Gelato SmartWallet Audit Report

Jun 13, 2025

Table of Contents

Summary 2

Overview 3

Issues 4

[WP-M1] Permissive signature verification implementation (not best security practice): Allows malleable (non-unique) signatures, prone to misuse 4

[WP-M2] The current implementation cannot receive ERC721, ERC1155, and other tokens due to the lack of required hooks. 9

[WP-M3] `execute()` SHOULD REVERT when `modeSelector != EXEC_MODE_DEFAULT && modeSelector != EXEC_MODE_OP_DATA` 16

[WP-M4] `Delegation.execute()` should handle the `calls[i].to == address(0)` properly 19

[WP-M5] The `validateUserOp()` implementation in the `Delegation` contract does not comply with the ERC-4337 specification for `IAccount.validateUserOp()` , which may lead to an unexpected revert: ”AA21 didn’t pay prefund.” 22

[WP-L6] Consider adding a no-op `fallback() external payable` to make EOA addresses set as 7702 Delegated Address behave more like regular EOA addresses 29

[WP-L7] When `mode` is `0x01000000000078210001...` , `opData` is optional, but `_execute()` does not handle the case when `opData` is missing. 31

[WP-I8] After an EOA clears all storage, the nonce is reset and signatures may be subject to replay attacks 35

[WP-D9] Consider documenting the source of ENTRY_POINT_V8 address 39

[WP-I10] Consider implementing ERC-7739: Readable Typed Signatures for Smart Accounts 41

[WP-I11] In cases of invalid signature length or failed recovery, `_verifySignature()` does not return false but `revert InvalidSignature()` , thus `_execute()` does not continue to check `allowUnauthorized` nor `revert Unauthorized()` 42

Appendix 45

Disclaimer 46

Summary

This report has been prepared for Gelato smart contract, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**Gelato**
Codebase	**https://github.com/gelatodigital/smartwallet-contracts**
Commit	**2e4018e9880ccfae7f49b26cba176aec32878158**
Language	**Solidity**

Audit Summary

Delivery Date	**Jun 13, 2025**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**11**

[WP-M1] Permissive signature verification implementation (not best security practice): Allows malleable (non-unique) signatures, prone to misuse

Medium

Issue Description

https://github.com/ethereum/EIPs/blob/bc623b870f0d49bf9a58fec39708782c788c30e4/EIPS/eip-2.md?plain=1#L28

> 1. All transaction signatures whose s-value is greater than `secp256k1n/2` are now considered invalid. *The ECDSA recover precompiled contract* ***remains unchanged*** *and will keep accepting high s-values*; this is useful e.g. if a contract recovers old Bitcoin signatures.

https://github.com/ethereum/EIPs/blob/bc623b870f0d49bf9a58fec39708782c788c30e4/EIPS/eip-2.md?plain=1#L48

> Allowing transactions with any s value with `0 < s < secp256k1n`, as is currently the case, opens a transaction malleability concern, as *one can* ***take any transaction****,* ***flip*** *the s value from* `s` *to* `secp256k1n - s`*, flip the v value (* `27 -> 28` *,* `28 -> 27` *),* ***and the resulting signature would still be valid****.*

For example:

In OpenZeppelin's common ERC-1271 implementation `SignatureChecker.isValidSignatureNow(address signer, bytes32 hash, bytes memory signature)`:

- When `signer.code.length == 0` (traditional EOA): Uses OpenZeppelin's `ECDSA.tryRecover(hash, signature)`, returns `false` when encountering `RecoverError.InvalidSignatureS`, avoiding signature malleability issues
- When `signer.code.length != 0` (e.g., using `Delegation.sol` as the EIP-7702 Delegated Address for `signer` EOA): Uses `signer.isValidSignature(hash, signature)` result
  - Current implementation of `Delegation.isValidSignature(bytes32 digest, bytes`

`calldata signature)` allows upper range `s` (greater than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0`), allows signature malleability, and considers flipped signatures as valid

This creates inconsistent behavior in `SignatureChecker.isValidSignatureNow()` , making it easier for users to mistakenly ignore malleability issues when they actually exist.

If an `isValidSignature()` user treats signatures as identifiers, similar to Gelato contract ExecWithSigsFacet's `wasSignatureUsedAlready[keccak256(_signature)]` , and directly uses `isValidSignature()` results without handling malleability, attackers can bypass restrictions by flipping signatures.

For reference, both ERC-1271 reference implementation and ERC-4337 reference implementation enforce unique signatures by not allowing s values greater than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0` .

Besides the malleability issue caused by upper range `s` values, supporting both 65 bytes and 64 bytes long `signature` also leads to malleability problems, see:

- https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHSA-4h98-2769-gh6h
- https://github.com/OpenZeppelin/openzeppelin-contracts/pull/3610

