

account-abstraction / contracts / core / EntryPoint.sol

⚠ This commit does not belong to any branch on this repository, and may belong to a fork outside of the repository.



783 lines (721 loc) · 29.3 KB

```
Raw 📮 🕹 🧷 🔻
                                                                                            <>
Code
         Blame
          // SPDX-License-Identifier: GPL-3.0
   1
   2
          pragma solidity ^0.8.23;
   3
          /* solhint-disable avoid-low-level-calls */
          /* solhint-disable no-inline-assembly */
   4
   5
   6
          import "../interfaces/IAccount.sol";
   7
          import "../interfaces/IAccountExecute.sol";
   8
          import "../interfaces/IPaymaster.sol";
   9
          import "../interfaces/IEntryPoint.sol";
   10
  11
          import "../utils/Exec.sol";
  12
          import "./StakeManager.sol";
          import "./SenderCreator.sol";
  13
  14
          import "./Helpers.sol";
          import "./NonceManager.sol";
  15
  16
          import "./UserOperationLib.sol";
  17
  18
          // we also require '@gnosis.pm/safe-contracts' and both libraries have 'IERC165.sol',
          import "@openzeppelin/contracts/utils/introspection/ERC165.sol" as OpenZeppelin;
  19
          import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
   20
  21
  22
          /*
           * Account-Abstraction (EIP-4337) singleton EntryPoint implementation.
  23
  24
           * Only one instance required on each chain.
   25
           */
  26
          contract EntryPoint is IEntryPoint, StakeManager, NonceManager, ReentrancyGuard, Open
  27
  28
              using UserOperationLib for UserOperation;
  29
              SenderCreator private senderCreator = new SenderCreator();
   30
  31
```

```
// Marker for inner call revert on out of gas
32
33
           bytes32 private constant INNER_OUT_OF_GAS = hex"deaddead";
34
           uint256 private constant REVERT REASON MAX LEN = 2048;
35
36
           uint256 private constant PENALTY_PERCENT = 10;
37
38
           /**
            * For simulation purposes, validateUserOp (and validatePaymasterUserOp)
39
            \ast must return this value in case of signature failure, instead of revert.
40
41
           uint256 public constant SIG_VALIDATION_FAILED = 1;
42
43
           /// @inheritdoc OpenZeppelin.IERC165
44
           function supportsInterface(bytes4 interfaceId) public view virtual override retur
45
               // note: solidity "type(IEntryPoint).interfaceId" is without inherited method
46
47
               return interfaceId == (type(IEntryPoint).interfaceId ^ type(IStakeManager).in
48
                   interfaceId == type(IEntryPoint).interfaceId ||
                   interfaceId == type(IStakeManager).interfaceId ||
49
50
                   interfaceId == type(INonceManager).interfaceId ||
51
                   super.supportsInterface(interfaceId);
52
53
54
           /**
55
            * Compensate the caller's beneficiary address with the collected fees of all Use
56
            * @param beneficiary - The address to receive the fees.
57
            * @param amount
                                 - Amount to transfer.
58
            */
           function compensate(address payable beneficiary, uint256 amount) internal {
59
               require(beneficiary != address(0), "AA90 invalid beneficiary");
60
               (bool success, ) = beneficiary.call{value: amount}("");
61
               require(success, "AA91 failed send to beneficiary");
62
           }
63
64
65
           /**
66
            * Execute a user operation.
67
            * @param opIndex
                                - Index into the opInfo array.
                                - The userOp to execute.
68
            * @param userOp
            * @param opInfo
                                - The opInfo filled by validatePrepayment for this userOp.
69
70
            * @return collected - The total amount this userOp paid.
