Security Assessment Report

Volta Multisig Smart Wallet

August 06, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Volta Multisig Smart Wallet smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 9 issues or questions.

program	type	commit
smart_contract_4337	Solidity	`79f945ec2d575e66bab5c2754391142e6306f254`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
SMART_CONTRACT_4337		
[H-01] Incorrect signature counting degrades quorum requirements	High	Resolved
[L-01] Missing deduplication checks when pushing addresses to arrays	Low	Acknowledged
[L-02] Missing `userOpHash` validation	Low	Acknowledged
[L-03] Potential ambiguity in `validationData` parsing	Low	Acknowledged
[I-01] `_validateExecuteCallData` fails to parse dynamic type parameters	Info	Acknowledged
[I-02] Gas optimization opportunities	Info	Acknowledged
[I-03] Missing data length check in `_validateExecuteCallData`	Info	Acknowledged
[I-04] Check `validUntil` against current timestamp	Info	Acknowledged
[I-05] Incorrect comment for the `execute` function	Info	Acknowledged

Findings in Detail

SMART_CONTRACT_4337
[H-01] Incorrect signature counting degrades quorum requirements

Identified in commit `79f945e`.

The `isValidSignature` function in the `VoltaAccount` contract handles multi-signature verification by validating signer participation against a threshold.

Verification passes if the signer count from `_countUniqueSigners` meets the minimum quorum:

```
/* contracts/VoltaAccount.sol */
349 | function isValidSignature(
350 |     bytes32 hash,
351 |     bytes calldata signature
352 | ) external view returns (bytes4) {
384 |     // Count how many valid, non-Volta owner signatures we have
385 |     (uint8 numSigners, ) = _countUniqueSigners(signature, hashWithSeparator);
387 |     // Signature is only valid if quorum is met
388 |     if (numSigners < minQuorum) {
389 |         return ERC1271_INVALID_SIGNATURE;
390 |     }
391 |     return ERC1271_MAGIC_VALUE;
392 | }
```

The `_countUniqueSigners` implementation contains an issue:

```
/* contracts/VoltaAccount.sol */
218 | function _countUniqueSigners(bytes calldata signature, bytes32 messageHash) internal view
219 | returns (uint8 numOwnerSignatures, bool voltaSigned) {
220 |     address[] memory signers = new address[](owners.length);
221 |     for (uint i = 0; i < signature.length; i += 65) {
222 |         bytes calldata sig = signature[i:65+i];
223 |         address signer = messageHash.recover(sig);
224 |         if (isVolta(signer)) {
225 |             if (i == 0) {
226 |                 voltaSigned = true;
227 |                 continue;
228 |             } else {
229 |                 revert("VAE: Volta must be the first signer");
230 |             }
231 |         }
232 |         if (_isAnOwner(signer)) {
233 |             if (_seenOwner(signers, signer)) {
```

```
234 |                 continue;
235 |             }
236 |             signers[i%65] = signer;
237 |             numOwnerSignatures++;
238 |         }
239 |     }
240 |     return (numOwnerSignatures, voltaSigned);
241 | }
```

Line 236 uses `signers[i%65] = signer;`, where incrementing `i` by 65 per loop causes `i%65` to remain `0`. This limits the signers array to a single element, allowing signature duplication and unintentional increases in `numOwnerSignatures`.

An attacker may leverage two owners signing alternately to inflate `numOwnerSignatures`, potentially bypassing the threshold:

```
/* contracts/VoltaAccount.sol */
387 |     // Signature is only valid if quorum is met
388 |     if (numSigners < minQuorum) {
389 |         return ERC1271_INVALID_SIGNATURE;
390 |     }
391 |     return ERC1271_MAGIC_VALUE;
392 | }
```

This flaw could reduce a typical M-of-N multi-signature to 2-of-N.

Among all operations, some selectors additionally require Volta to sign:

```
/* contracts/VoltaAccount.sol */
253 | function _requiresVolta(bytes4 selector) internal returns (bool) {
254 |     return selector == this.enable.selector ||
255 |            selector == this.disable.selector ||
256 |            selector == this.upgradeTo.selector ||
257 |            selector == this.upgradeToAndCall.selector ||
258 |            selector == this.upgradeSessionKeyValidator.selector ||
259 |            selector == this.upgradeExecutor.selector ||
260 |            selector == this.multiUpgrade.selector ||
261 |            selector == this.updateOwners.selector ||
262 |            selector == this.updateEntryPoint.selector;
263 | }
```

As `Volta` must be the first signer and cannot participate in the duplication, exploiting these selectors requires at least three signatures: one from `Volta` and two from other owners. effectively maintaining a 3-of-N requirement:

```
/* contracts/VoltaAccount.sol */
224 | if (isVolta(signer)) {
225 |     if (i == 0) {
226 |         voltaSigned = true;
227 |         continue;
228 |     } else {
229 |         revert("VAE: Volta must be the first signer");
230 |     }
231 | }
```

For the above selectors, since the Volta signature is required. The risk is relatively low.