Both ERC-1271 reference implementation and ERC-4337 reference implementation do not allow signatures with length of 64.

```
71      function isValidSignature(bytes32 digest, bytes calldata signature)
72          external
73          view
74          returns (bytes4)
75      {
76          // https://eips.ethereum.org/EIPS/eip-1271
77          return _verifySignature(digest, signature) ? bytes4(0x1626ba7e) :
    bytes4(0xffffffff);
78      }
79
80      function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash, uint256)
```

```
81          external
82          view
83          returns (uint256)
84      {
85          // https://eips.ethereum.org/EIPS/eip-4337
86          return _verifySignature(userOpHash, userOp.signature) ? 0 : 1;
87      }
```

```
208     function _verifySignature(bytes32 digest, bytes calldata signature)
209         internal
210         view
211         returns (bool)
212     {
213         return ECDSA.recoverCalldata(digest, signature) == address(this);
214     }
```

```
79      /// @dev Recovers the signer's address from a message digest `hash`, and the
    `signature`.
80      function recoverCalldata(bytes32 hash, bytes calldata signature)
81          internal
82          view
83          returns (address result)
84      {
85          /// @solidity memory-safe-assembly
86          assembly {
87              for { let m := mload(0x40) } 1 {
88                  mstore(0x00, 0x8baa579f) // `InvalidSignature()`.
89                  revert(0x1c, 0x04)
90              } {
91                  switch signature.length
92                  case 64 {
93                      let vs := calldataload(add(signature.offset, 0x20))
94                      mstore(0x20, add(shr(255, vs), 27)) // `v`.
95                      mstore(0x40, calldataload(signature.offset)) // `r`.
96                      mstore(0x60, shr(1, shl(1, vs))) // `s`.
97                  }
98                  case 65 {
99                      mstore(0x20, byte(0, calldataload(add(signature.offset,
    0x40)))) // `v`.
```

```
100                     calldatacopy(0x40, signature.offset, 0x40) // Copy `r` and
`s`.
101                 }
102                 default { continue }
103                 mstore(0x00, hash)
104                 result := mload(staticcall(gas(), 1, 0x00, 0x80, 0x01, 0x20))
105                 mstore(0x60, 0) // Restore the zero slot.
106                 mstore(0x40, m) // Restore the free memory pointer.
107                 // `returndatasize()` will be `0x20` upon success, and `0x00`
otherwise.
108                 if returndatasize() { break }
109             }
110         }
111     }
```

```
@@ 1,4 @@
5
6   import {ECDSA} from "./ECDSA.sol";
7   import {IERC1271} from "../../interfaces/IERC1271.sol";
8
9   /**
10   * @dev Signature verification helper that can be used instead of `ECDSA.recover`
to seamlessly support both ECDSA
11   * signatures from externally owned accounts (EOAs) as well as ERC-1271 signatures
from smart contract wallets like
12   * Argent and Safe Wallet (previously Gnosis Safe).
13   */
14  library SignatureChecker {
@@ 15,21 @@
22      function isValidSignatureNow(address signer, bytes32 hash, bytes memory
signature) internal view returns (bool) {
23          if (signer.code.length == 0) {
24              (address recovered, ECDSA.RecoverError err, ) = ECDSA.tryRecover(hash,
signature);
25              return err == ECDSA.RecoverError.NoError && recovered == signer;
26          } else {
27              return isValidERC1271SignatureNow(signer, hash, signature);
28          }
29      }
30
```

```
@@ 31,37 @@
38      function isValidERC1271SignatureNow(
39          address signer,
40          bytes32 hash,
41          bytes memory signature
42      ) internal view returns (bool) {
43          (bool success, bytes memory result) = signer.staticcall(
44              abi.encodeCall(IERC1271.isValidSignature, (hash, signature))
45          );
46          return (success &&
47              result.length >= 32 &&
48              abi.decode(result, (bytes32)) ==
bytes32(IERC1271.isValidSignature.selector));
49      }
50  }
```

Recommendation

Consider using OpenZeppelin's `ECDSA` library like the ERC-4337 reference implementation.

Status

✓ Fixed

[WP-M2] The current implementation cannot receive ERC721, ERC1155, and other tokens due to the lack of required hooks.

Medium

Issue Description

For EOA addresses with a 7702 Delegated Address (`7702DelegatedAddress`), `eoaWith7702DelegatedAddress.code.length` is 23 (the size of `0xef0100 || address`).

https://github.com/ethereum/EIPs/blob/8008342ee834baf5d9a93b22aeb1fcae43694c7e/EIPS/eip-7702.md?plain=1#L154-L155 :

> For code reading, only `CODESIZE` and `CODECOPY` instructions are affected reading. They operate directly on the executing code instead of the delegation. For example, when executing a delegated account `EXTCODESIZE` returns `23` (the size of `0xef0100 || address`) whereas `CODESIZE` returns the size of the code residing at `address` .
> *Note, this means during delegated execution `CODESIZE` and `CODECOPY` produce a different result compared to calling `EXTCODESIZE` and `EXTCODECOPY` on the authority.*

Typical ERC721 and ERC1155 implementations check for hooks like `onERC721Received()` , `onERC1155Received()` , and `onERC1155BatchReceived()` .

To enable EOA addresses to receive ERC721, ERC1155, and other tokens, these hook functions need to be implemented.

For reference:

- The ERC-4337 reference implementation repo's Simple7702Account.sol#L16 has implemented these hooks.
- The MetaMask Delegator smart contract has implemented these hooks with an onlyProxy modifier to prevent accidental token transfers to the Delegated Address. This allows EOA addresses to receive tokens while preventing the implementation address from receiving tokens.

```
17      /**
18       * @dev Performs an acceptance check for the provided `operator` by calling
    {IERC721Receiver-onERC721Received}
```

```
19      * on the `to` address. The `operator` is generally the address that initiated the token transfer (i.e. `msg.sender`).
20      *
21      * The acceptance call is not executed and treated as a no-op if the target address doesn't contain code (i.e. an EOA).
22      * Otherwise, the recipient must implement {IERC721Receiver-onERC721Received} and return the acceptance magic value to accept
23      * the transfer.
24      */
25     function checkOnERC721Received(
26         address operator,
27         address from,
28         address to,
29         uint256 tokenId,
30         bytes memory data
31     ) internal {
32         if (to.code.length > 0) {
33             try IERC721Receiver(to).onERC721Received(operator, from, tokenId, data) returns (bytes4 retval) {
34                 if (retval != IERC721Receiver.onERC721Received.selector) {
35                     // Token rejected
36                     revert IERC721Errors.ERC721InvalidReceiver(to);
37                 }
38             } catch (bytes memory reason) {
39                 if (reason.length == 0) {
40                     // non-IERC721Receiver implementer
41                     revert IERC721Errors.ERC721InvalidReceiver(to);
42                 } else {
43                     assembly ("memory-safe") {
44                         revert(add(32, reason), mload(reason))
45                     }
46                 }
47             }
48         }
49     }
```

```
184     /**
185      * @dev Version of {_update} that performs the token acceptance check by calling
186      * {IERC1155Receiver-onERC1155Received} or {IERC1155Receiver-onERC1155BatchReceived} on the receiver address if it
```

```
187       * contains code (eg. is a smart contract at the moment of execution).
188       *
189       * IMPORTANT: Overriding this function is discouraged because it poses a
    reentrancy risk from the receiver. So any
190       * update to the contract state after this function would break the
    check-effect-interaction pattern. Consider
191       * overriding {_update} instead.
192       */
193      function _updateWithAcceptanceCheck(
194          address from,
195          address to,
196          uint256[] memory ids,
197          uint256[] memory values,
198          bytes memory data
199      ) internal virtual {
200          _update(from, to, ids, values);
201          if (to != address(0)) {
202              address operator = _msgSender();
203              if (ids.length == 1) {
204                  uint256 id = ids.unsafeMemoryAccess(0);
205                  uint256 value = values.unsafeMemoryAccess(0);
206                  ERC1155Utils.checkOnERC1155Received(operator, from, to, id, value,
    data);
207              } else {
208                  ERC1155Utils.checkOnERC1155BatchReceived(operator, from, to, ids,
    values, data);
209              }
210          }
211      }
```

```
17      /**
18       * @dev Performs an acceptance check for the provided `operator` by calling
    {IERC1155Receiver-onERC1155Received}
19       * on the `to` address. The `operator` is generally the address that initiated
    the token transfer (i.e. `msg.sender`).
20       *
21       * The acceptance call is not executed and treated as a no-op if the target
    address doesn't contain code (i.e. an EOA).
22       * Otherwise, the recipient must implement
    {IERC1155Receiver-onERC1155Received} and return the acceptance magic value to
    accept
```

```
23          * the transfer.
24          */
25         function checkOnERC1155Received(
26             address operator,
27             address from,
28             address to,
29             uint256 id,
30             uint256 value,
31             bytes memory data
32         ) internal {
33             if (to.code.length > 0) {
34                 try IERC1155Receiver(to).onERC1155Received(operator, from, id, value,
    data) returns (bytes4 response) {
35                     if (response != IERC1155Receiver.onERC1155Received.selector) {
36                         // Tokens rejected
37                         revert IERC1155Errors.ERC1155InvalidReceiver(to);
38                     }
39                 } catch (bytes memory reason) {
40                     if (reason.length == 0) {
41                         // non-IERC1155Receiver implementer
42                         revert IERC1155Errors.ERC1155InvalidReceiver(to);
43                     } else {
44                         assembly ("memory-safe") {
45                             revert(add(32, reason), mload(reason))
46                         }
47                     }
48                 }
49             }
50         }
51
52         /**
53          * @dev Performs a batch acceptance check for the provided `operator` by
    calling {IERC1155Receiver-onERC1155BatchReceived}
54          * on the `to` address. The `operator` is generally the address that initiated
    the token transfer (i.e. `msg.sender`).
55          *
56          * The acceptance call is not executed and treated as a no-op if the target
    address doesn't contain code (i.e. an EOA).
57          * Otherwise, the recipient must implement
    {IERC1155Receiver-onERC1155Received} and return the acceptance magic value to
    accept
58          * the transfer.
59          */
```

```
60      function checkOnERC1155BatchReceived(
61          address operator,
62          address from,
63          address to,
64          uint256[] memory ids,
65          uint256[] memory values,
66          bytes memory data
67      ) internal {
68          if (to.code.length > 0) {
69              try IERC1155Receiver(to).onERC1155BatchReceived(operator, from, ids,
    values, data) returns (
70                  bytes4 response
71              ) {
72                  if (response != IERC1155Receiver.onERC1155BatchReceived.selector)
    {
73                      // Tokens rejected
74                      revert IERC1155Errors.ERC1155InvalidReceiver(to);
75                  }
76              } catch (bytes memory reason) {
77                  if (reason.length == 0) {
78                      // non-IERC1155Receiver implementer
79                      revert IERC1155Errors.ERC1155InvalidReceiver(to);
80                  } else {
81                      assembly ("memory-safe") {
82                          revert(add(32, reason), mload(reason))
83                      }
84                  }
85              }
86          }
87      }
```

Simple7702Account.sol

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.28;
3
4   import "@openzeppelin/contracts/utils/introspection/IERC165.sol";
5   import "@openzeppelin/contracts/interfaces/IERC1271.sol";
6   import "@openzeppelin/contracts/token/ERC1155/utils/ERC1155Holder.sol";
7   import "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";
8   import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
```

```
 9  import "../core/Helpers.sol";
10  import "../core/BaseAccount.sol";
11
12  /**
13   * Simple7702Account.sol
14   * A minimal account to be used with EIP-7702 (for batching) and ERC-4337 (for gas sponsoring)
15   */
16  contract Simple7702Account is BaseAccount, IERC165, IERC1271, ERC1155Holder, ERC721Holder {
    @@ 17,65 @@
66  }
```

EIP7702StatelessDeleGator/src/EIP7702/EIP7702DeleGatorCore.sol

```
102     /**
103      * @dev Prevents direct calls to the implementation.
104      * @dev Check that the execution is being performed through a delegatecall call.
105      */
106     modifier onlyProxy() {
107         if (address(this) == __self) revert UnauthorizedCallContext();
108         _;
109     }
```

EIP7702StatelessDeleGator/src/EIP7702/EIP7702DeleGatorCore.sol

```
323     /// @inheritdoc IERC721Receiver
324     function onERC721Received(address, address, uint256, bytes memory) external view override onlyProxy returns (bytes4) {
325         return this.onERC721Received.selector;
326     }
327
328     /// @inheritdoc IERC1155Receiver
329     function onERC1155Received(
330         address,
331         address,
332         uint256,
333         uint256,
```

```
334             bytes memory
335         )
336             external
337             view
338             override
339             onlyProxy
340             returns (bytes4)
341         {
342             return this.onERC1155Received.selector;
343         }
344
345         /// @inheritdoc IERC1155Receiver
346         function onERC1155BatchReceived(
347             address,
348             address,
349             uint256[] memory,
350             uint256[] memory,
351             bytes memory
352         )
353             external
354             view
355             override
356             onlyProxy
357             returns (bytes4)
358         {
359             return this.onERC1155BatchReceived.selector;
360         }
```

Status

✓ Fixed

[WP-M3] `execute()` SHOULD REVERT when `modeSelector != EXEC_MODE_DEFAULT && modeSelector != EXEC_MODE_OP_DATA`

Medium

Issue Description

According to `supportsExecutionMode()` L67 - L69, Delegation only supports two `modeSelector` values: `EXEC_MODE_DEFAULT` (0x00000000) and `EXEC_MODE_OP_DATA` (0x78210001).

However, in the `execute()` function, when `modeSelector != EXEC_MODE_DEFAULT` , there is no additional check on the `modeSelector` . It processes any non- `EXEC_MODE_DEFAULT` (0x00000000) `modeSelector` using the same logic as `EXEC_MODE_DEFAULT` (0x00000000).

Notes:

- According to https://eips.ethereum.org/EIPS/eip-7821#overview, different `mode` values require different encoding formats for `executionData` .
- The Reference Implementation at https://eips.ethereum.org/EIPS/eip-7821#reference-implementation shows that `execute()` should `revert UnsupportedExecutionMode();` when encountering an invalid mode.

