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.

drortirosh Merge branch 'develop' into executeUserOp ✓ last month

783 lines (721 loc) · 29.3 KB

Code Blame Raw

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.23;
/* solhint-disable avoid-low-level-calls */
/* solhint-disable no-inline-assembly */

import "../interfaces/IAccount.sol";
import "../interfaces/IAccountExecute.sol";
import "../interfaces/IPaymaster.sol";
import "../interfaces/IEntryPoint.sol";

import "../utils/Exec.sol";
import "./StakeManager.sol";
import "./SenderCreator.sol";
import "./Helpers.sol";
import "./NonceManager.sol";
import "./UserOperationLib.sol";

// we also require '@gnosis.pm/safe-contracts' and both libraries have 'IERC165.sol',
import "@openzeppelin/contracts/utils/introspection/ERC165.sol" as OpenZeppelin;
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";

/*
 * Account-Abstraction (EIP-4337) singleton EntryPoint implementation.
 * Only one instance required on each chain.
 */
contract EntryPoint is IEntryPoint, StakeManager, NonceManager, ReentrancyGuard, Open

    using UserOperationLib for UserOperation;

    SenderCreator private senderCreator = new SenderCreator();

    // Marker for inner call revert on out of gas
    bytes32 private constant INNER_OUT_OF_GAS = hex"deaddead";

    uint256 private constant REVERT_REASON_MAX_LEN = 2048;
    uint256 private constant PENALTY_PERCENT = 10;

    /**
     * For simulation purposes, validateUserOp (and validatePaymasterUserOp)
     * must return this value in case of signature failure, instead of revert.
     */
    uint256 public constant SIG_VALIDATION_FAILED = 1;

    /// @inheritdoc OpenZeppelin.IERC165
    function supportsInterface(bytes4 interfaceId) public view virtual override retur
        // note: solidity "type(IEntryPoint).interfaceId" is without inherited method
        return interfaceId == (type(IEntryPoint).interfaceId ^ type(IStakeManager).in
            interfaceId == type(IEntryPoint).interfaceId ||
            interfaceId == type(IStakeManager).interfaceId ||
            interfaceId == type(INonceManager).interfaceId ||
            super.supportsInterface(interfaceId);
    }

    /**
     * Compensate the caller's beneficiary address with the collected fees of all Use
     * @param beneficiary - The address to receive the fees.
     * @param amount      - Amount to transfer.
     */
    function _compensate(address payable beneficiary, uint256 amount) internal {
        require(beneficiary != address(0), "AA90 invalid beneficiary");
        (bool success, ) = beneficiary.call{value: amount}("");
        require(success, "AA91 failed send to beneficiary");
    }

    /**
     * Execute a user operation.
     * @param opIndex    - Index into the opInfo array.
     * @param userOp     - The userOp to execute.
     * @param opInfo     - The opInfo filled by validatePrepayment for this userOp.
     * @return collected - The total amount this userOp paid.
     */
    function _executeUserOp(
        uint256 opIndex,
        UserOperation calldata userOp,
        UserOpInfo memory opInfo
    )
    internal
    returns
    (uint256 collected) {
        uint256 preGas = gasleft();
        bytes memory context = getMemoryBytesFromOffset(opInfo.contextOffset);
        uint saveFreePtr;
        assembly {
            saveFreePtr := mload(0x40)
        }
        }
        bytes calldata callData = userOp.callData;
        bytes memory innerCall;
        bytes4 methodSig;
        assembly {
            let len := callData.length
            if gt(len,3) {
                methodSig := calldataload(callData.offset)
            }
        }
        if (methodSig == IAccountExecute.executeUserOp.selector) {
            bytes memory executeUserOp = abi.encodeCall(IAccountExecute.executeUserOp
            innerCall = abi.encodeCall(this.innerHandleOp, (executeUserOp, opInfo, co
        } else
        {
            innerCall = abi.encodeCall(this.innerHandleOp, (callData, opInfo, context
        }
        bool success;
        assembly {
            success := call(gas(), address(), 0, add(innerCall, 0x20), mload(innerCal
            collected := mload(0)
            mstore(0x40, saveFreePtr)
        }
        if (!success) {
            bytes32 innerRevertCode;
            assembly {
                let len := returndatasize()
                if eq(32,len) {
                    returndatacopy(0, 0, 32)
                    innerRevertCode := mload(0)
                }
            }
            // handleOps was called with gas limit too low. abort entire bundle.
            if (innerRevertCode == INNER_OUT_OF_GAS) {
                //report paymaster, since if it is not deliberately caused by the bun
                // it must be a revert caused by paymaster.
                revert FailedOp(opIndex, "AA95 out of gas");
            } else {
                emit PostOpRevertReason(
                    opInfo.userOpHash,
                    opInfo.mUserOp.sender,
                    opInfo.mUserOp.nonce,
                    Exec.getReturnData(REVERT_REASON_MAX_LEN)
                );
            }