However, the following selectors do not require the Volta signature:

- `execute`
- `executeBatch`
- `addDeposit`
- `withdrawDepositTo`
- `delegatedSKisValidSignature`
- `receive`

For these operations, attackers only needs two owner signatures, which is risky.

Recommendation

Consider replacing `signers[i%65]` with `signers[numOwnerSignatures]`.

Resolution

This issue has been resolved by commit `07ca43c`.

SMART_CONTRACT_4337

[L-01] Missing deduplication checks when pushing addresses to arrays

Identified in commit `79f945e`.

In `SessionKeyValidator`, the arrays `enabledUsers`, `targetAddresses`, and `selectors` are used to index into their respective mappings for easier iterations.

However, these arrays do not check for uniqueness when new entries are added.

As a result, if the input of the `enable` function contains duplicated entries, they may be inserted.

```
/* contracts/validators/SessionKeyValidator.sol */
013 | function enable(EnableUserInstruction[] calldata instructions) external payable {
014 |       for (uint256 i = 0; i < instructions.length; i++) {
025 |           if (userAccess.enabled) {
026 |               // ...
027 |           } else {
028 |               address[] storage enabledUsers = _getEnabledUsers();
029 |               enabledUsers.push(userAddress);
030 |           }
040 |           for (uint256 j = 0; j < instructions[i].rules.length; j++) {
041 |               require(instructions[i].rules[j].selectorRules.length <= 64,
                            "SKV: Too many selector rules specified");
043 |               address targetAddress = instructions[i].rules[j].targetAddress;
044 |               userAccess.targetAddresses.push(targetAddress);
050 |               for (uint256 k = 0; k < instructions[i].rules[j].selectorRules.length; k++) {
053 |                   bytes4 selector = instructions[i].rules[j].selectorRules[k].selector;
054 |                   userAccess.accessRules[targetAddress].selectors.push(selector);
```

It is recommended to add the deduplication checks before pushing to the array.

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[L-02] Missing userOpHash validation

Identified in commit `79f945e`.

In `VoltaAccount` and `SessionKeyValidator`, the `userOpHash` can be used to directly recover signer addresses without verifying whether they were properly derived from the given `UserOperation`.

The lack of validation for the `userOpHash` may result in an incorrect match between the signature and the actual `UserOperation` data, causing signature-based authentication to fail.

```
/* contracts/VoltaAccount.sol */
156 | bytes32 hash = userOpHash.toEthSignedMessageHash();
157 | if (userOp.signature.length > 65) {
158 |     return _validateOwners(userOp, hash);
159 | }

/* contracts/validators/SessionKeyValidator.sol */
102 | address signer = userOpHash.recover(userOp.signature);
103 | UserAccess storage userAccess = _getUserAccess(signer);
```

Although the `validateUserOp` can only be called by the `EntryPoint`, it is still recommended to include a check to ensure that the `userOpHash` is correctly derived from the `userOp` before relying on the recovered signer.

```
/* lib/account-abstraction/contracts/core/BaseAccount.sol */
043 | function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash, uint256
↪  missingAccountFunds)
044 | external override virtual returns (uint256 validationData) {
045 |     _requireFromEntryPoint();
046 |     validationData = _validateSignature(userOp, userOpHash);
047 |     _validateNonce(userOp.nonce);
048 |     _payPrefund(missingAccountFunds);
049 | }
```

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[L-03] Potential ambiguity in `validationData` parsing

Identified in commit `79f945e`.