```
19  contract Delegation is IERC7821, IERC1271, IERC4337, EIP712 {
    @@ 20,52 @@
53
54      receive() external payable {}
55
56      function execute(bytes32 mode, bytes calldata executionData) external payable
    {
57          _execute(mode, executionData, false);
58      }
59
60      function supportsExecutionMode(bytes32 mode) external pure returns (bool) {
61          (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
    _decodeExecutionMode(mode);
62
63          if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
64              return false;
```

```
65          }
66
67          if (modeSelector != EXEC_MODE_DEFAULT && modeSelector !=
    EXEC_MODE_OP_DATA) {
68              return false;
69          }
70
71          return true;
72      }
73
    @@ 74,121 @@
122
123     function _execute(bytes32 mode, bytes calldata executionData, bool
    allowUnauthorized)
124         internal
125     {
126         (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
    _decodeExecutionMode(mode);
127
128         if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
129             revert UnsupportedExecutionMode();
130         }
131
132         Call[] calldata calls = _decodeCalls(executionData);
133
134         if (modeSelector == EXEC_MODE_DEFAULT) {
135             // https://eips.ethereum.org/EIPS/eip-7821
136             // If `opData` is empty, the implementation SHOULD require that
    `msg.sender ==
137             // address(this)`.
138             // If `msg.sender` is an authorized entry point, then `execute` MAY
    accept calls from
139             // the entry point.
140             if (msg.sender != address(this) && msg.sender != ENTRY_POINT_V8 &&
    !allowUnauthorized) {
141                 revert Unauthorized();
142             }
143
144             _executeCalls(calls);
145         } else {
146             bytes calldata opData = _decodeOpData(executionData);
147             bytes calldata signature = _decodeSignature(opData);
```

```
148
149             uint256 nonce = _getAndUseNonce(_decodeNonceKey(opData));
150             bytes32 digest = _computeDigest(mode, calls, nonce);
151
152             // If `opData` is not empty, the implementation SHOULD use the
    signature encoded in
153             // `opData` to determine if the caller can perform the execution.
154             if (!_verifySignature(digest, signature) && !allowUnauthorized) {
155                 revert Unauthorized();
156             }
157
158             _executeCalls(calls);
159         }
160     }
161

    @@ 162,287 @@

288
289     function _decodeExecutionMode(bytes32 mode)
290         internal
291         pure
292         returns (bytes1 calltype, bytes1 execType, bytes4 modeSelector, bytes22
    modePayload)
293     {
294         // https://eips.ethereum.org/EIPS/eip-7579
295         // https://eips.ethereum.org/EIPS/eip-7821
296         assembly {
297             calltype := mode
298             execType := shl(8, mode)
299             modeSelector := shl(48, mode)
300             modePayload := shl(80, mode)
301         }
302     }
303

    @@ 304,312 @@

313 }
```

Status

✓ Fixed

[WP-M4] `Delegation.execute()` should handle the `calls[i].to == address(0)` properly

Medium

Issue Description

According to https://eips.ethereum.org/EIPS/eip-7821#overview and https://eips.ethereum.org/EIPS/eip-7821#replacing-address0-with-addressthis, setting `Call.to` to `address(0)` can represent `address(this)` for calldata compression optimization.

The Reference Implementation at https://eips.ethereum.org/EIPS/eip-7821#reference-implementation uses `address to = c.to == address(0) ? address(this) : c.to;` in its `_execute(Call[] memory calls)` function.

```
56      function execute(bytes32 mode, bytes calldata executionData) external payable
    {
57          _execute(mode, executionData, false);
58      }
```

```
123     function _execute(bytes32 mode, bytes calldata executionData, bool
    allowUnauthorized)
124         internal
125     {
126         (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
    _decodeExecutionMode(mode);
127
128         if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
129             revert UnsupportedExecutionMode();
130         }
131
132         Call[] calldata calls = _decodeCalls(executionData);
133
134         if (modeSelector == EXEC_MODE_DEFAULT) {
135             // https://eips.ethereum.org/EIPS/eip-7821
136             // If `opData` is empty, the implementation SHOULD require that
    `msg.sender ==
137             // address(this)`.
138             // If `msg.sender` is an authorized entry point, then `execute` MAY
    accept calls from
```

```
139             // the entry point.
140             if (msg.sender != address(this) && msg.sender != ENTRY_POINT_V8 &&
    !allowUnauthorized) {
141                 revert Unauthorized();
142             }
143
144             _executeCalls(calls);
145         } else {
146             bytes calldata opData = _decodeOpData(executionData);
147             bytes calldata signature = _decodeSignature(opData);
148
149             uint256 nonce = _getAndUseNonce(_decodeNonceKey(opData));
150             bytes32 digest = _computeDigest(mode, calls, nonce);
151
152             // If `opData` is not empty, the implementation SHOULD use the
    signature encoded in
153             // `opData` to determine if the caller can perform the execution.
154             if (!_verifySignature(digest, signature) && !allowUnauthorized) {
155                 revert Unauthorized();
156             }
157
158             _executeCalls(calls);
159         }
160     }
161
162     function _executeCalls(Call[] calldata calls) internal {
163         for (uint256 i = 0; i < calls.length; i++) {
164             (bool success, bytes memory data) =
165                 calls[i].to.call{value: calls[i].value}(calls[i].data);
166
167             if (!success) {
168                 assembly {
169                     revert(add(data, 0x20), mload(data))
170                 }
171             }
172         }
173     }
174
175     function _decodeCalls(bytes calldata executionData)
176         internal
177         pure
178         returns (Call[] calldata calls)
179     {
```

```
180         // If `opData` is empty, `executionData` is simply `abi.encode(calls)`.
181         // We decode this from calldata rather than abi.decode which avoids a
    memory copy
182         assembly {
183             let offset := add(executionData.offset,
    calldataload(executionData.offset))
184             calls.offset := add(offset, 0x20)
185             calls.length := calldataload(offset)
186         }
187     }
```

Status

✓ Fixed

[WP-M5] The `validateUserOp()` implementation in the `Delegation` contract does not comply with the ERC-4337 specification for `IAccount.validateUserOp()`, which may lead to an unexpected revert: ”AA21 didn’t pay prefund.”

Medium

Issue Description

https://eips.ethereum.org/EIPS/eip-4337#smart-contract-account-interface

The current implementation only checks the signature, but the specification requires:

https://github.com/ethereum/ERCs/blob/080f189cebda98c0f2aacf027f845992757fa09d/ERCS/erc-4337.md?plain=1#L135-L144