            uint256 actualGas = preGas - gasleft() + opInfo.preOpGas;
            collected = _postExecution(
                opIndex,
                IPaymaster.PostOpMode.postOpReverted,
                opInfo,
                context,
                actualGas
            );
        }
        }
    }

    /// @inheritdoc IEntryPoint
    function handleOps(
        UserOperation[] calldata ops,
        address payable beneficiary
    ) public nonReentrant {
        uint256 opslen = ops.length;
        UserOpInfo[] memory opInfos = new UserOpInfo[](opslen);

        unchecked {
            for (uint256 i = 0; i < opslen; i++) {
                UserOpInfo memory opInfo = opInfos[i];
                (
                    uint256 validationData,
                    uint256 pmValidationData
                ) = _validatePrepayment(i, ops[i], opInfo);
                _validateAccountAndPaymasterValidationData(
                    i,
                    validationData,
                    pmValidationData,
                    address(0)
                );
            }

            uint256 collected = 0;
            emit BeforeExecution();

            for (uint256 i = 0; i < opslen; i++) {
                collected += _executeUserOp(i, ops[i], opInfos[i]);
            }

            _compensate(beneficiary, collected);
        }
    }

    /// @inheritdoc IEntryPoint
    function handleAggregatedOps(
        UserOpsPerAggregator[] calldata opsPerAggregator,
        address payable beneficiary
    ) public nonReentrant {

        uint256 opasLen = opsPerAggregator.length;
        uint256 totalOps = 0;
        for (uint256 i = 0; i < opasLen; i++) {
            UserOpsPerAggregator calldata opa = opsPerAggregator[i];
            UserOperation[] calldata ops = opa.userOps;
            IAggregator aggregator = opa.aggregator;

            //address(1) is special marker of "signature error"
            require(
                address(aggregator) != address(1),
                "AA96 invalid aggregator"
            )
            );

            if (address(aggregator) != address(0)) {
                // solhint-disable-next-line no-empty-blocks
                try aggregator.validateSignatures(ops, opa.signature) {} catch {
                    revert SignatureValidationFailed(address(aggregator));
                }
            }

            totalOps += ops.length;
        }

        UserOpInfo[] memory opInfos = new UserOpInfo[](totalOps);

        uint256 opIndex = 0;
        for (uint256 a = 0; a < opasLen; a++) {
            UserOpsPerAggregator calldata opa = opsPerAggregator[a];
            UserOperation[] calldata ops = opa.userOps;
            IAggregator aggregator = opa.aggregator;

            uint256 opslen = ops.length;
            for (uint256 i = 0; i < opslen; i++) {
                UserOpInfo memory opInfo = opInfos[opIndex];
                (
                    uint256 validationData,
                    uint256 paymasterValidationData
                ) = _validatePrepayment(opIndex, ops[i], opInfo);
                _validateAccountAndPaymasterValidationData(
                    i,
                    validationData,
                    paymasterValidationData,
                    address(aggregator)
                );
                opIndex++;
            }
        }