Function `_validateSignature` returns a packed `validationData`, consisting of `sigAuthorizer`, `validUntil`, and `validAfter`.

```
/* lib/account-abstraction/contracts/core/BaseAccount.sol */
063 | * @return validationData signature and time-range of this operation
064 |  *      <20-byte> sigAuthorizer - 0 for valid signature, 1 to mark signature failure,
065 |  *         otherwise, an address of an "authorizer" contract.
066 |  *      <6-byte> validUntil - last timestamp this operation is valid. 0 for "indefinite"
067 |  *      <6-byte> validAfter - first timestamp this operation is valid
068 |  *      If the account doesn't use time-range, it is enough to return SIG_VALIDATION_FAILED
              value (1) for signature failure.
069 |  *      Note that the validation code cannot use block.timestamp (or block.number) directly.
070 |  */
071 | function _validateSignature(UserOperation calldata userOp, bytes32 userOpHash)
072 | internal virtual returns (uint256 validationData);
```

The `_validateOwners` function only returns the `sigAuthorizer`, whereas the delegatecall to `SessionKeyValidator` returns the full `validationData`, including time bounds.

```
/* contracts/VoltaAccount.sol */
142 | function _validateSignature(UserOperation calldata userOp, bytes32 userOpHash)
143 | internal override returns (uint256) {
156 |     bytes32 hash = userOpHash.toEthSignedMessageHash();
157 |     if (userOp.signature.length > 65) {
158 |         return _validateOwners(userOp, hash);
159 |     }
170 |     // Validate the signature as a session key
171 |     (bool success, bytes memory data) = sessionKeyValidator.delegatecall(
172 |         abi.encodeWithSelector(Validator.validateUserOp.selector, userOp, hash)
173 |     );
187 | }

/* contracts/VoltaAccount.sol */
189 | function _validateOwners(UserOperation calldata userOp, bytes32 userOpHash)
190 | internal returns (uint256 validationData) {
194 |     if (userOp.signature.length < uint256(65 * minQuorum)
              || userOp.signature.length > 65 * owners.length) {
195 |         return SIG_VALIDATION_FAILED;
196 |     }
202 |     if (numSigners < minQuorum) {
203 |         return SIG_VALIDATION_FAILED;
204 |     }
```

```
207 |     if (userOp.callData.length > 0) {
210 |         if (_requiresVolta(selector) && !voltaSigned) {
211 |             return SIG_VALIDATION_FAILED;
212 |         }
213 |     }
215 |     return 0;
216 | }

/* contracts/validators/SessionKeyValidator.sol */
095 | function validateUserOp(UserOperation calldata userOp, bytes32 userOpHash)
099 | {
114 |     if (funcSelector == 0x12c850df) {  // executeBatch
121 |         return _packValidationData(!valid, validUntil, validAfter);
122 |     } else if (funcSelector == 0xb61d27f6) { // execute
125 |         return _packValidationData(!valid, validUntil, validAfter);
126 |     }
```

When both `validUntil` and `validAfter` in `userAccess` are set to `0`, it may indicate an expired `userAccess` and should result in `outOfTimeRange = true`.

However, the `EntryPoint` currently interprets this as `validationData` returned by `_validateOwners`, incorrectly sets `outOfTimeRange` as `false`.

```
/* lib/account-abstraction/contracts/core/EntryPoint.sol */
486 | function _getValidationData(uint256 validationData) internal view returns (address aggregator, bool
↪  outOfTimeRange) {
487 |     if (validationData == 0) {
488 |         return (address(0), false);
489 |     }
490 |     ValidationData memory data = _parseValidationData(validationData);
491 |     // solhint-disable-next-line not-rely-on-time
492 |     outOfTimeRange = block.timestamp > data.validUntil || block.timestamp < data.validAfter;
493 |     aggregator = data.aggregator;
494 | }
```

It is recommended to differentiate between owner signatures and session key signature to avoid incorrect parsing of `validationData`.

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[I-01] `_validateExecuteCallData` fails to parse dynamic type parameters

Identified in commit `79f945e`.

The method for parsing data when `_validateExecuteCallData` verifies the validity of parameters is as follows:

```
/* contracts/validators/SessionKeyValidator.sol */
199 | bytes32 paramInCallData = bytes32(data[4+offset:4+offset+32]);
201 | bool valid = compareParams(paramInCallData, param, condition);
```

Each parameter read occupies 32 bytes. However, dynamic types like `bytes` or `string` may exceed this limit.

For example, the `abi.encode` encoding structure of the string `"hello,world"` is shown as follows:

- 32 bytes offset
- 32 bytes length
- length-sized data: The data size (length-sized) is not necessarily equal to 32 bytes.

```
0000000000000000000000000000000000000000000000000000000000000020
000000000000000000000000000000000000000000000000000000000000000b
68656c6c6f2c776f726c640000000000000000000000000000000000000000
```

The `SessionKeyValidator` rule system cannot effectively constrain parameters of dynamic types.

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[I-02] Gas optimization opportunities

Identified in commit `79f945e`.