````
119  ### Smart Contract Account Interface
120
121  The core interface required for the Smart Contract Account to have is:
122
123  ```solidity
124  interface IAccount {
125    function validateUserOp
126        (PackedUserOperation calldata userOp, bytes32 userOpHash, uint256
     missingAccountFunds)
127        external returns (uint256 validationData);
128  }
129  ```
130
131  The `userOpHash` is a hash over the `userOp` (except `signature`), `entryPoint`
     and `chainId`.
132
133  The Smart Contract Account:
134
135  * MUST validate the caller is a trusted `EntryPoint`
136  * MUST validate that the signature is a valid signature of the `userOpHash`, and
137    SHOULD return `SIG_VALIDATION_FAILED` (`1`) without reverting on signature
       mismatch. Any other error MUST revert.
138  * SHOULD not return early when returning `SIG_VALIDATION_FAILED` (`1`). Instead,
     it SHOULD complete the normal flow to enable performing a gas estimation for the
     validation function.
````

```
* MUST pay the `EntryPoint` (caller) at least the `missingAccountFunds` (which might be zero, in case the current `sender`'s deposit is sufficient)
* The `sender` MAY pay more than this minimum to cover future transactions. It can also call `withdrawTo` to retrieve it later at any time.
* The return value MUST be packed of `aggregator`/`authorizer`, `validUntil` and `validAfter` timestamps.
  * `aggregator`/`authorizer` - 0 for valid signature, 1 to mark signature failure. Otherwise, an address of an `aggregator`/`authorizer` contract, as defined in [ERC-7766](./eip-7766.md).
  * `validUntil` is 6-byte timestamp value, or zero for "infinite". The `UserOperation` is valid only up to this time.
  * `validAfter` is 6-byte timestamp. The `UserOperation` is valid only after this time.

The Smart Contract Account MAY implement the interface `IAccountExecute`

```solidity
interface IAccountExecute {
  function executeUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash) external;
}
```

This method will be called by the `EntryPoint` with the current UserOperation, instead of executing the `callData` itself directly on the `sender`.
```

For example, if `validateUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 missingAccountFunds)` doesn't pay for the `missingAccountFunds`, it would prevent the UserOperation from being executed. see EntryPoint.solL555-557

Reference Implementation: https://eips.ethereum.org/EIPS/eip-4337#reference-implementation https://github.com/eth-infinitism/account-abstraction/blob/v0.8.0/contracts/core/BaseAccount.sol#L80-L89

```
/**
 * Basic account implementation.
 * This contract provides the basic logic for implementing the IAccount interface - validateUserOp
 * Specific account implementation should inherit it and provide the account-specific logic.
```

```
17   */
18  abstract contract BaseAccount is IAccount {
    @@ 19,77 @@
78
79      /// @inheritdoc IAccount
80      function validateUserOp(
81          PackedUserOperation calldata userOp,
82          bytes32 userOpHash,
83          uint256 missingAccountFunds
84      ) external virtual override returns (uint256 validationData) {
85          _requireFromEntryPoint();
86          validationData = _validateSignature(userOp, userOpHash);
87          _validateNonce(userOp.nonce);
88          _payPrefund(missingAccountFunds);
89      }
90
91      /**
92       * Ensure the request comes from the known entrypoint.
93       */
94      function _requireFromEntryPoint() internal view virtual {
95          require(
96              msg.sender == address(entryPoint()),
97              "account: not from EntryPoint"
98          );
99      }
100
    @@ 101,141 @@
142
143     /**
144      * Sends to the entrypoint (msg.sender) the missing funds for this
    transaction.
145      * SubClass MAY override this method for better funds management
146      * (e.g. send to the entryPoint more than the minimum required, so that in
    future transactions
147      * it will not be required to send again).
148      * @param missingAccountFunds - The minimum value this method should send the
    entrypoint.
149      *                              This value MAY be zero, in case there is
    enough deposit,
150      *                              or the userOp has a paymaster.
151      */
152     function _payPrefund(uint256 missingAccountFunds) internal virtual {
```