        emit BeforeExecution();

        uint256 collected = 0;
        opIndex = 0;
        for (uint256 a = 0; a < opasLen; a++) {
            UserOpsPerAggregator calldata opa = opsPerAggregator[a];
            emit SignatureAggregatorChanged(address(opa.aggregator));
            UserOperation[] calldata ops = opa.userOps;
            uint256 opslen = ops.length;

            for (uint256 i = 0; i < opslen; i++) {
                collected += _executeUserOp(opIndex, ops[i], opInfos[opIndex]);
                opIndex++;
            }
        }
        emit SignatureAggregatorChanged(address(0));

        _compensate(beneficiary, collected);
    }

    /**
     * A memory copy of UserOp static fields only.
     * Excluding: callData, initCode and signature. Replacing paymasterAndData with p
     */
    struct MemoryUserOp {
        address sender;
        uint256 nonce;
        uint256 callGasLimit;
        uint256 verificationGasLimit;
        uint256 preVerificationGas;
        address paymaster;
        uint256 maxFeePerGas;
        uint256 maxPriorityFeePerGas;
    }

    struct UserOpInfo {
        MemoryUserOp mUserOp;
        bytes32 userOpHash;
        uint256 prefund;
        uint256 contextOffset;
        uint256 preOpGas;
    }

    /**
     * Inner function to handle a UserOperation.
     * Must be declared "external" to open a call context, but it can only be called
     * @param callData - The callData to execute.
     * @param opInfo   - The UserOpInfo struct.
     * @param context  - The context bytes.
     */
    function innerHandleOp(
        bytes memory callData,
        UserOpInfo memory opInfo,
        bytes calldata context
    ) external returns (uint256 actualGasCost) {
        uint256 preGas = gasleft();
        require(msg.sender == address(this), "AA92 internal call only");
        MemoryUserOp memory mUserOp = opInfo.mUserOp;

        uint callGasLimit = mUserOp.callGasLimit;
        unchecked {
            // handleOps was called with gas limit too low. abort entire bundle.
            if (
                gasleft() < callGasLimit + mUserOp.verificationGasLimit + 5000
            ) {
                assembly {
                    mstore(0, INNER_OUT_OF_GAS)
                    revert(0, 32)
                }
            }
        }
```

```

            IPaymaster.PostOpMode mode = IPaymaster.PostOpMode.opSucceeded;
            if (callData.length > 0) {
                bool success = Exec.call(mUserOp.sender, 0, callData, callGasLimit);
                if (!success) {
                    bytes memory result = Exec.getReturnData(REVERT_REASON_MAX_LEN);
                    if (result.length > 0) {
                        emit UserOperationRevertReason(
                            opInfo.userOpHash,
                            mUserOp.sender,
                            mUserOp.nonce,
                            result
                        );
                    }
                    mode = IPaymaster.PostOpMode.opReverted;
                }
            }

            unchecked {
                uint256 actualGas = preGas - gasleft() + opInfo.preOpGas;
                // Note: opIndex is ignored (relevant only if mode==postOpReverted, which
                return _postExecution(0, mode, opInfo, context, actualGas);
            }
        }

        /// @inheritdoc IEntryPoint
        function getUserOpHash(
            UserOperation calldata userOp
        ) public view returns (bytes32) {
            return
                keccak256(abi.encode(userOp.hash(), address(this), block.chainid));
        }