1. Replace "`i++`" with "`unchecked{++i;}`" in loops

Replace `i++` or `j++` with `unchecked{++i;}` or `unchecked{++j;}` in the code to reduce gas consumption.

```
/* contracts/VoltaAccount.sol */
123 | for (uint i = 1; i < owners.length; i++) { // skip Volta
245 | for (uint i = 0; i < seen.length; i++) {
424 | for (uint i = 0; i < _owners.length; i++) {

/* contracts/executors/DefaultExecutor.sol */
012 | for (uint256 i = 0; i < batchCall.calls.length; i++) {

/* contracts/validators/SessionKeyValidator.sol */
014 | for (uint256 i = 0; i < instructions.length; i++) {
040 | for (uint256 j = 0; j < instructions[i].rules.length; j++) {
073 | for (uint256 i = 0; i < instructions.length; i++) {
077 | for (uint256 j = 0; j < enabledUsers.length; j++) {
132 | for (uint256 i = 0; i < batchCall.calls.length; i++) {
189 | for (uint i = 0; i < ...; i++) {
195 | for (uint j = 0; j < ...; j++) {

/* contracts/validators/SessionKeyValidatorStorage.sol */
090 | for (uint256 i = 0; i < enabledUsers.length; i++) {
100 | for (uint256 j = 0; j < userAccess.targetAddresses.length; j++) {
```

Here is a simple example:

```
for (uint8 i = 0; i < numAssets;) {
    unchecked {++i;}
}
```

2. Read-only variables should use "`memory`" instead of "`storage`"

In the `SessionKeyValidator`, the `storage` modifier for these three `userAccess` variables could be changed to `memory` since the `userAccess` is not modified.

```
/* contracts/validators/SessionKeyValidator.sol */
103 | UserAccess storage userAccess = _getUserAccess(signer);
142 | UserAccess storage userAccess = _getUserAccess(signer);
176 | UserAccess storage userAccess = _getUserAccess(userAddress);
```

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[I-03] Missing data length check in `_validateExecuteCallData`

Identified in commit `79f945e`.

In `SessionKeyValidator`, the `_validateExecuteCallData` function accesses `data[0:4]` and `data[4 + offset : 4 + offset + 32]` without verifying that the data array is sufficiently long.

If the input data is shorter than expected, these operations may lead to a revert.

```
/* contracts/validators/SessionKeyValidator.sol */
175 | bytes4 selector = bytes4(data[0:4]);
176 | UserAccess storage userAccess = _getUserAccess(userAddress);
177 | require(userAccess.accessRules[targetAddress].selectorRules[selector].enabled,
178 |     "SKV: User not enabled for this selector");

199 | bytes32 paramInCallData = bytes32(data[4+offset:4+offset+32]);
```

It is recommended to check the `data.length` before accessing the selector or parameter bytes, and revert with "SKV: Invalid call data length" when the data is too short.

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[I-04] Check `validUntil` against current timestamp

Identified in commit `79f945e`.

In the `enable` function of `SessionKeyValidator`, there is no validation to ensure that the `validUntil` is greater than the current block timestamp, which allows expired access rules to be added without restriction.

```
/* contracts/validators/SessionKeyValidator.sol */
032 | userAccess.enabled = true;
033 | userAccess.globalMaxValue = instructions[i].globalMaxValue;
034 | userAccess.validAfter = uint48(instructions[i].validAfterUntil >> 48);
035 | userAccess.validUntil = uint48(instructions[i].validAfterUntil);
```

It is recommended to add a check ensuring that `validUntil` is in the future, in order to prevent adding already-expired user access entries.

Resolution

The team acknowledged this finding.

SMART_CONTRACT_4337

[I-05] Incorrect comment for the `execute` function

Identified in commit `79f945e`.

The comment on line 83 incorrectly states that the `execute` function can be called directly from "(min) owners".

However, the function is restricted by the `requireFromEntryPoint` modifier, which only permits calls from the `EntryPoint`.

```
/* contracts/VoltaAccount.sol */
082 | /**
083 |  * execute a transaction called directly from (min) owners, or by the entryPoint
084 |  */
085 | function execute(address dest, uint256 value, bytes calldata data) external requireFromEntryPoint {
086 |     // Note: no need to catch the return value, since the executor will revert if the call fails
087 |     executor.delegatecall(abi.encodeWithSelector(Executor.execute.selector, dest, value, data));
088 | }
```

Resolution

The team acknowledged this finding.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Volta (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.