeInfo.refreshInterval,
        lastUsedTime: sessionKeyData.nativeTokenSpendLimitTimeInfo.lastUsed
    });
} else {
    // The fields aren't cleared until the next time they are set, so report zeros.
    return SpendLimitInfo({hasLimit: false, limit: 0, limitUsed: 0, refreshInterval: 0, lastUsedTime:
↪   0});
}
```

This `if` / `else` can be removed and the `SpendLimitInfo` struct prepared in `if` clause can be returned directly since `hasLimit` already differentiates the two cases.

Recommendation: If these functions are supposed to be called onchain, consider removing the `if` / `else` clause and just return the struct prepared in the `if` clause. According Alchemy team, this has been done to avoid confusing frontends if a non-zero `limitUsed` is reported when a limit isn't set.

If these functions are just supposed to called off-chain from frontends, then the code is fine as-is, although frontends can check `hasLimit` first.

In that case the following optimization can be used:

```
  function getNativeTokenSpendLimitInfo(address account, address sessionKey) /*...*/ {
      if (hasLimit) {
          return SpendLimitInfo({hasLimit: true, /*...*/ });
      }
+     // else return all empty/zero/false values
-     else {
-         return SpendLimitInfo({hasLimit: false, limit: 0, limitUsed: 0, refreshInterval: 0,
↪  lastUsedTime: 0});
-     }
  }
```

Alchemy: Solved in PR 90.

Spearbit: Verified.

5.5.41 The project may fail to deploy or may not work properly on chains that are not compatible with the Shanghai hardfork

Severity: Informational

Context: SessionKeyPermissionsPlugin.sol#L2

Description: The contracts in the project use a version pragma of `^0.8.21`, indicating that they can be compiled with compiler version greater or equal to `0.8.21` but less than `0.9.0`. The `0.8.20` version of the compiler, sets the default version of the compiler to `Shanghai`, which introduces the `PUSH0` opcode.

However, the `PUSH0` opcode is not supported on some chains, which could pose an issue if the contracts are intended to be deployed on chains that do not support the `PUSH0` opcode, such as the `Arbitrum` or `Optimism Bedrock`.

Recommendation: It is advised to set the EVM version to paris in `foundry.toml`, by adding:

```
evm_version="paris"
```

Alchemy: Solved in PR 67.

Spearbit: Verified.

5.5.42 Keccaks for constants are evaluated at compile time

Severity: Informational

Context: MultiOwnerPlugin.sol#L72-L73, permissions/SessionKeyPermissionsBase.sol#L64-L67, AccountStorageV1.sol#L111-L113, AssociatedLinkedListSetLib.sol#L28

Description: `ERC6900_TYPEHASH` is assigned to the evaluated keccak256 value of `"ERC6900Message(bytes message)"`:

```
// keccak256("ERC6900Message(bytes message)");
bytes32 private constant ERC6900_TYPEHASH =
↪  0xa856bbdae1f2c6e4aa17a75ad7cc5650f184ec4b549174dd7258c9701d663fc6;
```

This is error-prone and requires extra effort to maintain synchronized. Also see the issue "Wrong keccak value for storage prefix".

It is equivalent to just assigning the `keccak256` expression to `ERC6900_TYPEHASH`, as Solidity evaluates the `keccak256` expression at compile time if gas optimization is turned on. Hence, this way is better than to assume that the hash evaluated manually is correct.

The same occurs here:

```
SessionKeyPermissionsBase.sol:
    bytes4 internal constant SESSION_KEY_ID_PREFIX = bytes4(0x1a01dae4); //
↪  bytes4(keccak256("SessionKeyId"))
    bytes4 internal constant SESSION_KEY_DATA_PREFIX = bytes4(0x16bff296); //
↪  bytes4(keccak256("SessionKeyData"))
    bytes4 internal constant CONTRACT_DATA_PREFIX = bytes4(0x634c29f5); //
↪  bytes4(keccak256("ContractData"))
    bytes4 internal constant FUNCTION_DATA_PREFIX = bytes4(0xd50536f0); //
↪  bytes4(keccak256("FunctionData"))