        /**
         * Copy general fields from userOp into the memory opInfo structure.
         * @param userOp  - The user operation.
         * @param mUserOp - The memory user operation.
         */
        function _copyUserOpToMemory(
            UserOperation calldata userOp,
            MemoryUserOp memory mUserOp
        ) internal pure {
            mUserOp.sender = userOp.sender;
            mUserOp.nonce = userOp.nonce;
            mUserOp.callGasLimit = userOp.callGasLimit;
            mUserOp.verificationGasLimit = userOp.verificationGasLimit;
            mUserOp.preVerificationGas = userOp.preVerificationGas;
            mUserOp.maxFeePerGas = userOp.maxFeePerGas;
            mUserOp.maxPriorityFeePerGas = userOp.maxPriorityFeePerGas;
            bytes calldata paymasterAndData = userOp.paymasterAndData;
            if (paymasterAndData.length > 0) {
                require(
                    paymasterAndData.length >= 20,
                    "AA93 invalid paymasterAndData"
                );
                mUserOp.paymaster = address(bytes20(paymasterAndData[:20]));
            } else {
                mUserOp.paymaster = address(0);
            }
        }

        /**
         * Get the required prefunded gas fee amount for an operation.
         * @param mUserOp - The user operation in memory.
         */
        function _getRequiredPrefund(
            MemoryUserOp memory mUserOp
        ) internal pure returns (uint256 requiredPrefund) {
            unchecked {
                // When using a Paymaster, the verificationGasLimit is used also to as a
                // Our security model might call postOp eventually twice.
                uint256 mul = mUserOp.paymaster != address(0) ? 2 : 1;
                uint256 requiredGas = mUserOp.callGasLimit +
                    mUserOp.verificationGasLimit *
                    mul +
                    mUserOp.preVerificationGas;

                requiredPrefund = requiredGas * mUserOp.maxFeePerGas;
            }
        }

        /**
         * Create sender smart contract account if init code is provided.
         * @param opIndex  - The operation index.
         * @param opInfo   - The operation info.
         * @param initCode - The init code for the smart contract account.
         */
        function _createSenderIfNeeded(
            uint256 opIndex,
            UserOpInfo memory opInfo,
            bytes calldata initCode
        ) internal {
            if (initCode.length != 0) {
                address sender = opInfo.mUserOp.sender;
                if (sender.code.length != 0)
                    revert FailedOp(opIndex, "AA10 sender already constructed");
                address sender1 = senderCreator.createSender{
                    gas: opInfo.mUserOp.verificationGasLimit
                }(initCode);
                if (sender1 == address(0))
                    revert FailedOp(opIndex, "AA13 initCode failed or OOG");
                if (sender1 != sender)
                    revert FailedOp(opIndex, "AA14 initCode must return sender");
                if (sender1.code.length == 0)
                    revert FailedOp(opIndex, "AA15 initCode must create sender");
                address factory = address(bytes20(initCode[0:20]));
                emit AccountDeployed(
                    opInfo.userOpHash,
                    sender,
                    factory,
                    opInfo.mUserOp.paymaster
                );
            }
        }

        /// @inheritdoc IEntryPoint
        function getSenderAddress(bytes calldata initCode) public {
            address sender = senderCreator.createSender(initCode);
            revert SenderAddressResult(sender);
        }

        /**
         * Call account.validateUserOp.
         * Revert (with FailedOp) in case validateUserOp reverts, or account didn't send
         * Decrement account's deposit if needed.
         * @param opIndex         - The operation index.
         * @param op              - The user operation.
         * @param opInfo          - The operation info.
         * @param requiredPrefund - The required prefund amount.
         */
        function _validateAccountPrepayment(
            uint256 opIndex,
            UserOperation calldata op,
            UserOpInfo memory opInfo,
            uint256 requiredPrefund
        )
            internal
            returns (
                uint256 gasUsedByValidateAccountPrepayment,
                uint256 validationData
            )
        {
            unchecked {
                uint256 preGas = gasleft();
                MemoryUserOp memory mUserOp = opInfo.mUserOp;
                address sender = mUserOp.sender;
                _createSenderIfNeeded(opIndex, opInfo, op.initCode);
                address paymaster = mUserOp.paymaster;
                uint256 missingAccountFunds = 0;
                if (paymaster == address(0)) {
                    uint256 bal = balanceOf(sender);
                    missingAccountFunds = bal > requiredPrefund
                        ? 0
                        : requiredPrefund - bal;
                }
                try
                    IAccount(sender).validateUserOp{
                        gas: mUserOp.verificationGasLimit
                    }(op, opInfo.userOpHash, missingAccountFunds)
                returns (uint256 _validationData) {
                    validationData = _validationData;
                } catch {
                    revert FailedOpWithRevert(opIndex, "AA23 reverted", Exec.getReturnDat
                }
                if (paymaster == address(0)) {
                    DepositInfo storage senderInfo = deposits[sender];
                    uint256 deposit = senderInfo.deposit;
                    if (requiredPrefund > deposit) {
                        revert FailedOp(opIndex, "AA21 didn't pay prefund");
                    }
                    senderInfo.deposit = uint112(deposit - requiredPrefund);
                }
                gasUsedByValidateAccountPrepayment = preGas - gasleft();
            }
        }