71
            */
72
           function executeUserOp(
73
               uint256 opIndex,
74
               UserOperation calldata userOp,
75
               UserOpInfo memory opInfo
76
           )
77
           internal
78
           returns
79
           (uint256 collected) {
80
               uint256 preGas = gasleft();
               bytes memory context = getMemoryBytesFromOffset(opInfo.contextOffset);
81
82
               uint saveFreePtr;
83
               assembly {
84
                   saveFreePtr := mload(0x40)
85
```

```
رن
 86
                bytes calldata callData = userOp.callData;
 87
                bytes memory innerCall;
 88
                bytes4 methodSig;
                assembly {
 89
                    let len := callData.length
 90
                    if gt(len,3) {
91
                        methodSig := calldataload(callData.offset)
 92
 93
                    }
 94
95
                if (methodSig == IAccountExecute.executeUserOp.selector) {
96
                    bytes memory executeUserOp = abi.encodeCall(IAccountExecute.executeUserOp
97
                    innerCall = abi.encodeCall(this.innerHandleOp, (executeUserOp, opInfo, co
98
                } else
99
                    innerCall = abi.encodeCall(this.innerHandleOp, (callData, opInfo, context
100
101
102
                bool success;
103
                assembly {
104
                    success := call(gas(), address(), 0, add(innerCall, 0x20), mload(innerCal
105
                    collected := mload(0)
                    mstore(0x40, saveFreePtr)
106
                }
107
108
                if (!success) {
                    bytes32 innerRevertCode;
109
110
                    assembly {
                        let len := returndatasize()
111
                        if eq(32,len) {
112
113
                            returndatacopy(0, 0, 32)
114
                            innerRevertCode := mload(0)
115
                    }
116
                    // handleOps was called with gas limit too low. abort entire bundle.
117
118
                    if (innerRevertCode == INNER_OUT_OF_GAS) {
                        //report paymaster, since if it is not deliberately caused by the bun
119
120
                        // it must be a revert caused by paymaster.
121
                        revert FailedOp(opIndex, "AA95 out of gas");
122
                    } else {
123
                        emit PostOpRevertReason(
124
                            opInfo.userOpHash,
125
                            opInfo.mUserOp.sender,
126
                            opInfo.mUserOp.nonce,
                            Exec.getReturnData(REVERT_REASON_MAX_LEN)
127
128
                        );
129
130
131
                    uint256 actualGas = preGas - gasleft() + opInfo.preOpGas;
132
                    collected = _postExecution(
133
                        opIndex,
134
                        IPaymaster.PostOpMode.postOpReverted,
135
                        opInfo,
136
                        context,
                        actualGas
137
138
                    );
120
```

```
TOS
            }
140
141
142
            /// @inheritdoc IEntryPoint
143
            function handleOps(
144
                UserOperation[] calldata ops,
145
                address payable beneficiary
146
            ) public nonReentrant {
                uint256 opslen = ops.length;
147
                UserOpInfo[] memory opInfos = new UserOpInfo[](opslen);
148
149
                unchecked {
150
151
                     for (uint256 i = 0; i < opslen; i++) {</pre>
152
                         UserOpInfo memory opInfo = opInfos[i];
153
                         (
154
                             uint256 validationData,
155
                             uint256 pmValidationData
156
                         ) = _validatePrepayment(i, ops[i], opInfo);
157
                         _validateAccountAndPaymasterValidationData(
158
                             i,
159
                             validationData,
160
                             pmValidationData,
161
                             address(0)
162
                         );
163
                    }
164
165
                    uint256 collected = 0;
                    emit BeforeExecution();
166
167
168
                     for (uint256 i = 0; i < opslen; i++) {</pre>
169
                         collected += _executeUserOp(i, ops[i], opInfos[i]);
170
                     }
171
                    _compensate(beneficiary, collected);
172
                }
173
            }
174
175
176
            /// @inheritdoc IEntryPoint
177
            function handleAggregatedOps(
178
                UserOpsPerAggregator[] calldata opsPerAggregator,
179
                address payable beneficiary
180
            ) public nonReentrant {
181
182
                uint256 opasLen = opsPerAggregator.length;
183
                uint256 totalOps = 0;
                for (uint256 i = 0; i < opasLen; i++) {</pre>
184
185
                     UserOpsPerAggregator calldata opa = opsPerAggregator[i];
                    UserOperation[] calldata ops = opa.userOps;
186
187
                    IAggregator aggregator = opa.aggregator;
188
189
                     //address(1) is special marker of "signature error"
190
                     require(
191
                         address(aggregator) != address(1),
192
                         "AA96 invalid aggregator"
100
```

```
193
                     );
194
195
                    if (address(aggregator) != address(0)) {
196
                         // solhint-disable-next-line no-empty-blocks
197
                         try aggregator.validateSignatures(ops, opa.signature) {} catch {
198
                             revert SignatureValidationFailed(address(aggregator));
199
                         }
                    }
200
201
202
                    totalOps += ops.length;
203
204
205
                UserOpInfo[] memory opInfos = new UserOpInfo[](totalOps);
206
                uint256 opIndex = 0;
207
208
                for (uint256 a = 0; a < opasLen; a++) {</pre>
209
                     UserOpsPerAggregator calldata opa = opsPerAggregator[a];
210
                    UserOperation[] calldata ops = opa.userOps;
211
                    IAggregator aggregator = opa.aggregator;
212
213
                    uint256 opslen = ops.length;
214
                     for (uint256 i = 0; i < opslen; i++) {</pre>
                         UserOpInfo memory opInfo = opInfos[opIndex];
215
216
217
                             uint256 validationData.