```
153         if (missingAccountFunds != 0) {
154             (bool success,) = payable(msg.sender).call{
155                     value: missingAccountFunds
156                 }("");
157             (success);
158             // Ignore failure (its EntryPoint's job to verify, not account.)
159         }
160     }
161 }
```

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.29;
3
4  // https://eips.ethereum.org/EIPS/eip-4337
5  interface IERC4337 {
6      struct PackedUserOperation {
7          address sender;
8          uint256 nonce;
9          bytes initCode;
10         bytes callData;
11         bytes32 accountGasLimits;
12         uint256 preVerificationGas;
13         bytes32 gasFees;
14         bytes paymasterAndData;
15         bytes signature;
16     }
17
18     function validateUserOp(
19         PackedUserOperation calldata userOp,
20         bytes32 userOpHash,
21         uint256 missingAccountFunds
22     ) external returns (uint256);
23 }
```

```
@@ 1,5 @@
6  import {IERC4337} from "./interfaces/IERC4337.sol";
@@ 7,17 @@
```

```
18
19  contract Delegation is IERC7821, IERC1271, IERC4337, EIP712 {
    @@ 20,81 @@
82
83      function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash, uint256)
84          external
85          view
86          returns (uint256)
87      {
88          // https://eips.ethereum.org/EIPS/eip-4337
89          return _verifySignature(userOpHash, userOp.signature) ? 0 : 1;
90      }
91
    @@ 92,218 @@
219
220     function _verifySignature(bytes32 digest, bytes calldata signature)
221         internal
222         view
223         returns (bool)
224     {
225         // If `signature` length is 64 or 65, treat it as secp256k1 signature
226         if (signature.length == 64 || signature.length == 65) {
227             return ECDSA.recoverCalldata(digest, signature) == address(this);
228         }
229
230         // `data` is `abi.encode(keyHash, signature)`.
231         bytes32 keyHash;
232         assembly {
233             keyHash := calldataload(signature.offset)
234
235             let offset := add(signature.offset, calldataload(add(signature.offset,
    0x20)))
236             signature.offset := add(offset, 0x20)
237             signature.length := calldataload(offset)
238         }
239
240         bytes storage pubkey = _getStorage().pubkey[keyHash];
241
242         (bytes32 x, bytes32 y) = P256.tryDecodePoint(pubkey);
243
244         return WebAuthn.verify(
```

```
245             abi.encode(digest), false,
    WebAuthn.tryDecodeAuthCompactCalldata(signature), x, y
246         );
247     }
248
    @@ 249,312 @@
313 }
```

```
520     /**
521      * Call account.validateUserOp.
522      * Revert (with FailedOp) in case validateUserOp reverts, or account didn't
    send required prefund.
523      * Decrement account's deposit if needed.
524      * @param opIndex         - The operation index.
525      * @param op              - The user operation.
526      * @param opInfo          - The operation info.
527      * @param requiredPrefund - The required prefund amount.
528      * @return validationData - The account's validationData.
529      */
530     function _validateAccountPrepayment(
531         uint256 opIndex,
532         PackedUserOperation calldata op,
533         UserOpInfo memory opInfo,
534         uint256 requiredPrefund
535     )
536     internal virtual
537     returns (
538         uint256 validationData
539     )
540     {
541         unchecked {
542             MemoryUserOp memory mUserOp = opInfo.mUserOp;
543             address sender = mUserOp.sender;
544             _createSenderIfNeeded(opIndex, opInfo, op.initCode);
545             address paymaster = mUserOp.paymaster;
546             uint256 missingAccountFunds = 0;
547             if (paymaster == address(0)) {
548                 uint256 bal = balanceOf(sender);
549                 missingAccountFunds = bal > requiredPrefund
550                     ? 0
```

```
551                     : requiredPrefund - bal;
552             }
553             validationData = _callValidateUserOp(opIndex, op, opInfo,
    missingAccountFunds);
554             if (paymaster == address(0)) {
555                 if (!_tryDecrementDeposit(sender, requiredPrefund)) {
556                     revert FailedOp(opIndex, "AA21 didn't pay prefund");
557                 }
558             }
559         }
560     }
```

Status

✓ Fixed

[WP-L6] Consider adding a no-op `fallback() external payable` to make EOA addresses set as 7702 Delegated Address behave more like regular EOA addresses

Low

Issue Description

Regular EOA addresses do not revert on any calldata.

The current implementation reverts on inbound transactions with calldata (e.g., sending messages).

For reference, Simple7702Account.solL60-62 in the ERC-4337 reference implementation repo has such a `fallback() external payable` .

```
@@ 1,10 @@

/**
 * Simple7702Account.sol
 * A minimal account to be used with EIP-7702 (for batching) and ERC-4337 (for gas sponsoring)
 */
contract Simple7702Account is BaseAccount, IERC165, IERC1271, ERC1155Holder, ERC721Holder {
@@ 17,58 @@

    // accept incoming calls (with or without value), to mimic an EOA.
    fallback() external payable {
    }

    receive() external payable {
    }
}
```

Status

✓ Fixed

[WP-L7] When `mode` is `0x0100000000078210001...`, `opData` is optional, but `_execute()` does not handle the case when `opData` is missing.

Low

Issue Description

According to https://eips.ethereum.org/EIPS/eip-7821#overview and https://eips.ethereum.org/EIPS/eip-7821#optional-encoding-of-opdata-in-executiondata, `0x0100000000078210001...` Single batch mode supports ***optional*** `opData`.

```
    /// Authorization checks:
    /// - If `opData` is empty, the implementation SHOULD require that
    ///   `msg.sender == address(this)`.
```

The Reference Implementation at L77-L83 first checks if `opData` exists, then at L86-L90 only decodes `executionData` as `(Call[], bytes)` when `opData` is present.

Delegation.sol

```
56      function execute(bytes32 mode, bytes calldata executionData) external payable
    {
57          _execute(mode, executionData, false);
58      }
```

Delegation.sol

```
123     function _execute(bytes32 mode, bytes calldata executionData, bool
    allowUnauthorized)
124         internal
125     {
126         (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
    _decodeExecutionMode(mode);
127
128         if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
```

```
129             revert UnsupportedExecutionMode();
130         }
131
132         Call[] calldata calls = _decodeCalls(executionData);
133
134         if (modeSelector == EXEC_MODE_DEFAULT) {
135             // https://eips.ethereum.org/EIPS/eip-7821
136             // If `opData` is empty, the implementation SHOULD require that
    `msg.sender ==
137             // address(this)`.
138             // If `msg.sender` is an authorized entry point, then `execute` MAY
    accept calls from
139             // the entry point.
140             if (msg.sender != address(this) && msg.sender != ENTRY_POINT_V8 &&
    !allowUnauthorized) {
141                 revert Unauthorized();
142             }
143
144             _executeCalls(calls);
145         } else {
146             bytes calldata opData = _decodeOpData(executionData);
147             bytes calldata signature = _decodeSignature(opData);
148
149             uint256 nonce = _getAndUseNonce(_decodeNonceKey(opData));
150             bytes32 digest = _computeDigest(mode, calls, nonce);
151
152             // If `opData` is not empty, the implementation SHOULD use the
    signature encoded in
153             // `opData` to determine if the caller can perform the execution.
154             if (!_verifySignature(digest, signature) && !allowUnauthorized) {
155                 revert Unauthorized();
156             }
157
158             _executeCalls(calls);
159         }
160     }
```

```
189     function _decodeOpData(bytes calldata executionData)
190         internal
191         pure
192         returns (bytes calldata opData)
```

```
193     {
194         // If `opData` is not empty, `executionData` is `abi.encode(calls, opData)`.
195         // We decode this from calldata rather than abi.decode which avoids a memory copy
196         assembly {
197             let offset := add(executionData.offset, calldataload(add(executionData.offset, 0x20)))
198             opData.offset := add(offset, 0x20)
199             opData.length := calldataload(offset)
200         }
201     }
```

```
1  // SPDX-License-Identifier: CC0-1.0
2  pragma solidity ^0.8.4;
3
4  /// @notice Minimal batch executor mixin.
5  abstract contract ERC7821 {
@@ 6,26 @@
27
@@ 28,61 @@
62     function execute(bytes32 mode, bytes memory executionData)
63         public
64         payable
65         virtual
66     {
67         uint256 id = _executionModeId(mode);
68         if (id == 3) {
@@ 69,73 @@
74             return;
75         }
76         if (id == uint256(0)) revert UnsupportedExecutionMode();
77         bool tryWithOpData;
78         /// @solidity memory-safe-assembly
79         assembly {
80             let t := gt(mload(add(executionData, 0x20)), 0x3f)
81             let executionDataLength := mload(executionData)
82             tryWithOpData := and(eq(id, 2), and(gt(executionDataLength, 0x3f), t))
83         }
```

```
84          Call[] memory calls;
85          bytes memory opData;
86          if (tryWithOpData) {
87              (calls, opData) = abi.decode(executionData, (Call[], bytes));
88          } else {
89              calls = abi.decode(executionData, (Call[]));
90          }
91          _execute(calls, opData);
92      }
93
    @@ 94,162 @@
163 }
```