        /**
         * In case the request has a paymaster:
         *  - Validate paymaster has enough deposit.
         *  - Call paymaster.validatePaymasterUserOp.
         *  - Revert with proper FailedOp in case paymaster reverts.
         *  - Decrement paymaster's deposit.
         * @param opIndex                            - The operation index.
         * @param op                                 - The user operation.
         * @param opInfo                             - The operation info.
         * @param requiredPreFund                    - The required prefund amount.
         * @param gasUsedByValidateAccountPrepayment - The gas used by _validateAccountPr
         */
        function _validatePaymasterPrepayment(
            uint256 opIndex,
            UserOperation calldata op,
            UserOpInfo memory opInfo,
            uint256 requiredPreFund,
            uint256 gasUsedByValidateAccountPrepayment
        ) internal returns (bytes memory context, uint256 validationData) {
            unchecked {
                MemoryUserOp memory mUserOp = opInfo.mUserOp;
                uint256 verificationGasLimit = mUserOp.verificationGasLimit;
                require(
                    verificationGasLimit > gasUsedByValidateAccountPrepayment,
                    "AA41 too little verificationGas"
                );
                uint256 gas = verificationGasLimit -
                    gasUsedByValidateAccountPrepayment;

                address paymaster = mUserOp.paymaster;
                DepositInfo storage paymasterInfo = deposits[paymaster];
                uint256 deposit = paymasterInfo.deposit;
                if (deposit < requiredPreFund) {
                    revert FailedOp(opIndex, "AA31 paymaster deposit too low");
                }
                paymasterInfo.deposit = uint112(deposit - requiredPreFund);
                try
                    IPaymaster(paymaster).validatePaymasterUserOp{gas: gas}(
                        op,
                        opInfo.userOpHash,
                        requiredPreFund
                    )
                returns (bytes memory _context, uint256 _validationData) {
                    context = _context;
                    validationData = _validationData;
                } catch {
                    revert FailedOpWithRevert(opIndex, "AA33 reverted", Exec.getReturnDat
                }
            }
        }

        /**
         * Revert if either account validationData or paymaster validationData is expired
         * @param opIndex                 - The operation index.
         * @param validationData          - The account validationData.
         * @param paymasterValidationData - The paymaster validationData.
         * @param expectedAggregator      - The expected aggregator.
         */
        function _validateAccountAndPaymasterValidationData(
            uint256 opIndex,
            uint256 validationData,
            uint256 paymasterValidationData,
            address expectedAggregator
        ) internal view {
            (address aggregator, bool outOfTimeRange) = _getValidationData(
                validationData
            );
            if (expectedAggregator != aggregator) {
                revert FailedOp(opIndex, "AA24 signature error");
            }
            if (outOfTimeRange) {
                revert FailedOp(opIndex, "AA22 expired or not due");
            }
            // pmAggregator is not a real signature aggregator: we don't have logic to ha
            // Non-zero address means that the paymaster fails due to some signature chec
            address pmAggregator;
            (pmAggregator, outOfTimeRange) = _getValidationData(
                paymasterValidationData
            );
            if (pmAggregator != address(0)) {
                revert FailedOp(opIndex, "AA34 signature error");
            }
            if (outOfTimeRange) {
                revert FailedOp(opIndex, "AA32 paymaster expired or not due");
            }
        }