218
                             uint256 paymasterValidationData
219
                         ) = _validatePrepayment(opIndex, ops[i], opInfo);
220
                         _validateAccountAndPaymasterValidationData(
221
222
                             validationData,
223
                             paymasterValidationData,
224
                             address(aggregator)
225
                         );
226
                         opIndex++;
227
                    }
228
229
230
                emit BeforeExecution();
231
232
                uint256 collected = 0:
233
                opIndex = 0;
234
                for (uint256 a = 0; a < opasLen; a++) {</pre>
235
                    UserOpsPerAggregator calldata opa = opsPerAggregator[a];
236
                     emit SignatureAggregatorChanged(address(opa.aggregator));
237
                    UserOperation[] calldata ops = opa.userOps;
238
                    uint256 opslen = ops.length;
239
240
                    for (uint256 i = 0; i < opslen; i++) {</pre>
                         collected += executeUserOp(opIndex, ops[i], opInfos[opIndex]);
241
242
                         opIndex++;
243
                     }
244
                }
                emit SignatureAggregatorChanged(address(0));
245
246
```

```
247
                compensate(beneficiary, collected);
248
            }
249
250
            /**
251
             * A memory copy of UserOp static fields only.
252
             * Excluding: callData, initCode and signature. Replacing paymasterAndData with p
253
             */
254
            struct MemoryUserOp {
                address sender;
255
                uint256 nonce;
256
257
                uint256 callGasLimit;
258
                uint256 verificationGasLimit;
259
                uint256 preVerificationGas;
260
                address paymaster;
261
                uint256 maxFeePerGas;
262
                uint256 maxPriorityFeePerGas;
263
            }-
264
265
            struct UserOpInfo {
266
                MemoryUserOp mUserOp;
267
                bytes32 user0pHash;
268
                uint256 prefund;
269
                uint256 contextOffset;
                uint256 pre0pGas;
270
            }
271
272
273
            /**
274
             * Inner function to handle a UserOperation.
             * Must be declared "external" to open a call context, but it can only be called
275
276
             st @param callData - The callData to execute.
277
             * @param opInfo - The UserOpInfo struct.
278
             * @param context - The context bytes.
279
             */
280
            function innerHandleOp(
                bytes memory callData,
281
                UserOpInfo memory opInfo,
282
283
                bytes calldata context
284
            ) external returns (uint256 actualGasCost) {
285
                uint256 preGas = gasleft();
                require(msg.sender == address(this), "AA92 internal call only");
286
287
                MemoryUserOp memory mUserOp = opInfo.mUserOp;
288
289
                uint callGasLimit = mUserOp.callGasLimit;
290
                unchecked {
291
                    // handleOps was called with gas limit too low. abort entire bundle.