Status

✓ Fixed

[WP-I8] After an EOA clears all storage, the nonce is reset and signatures may be subject to replay attacks

Informational

Issue Description

Consider:

- Warn users about potential impacts of clearing storage-location erc7201:gelato.delegation.storage
- Set the version number in domain separation during initialization. Cleaning storage and reinitialization will invalidate all previous signatures

Related Info: EIP-7702 https://github.com/ethereum/EIPs/blob/015f08bba346696a02379f1dec40cd38db38b2c9/EIPS/eip-7702.md?plain=1#L551-L553

```
535 ### Storage management
536
537 Changing an account's delegation is a security-critical operation that should
538 not be done lightly, especially if the newly delegated code is not purposely
539 designed and tested as an upgrade to the old one.
540
541 In particular, in order to ensure a safe migration of an account from one
542 delegate contract to another, it's important for these contracts to use storage
543 in a way that avoids accidental collisions among them. For example, using
544 [ERC-7201](./eip-7201.md) a contract may root its storage layout at a slot
545 dependent on a unique identifier. To simplify this, smart contract languages may
546 provide a way of re-rooting the entire storage layout of existing contract
547 source code.
548
549 If all contracts previously delegated to by the account used the approach
550 described above, a migration should not cause any issues. However, if there is
551 any doubt, it is recommended to first clear all account storage, an operation
552 that is not natively offered by the protocol but that a special-purpose delegate
553 contract can be designed to implement.
```

```
19  contract Delegation is IERC7821, IERC1271, IERC4337, EIP712 {
@@ 20,24 @@
25
26      // https://eips.ethereum.org/EIPS/eip-7201
27      /// @custom:storage-location erc7201:gelato.delegation.storage
28      struct Storage {
29          mapping(uint192 => uint64) nonceSequenceNumber;
30          mapping(bytes32 => bytes) pubkey;
31      }
32
33      // keccak256(abi.encode(uint256(keccak256("gelato.delegation.storage")) - 1))
&
34      // ~bytes32(uint256(0xff));
35      bytes32 private constant STORAGE_LOCATION =
36          0x1581abf533ae210f1ff5d25f322511179a9a65d8d8e43c998eab264f924af900;
37
@@ 38,54 @@
55
56      function execute(bytes32 mode, bytes calldata executionData) external payable
{
57          _execute(mode, executionData, false);
58      }
59
@@ 60,121 @@
122
123     function _execute(bytes32 mode, bytes calldata executionData, bool
allowUnauthorized)
124         internal
125     {
126         (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
_decodeExecutionMode(mode);
127
128         if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
129             revert UnsupportedExecutionMode();
130         }
131
132         Call[] calldata calls = _decodeCalls(executionData);
133
134         if (modeSelector == EXEC_MODE_DEFAULT) {
```

```
@@ 135,144 @@
145         } else {
146             bytes calldata opData = _decodeOpData(executionData);
147             bytes calldata signature = _decodeSignature(opData);
148 
149             uint256 nonce = _getAndUseNonce(_decodeNonceKey(opData));
150             bytes32 digest = _computeDigest(mode, calls, nonce);
151 
152             // If `opData` is not empty, the implementation SHOULD use the
    signature encoded in
153             // `opData` to determine if the caller can perform the execution.
154             if (!_verifySignature(digest, signature) && !allowUnauthorized) {
155                 revert Unauthorized();
156             }
157 
158             _executeCalls(calls);
159         }
160     }
161 
@@ 162,247 @@
248 
249     function _computeDigest(bytes32 mode, Call[] calldata calls, uint256 nonce)
250         internal
251         view
252         returns (bytes32)
253     {
254         bytes32[] memory callsHashes = new bytes32[](calls.length);
255         for (uint256 i = 0; i < calls.length; i++) {
256             callsHashes[i] = keccak256(
257                 abi.encode(CALL_TYPEHASH, calls[i].to, calls[i].value,
    keccak256(calls[i].data))
258             );
259         }
260 
261         bytes32 executeHash = keccak256(
262             abi.encode(EXECUTE_TYPEHASH, mode,
    keccak256(abi.encodePacked(callsHashes)), nonce)
263         );
264 
265         return _hashTypedData(executeHash);
266     }
267 
```

```
268     function _getAndUseNonce(uint192 key) internal returns (uint256) {
269         uint64 seq = _getStorage().nonceSequenceNumber[key];
270         _getStorage().nonceSequenceNumber[key]++;
271         return _encodeNonce(key, seq);
272     }
273
```

```
@@ 274,302 @@
```

```
303
304     function _domainNameAndVersion()
305         internal
306         pure
307         override
308         returns (string memory name, string memory version)
309     {
310         name = "GelatoDelegation";
311         version = "0.0.1";
312     }
313 }
```

Status

ⓘ Acknowledged

[WP-D9] Consider documenting the source of ENTRY_POINT_V8 address

Issue Description

Given the high privileges of ENTRY_POINT_V8 (ability to make arbitrary calls as wallet without signature), its address correctness is critical for wallet security. Consider explicitly documenting its source for user verification.