AccountStorageV1.sol:
    /// bytes = keccak256(
    /// abi.encode(uint256(keccak256("Alchemy.UpgradeableModularAccount.Storage_V1")) - 1)
    /// ) & ~bytes32(uint256(0xff));
    bytes32 internal constant _V1_STORAGE_SLOT =
        0xade46bbfcf6f898a43d541e42556d456ca0bf9b326df8debc0f29d3f811a0300;

AssociatedLinkedListSetLib.sol:
    bytes4 internal constant _ASSOCIATED_STORAGE_PREFIX = 0x9cc6c923;
    // bytes4(keccak256("AssociatedLinkedListSet"))
```

Recommendation: Update the assignment as:

```
bytes32 private constant ERC6900_TYPEHASH = keccak256("ERC6900Message(bytes message)");
```

Change the other occurrences in a similar way, except for the cases where its used in assembly.

Alchemy: Solved in PR 98.

Spearbit: Verified.

5.5.43 One `assembly` block without `("memory-safe")`

Severity: Informational

Context: AccountStorageV1.sol#L116-L120

Description: Function `_getAccountStorage()` has an `assembly` block without `("memory-safe")`, while all other blocks have `("memory-safe")`.

Note: the original code OZ _getInitializableStorage() doesn't have `("memory-safe")` either. Using the same pattern everywhere is more consistent.

```
function _getAccountStorage() internal pure returns (AccountStorage storage storage_) {
    assembly {
        storage_.slot := _V1_STORAGE_SLOT
    }
}
```

Recommendation: Consider adding `("memory-safe")` here too.

Alchemy: Solved in PR 84.

Spearbit: Verified.

5.5.44 Missing `OwnerUpdated` event emission in `onInstall` and `onUninstall` functions

Severity: Informational

Context: MultiOwnerPlugin.sol#L82-L94

Description: The `OwnerUpdated` event is emitted when the `updateOwners` function is called and the owners of the account are updated. However, this event is not emitted during the execution of the `onInstall` and the `onUninstall` functions.

Recommendation: It is recommended to emit the `OwnerUpdated` event in both the `onInstall` and the `onUninstall` functions.

Alchemy: Solved in PR 85.

Spearbit: Verified.

5.5.45 Comments for `onHookApply()` could be extended

Severity: Informational

Context: IPlugin.sol#L175-L184

Description: The function `onHookApply()` could run multiple times if multiple hooks are use from the plugin. This might not be obvious to plugin developers.

```
/// @notice A hook that runs when a hook this plugin owns is installed onto another plugin
/// @dev Optional, use to implement any required setup logic
/// @param pluginAppliedOn The plugin that the hook is being applied on
/// @param injectedHooksInfo Contains pre/post exec hook information
/// @param data Any optional data for setup
function onHookApply(/*...*/) external;
```

Recommendation: Consider extending the comment to indicate this function can run multiple times.

Alchemy: The `onHookApply()` functionality was removed in PR 76, which resolves the need to document this behavior.

Spearbit: Verified.

5.5.46 Wrong keccak value for storage prefix

Severity: Informational

Context: AssociatedLinkedListSetLib.sol#L28

Description: `_ASSOCIATED_STORAGE_PREFIX` is computed incorrectly:

```
bytes4 internal constant _ASSOCIATED_STORAGE_PREFIX = 0x9cc6c923; //
↪  bytes4(keccak256("AssociatedLinkedListSet"))
```

This is the keccak of `"AssociatedEnumerableSet"`.

Recommendation: Update `_ASSOCIATED_STORAGE_PREFIX` to `0xf938c976`. Or even better, use the `bytes4(keccak256(...))` calculation. Also see the issue "Keccaks for constants are evaluated at compile time".

Alchemy: Solved in PR 60.

Spearbit: Verified.

5.5.47 `executeFromPluginExternal()` can be made more readable

Severity: Informational

Context: UpgradeableModularAccount.sol#L306-L341

Description: Function `executeFromPluginExternal()` can be made more readable with a temporary variable. This will also save some gas.