                    if (
292
293
                        gasleft() < callGasLimit + mUserOp.verificationGasLimit + 5000</pre>
294
                    ) {
295
                        assembly {
                            mstore(0, INNER_OUT_OF_GAS)
296
                            revert(0, 32)
297
298
                    }
299
300
                }
```

```
301
302
                IPaymaster.PostOpMode mode = IPaymaster.PostOpMode.opSucceeded;
303
                if (callData.length > 0) {
304
                    bool success = Exec.call(mUserOp.sender, 0, callData, callGasLimit);
305
                    if (!success) {
306
                        bytes memory result = Exec.getReturnData(REVERT_REASON_MAX_LEN);
307
                        if (result.length > 0) {
308
                            emit UserOperationRevertReason(
309
                                opInfo.userOpHash,
310
                                mUserOp.sender,
                                mUserOp.nonce,
311
312
                                result
                            );
313
314
                        }
315
                        mode = IPaymaster.PostOpMode.opReverted;
316
                    }
317
                }
318
319
                unchecked {
                    uint256 actualGas = preGas - gasleft() + opInfo.preOpGas;
320
321
                    // Note: opIndex is ignored (relevant only if mode==postOpReverted, which
322
                    return _postExecution(0, mode, opInfo, context, actualGas);
323
324
            }
325
326
            /// @inheritdoc IEntryPoint
327
            function getUserOpHash(
                UserOperation calldata userOp
328
329
            ) public view returns (bytes32) {
330
                return
331
                    keccak256(abi.encode(user0p.hash(), address(this), block.chainid));
332
            }
333
334
335
             * Copy general fields from userOp into the memory opInfo structure.
336
             * @param userOp - The user operation.
337
             * @param mUserOp - The memory user operation.
338
339
            function copyUserOpToMemory(
340
                UserOperation calldata userOp,
341
                MemoryUserOp memory mUserOp
342
            ) internal pure {
343
                mUserOp.sender = userOp.sender;
344
                mUserOp.nonce = userOp.nonce;
345
                mUserOp.callGasLimit = userOp.callGasLimit;
346
                mUserOp.verificationGasLimit = userOp.verificationGasLimit;
347
                mUserOp.preVerificationGas = userOp.preVerificationGas;
348
                mUserOp.maxFeePerGas = userOp.maxFeePerGas;
349
                mUserOp.maxPriorityFeePerGas = userOp.maxPriorityFeePerGas;
                bytes calldata paymasterAndData = userOp.paymasterAndData;
350
351
                if (paymasterAndData.length > 0) {
                    require(
352
353
                        paymasterAndData.length >= 20,
                        "AA93 invalid paymasterAndData"
354
```

```
355
                    );
356
                    mUserOp.paymaster = address(bytes20(paymasterAndData[:20]));
357
                } else {
                    mUserOp.paymaster = address(0);
358
359
                }
360
            }
361
362
            /**
363
             * Get the required prefunded gas fee amount for an operation.
364
             * @param mUserOp - The user operation in memory.
365
366
            function getRequiredPrefund(
367
                MemoryUserOp memory mUserOp
368
            ) internal pure returns (uint256 requiredPrefund) {
369
                unchecked {
370
                    // When using a Paymaster, the verificationGasLimit is used also to as a
                    // Our security model might call postOp eventually twice.
371
372
                    uint256 mul = mUserOp.paymaster != address(0) ? 2 : 1;
373
                    uint256 requiredGas = mUserOp.callGasLimit +
374
                        mUserOp.verificationGasLimit *
                        mul +
375
376
                        mUserOp.preVerificationGas;
377
378
                    requiredPrefund = requiredGas * mUserOp.maxFeePerGas;
379
                }
380
            }
381
382
            /**
383
             * Create sender smart contract account if init code is provided.
             * @param opIndex - The operation index.
384
             * @param opInfo - The operation info.
385
386
             * @param initCode - The init code for the smart contract account.