For example, consider referencing https://github.com/ethereum/ERCs/blob/9f18295f250a18a33eac52c8ea60b0516eb6d4b8/ERCS/erc-4337.md?plain=1#L581-L583, https://github.com/eth-infinitism/account-abstraction/releases/tag/v0.8.0, and https://github.com/eth-infinitism/account-abstraction/blob/v0.8.0/README.md?plain=1#L76

```
53      function execute(bytes32 mode, bytes calldata executionData) external payable
    {
54          _execute(mode, executionData, false);
55      }
```

```
111     function _execute(bytes32 mode, bytes calldata executionData, bool
    allowUnauthorized)
112         internal
113     {
114         (bytes1 callType, bytes1 execType, bytes4 modeSelector,) =
    _decodeExecutionMode(mode);
115
116         if (callType != CALL_TYPE_BATCH || execType != EXEC_TYPE_DEFAULT) {
117             revert UnsupportedExecutionMode();
118         }
119
120         Call[] calldata calls = _decodeCalls(executionData);
121
122         if (modeSelector == EXEC_MODE_DEFAULT) {
123             // https://eips.ethereum.org/EIPS/eip-7821
124             // If `opData` is empty, the implementation SHOULD require that
    `msg.sender ==
125             // address(this)`.
126             // If `msg.sender` is an authorized entry point, then `execute` MAY
    accept calls from
```

```
127             // the entry point.
128             if (msg.sender != address(this) && msg.sender != ENTRY_POINT_V8 &&
    !allowUnauthorized) {
129                 revert Unauthorized();
130             }
131
132             _executeCalls(calls);
133         } else {
    @@ 134,146 @@
147         }
148     }
149
150     function _executeCalls(Call[] calldata calls) internal {
151         for (uint256 i = 0; i < calls.length; i++) {
152             (bool success, bytes memory data) =
153                 calls[i].to.call{value: calls[i].value}(calls[i].data);
154
155             if (!success) {
156                 assembly {
157                     revert(add(data, 0x20), mload(data))
158                 }
159             }
160         }
161     }
```

```
11  // https://eips.ethereum.org/EIPS/eip-4337
12  address constant ENTRY_POINT_V8 = 0x4337084D9E255Ff0702461CF8895CE9E3b5Ff108;
```

Status

✓ Fixed

[WP-I10] Consider implementing ERC-7739: Readable Typed Signatures for Smart Accounts

Informational

Issue Description

When the EOA address is also the owner of other smart contracts that implement ERC-1271, the same signature will be considered valid for all these smart contracts. See also: https://ethereum-magicians.org/t/erc-7739-readable-typed-signatures-for-smart-accounts/20513

As a reference, Uniswap's Calibur MinimalDelegation implemented ERC-7739: https://github.com/Uniswap/calibur/blob/main/src/MinimalDelegation.sol#L46

Status

Acknowledged

[WP-I11] In cases of invalid signature length or failed recovery, `_verifySignature()` does not return false but `revert InvalidSignature()`, thus `_execute()` does not continue to check `allowUnauthorized` nor `revert Unauthorized()`

Informational

Issue Description

`ECDSA.recoverCalldata(digest, signature)` will `revert InvalidSignature()` at L89 when the signature "length is not 64/65" or when "0x01(ecRecover) Precompiled Contract's returndatasize is 0 (an address cannot be recovered or not enough gas was given)".

The current implementation might be intentional, based on the observations:

- `allowUnauthorized` is only set to true in Simulation.sol
- Using different custom errors for invalid signature and signature mismatch makes it easier to identify the revert reason

If it is *indeed* necessary to check `allowUnauthorized` or `revert Unauthorized()` when signature invalid, consider using OpenZeppelin's `ECDSA.tryRecover(bytes32 hash, bytes memory signature)`.

```
53      function execute(bytes32 mode, bytes calldata executionData) external payable
    {
54          _execute(mode, executionData, false);
55      }
```

```
111     function _execute(bytes32 mode, bytes calldata executionData, bool
    allowUnauthorized)
112         internal
113     {
    @@ 114,120 @@
121
122         if (modeSelector == EXEC_MODE_DEFAULT) {
```

```
@@ 123,132 @@
133         } else {
134             bytes calldata opData = _decodeOpData(executionData);
135             bytes calldata signature = _decodeSignature(opData);
136 
137             uint256 nonce = _getAndUseNonce(_decodeNonceKey(opData));
138             bytes32 digest = _computeDigest(mode, calls, nonce);
139 
140             // If `opData` is not empty, the implementation SHOULD use the signature encoded in
141             // `opData` to determine if the caller can perform the execution.
142             if (!_verifySignature(digest, signature) && !allowUnauthorized) {
143                 revert Unauthorized();
144             }
145 
146             _executeCalls(calls);
147         }
148     }
```

```
208     function _verifySignature(bytes32 digest, bytes calldata signature)
209         internal
210         view
211         returns (bool)
212     {
213         return ECDSA.recoverCalldata(digest, signature) == address(this);
214     }
```

```
79     /// @dev Recovers the signer's address from a message digest `hash`, and the `signature`.
80     function recoverCalldata(bytes32 hash, bytes calldata signature)
81         internal
82         view
83         returns (address result)
84     {
85         /// @solidity memory-safe-assembly
86         assembly {
87             for { let m := mload(0x40) } 1 {
88                 mstore(0x00, 0x8baa579f) // `InvalidSignature()`.
89                 revert(0x1c, 0x04)
90             } {
91                 switch signature.length
```

```
92                  case 64 {
93                      let vs := calldataload(add(signature.offset, 0x20))
94                      mstore(0x20, add(shr(255, vs), 27)) // `v`.
95                      mstore(0x40, calldataload(signature.offset)) // `r`.
96                      mstore(0x60, shr(1, shl(1, vs))) // `s`.
97                  }
98                  case 65 {
99                      mstore(0x20, byte(0, calldataload(add(signature.offset,
    0x40)))) // `v`.
100                     calldatacopy(0x40, signature.offset, 0x40) // Copy `r` and
    `s`.
101                 }
102                 default { continue }
103                 mstore(0x00, hash)
104                 result := mload(staticcall(gas(), 1, 0x00, 0x80, 0x01, 0x20))
105                 mstore(0x60, 0) // Restore the zero slot.
106                 mstore(0x40, m) // Restore the free memory pointer.
107                 // `returndatasize()` will be `0x20` upon success, and `0x00`
    otherwise.
108                 if returndatasize() { break }
109             }
110         }
111     }
```

Status

✓ Fixed

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Smart Contract technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.