```
function executeFromPluginExternal(/*...*/) /*...*/ {
    // ...
    if (storage_.permittedExternalCalls[IPlugin(callingPlugin)][target].addressPermitted) {
        isTargetCallPermitted = (
            storage_.permittedExternalCalls[IPlugin(callingPlugin)][target].anySelectorPermitted
                || data.length == 0
                ||
↪   storage_.permittedExternalCalls[IPlugin(callingPlugin)][target].permittedSelectors[bytes4(data)]
        );
    } else {
       // ...
    }
    // ...
}
```

Recommendation: Consider changing the code to:

```
function executeFromPluginExternal(/*...*/) /*...*/ {
    // ...
    PermittedExternalCallData memory permittedExternalCallData = // could also be storage
        storage_.permittedExternalCalls[IPlugin(callingPlugin)][target];

    if (permittedExternalCallData.addressPermitted) {
        isTargetCallPermitted = (
           permittedExternalCallData.anySelectorPermitted
                || data.length == 0
                || permittedExternalCallData.permittedSelectors[bytes4(data)]
        );
    } else {
       // ...
    }
    // ...
}
```

Alchemy: Solved in PR 87.

Spearbit: Verified.

5.5.48 `unchecked` increment pattern is no longer necessary

Severity: Informational

Context: UpgradeableModularAccount.sol#L117-L119

Description: The code has several occurances of the pattern where the index variable of a `for` loop is incremented via `unchecked`. This is no longer necessary from Solidity 0.8.22. Removing this pattern improves readability and makes maintenance easier.

```
function initialize(/*...*/) /*...*/ {
    // ...
    for (uint256 i = 0; i < length;) {
        // ...
        unchecked {
            ++i;
        }
    }
    // ...
}
```

Recommendation: Consider removing the `unchecked` pattern for the `for` index variables. This change might also be good for the reference implementation of ERC-6900.

Note: Its best practice not to use to latest version of Solidity in production. However because Solidity 0.8.23 is also available, using version 0.8.22 can be considered. There is no urgency to change this.

Alchemy: Fixed in PR 88 and PR 89.

Spearbit: Verified.

5.5.49 `_resolveManifestFunction()` can be refactored to enhance readability

Severity: Informational

Context: PluginManagerInternals.sol#L929-L955

Description: The function `_resolveManifestFunction()` is somewhat difficult to read. It could be refactored to make it easier to read and maintain.

```
function _resolveManifestFunction(/*...*/) /*...*/ {
    if (manifestFunction.functionType == ManifestAssociatedFunctionType.SELF) {
        return FunctionReferenceLib.pack(plugin, manifestFunction.functionId);
    } else if (manifestFunction.functionType == ManifestAssociatedFunctionType.DEPENDENCY) {
        return dependencies[manifestFunction.dependencyIndex];
    } else if (manifestFunction.functionType ==
↪   ManifestAssociatedFunctionType.RUNTIME_VALIDATION_ALWAYS_ALLOW)
    {
        if (allowedMagicValue == ManifestAssociatedFunctionType.RUNTIME_VALIDATION_ALWAYS_ALLOW) {
            return FunctionReferenceLib._RUNTIME_VALIDATION_ALWAYS_ALLOW;
        } else {
            revert InvalidPluginManifest();
        }
    } else if (manifestFunction.functionType == ManifestAssociatedFunctionType.PRE_HOOK_ALWAYS_DENY) {
        if (allowedMagicValue == ManifestAssociatedFunctionType.PRE_HOOK_ALWAYS_DENY) {
            return FunctionReferenceLib._PRE_HOOK_ALWAYS_DENY;
        } else {
            revert InvalidPluginManifest();
        }
    }
    return FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE; // Empty checks are done elsewhere
}
```

Recommendation: Consider changing the code to:

```
function _resolveManifestFunction(/*...*/) /*...*/ {
    if (manifestFunction.functionType == ManifestAssociatedFunctionType.SELF)
        return FunctionReferenceLib.pack(plugin, manifestFunction.functionId);

    if (manifestFunction.functionType == ManifestAssociatedFunctionType.DEPENDENCY)
        return dependencies[manifestFunction.dependencyIndex];

    if (manifestFunction.functionType ==
↪   ManifestAssociatedFunctionType.RUNTIME_VALIDATION_ALWAYS_ALLOW) {
        if (allowedMagicValue == ManifestAssociatedFunctionType.RUNTIME_VALIDATION_ALWAYS_ALLOW)
            return FunctionReferenceLib._RUNTIME_VALIDATION_ALWAYS_ALLOW;
        revert InvalidPluginManifest();
    }
    if (manifestFunction.functionType == ManifestAssociatedFunctionType.PRE_HOOK_ALWAYS_DENY) {
        if (allowedMagicValue == ManifestAssociatedFunctionType.PRE_HOOK_ALWAYS_DENY)
            return FunctionReferenceLib._PRE_HOOK_ALWAYS_DENY;
        revert InvalidPluginManifest();
    }
    return FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE; // Empty checks are done elsewhere
}
```

Alchemy: Solved in PR 97.

Spearbit: Verified.

5.5.50 `_resolveManifestFunction()` could revert

Severity: Informational

Context: PluginManagerInternals.sol#L939

Description: The array `dependencies[]` is accessed via index `manifestFunction.dependencyIndex`. It this is out of bound then the statement will revert without a clear error message.

```
function _resolveManifestFunction(/*...*/, FunctionReference[] memory dependencies, /*...*/) /*...*/ {
    // ...
    return dependencies[manifestFunction.dependencyIndex];
    // ...
}
```

Recommendation: Consider explicitly checking `manifestFunction.dependencyIndex` is within bounds and give an error message otherwise.

Alchemy: Solved in PR 77 and PR 97.

Spearbit: Verified.

5.5.51 Other smart contract wallets could be plugins

Severity: Informational

Context: PluginManagerInternals.sol#L322-L334

Description: The main check to make sure a plugin a valid plugin is to check `supportsInterface()`. This could allow smart contract wallets and MSCAs to be installed as a plugin if they expose the appropriate `interfaceId` and functions.

Note: using a MSCA as a plugin can be prevented via the recommendations of issue "Internal plugin functions can be setup to be externally accessible".

We don't see relevant downsides of this, but it might be unexpected, hence the informational severity.

```
function _installPlugin(./*..*/) /*...*/ {
    // ...
    // Check if the plugin exists, also invalidate null address.
    if (!storage_.plugins.tryAdd(CastLib.toSetValue(plugin))) {
        revert PluginAlreadyInstalled(plugin);
    }
    // Check that the plugin supports the IPlugin interface.
    if (!ERC165Checker.supportsInterface(plugin, type(IPlugin).interfaceId)) {
        revert PluginInterfaceNotSupported(plugin);
    }
    // ...
}
```

Recommendation: If any new issues are found, limitations could be added to `_installPlugin()`, or a plugin could be created to prevent installing these types of plugins via hooks on `UpgradeableModularAccount.installPlugin.selector`.

Alchemy: Solved in PR 45.

Spearbit: Verified.

5.5.52 `_addHooks()` can call `_assertNotNullFunction()`

Severity: Informational

Context: PluginManagerInternals.sol#L202-L231, PluginManagerInternals.sol#L957-L961

Description: The function `_addHooks()` does a check for `_EMPTY_FUNCTION_REFERENCE` and then reverts. This code could be replaced with a call to `_assertNotNullFunction()`, which also implements this check.