387
             */
            function _createSenderIfNeeded(
388
389
                uint256 opIndex,
390
                UserOpInfo memory opInfo,
391
                bytes calldata initCode
392
            ) internal {
                if (initCode.length != 0) {
393
394
                    address sender = opInfo.mUserOp.sender;
                    if (sender.code.length != 0)
395
396
                        revert FailedOp(opIndex, "AA10 sender already constructed");
                    address sender1 = senderCreator.createSender{
397
398
                        gas: opInfo.mUserOp.verificationGasLimit
399
                    }(initCode);
                    if (sender1 == address(0))
400
401
                        revert FailedOp(opIndex, "AA13 initCode failed or 00G");
402
                    if (sender1 != sender)
                        revert FailedOp(opIndex, "AA14 initCode must return sender");
403
                    if (sender1.code.length == 0)
404
                        revert FailedOp(opIndex, "AA15 initCode must create sender");
405
406
                    address factory = address(bytes20(initCode[0:20]));
407
                    emit AccountDeployed(
408
                        opInfo.userOpHash,
```

```
409
                         sender,
410
                         factory,
411
                        opInfo.mUserOp.paymaster
412
                   );
               }
413
414
            }
415
            /// @inheritdoc IEntryPoint
416
417
            function getSenderAddress(bytes calldata initCode) public {
                address sender = senderCreator.createSender(initCode);
418
419
                revert SenderAddressResult(sender);
420
            }
421
422
            /**
423
             * Call account.validateUserOp.
424
             * Revert (with FailedOp) in case validateUserOp reverts, or account didn't send
             * Decrement account's deposit if needed.
425
426
             * @param opIndex

    The operation index.

427
             * @param op
                                       - The user operation.

    The operation info.

428
             * @param opInfo
429
             * @param requiredPrefund - The required prefund amount.
430
             */
431
            function _validateAccountPrepayment(
432
                uint256 opIndex,
433
                UserOperation calldata op,
                UserOpInfo memory opInfo,
434
                uint256 requiredPrefund
435
436
437
                internal
438
                returns (
439
                    uint256 gasUsedByValidateAccountPrepayment,
                    uint256 validationData
440
441
                )
442
            {
                unchecked {
443
444
                    uint256 preGas = gasleft();
                    MemoryUserOp memory mUserOp = opInfo.mUserOp;
445
446
                    address sender = mUserOp.sender;
447
                    _createSenderIfNeeded(opIndex, opInfo, op.initCode);
448
                    address paymaster = mUserOp.paymaster;
449
                    uint256 missingAccountFunds = 0;
                    if (paymaster == address(0)) {
450
                        uint256 bal = balanceOf(sender);
451
452
                        missingAccountFunds = bal > requiredPrefund
453
                            ? 0
454
                             : requiredPrefund - bal;
                    }
455
456
                    try
457
                        IAccount(sender).validateUserOp{
458
                             gas: mUserOp.verificationGasLimit
459
                        }(op, opInfo.userOpHash, missingAccountFunds)
                    returns (uint256 _validationData) {
460
                        validationData = _validationData;
461
462
                    } catch {
```

```
463
                         revert FailedOpWithRevert(opIndex, "AA23 reverted", Exec.getReturnDat
464
465
                    if (paymaster == address(0)) {
466
                        DepositInfo storage senderInfo = deposits[sender];
467
                        uint256 deposit = senderInfo.deposit;
468
                        if (requiredPrefund > deposit) {
                             revert FailedOp(opIndex, "AA21 didn't pay prefund");
469
470
                        }
471
                        senderInfo.deposit = uint112(deposit - requiredPrefund);
472
                    }
473
                    gasUsedByValidateAccountPrepayment = preGas - gasleft();
474
475
            }
476
477
            /**
478
             * In case the request has a paymaster:
479
             * - Validate paymaster has enough deposit.
480
             * - Call paymaster.validatePaymasterUserOp.
481
             * - Revert with proper FailedOp in case paymaster reverts.
             * - Decrement paymaster's deposit.
482
483
             * @param opIndex
                                                           - The operation index.
                                                           - The user operation.
484
             * @param op
485
                                                           - The operation info.
             * @param opInfo
                                                           - The required prefund amount.