        /**
         * Parse validationData into its components.
         * @param validationData - The packed validation data (sigFailed, validAfter, val
         */
        function _getValidationData(
            uint256 validationData
        ) internal view returns (address aggregator, bool outOfTimeRange) {
            if (validationData == 0) {
                return (address(0), false);
            }
            ValidationData memory data = _parseValidationData(validationData);
            // solhint-disable-next-line not-rely-on-time
            outOfTimeRange = block.timestamp > data.validUntil || block.timestamp < data.
            aggregator = data.aggregator;
        }

        /**
         * Validate account and paymaster (if defined) and
         * also make sure total validation doesn't exceed verificationGasLimit.
         * This method is called off-chain (simulateValidation()) and on-chain (from hand
         * @param opIndex - The index of this userOp into the "opInfos" array.
         * @param userOp  - The userOp to validate.
         */
        function _validatePrepayment(
            uint256 opIndex,
            UserOperation calldata userOp,
            UserOpInfo memory outOpInfo
        )
            internal
            returns (uint256 validationData, uint256 paymasterValidationData)
        {
            uint256 preGas = gasleft();
            MemoryUserOp memory mUserOp = outOpInfo.mUserOp;
            _copyUserOpToMemory(userOp, mUserOp);
            outOpInfo.userOpHash = getUserOpHash(userOp);

            // Validate all numeric values in userOp are well below 128 bit, so they can
            // and multiplied without causing overflow.
            uint256 maxGasValues = mUserOp.preVerificationGas |
                mUserOp.verificationGasLimit |
                mUserOp.callGasLimit |
                userOp.maxFeePerGas |
                userOp.maxPriorityFeePerGas;
            require(maxGasValues <= type(uint120).max, "AA94 gas values overflow");

            uint256 gasUsedByValidateAccountPrepayment;
            uint256 requiredPreFund = _getRequiredPrefund(mUserOp);
            (
                gasUsedByValidateAccountPrepayment,
                validationData
            ) = _validateAccountPrepayment(
                opIndex,
                userOp,
                outOpInfo,
                requiredPreFund
            );

            if (!_validateAndUpdateNonce(mUserOp.sender, mUserOp.nonce)) {
                revert FailedOp(opIndex, "AA25 invalid account nonce");
            }
            bytes memory context;
            if (mUserOp.paymaster != address(0)) {
                (context, paymasterValidationData) = _validatePaymasterPrepayment(
                    opIndex,
                    userOp,
                    outOpInfo,
                    requiredPreFund,
                    gasUsedByValidateAccountPrepayment
                );
            }
            unchecked {
                uint256 gasUsed = preGas - gasleft();

                if (userOp.verificationGasLimit < gasUsed) {
                    revert FailedOp(opIndex, "AA40 over verificationGasLimit");
                }
                outOpInfo.prefund = requiredPreFund;
                outOpInfo.contextOffset = getOffsetOfMemoryBytes(context);
                outOpInfo.preOpGas = preGas - gasleft() + userOp.preVerificationGas;
            }
        }

        /**
         * Process post-operation, called just after the callData is executed.
         * 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
         * @param opIndex   - Index in the batch.
         * @param mode      - Whether is called from innerHandleOp, or outside (postOpRev
         * @param opInfo    - UserOp fields and info collected during validation.
         * @param context   - The context returned in validatePaymasterUserOp.
         * @param actualGas - The gas used so far by this user operation.
         */
        function _postExecution(
            uint256 opIndex,
            IPaymaster.PostOpMode mode,
            UserOpInfo memory opInfo,
            bytes memory context,
            uint256 actualGas
        ) private returns (uint256 actualGasCost) {
            uint256 preGas = gasleft();
            unchecked {
                address refundAddress;
                MemoryUserOp memory mUserOp = opInfo.mUserOp;
                uint256 gasPrice = getUserOpGasPrice(mUserOp);

                address paymaster = mUserOp.paymaster;
                if (paymaster == address(0)) {
                    refundAddress = mUserOp.sender;
                } else {
                    refundAddress = paymaster;
                    if (context.length > 0) {
                        actualGasCost = actualGas * gasPrice;
                        if (mode != IPaymaster.PostOpMode.postOpReverted) {
                            try IPaymaster(paymaster).postOp{
```