```
function _addHooks(/*...*/) /*...*/ {
    // ...
    if (postExecHook == FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE) {
        revert NullFunctionReference();
    }
    // ...
}
function _assertNotNullFunction(FunctionReference functionReference) internal pure {
    if (functionReference == FunctionReferenceLib._EMPTY_FUNCTION_REFERENCE) {
        revert NullFunctionReference();
    }
}
```

Recommendation: Consider calling `_assertNotNullFunction()` from `_addHooks()`.

Alchemy: Solved in PR 83.

Spearbit: Verified.

5.5.53 Unused code

Severity: Informational

Context: PluginManagerInternals.sol#L52

Description: Some pieces of code are not used.

- PluginManagerInternals.sol#L52:

```
error ExecutionFunctionNotSet(bytes4 selector);
```

Recommendation: Double check the unused code is indeed no longer necessary and if so, remove it.

Alchemy: Solved in PR 61.

Spearbit: Verified.

5.5.54 Upgrade `AccountStorageInitializable` to the latest version of the OpenZeppelin library

Severity: Informational

Context: AccountStorageInitializable.sol#L19-L37, OZ Initializable.sol#L104-L132

Description: The code of `AccountStorageInitializable` is derived from the OpenZeppelin 4.9 library. The latest OpenZeppelin 5.0 library also uses a diamond storage pattern, has slightly more readable code and doesn't use `isContract()` anymore. This is also easier for future maintenance:

```
modifier initializer() {
    // ...
    bool isTopLevelCall = !storage_.initializing;
    if (
            isTopLevelCall && storage_.initialized < 1
                || !Address.isContract(address(this)) && storage_.initialized == 1
        ) {
        // ...
    } else {
        revert AlreadyInitialized();
    // ...
}
```

OpenZeppelin 5.0 library:

```
modifier initializer() {
    // ...
    bool isTopLevelCall = !$._initializing;
    uint64 initialized = $._initialized;
    bool initialSetup = initialized == 0 && isTopLevelCall;
    bool construction = initialized == 1 && address(this).code.length == 0;
    if (!initialSetup && !construction) {
        revert InvalidInitialization();
    }
    // ...
}
```

Recommendation: Consider using forking the code of the latest OpenZeppelin 5.0 library and adapting it in a similar way like in `AccountStorageInitializable`. Especially, `INITIALIZABLE_STORAGE` should not be copied because that could clash with other implementations of smart contract accounts (in case of conversion to another SCA).

Alchemy: Opened OpenZeppelin issue 4782. Solady has made this change: Vectorized PR 795. The OZ change will not be ready in time. Switching to Solady is out of scope for our fix period currently. So acknowledged for now.

Spearbit: Acknowledged.

5.5.55 Typos

Severity: Informational

Context: AccountExecutor.sol#L165, AccountStorageInitializable.sol#L11, UpgradeableModularAccount.sol#L323, UpgradeableModularAccount.sol#L348, UpgradeableModularAccount.sol#L746, IPluginExecutor.sol#L6, AccountStorageV1.sol#L45, SessionKeyPermissionsPlugin.sol#L283, SessionKeyPermissionsPlugin.sol#L671

Description: A few typos have been found. There are listed in the recommendation section with the suggested fix.

Recommendation:

- AccountExecutor.sol#L165:

```
- // Totoal: 164 bytes
+ // Total: 164 bytes
```

- AccountStorageInitializable.sol#L11:

```
- Initialiazble
+ Initializable
```

- UpgradeableModularAccount.sol#L323:

```
- // b. Is the calldata is empty?
+ // b. Is the calldata empty?
```

- UpgradeableModularAccount.sol#L348:

```
- execfuteFromPluginExternal
+ executeFromPluginExternal
```

- UpgradeableModularAccount.sol#L746:

```
- exeuction
+ execution
```

- IPluginExecutor.sol#L6:

```
- cals
+ calls
```

- AccountStorageV1.sol#L45:

```
- IPliginManager
+ IPluginManager
```

- SessionKeyPermissionsPlugin.sol#L283:

```
- verificaiton
+ verification
```

- SessionKeyPermissionsPlugin.sol#L671:

```
- Unrecognzied
+ Unrecognized
```

Alchemy: Solved in PR 12.

Spearbit: Verified.