486
             * @param requiredPreFund
487
             * @param gasUsedByValidateAccountPrepayment - The gas used by _validateAccountPr
488
            function _validatePaymasterPrepayment(
489
490
                uint256 opIndex,
491
                UserOperation calldata op,
492
                UserOpInfo memory opInfo,
493
                uint256 requiredPreFund,
494
                uint256 gasUsedBvValidateAccountPrepayment
495
            ) internal returns (bytes memory context, uint256 validationData) {
496
                unchecked {
497
                    MemoryUserOp memory mUserOp = opInfo.mUserOp;
498
                    uint256 verificationGasLimit = mUserOp.verificationGasLimit;
499
                    require(
500
                        verificationGasLimit > gasUsedByValidateAccountPrepayment,
501
                        "AA41 too little verificationGas"
502
                    );
503
                    uint256 gas = verificationGasLimit -
504
                        gasUsedByValidateAccountPrepayment;
505
506
                    address paymaster = mUserOp.paymaster;
507
                    DepositInfo storage paymasterInfo = deposits[paymaster];
508
                    uint256 deposit = paymasterInfo.deposit;
509
                    if (deposit < requiredPreFund) {</pre>
510
                         revert FailedOp(opIndex, "AA31 paymaster deposit too low");
511
512
                    paymasterInfo.deposit = uint112(deposit - requiredPreFund);
513
                    try
514
                        IPaymaster(paymaster).validatePaymasterUserOp{gas: gas}(
515
516
                             opInfo.userOpHash,
```

```
517
                            requiredPreFund
518
519
                    returns (bytes memory _context, uint256 _validationData) {
520
                        context = context;
                        validationData = _validationData;
521
522
                    } catch {
523
                        revert FailedOpWithRevert(opIndex, "AA33 reverted", Exec.getReturnDat
524
               }
525
526
            }
527
528
            /**
529
             * Revert if either account validationData or paymaster validationData is expired
530
             * @param opIndex
                                              - The operation index.
531
             * @param validationData
                                              - The account validationData.
532
             * @param paymasterValidationData - The paymaster validationData.
533
             * @param expectedAggregator

    The expected aggregator.

534
            function _validateAccountAndPaymasterValidationData(
535
536
                uint256 opIndex,
537
                uint256 validationData,
538
                uint256 paymasterValidationData,
539
                address expectedAggregator
540
            ) internal view {
541
                (address aggregator, bool outOfTimeRange) = _getValidationData(
542
                    validationData
543
                );
                if (expectedAggregator != aggregator) {
544
545
                    revert FailedOp(opIndex, "AA24 signature error");
546
                if (outOfTimeRange) {
547
548
                    revert FailedOp(opIndex, "AA22 expired or not due");
549
                // pmAggregator is not a real signature aggregator: we don't have logic to ha
550
                // Non-zero address means that the paymaster fails due to some signature chec
551
                address pmAggregator;
552
                (pmAggregator, outOfTimeRange) = _getValidationData(
553
                    paymasterValidationData
554
555
                );
                if (pmAggregator != address(0)) {
556
                    revert FailedOp(opIndex, "AA34 signature error");
557
558
                }
559
                if (outOfTimeRange) {
                    revert FailedOp(opIndex, "AA32 paymaster expired or not due");
560
561
           }
562
563
564
565
             * Parse validationData into its components.
             * @param validationData - The packed validation data (sigFailed, validAfter, val
566
567
             */
568
            function _getValidationData(
569
                uint256 validationData
570
            ) internal view returns (address aggregator, bool outOfTimeRange) {
```

```
571
                if (validationData == 0) {
572
                    return (address(0), false);
573
                ValidationData memory data = parseValidationData(validationData);
574
                // solhint-disable-next-line not-rely-on-time
575
576
                outOfTimeRange = block.timestamp > data.validUntil || block.timestamp < data.</pre>
577
                aggregator = data.aggregator;
578
            }
579
580
            /**
             * Validate account and paymaster (if defined) and
581
582
             * also make sure total validation doesn't exceed verificationGasLimit.
             * This method is called off-chain (simulateValidation()) and on-chain (from hand
583
             * @param opIndex - The index of this userOp into the "opInfos" array.
584
             * @param userOp - The userOp to validate.