```
                        try IPaymaster(paymaster).postOp{
                            gas: mUserOp.verificationGasLimit
                        }(mode, context, actualGasCost)
                        // solhint-disable-next-line no-empty-blocks
                        {} catch {
                            bytes memory reason = Exec.getReturnData(REVERT_REASON_MA
                            revert PostOpReverted(reason);
                        }
                    }
                }
            }
            actualGas += preGas - gasleft();

            // Calculating a penalty for unused execution gas
            {
                uint256 executionGasLimit = mUserOp.callGasLimit;
                // Note that 'verificationGasLimit' here is the limit given to the 'p
                if (context.length > 0){
                    executionGasLimit += mUserOp.verificationGasLimit;
                }
                uint256 executionGasUsed = actualGas - opInfo.preOpGas;
                // this check is required for the gas used within EntryPoint and not
                if (executionGasLimit > executionGasUsed) {
                    uint256 unusedGas = executionGasLimit - executionGasUsed;
                    uint256 unusedGasPenalty = (unusedGas * PENALTY_PERCENT) / 100;
                    actualGas += unusedGasPenalty;
                }
            }

            actualGasCost = actualGas * gasPrice;
            if (opInfo.prefund < actualGasCost) {
                revert FailedOp(opIndex, "AA51 prefund below actualGasCost");
            }
            uint256 refund = opInfo.prefund - actualGasCost;
            _incrementDeposit(refundAddress, refund);
            bool success = mode == IPaymaster.PostOpMode.opSucceeded;
            emit UserOperationEvent(
                opInfo.userOpHash,
                mUserOp.sender,
                mUserOp.paymaster,
                mUserOp.nonce,
                success,
                actualGasCost,
                actualGas
            );
        } // unchecked
    }

    /**
     * The gas price this UserOp agrees to pay.
     * Relayer/block builder might submit the TX with higher priorityFee, but the use
     * @param mUserOp - The userOp to get the gas price from.
     */
    function getUserOpGasPrice(
        MemoryUserOp memory mUserOp
        MemoryUserOp memory mUserOp
    ) internal view returns (uint256) {
        unchecked {
            uint256 maxFeePerGas = mUserOp.maxFeePerGas;
            uint256 maxPriorityFeePerGas = mUserOp.maxPriorityFeePerGas;
            if (maxFeePerGas == maxPriorityFeePerGas) {
                //legacy mode (for networks that don't support basefee opcode)
                return maxFeePerGas;
            }
            return min(maxFeePerGas, maxPriorityFeePerGas + block.basefee);
        }
    }

    /**
     * The minimum of two numbers.
     * @param a - First number.
     * @param b - Second number.
     */
    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    /**
     * The offset of the given bytes in memory.
     * @param data - The bytes to get the offset of.
     */
    function getOffsetOfMemoryBytes(
        bytes memory data
    ) internal pure returns (uint256 offset) {
        assembly {
            offset := data
        }
    }

    /**
     * The bytes in memory at the given offset.
     * @param offset - The offset to get the bytes from.
     */
    function getMemoryBytesFromOffset(
        uint256 offset
    ) internal pure returns (bytes memory data) {
        assembly {
            data := offset
        }
    }

    /// @inheritdoc IEntryPoint
    function delegateAndRevert(address target, bytes calldata data) external {
        (bool success, bytes memory ret) = target.delegatecall(data);
        revert DelegateAndRevert(success, ret);
    }
}
```