585
586
587
            function validatePrepayment(
588
                uint256 opIndex,
                UserOperation calldata userOp,
589
                UserOpInfo memory outOpInfo
590
591
592
                internal
                returns (uint256 validationData, uint256 paymasterValidationData)
593
594
595
                uint256 preGas = gasleft();
596
                MemoryUserOp memory mUserOp = outOpInfo.mUserOp;
                _copyUserOpToMemory(userOp, mUserOp);
597
598
                outOpInfo.userOpHash = getUserOpHash(userOp);
599
                // Validate all numeric values in userOp are well below 128 bit, so they can
600
601
                // and multiplied without causing overflow.
                uint256 maxGasValues = mUserOp.preVerificationGas |
602
                    mUserOp.verificationGasLimit |
603
604
                    mUserOp.callGasLimit
605
                    userOp.maxFeePerGas
606
                    userOp.maxPriorityFeePerGas;
                require(maxGasValues <= type(uint120).max, "AA94 gas values overflow");</pre>
607
608
                uint256 gasUsedByValidateAccountPrepayment;
609
                uint256 requiredPreFund = _getRequiredPrefund(mUserOp);
610
611
612
                    gasUsedByValidateAccountPrepayment,
613
                    validationData
614
                ) = _validateAccountPrepayment(
615
                    opIndex,
616
                    userOp,
617
                    outOpInfo,
618
                    requiredPreFund
                );
619
620
                if (!_validateAndUpdateNonce(mUserOp.sender, mUserOp.nonce)) {
621
622
                    revert FailedOp(opIndex, "AA25 invalid account nonce");
623
                }
624
```

```
bytes memory context;
625
626
                if (mUserOp.paymaster != address(0)) {
627
                    (context, paymasterValidationData) = _validatePaymasterPrepayment(
628
                        opIndex.
629
                        userOp,
630
                        outOpInfo,
631
                        requiredPreFund,
632
                        gasUsedByValidateAccountPrepayment
633
                    );
634
                }
635
                unchecked {
636
                    uint256 gasUsed = preGas - gasleft();
637
638
                    if (userOp.verificationGasLimit < gasUsed) {</pre>
639
                        revert FailedOp(opIndex, "AA40 over verificationGasLimit");
640
                    }
641
                    outOpInfo.prefund = requiredPreFund;
642
                    outOpInfo.contextOffset = getOffsetOfMemoryBytes(context);
                    outOpInfo.preOpGas = preGas - gasleft() + userOp.preVerificationGas;
643
644
                }
645
            }
646
647
648
             * Process post-operation, called just after the callData is executed.
649
             * If a paymaster is defined and its validation returned a non-empty context, its
             * The excess amount is refunded to the account (or paymaster - if it was used in
650
651
             * @param opIndex - Index in the batch.
652
             * @param mode

    Whether is called from innerHandleOp, or outside (postOpRev

653
             * @param opInfo
                                - UserOp fields and info collected during validation.
654
             * @param context - The context returned in validatePaymasterUserOp.
             * @param actualGas - The gas used so far by this user operation.
655
656
657
            function _postExecution(
658
                uint256 opIndex.
659
                IPaymaster.PostOpMode mode,
660
                UserOpInfo memory opInfo,
661
                bytes memory context,
662
                uint256 actualGas
663
            ) private returns (uint256 actualGasCost) {
664
                uint256 preGas = gasleft();
665
                unchecked {
                    address refundAddress;
666
                    MemoryUserOp memory mUserOp = opInfo.mUserOp;
667
                    uint256 gasPrice = getUserOpGasPrice(mUserOp);
668
669
670
                    address paymaster = mUserOp.paymaster;
671
                    if (paymaster == address(0)) {
                        refundAddress = mUserOp.sender;
672
673
                    } else {
674
                        refundAddress = paymaster;
675
                        if (context.length > 0) {
676
                            actualGasCost = actualGas * gasPrice;
677
                            if (mode != IPaymaster.PostOpMode.postOpReverted) {
                                try TDaymactar/naymactar) nactOnf
670
```

```
LIY TEANHASTEL (hahmastel / postohi
0/0
679
                                     gas: mUserOp.verificationGasLimit
680
                                 }(mode, context, actualGasCost)
681
                                 // solhint-disable-next-line no-empty-blocks
682
                                 {} catch {
683
                                     bytes memory reason = Exec.getReturnData(REVERT_REASON_MA
684
                                     revert PostOpReverted(reason);
685
                            }
686
                        }
687
688
689
                    actualGas += preGas - gasleft();
690
691
                    // Calculating a penalty for unused execution gas
692
                    {
                        uint256 executionGasLimit = mUserOp.callGasLimit;
693
                        // Note that 'verificationGasLimit' here is the limit given to the 'p
694
695
                        if (context.length > 0){
696
                            executionGasLimit += mUserOp.verificationGasLimit;
697
                        }
698
                        uint256 executionGasUsed = actualGas - opInfo.pre0pGas;
699
                        // this check is required for the gas used within EntryPoint and not
700
                        if (executionGasLimit > executionGasUsed) {
701
                            uint256 unusedGas = executionGasLimit - executionGasUsed;
702
                            uint256 unusedGasPenalty = (unusedGas * PENALTY_PERCENT) / 100;
703
                            actualGas += unusedGasPenalty;
704
                        }
705
                    }
706
707
                    actualGasCost = actualGas * gasPrice;
                    if (opInfo.prefund < actualGasCost) {</pre>
708
709
                         revert FailedOp(opIndex, "AA51 prefund below actualGasCost");
710
711
                    uint256 refund = opInfo.prefund - actualGasCost;
                    _incrementDeposit(refundAddress, refund);
712
                    bool success = mode == IPaymaster.PostOpMode.opSucceeded;
713
714
                    emit UserOperationEvent(
715
                        opInfo.userOpHash,
716
                        mUserOp.sender,
717
                        mUserOp.paymaster,
718
                        mUserOp.nonce,
719
                        success,
720
                        actualGasCost,
721
                        actualGas
722
                    );
                } // unchecked
723
724
            }
725
726
            /**
727
             * The gas price this UserOp agrees to pay.
             * Relayer/block builder might submit the TX with higher priorityFee, but the use
728
729
             * @param mUserOp - The userOp to get the gas price from.
730
             */
731
            function getUserOpGasPrice(
722
```

```
Memoryuserup memory muserup
132
733
            ) internal view returns (uint256) {
                unchecked {
734
735
                    uint256 maxFeePerGas = mUserOp.maxFeePerGas;
                    uint256 maxPriorityFeePerGas = mUserOp.maxPriorityFeePerGas;
736
737
                    if (maxFeePerGas == maxPriorityFeePerGas) {
738
                        //legacy mode (for networks that don't support basefee opcode)
                        return maxFeePerGas;
739
740
                    return min(maxFeePerGas, maxPriorityFeePerGas + block.basefee);
741
742
                }
            }-
743
744
745
            /**
746
             * The minimum of two numbers.
747
             * @param a - First number.
748
             * @param b - Second number.
749
             */
            function min(uint256 a, uint256 b) internal pure returns (uint256) {
750
751
                return a < b ? a : b;
752
            }
753
754
            /**
             * The offset of the given bytes in memory.
755
756
             * @param data - The bytes to get the offset of.
757
             */
758
            function getOffsetOfMemoryBytes(
759
                bytes memory data
            ) internal pure returns (uint256 offset) {
760
761
                assembly {
762
                    offset := data
               }
763
            }-
764
765
766
            /**
767
             st The bytes in memory at the given offset.
             * @param offset - The offset to get the bytes from.
768
769
             */
770
            function getMemoryBytesFromOffset(
771
                uint256 offset
772
            ) internal pure returns (bytes memory data) {
773
                assembly {
774
                    data := offset
775
776
            }
777
778
            /// @inheritdoc IEntryPoint
779
            function delegateAndRevert(address target, bytes calldata data) external {
780
                (bool success, bytes memory ret) = target.delegatecall(data);
781
                revert DelegateAndRevert(success, ret);
782
            }
783
       }
```